The Refined-Language Approach To Compiling For Parallel Supercomputers

by

Henry G. Dietz

May 1987

CIRRICULUM VITAE

May 1987

Henry G. Dietz

Born November 12, 1959 in Rockville Centre, New York, Dietz received his primary and secondary education in the Garden City, New York, public school system. From Fall 1977 through Spring 1979 he attended Columbia University. From Fall 1979 he attended the Polytechnic Institute of New York (now Polytechnic University), where he earned both B.S. and M.S. degrees in Computer Science. He successfully defended his Ph.D. dissertation in August 1986.

While a graduate student at Polytechnic University, Dietz was supported by several special (research) fellowships. There, in addition to overseeing the Sperry Microprocessor Systems Laboratory, he led a number of research efforts involving computer aided instruction, semantic translation, parallel computer architecture, digital halftoning, and parallelizing compilers. He is author, or co-author, of over one dozen research reports and monographs, as well as author or co-author of numerous published papers.

As an Academic Associate at Polytechnic University and as an Adjunct Professor at both Polytechnic University and Stevens Institute of Technology, he taught and reorganized a number of courses including both schools' graduate courses in Compiler Design and Construction. Since August 1986, Dietz has held an appointment as Assistant Professor of Electrical Engineering (Computer Engineering) at Purdue University.

In September 2017, Dr. Dietz is a Professor, and James F. Hardymon Chair in Networking, in the Electrical and Computer Engineering Department at the University of Kentucky in Lexington, Kentucky. His current contact information is posted at `http://aggregate.org/hankd`.

To my parents,

Henry and Gloria Dietz.

AN ABSTRACT

The Refined-Language Approach To Compiling For Parallel Supercomputers

by

Henry G. Dietz

Advisor: A. David Klappholz

Submitted in Partial Fulfillment of the Requirements

for the Degree of Doctor of Philosophy (Computer Science)

June 1987

To produce efficient code for a program which is to be executed using a particular parallel supercomputer, it is vital that the compiler employ detailed knowledge of both the algorithm and the target machine. However, the specifications of algorithms in conventional, sequential, languages often are not sufficiently precise; information which the programmer knew, and which the compiler must use to generate efficient parallel code, is either hidden or lost because the language does not provide a way to express that information. In addition, although many compilation technologies have been developed to support specific classes of speedup-oriented parallel supercomputer, relatively little has been done to unite these different compilation technologies in a consistent view of code generation for all parallel machines; this has often led to extremely clever code generation resulting in code which is inappropriate for the target machine and therefore slow.

The refined-language approach is an holistic strategy for solving the problem of compiling, and programming, for speedup-oriented parallel supercomputers. So that information relevant to parallelization of an algorithm is not lost when the program is written, we provide guidelines for “refining” the definition of programming languages to permit the programmer to explicitly state, as part of his/her programs, this information. We further propose a framework for automatic parallelization and related compiler technologies which spans many classes of architecture and highlights the close relationship between conventional optimization and optimizing parallelization. Several new parallelization techniques are embedded in this discussion.

A prototype refined-C (RC) compiler, based on full ANSI C and targeted to a MIMD supercomputer, has been built. Although this compiler has not been thoroughly benchmarked, it has demonstrated both that language refinement greatly improves the speed and accuracy of compile-time analysis and that, at least in some cases, efficient parallel code can be generated for constructs which would have been sequential using other techniques.

Table Of Contents

Table Of Figures

Table Of Listings

This page is intentionally blank.

Introduction

Programming a parallel computer in an explicitly-parallel language is a difficult, and extremely error-prone, endeavor. It would be far more productive if it were possible to program parallel machines in conventional-looking sequential high-level languages. Previous attempts to make this possible have been concerned primarily with automatic parallelization of standard FORTRAN code.

Many of the parallelization techniques which have been developed for this purpose, especially those for parallelizing loops, are very powerful; constructs such as simple FORTRAN `DO` loops can be transformed into nearly ideal parallel code for any of a number of machine types. However, the total amount of parallelism derived from typical FORTRAN code is not sufficient to fully utilize the power of some advanced parallel computers currently available and will be grossly insufficient for the next generation of parallel machines. Results using conventional sequential languages other than FORTRAN have been even more disappointing, in terms of the amount of useful parallelism found; virtually no parallelizing compilers have been built for other languages.

The refined-language methodology is a comprehensive approach toward dramatically increasing the amount of useful parallelism which will be automatically derived from typical sequentially control-structured code. Because the methodology deals directly with the causes of parallelization analysis failures, it is far less sensitive to the choice of base language: Ada, C, FORTRAN, Lisp, Pascal, Prolog, etc. In fact, the C programming language, which is well-known to be nearly undecipherable using conventional parallelization analysis, was the first language to which the methodology was applied, and with remarkable success.

1. Refined Languages

The refined-language approach begins with a conventional HLL (High-Level Language) which has been stripped of its support for the explicit specification of parallel execution with inter-process communication and synchronization. It is impossible for a programmer to write a race condition in such a language — either deliberately or (far more likely) by accident.

Since the definition of “race condition” has become somewhat fuzzy, it is important to define it as it is used in this paper: a race condition occurs wherever contention for a named object is neither explicitly nor implicitly resolved in the program. In other words, if two or more segments of a program attempt to obtain access to the same resource (e.g., try to write to the same memory location), unless the program specifies which segment takes precedence, the program embodies a race. Sequentially control-structured code orders all operations in the program, hence it insures that there will be no races. It does not, however, insure that the same answer will be computed every time a program is recompiled and run with the same input — data-dependent computational failures such as round-off problems caused by re-ordering computations (without violating the ordering of accesses to any single resource) are not race conditions and certainly may occur within code generated from a sequential program. These data-dependent failures (discussed in section 7:1.2.3) are not as troublesome as races — they are far easier to debug since they do not become more complex in proportion to the parallelism-width of the target machine, as races do. In summary, by disallowing explicit parallelism, we insure debugability.

The refined-language designer next examines the remaining constructs, which include most of the constructs of the base language, to identify particular constructs which deter a flow-analyzing compiler from discovering precisely which data are accessed by what code in what way. This**data access** information guides the compiler in preserving correctness in automatic parallelization; imprecision forces the compiler to preserve correctness of code by not transforming it — leaving it sequential. Therefore, any constructs which blur the compiler's understanding must be either removed from the language or restricted in their usage so that they no longer produce this ill effect. Finally, a few new constructs may be added to help compensate for any loss of expressive power and/or to aid in specification of precise data-access flow.

The resulting language — a refined version of the base language — is a dialect of the base language. A programmer accustomed to the base language will find the refined version to be very familiar and easy to use, if not directly compatible with his/her old code and some of his/her programming habits. The key benefit is that the use of a refined language makes it possible for a target-machine-dependent, flow-analyzing, compiler to transform each refined-language program into reasonably efficient, race-free, parallel code for nearly *any* machine. Better efficiency will result from better algorithms, but each algorithm will execute nearly as well as the match between algorithm and machine permits.

2. Compilation Of Refined Languages

The bulk of the present work discusses the refined-language methodology as it applies to compiling for MIMD[1] computers, but only because:

- MIMDs are difficult targets for other automatic parallelization techniques, hence, success with a MIMD target machine clearly demonstrates the value of the methodology.
- As an historical accident, the first few refined language compilers are being implemented for MIMDs, of one flavor or another. Test mock-ups of the compilation techniques discussed in the present work were built as fragments of an RC (Refined C) compiler targeted for a machine which is a dynamically-scheduled MIMD with a small VLIW multiprocessor at each node [PaK86] [DiK85]. The first complete RC compiler [Ste86] is targeted for the IBM RP3 [PfB85], which is also a MIMD.

At its heart, the refined-language methodology incorporates the understanding that precise static analysis of data access flow is the key to transformability — particularly transformation into efficient target-machine-dependent parallel code. Since a refined-language program is expected to undergo transformation, and since much more information is available in a refined-language program, targeting code to any MIMD, SIMD, VLIW, vector/array, pipelined, systolic, etc. machine is tedious, but straightforward. (In fact, since

1. Detailed discussion of parallel supercomputer architectures appears in Chapter 6.

program transformability for conventional code optimization is also enhanced, a refined-language program can often be compiled into more efficient code for a sequential computer than can be generated from the corresponding base-language program.)

In the refined-language methodology, a program is not seen as representing a unique executable formulation, but, rather, as representing a large class of allowable executable formulations. Given that a refined-language program explicitly contains data access flow information relevant to parallelization, the portion of this information needed for a particular machine is easily extracted by conventional flow analysis techniques and can be applied to generate an efficient (parallel) formulation for that machine.

Although previously proposed techniques for compiling base-language code (most notably FORTRAN) for various parallel machines could be used for refined language compilation, the refined-language methodology includes a non-deterministic approach to parallel code generation (henceforth called “process packaging”) for various types of target machine. This approach is consistent with the idea that a refined language program specifies many possible parallel-executable codings, and that the generated code should be the best of the alternatives for the particular target machine. Indeed, the recognition that complex machine dependencies dominate the generation of efficient parallel code is the fundamental reason that refined languages attempt to remove this responsibility from the programmer. It is better to suffer the creation of a complex compiler once (per machine) than to have every programmer suffer in creating every application program.

3. Programming Support

It has often been said that the best way to write a piece of code is to modify similar code which was written earlier. This poses two problems: how is code reused and how can it be modified or improved?

3.1. Reuse Of Existing Code

Reuse of code modules is not practical unless the modules are written in the same language being used to write the new code. Applied to most languages, the refined-language methodology will make such direct reuse impossible; verbatim, a base-language program is not necessarily semantically or syntactically valid as a program in the corresponding refined-language.

To solve this problem, a software tool is created to convert base-language programs into their refined language *equivalents*.[2] This tool, generically called PREFINE,[3] performs analysis very similar to that performed by a compiler attempting to generate parallelized code from a base-language program. Rather than generating parallel code, it gathers information relevant to paralellization and uses this to create an equivalent program in the corresponding refined language.

For example, to parallelize subroutine invocations, the compiler must be able to determine the data accesses made by called subroutines. However, these subroutines may be in separate compilation modules within other files. PREFINE performs the (relatively expensive) required analysis *over all modules of the program* to determine the data-access flow. The refined-language program it generates contains explicit specification of the data access flow.

A refined language compiler will find no more and no less to be executed in parallel within a PREFINEd program than would a similar parallelizing compiler analyzing the base-language version of the program — but it will do so using only localized, relatively inexpensive, flow-analysis. This conceptually small difference not only makes re-use of old code practical, but makes separate compilation of code modules feasible[4] — whereas the analysis necessary when using a conventional-language program would traditionally require the entire program to be examined whenever any module was recompiled.[5]

3.2. Modification Of Code

The advantage of using a refined language is not merely that expensive inter-module analysis is converted into inexpensive localized analysis. A refined language provides constructs which permit specification of data access flow information beyond that which could have been determined from a conventional program, regardless of the amount of effort invested in analysis.

As an example, consider the quicksort algorithm in a typical base-language version (see section 3:3.5). The major source of parallelism for many, if not most, machines is the parallel execution of the two

2. Since some base-language constructs are removed in the corresponding refined language, mechanical translation is not always feasible. For most languages, the occurrence of constructs which have no direct equivalent is very infrequent and the translation program can simply flag each such construct so that the programmer may rewrite it appropriately.
3. There is a different version of PREFINE for each base language.
4. A very few target architectures, such as some VLIW machines, may require code for the entire program to be generated as an integrated whole . . . thereby making separate compilation impossible despite the fact that the analysis can be performed on a single module at a time.
5. Work on programming environments, such as [CoK86] [AlB86], has recently developed the theory by which the analysis may be performed in a similar incremental fashion — provided that each program is written and modified using only the tools integrated into the environment. These tools collect essentially the same information that can be directly expressed in a refined-language program, but they maintain it in a compilation database which the user cannot directly access.

recursive calls. However, as quicksort appears in most conventional programs, it is impossible for the compiler to determine that the two recursive calls operate on disjoint portions of the array being sorted. In a refined language, the programmer can express this fact directly — and the compiler can confirm that the programmer was right before deciding to generate code which invokes the two calls in parallel.

If the obvious generalization of this problem is viewed as the compiler sees it, it becomes clear that the parallelizing compiler *knows* where it has only approximate data-access flow information and knows roughly what the penalty is for the imprecision, in terms of loss of parallelism for the target machine. A software tool, generically called REFLEX[6], will make use of this knowledge to prompt the programmer for more precise information to resolve the most costly ambiguities. The information the programmer supplies will become an integral part of the program. It is integrated not just in that it appears in the same file, but also in that it reads as part of the program with meaning and purpose which are evident in that context.

Together with PREFINE, REFLEX will allow old code to be reused and also to be systematically improved. Data-access flow is key not only to the compiler's understanding of the code, but also to the programmer's understanding: each improvement not only makes the program run faster, but makes it easier for a human to understand. It is not entirely by coincidence that many of the refined-language constructs closely resemble those new constructs proposed in the name of software engineering and software reliability.

4. Overview Of This Document

As we have indicated, the refined language methodology is not about only one aspect of the problem of programming parallel computers, but is a complete and systematic view of the entire task. It is convenient in this presentation, however, to partition the material into two parts.

In the first part (Chapters 2 and 3) of the current work, the programming language design issues are explored and several examples of refined languages are given. Chapter 2 compares the various alternative approaches to the programming of parallel computers and discusses the relationships between them. In Chapter 3, the process of designing a refined version of a language is discussed in detail, with examples from RF and RC (refined FORTRAN and C).

The concept of and rationale for refined languages is entirely new, although many have recognized the problems and the constructs used in solving them had, in several cases, been proposed by others for

6. There will be different versions of REFLEX, one for each refined language on each machine. Although the basic design of REFLEX is well understood, it has not yet been implemented.

completely different reasons. Data access flow, a new model of computation, underlies and motivates these refinements.

A scheme for compilation of refined-language programs into efficient code for MIMDs, SIMDs, VLIWs, and many other classes of parallel machine is given in the second part (Chapters 4 through 8). Chapter 4 gives an overview of the complete compilation process. Chapter 5 outlines the principles of concurrency detection, the machine-independent portion of automatic parallelization. In Chapter 6, as an introduction to machine-dependent parallelization concerns, we present an overview of parallel computer architecture “as seen by” a parallelizing compiler. Chapters 7 and 8 discuss machine-dependent parallelization techniques based, respectively, on loops and irregular code.

While many of the compilation techniques discussed were developed by others, the techniques operating on irregular code are new, as is the statement of the organization of the entire process. The formulation of the compilation analysis using the terminology of conventional (uniprocessor-oriented) optimization is also unusual, and we believe many observations given in the discussion are unique in this context.

Chapter 9 summarizes the contributions of the refined-language approach to compiling for parallel supercomputers. Several directions for further research are also discussed.

Languages

For Parallel Computers

There have been many different approaches to programming parallel computers. The reason is simply that the goals of these techniques are not the same — which is also why there are many different languages for conventional machines. In the case of refined languages, our primary goal is to obtain efficient speedup in execution of programs through proper use of machine parallelism. This is not to say, for example, that refined languages could not be used to program machines which employ parallelism only in support of fault-tolerance or toward distributing control and sensing operations in real-time. Rather, it acknowledges that the fundamental trade-offs are optimized for obtaining efficient speedup through parallelism, and that other kinds of application may be better served by different optimizations.

Therefore, in considering competing approaches, we consider only those approaches which have sought the same goal.

1. Speedup-Oriented Languages

Rather than considering each individual speedup-oriented language which has been proposed for programming parallel computers as a separate approach, we will group similar languages together and describe the properties of these groups. To make the interrelationships of these groups clearer, it is useful to consider the "family tree" of concepts used in languages for programming parallel computers, as illustrated by the following graph:

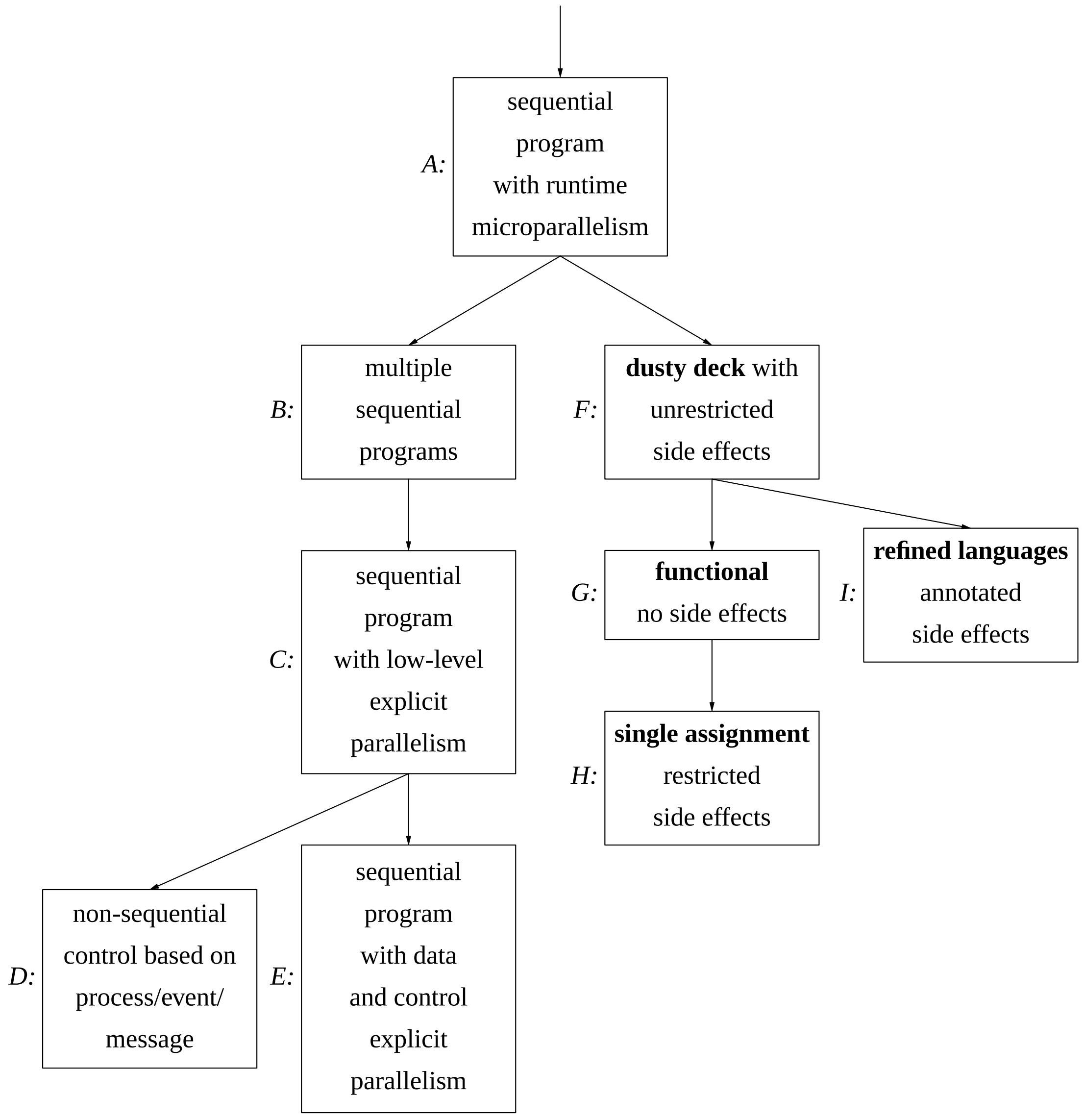

A:
sequential program with runtime microparallelism
B:
multiple sequential programs
F:
dusty deck with unrestricted side effects
C:
sequential program with low-level explicit parallelism
G:
functional no side effects
I:
refined languages annotated side effects
H:
single assignment restricted side effects
D:
non-sequential control based on process/event/ message
E:
sequential program with data and control explicit parallelism

Figure 2:1: “Family Tree” Of Languages For Parallel Machines

In the above graph, each node represents a group of languages which share a fundamental approach to the exploitation of parallelism in program execution.

In characterizing each group, it is useful to point-out that the groups labeled *B*, *C*, *D*, and *E* are all based on direct programmer specification of parallelism, whereas *F*, *G*, *H*, and *I* (the refined-languages proposed in the present work) are based on automatic compile-time detection of parallelism in sequential code.

Group *A*, the earliest technique used to exploit hardware parallelism, involves the use of automatic run-time detection of parallelism in sequential code. For example, a `Load` instruction followed by an `Add` might be accomplished with the last few cycles of the `Load` instruction overlapping the first few cycles of the `Add`, if the `Add` instruction does not use the target register of the `Load`. This is a particularly successful approach in that usable parallelism can be detected regardless of the source language; unfortunately, the speedup obtained in this way is nearly always relatively small. The computer hardware does not have the ability to examine the entire code sequence simultaneously, hence, it can only parallelize execution of non-interfering instructions within a small window on the execution stream.

Assuming that the machine is being programmed using an HLL, the easiest code sequence to generate would usually be one in which nearly all operations are linked by sequential re-use of their registers. To help insure that some overlap will be possible, compilers for microparallel machines have long performed a few simple optimizations which locally re-arrange instructions to permit greater overlap in execution [CoS70]. These optimizations closely resemble those used more recently to fill pipeline delay slots on RISC machines [Gro83], except that the compiler for a RISC processor may need to insert null “padding” operations.

1.1. Explicit Specification Of Parallelism

It is quite obvious that, for example, computing the trajectories of ten independent missiles can be sped-up by at least a factor of ten using parallelism which *seems to be inherent in the specification of the problem*: instead of computing the trajectory for each missile in sequence on a single machine, all ten can be computed simultaneously by computing one trajectory on each of ten separate processors. Programming techniques based on the use of explicit parallelism generally embrace the concept that, as with the trajectory problem just given, most problems have a natural parallel decomposition — often called the “maximally parallel” form. Groups *B*, *C*, *D*, and *E* differ only in the features incorporated in the languages to permit specification of this parallelism.

1.1.1. *B:* Multiple Sequential Programs

Perhaps the easiest explicit-parallelism technique to support is the use of multiple sequential programs. Typically, programs can communicate with each other only through the file system, but this is sufficient for parallelizations like that described above. Of course, the cost of frequent communication or synchronization is very high — each process must be relatively large and should execute independently of its siblings, otherwise these costs will dominate the program's execution time.

Good examples of systems which exploit this approach would include the many commercial microcomputers and minicomputers which support from two to about a dozen processors: the Discovery multimicroprocessor, the dual-processor Prime and Gould machines, the DEC VAX VMS network, etc.

1.1.2. *C:* Low-Level Explicit Parallelism

The high communication and synchronization overhead of technique *B* makes only very large-grain parallelism useful. A number of sequential languages have been modified to provide relatively low-overhead primitives for these purposes, thereby permitting use of finer-grain parallelism. Primitives, often in the form of built-in functions, usually include simple semaphore operations, spawn/fork and join/wait, and some way to establish communication between processes (memory-mapping commands, simple message buffers, pipes, or sockets). These primitives operate at a very low-level, making the correct specification of a complex parallel execution stream a time consuming and highly error-prone programming task. A small error in the use of the primitives will often cause a race condition or deadlock — the debugging of which is an art few have mastered.

Good examples of this approach include the BBN Uniform Programming System [Hof79], IBM's EPEX [Nor86], Pisces 1 and 2 [Pra85] [Pra86], Poker [SnS86], The Force [Jor87], and several parallel computer versions of Berkeley UNIX[7] (such as that of the Sequent Balance [Ost85]).

1.1.3. *D:* Non-Sequential Control

Since the expression of parallelism is the key problem, a number of approaches have been developed from high-level formalisms (such as [Dij75]) for describing the parallel execution of code. Probably the best known such formalism language is Hoare's CSP [Hoa78] [DeS86], which abstracts the description of parallel execution structure to a high enough level so that it is relatively straightforward to use. Other well-known examples include OCCAM [Inm84] and LINDA [Gel86].

A basic limitation of this approach is that it specifies so much that it makes it extremely difficult, if not impossible, for a compiler to generate a process structure other than that represented by the program. For example, most programs written to use message passing cannot be converted into forms which would have good execution efficiency on a vector processor, or on any other machine which cannot efficiently

7. UNIX is a trademark of AT&T Bell Labs.

simulate message passing. Programming efficiency is good once the programmer is familiar with the parallelism model, but a change in the underlying hardware can easily bring programming efficiency to zero. Reasonable execution efficiency depends critically on achieving a close match between the programmer's code and the computer's hardware — although the language provides efficient ways of specifying details, the compiler cannot insulate the programmer from having to specify them for the particular machine.

1.1.4. *E:* Data And Control Parallelism

Instead of stressing just the model of parallelism as in group *D*, this approach stresses modeling of the entire machine environment. For example, it is at least as difficult to decide where data is to be placed in a parallel machine with multiple local memories as it is to organize the parallelism of the control structure.

Actus [Per79] is one language which permits such specifications:

```
var a: array[1:4, 1..5] of integer;
```

means that the array a is to be arranged in memory so that parallel data references can occur across the first index of a. In other words, "the parallel dots indicate the index which is . . . spread across the processors in an array processor or stored contiguously in a vector processor" [Per79].

Actus also incorporates mechanisms for specification of enable bit-vectors, index ranges, and various vector functions which are modeled after the structure of array and vector machines. Languages supporting similar (vector-oriented) constructs include Glypnir [LaL75], Parallel Pascal [Ree84], Vector C [LiS85] [Mac85], Parallel C [KuS85], Concurrent C [GeR85], and, to some extent, ANSI FORTRAN 8x.

There are quite a few such languages. For much the same reason that modern computers still bother to define an assembly language, parallel supercomputers tend to use explicitly parallel languages.

PIE [SeR85] provides a software development environment with similar features.

1.2. Sequential Control Structures

Since the starting point in using a sequential language for programming parallel machines is a sequential program, and since no explicit parallelism invocation or synchronization constructs exist in a sequential language, *it is naturally impossible for the programmer to write a race condition in such a language*. If the compiler which transforms the code into a parallel form uses only correctness-preserving transformations, the resulting parallel program will be free of race conditions: the programmer is guaranteed that the parallel code will produce the same result as the sequential program — hence, the program will be debuggable.

Also in common to the various versions of the sequential language approach is the fact that the source program is extremely portable, both across a wide variety of parallel machines with different **efficiency-critical features** (ECFs)[8] and across most single-processor machines, with their various ECFs. The compiler, or at least some program which is always part of the compilation process (e.g. post-processor, assembler, etc.), carries the heavy burden of proper utilization of the target machine's ECFs. Since the compiler is responsible for choice of synchronization method, etc., it can easily generate the type most appropriate for the machine — or at least it can *consistently* generate code far closer to the ideal form than a human normally would.[9] This is not because a highly skilled programmer couldn't do better, but because few programmers are sufficiently skillful and very few of them can afford to spend the time needed to hand-optimize every piece of code they write.

Many different sequential languages have been proposed for programming parallel computers. The most significant feature of each of these sequential languages is, not surprisingly, the measures it employs toward enabling a compiler to detect operations which can be safely executed in parallel.

1.2.1. A Classification Scheme

In order to make objective comparison of the various sequential language approaches possible, we introduce a new classification scheme for sequential languages which are intended as input to concurrency-detecting compilers. The classification scheme is based on the ways in which *side-effects* may be expressed in the language, since imprecision in analysis of side-effects is the fundamental limit on a compiler's ability to transform a program into an appropriate parallel form. The structure of side-effects also defines the type of flow analysis which will be needed to support this transformation. This is easiest to understand by first explaining what flow analysis for concurrency detection must do.

8. An ECF is a feature of the target machine whose consideration is critical in obtaining good execution efficiency. For example, the number and kind of registers in a processor is usually an ECF. The kind of parallelism which is most efficiently used by a machine is an ECF. In general, any feature whose efficient utilization requires machine-dependent code is an ECF.

9. Notice here that we are talking about the code's form, not the appropriateness of the algorithm for the given machine. Current compiler technology provides only minor adjustment of algorithms to better match hardware — major algorithm changes remain the programmer's responsibility.

A side-effect is an execution-time binding of a value to a name. At runtime, values are manipulated by their association with names. In order to safely parallelize a program, a compiler must be able to insure that the same values are used in each parallelized computation as in the corresponding sequential computation. Therefore, for a compiler to generate a correct parallelization, the compiler must be able to fully understand how each side-effect affects each name.

On the surface, this seems simple enough because most side-effects are accomplished by the execution of assignment statements. But the value being associated with the name is usually not knowable at compile time except in terms of its relationships to other unknown values — the key questions are most often “Is this value the same as that one?” and “Is it certain that this value cannot be the same as that one?” questions which are often difficult or impossible to answer. For a conventional optimizing compiler, answering “maybe.” rarely carries a large penalty, but it will often cause a parallelizing compiler to find *no* useful parallelism at all.

Even discovering which name is being used can be very complex or impossible: consider assignment through a pointer or to an array variable subscripted by yet another variable. There are also problems caused by permitting separate compilation: the side-effect bearing code within another module may be concealed from the compiler while it is analyzing the code which uses the module . . . what the compiler doesn’t see can hurt the parallelization a lot.

The four fundamental variations on the theme of side-effects in sequential languages are: unrestricted side-effects, no side-effects, restricted side-effects, and annotated side-effects.

1.2.2. *F:* Unrestricted Side-Effects

Historically, the first kind of sequential language considered for parallel programming is the kind collectively known as FORTRAN. Most conventional languages which were not designed especially for the purpose of programming parallel machines (nor to help flow analysis in any way) permit use of unconstrained side-effects[10]; programs written in them may be analyzed in essentially the same way as standard FORTRAN code.

An obvious benefit of using such a language is that it is possible to take *existing* sequential programs and to obtain some parallelism from them. Unfortunately, the high frequency of occurrence of obscure side-effects in most conventional languages often makes the compiler’s flow analysis very difficult and/or

10. Some allow for more varieties of side-effects, but none, in their application to parallel machines, attempt to reduce the creation of side-effects.

the resulting data access information insufficiently precise for a large amount of parallelism to be detected. For example, consider the following fragment of code:

```
a[i] := b;
a[k] := a[j] * c;
```

Does the second statement *use* the value computed in the first? Does the second statement *kill* the value of `a[i]`? The answers to these questions depend on being able to determine if either (or both) of `a[j]` or `a[k]` could be an *alias* for `a[i]`. This, in turn, depends on knowing the relationships between `i`, `j`, and `k` — but `i`, `j`, and `k` may have relationships which are:

- able to be determined at compile-time only *by performing flow analysis on the entire program as a whole*, rather than individually on smaller regions of the program (for example, this would be the case if `i`, `j`, or `k` were assigned the value of an expression which was partly computed by another function) — the effort taken in performing flow analysis grows quickly as the size of the region involved increases, hence the *analysis becomes extremely expensive*; or
- *theoretically not able to be determined from the program at compile-time* (for example, if `i`, `j`, or `k`'s value is partly derived from input to the program) — *which does not necessarily imply that a dependency exists* (it may, for example, be impossible to determine by compile-time analysis that the two recursive calls in *quicksort* operate on unrelated portions of the array being sorted, yet, this is the case [DiK85a]).

In either of these situations, the compiler's flow analysis is forced to settle for answering both questions "maybe." Any other assumption, carried through into automatic parallelization, could result in a race condition if the compiler's guess was wrong.

Fortunately, such situations do not occur very often in FORTRAN programs, which are remarkably static compared to typical programs in other sequential languages. For example, FORTRAN doesn't provide pointers, recursion, nor dynamic memory allocation. This fact, combined with the overall simplicity of the language[11] and the multitude of existing application programs written in FORTRAN, has made the application of "dusty-deck" techniques to other conventional languages very rare.

Other unrestricted side-effect languages, especially languages like C, for which typical programs contain a multitude of pointer references and other constructs which cause these analysis problems, are parallelized with much poorer consistency than FORTRAN.

11. This refers mostly to ANSI FORTRAN 66, but even ANSI FORTRAN 77 is simpler than most languages, and so will be ANSI FORTRAN 8x.

The major execution efficiency benefit to using a conventional language is that so much is known about optimization of sequential code for these languages that, even if little parallelism is found, the execution efficiency *of each process* (considered individually) is likely to be very good.

The sections 2:1.2.1, 2:1.2.2, 2:1.2.3, and 2:1.2.4 briefly outline the approaches taken by some of the more active research groups in the field of “dusty deck” parallelization.

1.2.2.1. PARAPHRASE

PARAPHRASE [KuM72] [KuS84] is probably the best-known program for converting sequential programs into parallel code. It was probably the first such program and it continues to be improved. Along the way, it has motivated many other parallelizing compilers, such as PFC [AlK82], the KAP [KAI85] translators, and [All86].

The analysis and transformation schemes used by PARAPHRASE are primarily (though not exclusively) intended for number-crunching FORTRAN programs and vector-oriented target machines. The array of transformations performed definitely places it among the most sophisticated compilers. However, PARAPHRASE is not perfect:

- No extraordinary attempts are made to resolve aliasing ambiguities arising from unconstrained side-effects, be they intra- or inter- compilation unit. If a subprogram call (to a non-intrinsic routine) is made from within a parallelizable construct, PARAPHRASE fails to detect the parallelism. Ongoing research may soon correct this deficiency [Ban86].
- PARAPHRASE finds `DO` loops and performs dependence analysis on loop bodies at the level of simple FORTRAN statements. This makes it impossible for PARAPHRASE to consider low-level parallel machine ECFs (such as register allocation, etc.). For this reason, it would be extremely difficult to adapt PARAPHRASE to generate good code for machines which have many low-level ECFs — such as VLIWs and even some vector machines, as discussed in Chapter 6.

An improved version of PARAPHRASE, Mini-KAP/AF [KAI85], is currently available commercially.

1.2.2.2. BULLDOG

The VLIW (Very Long Instruction Word) machine [Fis84] [FiE84], is either a very wide microcoded SISD or a very static MIMD, depending on how one chooses to view it. In any case, it is a machine plagued by very strong dependence on very low-level parallel machine ECFs. The BULLDOG [Ell85] compiler was developed as an integral part of a VLIW machine design, to convert programs written in a small

FORTRAN subset into highly parallel code for this architecture.

BULLDOG uses numerous sophisticated program transformations, most of them cleverly adapted from previous work done in microcode compaction. It is among the few concurrency detecting compilers which do not resort to pattern matching for key transformations — it is completely analytic (and non-deterministic in its choice of parallelization). Most significant of all, BULLDOG embodies a very powerful technique for resolving the two kinds of *aliasing* problems described in the introduction to section 2:1.2.2.

The array subscript problem cited above can be restated as the pair of questions:

- Is it possible that `i - j = 0` ?
- Is it possible that `i - k = 0` ?

In BULLDOG, each *aliasing* question is answered by converting the problem into an equation and trying to solve it. This makes compilation slow, but works remarkably well.

When an inequality can be neither solved nor proven to have no solution, the compiler presents the inequality to the programmer and permits the programmer to insert assertions, also in the form of equations or inequalities. For example, in the case cited above, the programmer's response might be:

```
ASSERT i < j
ASSERT i < k
```

which allows BULLDOG to solve for the answer to the aliasing question. Most importantly, generation of parallel code based on the assertions is *safe*, since the assertions can be validated at run-time. These assertions can safely answer even questions whose answers are theoretically unknowable at compile-time.

The main difficulty in using this approach is that the assertions are often intuitively unrelated to the problem solved by a program. When the compiler prompts the programmer for an assertion, the programmer might not immediately know the answer nor understand why that particular question was asked. Therefore, the assertions do little toward making the program easier for a human to understand.

Early work on BULLDOG has been expanded to include non-VLIW architectures [Nic85]. Relative to the handling of side-effect analysis problems, [All83] presents a similar approach.

1.2.2.3. PTOOL

Programming environments, such as PTOOL [AlB86], attempt to make inter-compilation-unit flow analysis acceptably efficient. This is accomplished by incrementally collecting and merging information from separate compilation units as they are compiled.

While PTOOL does not yet attempt automatic parallelization of sequential (FORTRAN) code, it does provide a set of software tools with which the development of parallel code can be done far more efficiently than if it were attempted purely by hand. In principle, there is no reason that an automatically parallelizing compiler could not be incorporated in a programming environment. This work is currently underway [Bur84] [BuC86] [CoK86] [TrI86].

Integration of inter-procedural analysis is a very important step in "dusty deck" parallelization technology. However, once such a compiler has been constructed, it will only be able to solve one of the two problems presented in the introduction to section 2:1.2.2 — alone, programming environments provide no mechanism for the programmer to express his knowledge of the run-time referencing behavior of the program.

1.2.2.4. PREFINE

Independently from the programming environment researchers, we also observed that the cost of performing inter-compilation-unit analysis is not excessive provided that the bulk of information need not be determined anew with each compile. However, like the researchers who produced BULLDOG, we feel that this information should be a visible part of the program — useful in helping people, as well as the compiler, understand the program. In order to do this, however, the great bulk of the interprocedural and other data access information must be severely reduced.

C-PREFINE (CP) is a program which collects *only* the data access information needed for concurrency detection across compilation units. When given a C program as input, CP translates it into a valid refined C (RC) program, placing interprocedural information in an **interface specification** file which is included as a user-visible part of the RC program generated. From this point on, the program would be maintained as an RC program, and the C version ignored.

The RC compiler, discussed later, actually parallelizes the program. Another software tool aids the programmer in improving the RC code in much the same way the BULLDOG helps improve FORTRAN code.

There are several major distinctions between this and other approaches. One is that, unlike most "dusty deck" techniques, a version of PREFINE can be constructed for nearly any language. (A FORTRAN version is under development.) Another is that the notation used for the equivalent of BULLDOG's assertions in refined languages is directly related to the structure of the program, hence, the refined version of a program is nearly always easier for people, as well as compilers, to understand. Further, PREFINE completely isolates parallelization analysis from flow (including dependence) analysis.

1.2.3. Restricted/No Side-Effects

Considered as a group, most, if not all, the compilers which analyze and parallelize sequential languages permitting unrestricted side-effects share the following characteristics:

(1) They use complex flow analysis and transformation algorithms, many of which execute very slowly, and

(2) The solutions to the *aliasing* problems discussed above are obtained either not at all or only by the programmer's providing additional information about *aliases*.

Partly in response to these difficulties, and partly in an attempt to express the concept of dataflow [ArC84] execution in a high-level language, a number of sequential languages with drastically restricted side-effects have been designed. It is commonly held that these nearly side-effect free languages are easier to transform into parallel code than are languages which permit a richer variety of side-effects.

Relative to (2) above, languages which restrict side-effects are completely effective. Relative to (1), these languages are a moderate success.

Compilers must still perform flow and parallelization analysis, but they are simplified in that:

- programmer names correspond directly to unique values and
- in most cases, restricted side-effect languages have such simple structures that transformation into parallel forms can be accomplished entirely by simple pattern matching [Omo84].

Viewing this from a different angle, most of the compilers for restricted side-effect languages implemented thus far completely disregard nearly all ECFs — and the penalty is often great.

It is interesting to note that this is really the connection to the dataflow model. Dataflow machines have very few ECFs; efficiency depends mostly on dynamic properties which are inaccessible to the compiler. Hence, a “dumb” compiler performs nearly as well as a “smart” one for most dataflow machines.

In order for side-effects to be constrained to a level which permits use of a very simple compiler, restricted side-effect languages use call-by-value and every “assignment” statement becomes an explicit copy operation. It is surprisingly difficult to mechanically convert call-by-value into call-by-address (so that the cost of copying values may be eliminated) or to convert explicit copying into pointer references.

The reason is that, at the very least, copy elimination requires detailed global analysis — exactly the problem that restricted side-effect languages attempt to avoid. The “copy problem” is also magnified by the fact that many restricted side-effect languages do not provide for compile-time knowledge of data types.

The utility of the different programming style imposed by restricting side-effects is debatable, but it is fair to say that operations such as input and output (since they are history sensitive — i.e. they are inherently side-effects) are particularly awkward to express. Ironically, although restricted side-effect languages allow the compiler to easily generate programs which are free from race conditions, debugging is difficult — precisely because most debugging practices are based on use of side-effects to monitor a program’s behavior [CuW82].

The major classes of restricted side-effect languages are **single-assignment** languages and **functional** and **applicative** languages.

1.2.3.1. *H:* Single Assignment Languages

In restricting side-effects, the restriction which is almost always made is that *each name must uniquely identify a value*. In other words, only a *single assignment* is allowed to each variable. In this way, the *alias* problem is solved — because the questions which caused the difficulty never need be asked. The only violation of the single assignment rule appears within looping constructs, where the keyword `new` is used to indicate that a conceptually new name is in use.

Examples of single assignment languages are Id [ArG78], Val [McS82], Sisal [McS85] [SkG85], and Blaze [MeV85].

1.2.3.2. *G:* Functional And Applicative Languages

A purely functional or applicative language goes a step further — it does not permit any side-effects. With no side-effects, there are, of course, no variables and therefore no names for data. This certainly insures that there will be no name-resolution problems (*aliases*), but it also demands a dramatically different programming style.

Examples of this type of language are FP [Bac78] and HOPE [Moo82]. There have been several research efforts toward reducing the recomputation incurred by the lack of variables, mostly in the area of applicative caching [KeS81]; however, the single-assignment solution clearly dominates.

1.2.4. *I:* Annotated Side-Effects

The topic of this thesis, the refined-language approach [DiK84], is the automatic parallelization technique based on the concept of sequential code with annotated side-effects. Relative to side-effects, the annotations guarantee that what the compiler gets is what it sees.

The annotation of side-effects is based on the specification of precise **data access permissions** — arbitrary side-effects can be specified, but only provided that they are consistent with the access rights granted to a particular region of sequential code and the access rights carried by names within that code.

1.2.4.1. Access Flow

The new concept of data **access flow**, the heart of the refined-language approach, is very simple and has a strong similarity to dataflow.

In dataflow, data flows into computational nodes where it is absorbed and new data are generated — hence causing the copy problem discussed above. In access flow, permission to access names in certain ways flows through computational nodes — access rights are not absorbed and then regenerated. It is this property which permits refined languages to avoid the copy problem.

In access flow, if access rights to some data structure are divided, eventually these rights must join together at another node, since all rights must be coalesced for the program to return these rights to the operating system (for the program to terminate). Typical access rights are:

(1) Permission to read,

(2) Permission to write (possibly reading what was written),

(3) Permission to read and then write (modify),

(4) Permission to allocate (to create space associated with an existing name), and

(5) Permission to execute.

each node may transmit, merge, or partition, the access rights which flow into it. Access rights cannot be created nor destroyed.

1.2.4.2. Refined Languages

Access flow graphs are not a very convenient representation for a programmer to manipulate — the large number of arcs and high connectivity even makes them awkward to view — conventional languages are far more “user friendly.” Hence, refined language programs are conventional language programs which are annotated with any access flow information that could not have been mechanically obtained, with reasonable effort, from the conventional language program. In some cases, the information would have required very expensive analysis to obtain, hence it is given explicitly to save compile time (as in the case of interprocedural access rights). In other cases, the access rights could not have been determined with reasonable precision no matter how sophisticated the compile-time analysis, hence the programmer's insight into the run-time action of the algorithm is explicitly stated (much as the assertions to BULLDOG, but in terms of partitioning access rights to a data structure rather than in terms of apparently unrelated assertions involving the values of variables).

Definitions of both RC[12] (refined C) [DiK85a] and RF77 (refined ANSI FORTRAN 77) [DiS85] [DiK86] have been presented, and compilers for these languages are under development. Other languages soon to be refined include Pascal, Ada, and (with some restrictions and changes imposed) Lisp: almost any language can be modified to permit specification of more precise **access flow** information.

In refining a language, constructs which obscure the information needed to solve *aliasing* problems (resulting from the use of global data, pointers, references to portions of a data structure, etc.) are removed from the language and replaced by modified versions which do not inhibit the analysis. The replacements can be made to look much like the original constructs and can provide all (or nearly all) of their expressive power. All other language constructs remain as they were.

Since each refined language is very similar to a *dusty-deck* language, compilation into efficient parallel code *can be identical to* compilation of *dusty-deck* programs into efficient parallel code. Any of the compilation techniques of [KuM72] [AlK82] [Ell85] [Vei85] [KuS84] [Nic85] [KAI85] [DiK86] [ScK86], could be used and the result would be at least as good as, and probably much better than, that obtained from the *dusty-deck* language — because the only difference is that it is natural, in writing a refined-language program, for the programmer to supply the compiler with precise information which can be used to *safely* resolve additional aliasing questions.

Of particular concern is a proposed MIMD design which consists of a dynamically-scheduled single-stage network [PaK86] interconnecting processor/memory nodes (requiring large-grain dynamic parallelism) each of which contains some number of specialized RISC processors [DiK85] in a VLIW configuration (requiring very fine-grain static parallelism). In order to support this architecture, compilation techniques capable of dealing with a wide variety of parallel machine ECF levels have had to be developed.

12. The syntax of RC has been slightly altered from that presented in [DiK84] to better conform to the draft specification of ANSI C. A revised definition of RC appears in section 3:3.

The result is that:

- Some progress has been made toward specification of exactly what compiler techniques are associated with various specific ECFs (as discussed in Chapter 6),
- Some new parallelization technology, and terminology, has been developed, incorporating such concepts as non-deterministic selection of target parallel code structure (a heavily-pruned search for the best coding, similar to [KrA82] for single-processor computers, is performed), and
- We have been able to describe concurrency detection and parallelism transformation in terms of standard, uniprocessor optimization-oriented, compiler analysis [CoS70] [WuJ75] [AhU77] [AhS86] [Die84].

Further, in much the same way that BULLDOG provides for iterative improvement of the precision with which *aliasing* questions can be solved, a software tool, REFLEX (currently under development), will permit refined language programs to be improved. The main differences between REFLEX and BULLDOG are:

- The refined language improvements are in terms of obvious, programmer visible, objects — not seemingly unrelated equations, and
- REFLEX will not ask the programmer to be more precise about every ambiguous reference; rather, it will request clarification only for those *aliasing* questions which *would result in large execution-time penalties on the target machine* if they remained unanswered.

The refined-language approach attempts to make every aspect of programming/compiling for parallel machines more efficient — except, perhaps, for the effort the compiler expends in chosing the best parallel execution structure.

2. Efficiency

The efficiency of a programming technique for speedup using parallel computers, or any other class of machine, has many dimensions. In a discussion of programming language design, the most relevant ones are programming, compilation, and execution efficiency.

2.1. Programming Efficiency

One aspect, perhaps the most important one in the long run, is the efficiency with which the author's programming time and effort are converted into a properly functioning program.

Programming efficiency is, therefore, partly dependent on the expressive power of the language. For example, the ease with which APL can be used to represent operations like vector summation makes it more efficient for this task than, say, FORTRAN (which requires an entire loop as opposed to a single compound APL operation). The token-length of the APL program is smaller, hence, the APL program exhibits a higher information density. Fewer ancillary "concepts" need be encoded by the programmer, which implies that the APL version can be created both more easily and faster.

However, good expressive power alone may not be enough. Despite its terseness, it is generally agreed that APL isn't the most programmer-efficient language even for programs which would be far shorter in APL than in any other language. In many cases, APL's expressive power is obtained at the expense of the clarity with which algorithms may be expressed — algorithms often are re-cast into APL's powerful operations even though conceptually they perform operations which are quite different from those required. This is the root of APL's reputation as a "write-only language."

In parallel computing, as in sequential computing, the most powerful language is the language which best matches the task at hand — there is no reason to assume that one language is best for all kinds of programs. If the expressive power of two languages is roughly equal for a given application domain, then the relative ease of understanding and modifying programs in the languages will determine which will lead to greater programming efficiency.

Consider a very powerful language whose constructs are all closely tied to deep understanding of the machine hardware: if the programmer has this knowledge, the language is very efficient; without it, the programmer may find himself reduced to essentially zero productivity while he struggles to learn the pertinent details. If the details change, he becomes unproductive — and existing programs may need to be drastically modified.

A refined language has essentially the same expressive power as its base language. Different refined languages are tailored to different tasks: refined FORTRAN serves a different application domain from that of refined C or of refined Lisp. Common to all of them is the ability to explicitly specify data-access flow, which adds the power to easily express explicitly-parallel algorithms; yet, the parallel structure of the resulting code is determined solely by the compiler. The programmer need not have any understanding of the target-machine (although such understanding can be applied to create algorithms which are more appropriate for the machine, hence faster to execute). In sum, programming efficiency using a refined language is very high and continues to be high even if the hardware changes.

2.2. Compilation Efficiency

Once a program has been written, it is the task of the compiler to generate executable code. The transformation of typical programs into executable code may be very simple or very complex. If it is simple, the compiler will perform its task quickly, making program development somewhat easier. If it is complex, the compiler itself will be more complex and more difficult to create; in addition, the program development cycle will be extended by long compile times. A particular language is compilation inefficient if equivalent programs written using another notation are more easily translated into an equally beneficial executable form.

Refined-language programs are more difficult to compile than programs in many other languages used for programming parallel computers. Relative to other sequential languages, however, refined languages are, by design, very cooperative with compilers which seek precise data-access flow information in support of automatic parallelization. A refined-language compiler is complex only because it is isolating the user from target-machine details, hence enlarging the class of potential users of parallel machines — a very worthwhile trade.

2.3. Execution Efficiency

If the execution time of an executable representation is greater than that obtained using another model, or if the execution time is similar but the resource requirements far greater, then the execution efficiency is relatively low.

In parallel computing, this is not quite as clear a distinction as it at first seems. To see this, consider the following code fragment:

```
IF (A > B * C) THEN D = E * F;
```

- Traditionally, a parallel supercomputer is dedicated to a single program at a time, which implies that any hardware with no code assigned to it is wasted. In the example above, this suggests that it might be worthwhile executing `E * F` in parallel with `A > B * C`. Extra work is being done in the case that `A > B * C` is `FALSE` (namely, `E * F` is needlessly computed). However, if there was otherwise idle hardware able to compute `E * F`, this carries no penalty and, if `A > B * C` is `TRUE`, it speeds-up the program.
- If the target parallel computer is multiprogrammed — which is the preferred environment once parallel hardware becomes relatively inexpensive — hardware which is not being used by one program might be used by another. Continuing the above line of reasoning, execution of `E * F` in parallel

> with `A > B * C` when this condition happens to be `FALSE` is wasting hardware that another program, or even another part of the same program, might have put to profitable use. The desirability of this parallelization depends on the probability of `A > B * C` being `TRUE`, which has long been acknowledged as impossible to predict.[13]

Aside from the discrepancy between uniprogramming and multiprogramming efficiency, if efficient execution is to be obtained, nearly every aspect of a parallel target machine will have major impact on the code generated. Later, in Chapter 6, we relate many basic architectural features to specific parallel code structures and code improvements; for the moment, it is sufficient to point-out that execution efficiency depends on the ability of the compiler to closely match the preferred model for the target machine in question.

A compilation technique may, for example, result in very execution efficient code for a vector machine, but very inefficient code for MIMDs. Such a technique is not execution efficient in general, since it is tied to a target-machine model, and therefore does not constitute a general solution to the problem of programming a parallel computer. (Commercially, such languages may actually be favored because they make it more difficult and less profitable for a customer to switch to another vendor's machine, but this is not an attitude to be encouraged.)

The execution efficiency of code generated from a refined-language program can be extremely high, since the data-access flow information is readily available to the compiler and is very precise. Restating this, execution efficiency of a refined language is a function of how well its compiler models the target machine — the other variables have been eliminated.

The three different kinds of efficiency trade off against each other and, even considering just these few aspects, no methodology excels along all dimensions. The refined-language methodology tends to be implemented with only moderate compilation efficiency because, unlike other approaches, it enables the user to obtain outstandingly efficient execution with just a bit more effort invested in the compiler's model of the target machine.

---

13. It is often suggested that it might be beneficial to allow programmer to specify the probability of taking each conditional branch in the program [CoS70]. In [Fis84], it was proposed that each program be compiled twice. The first time, the compiler would insert code to collect statistics on branching behavior and then the code would be executed with user-supplied test data. The second compile would use the probabilities determined by the statistics accumulated in the test executions. Neither approach has ever met with wide acceptance; guessing that all branches are roughly equiprobable is far more common.

Refined Languages

A **refined language** is a language which permits the programmer to explicitly manipulate data access rights, thereby minimizing ambiguities in analysis for parallelization. To create a refined language, one begins with a conventional HLL — nearly any HLL: Ada, C, FORTRAN, Pascal, PL/I, etc.

Most conventional languages support few, if any, explicitly-parallel constructs. If a language supports constructs which are explicitly parallel *and* have no equivalent sequential semantics, these constructs are removed.[14] The exclusive use of constructs which have sequential orderings insures that race conditions and deadlocks cannot be written; what could otherwise be a race condition is resolved at compile time by the sequential ordering.

Next, the remaining constructs are examined to identify those which obscure the data access flow of a program from a flow-analyzing compiler — in other words, constructs which make compile-time analysis of side-effects ineffectual. These constructs are replaced and/or augmented by constructs with similar syntax which provide essentially the same expressive power, yet do not inhibit the analysis. The “refined” constructs accomplish this by providing the programmer with ways to express the knowledge of data access flow that he must have had in order to have written a particular program; they do not require him to think about his program in a new way and they do not require him to change his programming style.

There are other “fixes” which could be made to a language so that analysis would be more precise, but only the above “refinements” are essential to the success of automatic parallelization. For example, it is extremely useful for a compiler to know approximate branch frequencies [McH86]. Constructs could certainly be invented to permit such specifications, but such knowledge is of little use in detecting parallelism; it would, however, improve the quality of the code in general. Only refinements which clarify the *data-access flow* directly improve the ability of a compiler to generate parallel code and, because we wish to minimize the differences relative to each base language, we do not impose other changes.

In section 3:1, the problematic constructs in FORTRAN are discussed in depth. Section 3:2 presents the definition of RF — refined FORTRAN. Section 3:3 discusses both the problematic constructs in C and the definition of RC. In refining FORTRAN and C, all the techniques needed to refine most conventional

14. Most explicitly-parallel constructs have well-defined sequential semantics. For example, the vector assignment “A = B * C” has the sequential semantics of “for I = LB TO UB DO BEGIN T[I] = B[I] * C[I]; A[I] = T[I] END.” Only parallel constructs like Spawn/Join cannot be represented in equivalent sequential form.

languages are displayed; section 3:4 describes the few additional concerns which arise when either a language is based on inherently sequential data structures (such as Lisp) or a language is characterized by its explicitly-parallel control constructs (such as CSP).

1. Problems In Standard FORTRAN

From the point of view of automatic parallelism detection, it is convenient that typical (unrefined) FORTRAN programs are far simpler and more static than programs written in most other languages. Programs are simpler because FORTRAN is a relatively spartan language — there are not as many different ways to say the same thing as in most other languages. Programs are more static in that more information than usual about the run-time behavior of a program can be determined at compile-time. For example, the amount of data space required by a typical FORTRAN program is known at compile time; in most languages, the ability to perform recursive calls and to dynamically request chunks of memory makes determining the run-time size effectively impossible.

In these respects, FORTRAN, in any of its major dialects, is an ideal language for a compiler to analyze. This fact is evidenced by the multitude of parallelizing compilers for “dusty deck” FORTRAN [AlK82] [Ell85] [KuS84] [Nic85] [KAI85] [ScK86] and the lack of parallelizing compilers for almost any other language.

Since FORTRAN contains no explicit parallelism-invocation or synchronization constructs, it is naturally impossible to write a race condition in the language. Likewise, a flow-analyzing compiler re-structuring code into a parallel form by using only correctness-preserving transformations will be incapable of generating a race condition. If such a compiler is given pure ANSI FORTRAN code, the programmer is guaranteed that the parallelized program will produce the same result as the sequential program — the program will be debuggable. Perhaps even more important, using a compiler with a “back-end” appropriate for each parallel or sequential machine, the program will be completely portable.

Unfortunately, the amount of useful parallelism found by a flow-analyzing compiler examining a typical (unrefined) FORTRAN program is not necessarily all (or even a large fraction of all) that is present in the program [KuM72]. This discrepancy is caused by certain language constructs which obscure (from the compiler’s flow analysis) potential parallelism.

A construct which blurs the compiler’s picture of which data items might be accessed by any particular reference will result in the compiler making a “safe” assumption. For example, in:

```
C      For this and the following examples, assume
C      that labels 10 and 20 are never referenced;
C      they appear only to relate each example to
C      the discussion in the text.
 10    A = B * C
 20    D = E * F
```

the statements labeled 10 and 20 can be executed either sequentially or in parallel, but only in parallel if the answers to the following questions are "no":

(1) Is `B` or `C` an alias for `D`? Is`E` or `F` an alias for `A`?

(2) Are `A` and `D` aliases for each other?

If any part of (1) were answered "yes," executing statements 10 and 20 in parallel could produce an incorrect result — a write/read race condition would exist. If (2) were answered "yes," executing statements 10 and 20 in parallel would produce unreliable results — a write/write race condition would exist.[15] If static analysis cannot answer either (1) or (2), or if either answer is "sometimes yes," the only "safe" assumption is that parallel code should not be generated for statements 10 and 20.

The previous example is somewhat contrived, because it is (usually) trivial for a flow-analyzing compiler to determine the answers to each of the indicated questions — any of the "dusty deck" parallelizing compilers mentioned above could answer them. However, minor variations on this example will demonstrate the analysis problems caused by each of the FORTRAN constructs which must be *refined*.

1.1. References To Global Data (COMMONs)

Suppose that `A`, `B`, and `C` are defined as distinct variables which reside in a `COMMON` and that the statement labeled 20 is changed as follows:

```
 10    A = B * C
 20    CALL SUBR
```

Assume further that `SUBR` is a `SUBROUTINE` which is defined in a separate file. The questions the compiler must answer are essentially the same:

(1) Does `SUBR` (or any subprogram invoked by `SUBR`) contain stores into `B` or `C` or loads of `A`?

15. It is interesting to note that, in general, if an answer is known to be "yes" then a sequential code optimization is possible. For example, if A and D are aliases for each other, statement 10 is a dead computation and can be eliminated.

(2) Does SUBR (or any subprogram invoked by SUBR) contain stores into A?

In order for the compiler to generate code which can safely execute statements 10 and 20 in parallel, the answers to both questions must be known to be "no."

The time complexity of typical flow-analysis techniques used to attempt to answer these questions forces flow-analysis to be localized to small regions of a program (for example, a SUBROUTINE or FUNCTION at a time) — analysis of larger regions would take an unacceptably long time. Since the last example may require this analysis to be performed on the entire program,[16] the compiler would probably be unable to answer these questions. This, in turn, would force the compiler to make the "safe" assumption that every SUBROUTINE or FUNCTION call might affect every variable that appears in any COMMON. Sequential code would result.

A less obvious, but very similar, situation exists relative to the use of I/O channels by subprograms. An I/O channel number acts like a global variable. Consider:

```
10    WRITE (6,*) A
20    CALL SUBR
```

It is impossible to tell if SUBR uses I/O channel 6 without looking, at the very least, at the code for SUBR.

1.2. References Via Pointers (Call-By-Address)

Let us now assume that A, B, and C are all defined as distinct variables local to the SUBROUTINE in which the following code appears:

```
10    A = B * C
20    CALL SUBR(A, B)
```

As before, SUBR is assumed to be defined in a separate file; but, since we have already considered the problem of global data, we will assume that there are no COMMONs in SUBR. AlthoughFORTRAN does not explicitly support *pointers*, it does use call-by-address in passing arguments to FUNCTIONs and SUBROUTINEs. The compiler must now find answers to:

(1) Does SUBR (or any subprogram invoked by SUBR) store into B or load from A?

16. Recently, substantial advances have been made toward limiting the scope of analysis by constructing "programming environments" which incrementally collect the needed information. Good examples of this approach are [Bur84] [BuC86] [CoK86] [TrI86], which were briefly discussed in Chapter 2. However, this mechanism alone is not capable of solving the problems described in section 3:1.3.

(2) Does SUBR (or any subprogram invoked by SUBR) store into A?

which, of course, would normally be very expensive for the compiler to answer.

1.3. References To Indexed Data Structures

Let us return to our original example, again, slightly modified:

```
      IMPLICIT INTEGER A-Z
      DIMENSION G(100)
10    G(A) = G(B) * G(C)
20    G(D) = G(E) * G(F)
```

The questions to be answered are now:

(1) Is G(B) or G(C) the same element as G(D)? IsG(E) or G(F) the same element as G(A)?

(2) Is G(A) the same element as G(D)?

However, these questions are much harder to answer. In fact, answering them may require arbitrarily complex theorem proving or may be impossible, as in the case where a value is READ for one of the variables.

If the value of any of A, B, C, D, E, and F is affected by a parameter entering the SUBROUTINE which contains the above code, then the corresponding argument to *every* CALL of that SUBROUTINE must be examined. This may be theoretically possible, but it certainly is not practical if it must be repeated each time a module is modified (thereby slowing program development and debugging).

Since it can be very difficult to determine which element(s) of a data structure can be accessed by an indexed reference, it is often necessary to assume that such a reference potentially affects any (every) element of the array.

Several attempts have been made to resolve this problem by having the programmer insert assertions, as described in section 2:1.2.2.2.

The RF equivalent to assertions (the concept of *partitioning*, in section 3:2.3) is both efficient and safe, but, most importantly, it makes sense in terms of expressing an algorithm; the only information the programmer needs to express is information which he must have known in order to have understood the algorithm.

2. The FORTRAN Refinements

In each of the situations described above, the compiler's inability to resolve exactly which data items might be accessed by a particular reference must result in the "safe" assumption — that all possibly touched items are not *available* across such a reference — which typically forces the generation of sequential code. If we can enable the programmer to specify what really happens in these cases, fewer precedence constraints need be artificially imposed by the compiler, and less of the generated code will be forced to be sequential.

Each refinement can be viewed as providing the programmer with a language construct which, while being intuitive and natural to the programmer, allows him to provide exactly the extra information the compiler needs.

RF (Refined FORTRAN), looks and "feels" like FORTRAN, but, unlike the latter, can be compiled into *reasonably efficient* race-free code for any kind of machine, parallel or sequential (assuming that a compiler has been constructed for the machine in question).

In the present work, refinements are discussed relative to ANSI FORTRAN 77, since it is very similar to most FORTRAN dialects in popular use, yet its specification is readily available.

Because the ANSI FORTRAN 8x specification includes vector operations, it might seem, at first, that refining ANSI FORTRAN 77 is attempting to "re-invent the wheel." But, although vector notation causes no new problems for flow analysis and concurrency detection, neither does it solve any old ones. Vector operations are helpful in generating good code for some parallel computers, but do not make generation of good code significantly easier for most target machines.

Clearly, vector notation, taken literally, would tend to make processes too small for profitable parallel execution using a MIMD. In fact, the ANSI FORTRAN 8x vector notation is not even capable of encoding many of the most efficient parallel execution forms used by existing vector processors [ScK86]. Chapter 6 details the relationship between various kinds of target machine and "vector" code.

Vector *notation*, as it is supported by ANSI FORTRAN 8x, is little more than a convenience feature for the programmer — much as the data type `COMPLEX` is a convenience. In any case, the refinements made to ANSI FORTRAN 77 are compatible with the vector notation of ANSI FORTRAN 8x; hence, RF easily can be extended into RF 8x.

2.1. Access Permissions To Globals

As was pointed-out above, FORTRAN supports two kinds of global data:

- `COMMON`s, which are used to group sets of variables together by name and to give `FUNCTION`s and `SUBROUTINE`s access to the variables by these group names.
- I/O channel (logical unit, logical file) numbers, each of which is used to give the set of data within a file a name (number) and to give `FUNCTION`s and `SUBROUTINE`s access to the data by that name.

The kind of language construct needed to solve the problems associated with global data references closely resembles a `COMMON`. While`COMMON` statements pro vide the naming features needed, they are scattered throughout a source program and potentially across a large number of files. To make compilation speed acceptable, the information must be available without having to scan the entire source program.

In an RF program, information concerning global data is placed in a separate **interface specification** file,[17] which is included by all files that constitute the source program — much as C programs `#include` “header files.” The interface specification file contains the definitions of global variables and the access permission each `FUNCTION` and `SUBROUTINE` has to each global.

Borrowing the terminology of Ada [Ada80], there are two primitive kinds of access permission relevant in performing concurrency detection and generation of efficient parallel code:

`IN`	Permission for a variable’s *rvalue* to flow into a `FUNCTION` or `SUBROUTINE`; permission to `READ` from a file.
`OUT`	Permission for the *rvalue* of a variable to be different when the variable flows out of the `FUNCTION` or `SUBROUTINE` from what it was at entry; permission to `WRITE` to a file.

Also, as in Ada, these access permissions may be combined:

`IN OUT`	Permission for a variable’s *rvalue* to both flow into a `FUNCTION` or `SUBROUTINE` and to be different when the variable flows out; permission to both `READ` from and `WRITE` to a file.

2.1.1. `COMMON` Permissions

Each individual entry within an RF *interface specification* takes one of the following forms:

`IN /`*global_name*`/` *subprogram_list*
`OUT /`*global_name*`/` *subprogram_list*
`IN OUT /`*global_name*`/` *subprogram_list*

17. PREFINE, discussed in section 2:1.2.2.4, automatically converts a FORTRAN program into its RF equivalent — automatically creating an appropriate interface specification file.

Note that *global_name* may be either the name of a COMMON or it can be blank, representing the unnamed COMMON.

The following is a skeletal example of the use of IN, OUT, and IN OUT:

```
C      this would be the interface specification
C      which appears in the file TEST.H
C      SUBR1 and FUNC2 can both examine any var in A
       IN /A/ SUBR1,FUNC2
C      SUBR2 can change any var in COMMON A
       OUT /A/ SUBR2
C      FUNC1 can examine and change any variable in A
       IN OUT /A/ FUNC1

C      the following appears in the file TEST1.RF
#INCLUDE TEST.H
       FUNCTION FUNC1(. . .)
       COMMON /A/ X,Y,Z
        . . .
       END

C      the following appears in TEST2.RF
#INCLUDE TEST.H
       FUNCTION FUNC2(. . .)
       COMMON /A/ X,Y,Z
        . . .
       END

C      the following appears in TEST3.RF
#INCLUDE TEST.H
       SUBROUTINE SUBR1(. . .)
       COMMON /A/ X,Y,Z
        . . .
       END
```

```
C      the following appears in TEST4.RF
#INCLUDE TEST.H
       SUBROUTINE SUBR2(. . .)
       COMMON /A/ X,Y,Z
        . . .
       END
```

It is important to note that the RF compiler will flag any attempt to reference a global for which permission was not explicitly or implicitly[18] granted: if any subprogram attempts to exceed the access rights granted by the *interface specification*, a fatal compile-time error will result. According to the *interface specification* given in the previous example, the following definition of `SUBR1` is in error because only `IN` rights were granted for members of the `COMMON A`:

```
       SUBROUTINE SUBR1(B)
       COMMON /A/ X,Y,Z
       X = 5.0
       END
```

By the same principle, any attempt to `CALL` a `SUBROUTINE` or `FUNCTION` which has access privileges beyond those of the caller also constitutes a fatal compile-time error:

```
       SUBROUTINE SUBR1(B)
       COMMON /A/ X,Y,Z
       CALL SUBR2
       END
```

2.1.2. I/O Channel Permissions

As we pointed-out at the start of this section, I/O channels are a form of global, named by the `INTEGER` channel number (which is considered to be a *pre-defined* `COMMON` name). For example, the ability to `READ` from I/O channel 7 would be granted to subroutine `SUBR3` by:

```
       IN /7/ SUBR3
```

The parallelization of I/O operations is traditionally one of the most difficult and unreliable language features, as evidenced by the lack of I/O operations in many new parallel-processing languages. In creating RF, this problem is even more difficult because a large part of the flavor of FORTRAN is its style of I/O —

18. Permission is implicitly granted, for example, for the MAIN to access all global data.

drastically changing the I/O would make RF drastically different from FORTRAN. We have chosen to maintain the FORTRAN I/O style, at a slight cost in reliability of parallelization.

To function properly, I/O operations would have to be based on naming *files* — but FORTRAN, and hence RF, I/O is based on naming *channels*. RF*compiler s assume that operations performed using different I/O channels are independent of each other*. Strictly speaking, this is not always true — a single file may be associated with several I/O channels: in the case of a `WRITE`, the compiler might accidentally create a race condition by assuming that operations on two different channels can proceed simultaneously. In fact, this can also cause unpredictable results on some single-processor machines, due to I/O buffering problems.

The problem with FORTRAN I/O is not really a language problem, but one of poor operating system design. Traditionally, although a file system identifies files by name, the only way in which a user program can access a file is by establishing a conceptual data path to the file's contents — but there may be multiple paths to the same object and, because paths are bound to files at run time (and may be re-bound at any time during program execution), all these aliases are unresolvable at compile time. It should be possible, however, to directly reference the file's contents by using the file name. Since file names are unique (by definition), there would be no unresolvable ambiguity in aliasing if files were referenced exclusively by name.

2.2. Argument Passing And Parameter Definition

Each FORTRAN `FUNCTION` or `SUBROUTINE` is able to accept any number of call-by-address arguments. Since call-by-address is used, each argument passed to a subprogram could be carrying `IN`, `OUT`, or `IN OUT`, permissions to the *rvalue*: for the same reasons that a parallelism-detecting compiler must know the access permissions that subprograms have to `COMMON`s and I/O channels, the compiler must know what permissions they have to their parameters. As with global information, the same specification must be available to the compiler during compilation of both the `CALL`er and the `CALL`ed subprogram. The `CALL`ed routine cannot require access privileges not granted by the `CALL`er.

The access privileges granted by the caller should generally match those rights required by the called subprogram of its parameters.[19] By placing this information in the *interface specification*, it need be given only once for each `FUNCTION` or `SUBROUTINE`. RF uses the following syntax to state which access

19. In fact, these permissions often do not match. An individual call might pass more restricted rights than the subprogram normally requires, but that are known to be sufficient for that call. Capitalizing on this was deemed too risky.

rights are carried by each argument of a subprogram:

function_spec ::= `ARGUMENT` *FORTRAN_type arg_spec*

subroutine_spec ::= `ARGUMENT` *arg_spec*

arg_spec ::= *sub_name* (*a_p_list*)

a_p_list ::= *a_p_list* , *perm type_size*

| *perm type_size*

perm ::= `IN` | `OUT` | `IN OUT` | *null_string*

type_size ::= *FORTRAN_type* (*dim_list*)

| *FORTRAN_type*

| *null_string*

where *FORTRAN_type* is any data type supported by standard FORTRAN.

Normally, `ARGUMENT` specifications will state the access permission carried by each argument; however, if the access permission carried by an argument is not specified then the permission is assumed to be `IN`. The optional*type_size* specifications, if gi ven, allow the RF translator to perform type checking on arguments — a feature not related to parallel execution, but desirable for other reasons.

As an example of access permission specification, a subroutine to add the values of two variables and store the result in a third might be written as:

```
C       this would be the interface file TEST2.H
        ARGUMENT ADDSUB(IN, IN, OUT)

C       the following appears in the file TEST2.RF
#INCLUDE TEST2.H
        SUBROUTINE ADDSUB(A, B, C)
        C = A + B
        RETURN
        END
```

Where ADDSUB would be called as, for example:

```
        CALL ADDSUB(5, D, E)
```

A better (but still artificial) example of the use of `ARGUMENT` is the following matrix multiplication code:

```
C      this would be the interface spec., "MATMUL.H"
C      MATMUL and DOTPRO examine COMMON AB
       IN /AB/ MATMUL, DOTPRO
C      DOTPRO can examine,examine,& change its args
       ARGUMENT DOTPRO(IN,IN,OUT)
C      MATMUL can change its arg
       ARGUMENT MATMUL(OUT)

C      the following appears in some other file
#INCLUDE MATMUL.H
       SUBROUTINE MATMUL(C)
       REAL C(100,100)
       COMMON /AB/ A(100,100),B(100,100)
       DO 10 I=1,100
           DO 20 J=1,100
               CALL DOTPRO(I, J, C(I,J))
 20        CONTINUE
 10    CONTINUE
       STOP
       END

C      the following appears in yet another file
#INCLUDE MATMUL.H
       SUBROUTINE DOTPRO(I, J, C)
       COMMON /AB/ A(100,100),B(100,100)
       SUM = 0.0
       DO 10 K=1,100
           SUM = SUM + A(I,K) * B(K,J)
 10    CONTINUE
       C = SUM
       RETURN
       END
```

The constructs presented in this and the previous section enable the RF compiler to determine which subprograms can be executed in parallel with one another *without* requiring expensive global analysis. The constructs presented in the next section permit the programmer to express information beyond that which can be expressed in a “dusty deck” language, hence greatly increasing parallelism.

2.3. FORTRAN Indexing And Partitions

Partitioning is the technique used by refined languages to create new names for arbitrary, mutually exclusive, portions of a data structure. Once these names exist, it is a trivial matter to independently state access permissions for each piece (called a **partition element**). Data may be grouped into partition elements by arbitrary formulas within a RF `PARTITION` statement. These formulas are called **membership test formulas**.

Although the generality of RF partitions is new, the concept is not entirely alien to FORTRAN programmers. For example, a FORTRAN array's membership test formula simply checks that the subscripting values are within bounds. Indeed, FORTRAN's pseudo-array-dimensioning is a mechanism for a restricted kind of partitioning with static indexing: any portion of an array which is stored consecutively in memory can be treated as a separate entity by passing the first element of the sequence to a subprogram which declares that argument as an array (rather than as a single element). This feature of FORTRAN is quite commonly used, since it also makes it possible to write functions in FORTRAN which will operate on data structures whose sizes are not known at compile time (this aspect is discussed further in section 3:3.4.1). However, rather than aiding compiler analysis, pseudo-array-dimensioning hinders analysis because it implies that giving access to what appears to be a single element of a data structure may actually be giving access to any portion of the data structure. In addition, pseudo-array-dimensioning does not aid the compiler in determining whether the access rights passed to various subprograms are *mutually exclusive* — the fundamental motivation of partitioning.

Many other languages also support constructs which are in some way similar to RF partitions. In PL/I and several other languages, there is the concept of a **slice** of a vector or matrix: although the elements of a vector/matrix slice need not be contiguous in memory, they must be addressable by a linear indexing formula and there is no guarantee of mutual exclusion — one datum may appear in several slices simultaneously. Several languages using explicitly parallel control constructs incorporate the concept of **index sets** (for example, [Per79] and [LuB80]), but these also are typically restricted to linear indexing formulas (or set operations thereupon) and do not directly provide for mutual exclusion. Although it is not exceedingly difficult for the compiler to *mechanically prove* that two index sets constructed by set operations on linear addressing formulas are mutually exclusive, the fact that this property isn't obvious to a person examining a source listing is enough to make such notations less desirable than the RF construct.

RF partitioning of a data structure is specified using the (apparently) executable `PARTITION` statement:

partstat ::= `PARTITION (` *structure* `,` *partlist part* `)`

partlist ::= *part* `(` *condition* `) ,` *partlist*

   | *null_string*

where *structure* is the name of the structure being partitioned (including dummy variables naming each subscript) and *part* is a name for a partition element.

The following code illustrates the definition of a `PARTITION` which creates partition-elements naming the portions of the square array A which are, respectively, above, on, and below, the diagonal:

```
      REAL A(100,100)
       . . .
      PARTITION(A(I,J), AUPPER(J .GT. I),
     1    ADIAG(I .EQ. J), ALOWER)
```

The *conditions* within a `PARTITION` statement are evaluated left-to-right *on only those data which have not yet been placed in a partition element*. Therefore, all partition elements are guaranteed to be mutually exclusive: *each datum belongs to only one*. Therefore, arbitrary operations different partition elements can be executed simultaneously without the possibility of races or deadlocks. It is this fact which enables programmers to express the parallelism they envision by partitioning access rights to data structures.

In RF, the shape and indexing structure of the *original data structure* is preserved for each partition element, regardless of the partitioning specification. We call this partitioning with *static indexing*.[20]

For example:

```
C      the following reference zeroes the value
C      of what was originally called A(3,4)
       AUPPER(3,4) = 0.0
```

RF's static indexing means, for example, when a square matrix is partitioned into partition elements above and below the diagonal, each partition element has the same indexing formulas and shape as the

20. The alternative scheme, called *dynamic indexing*, is discussed in section 3:3.3 (RC partitions). For RF, the fact that dynamic indexing could generate non-rectangular data structures makes it inappropriate — RF programmers will not want to think in terms of such data structures because conventional FORTRAN doesn't provide any way of building them.

original matrix, but some of the data of the original matrix are inaccessible through indexing each partition-element name. A reference to a datum by indexing through the partition element `AUPPER`, where the datum is conceptually contained in the partition element `ALOWER`, is an error. A combination of compile time and the run time checks can detect and report all such errors.

Consider:

```
C       the following reference is valid
        AUPPER(5,7) = 3.14159265
C       the following reference is not valid
C       and would cause a fatal compile-time error
        ALOWER(5,7) = 3.14159265
```

to support these checks, each partition element must have a *membership test* associated with it.

The user may explicitly test whether a particular datum is a member of a partition-element by using the unary prefix operator `.MEMBER.`, which simply applies the membership test formula for the following subscripted reference and returns a logical value of `.TRUE.` if that reference is valid. Thus:

```
C       the condition below is obviously .TRUE.
        IF (.MEMBER. AUPPER(1,99)) GOTO 30
C       this might or might not be .TRUE.
        IF (.MEMBER. AUPPER(4*K-1,L)) GOTO 30
C       the condition below is obviously .FALSE.
        IF (.MEMBER. AUPPER(I,I)) GOTO 30
```

There are at least two ways in which partitioning using static indexing can be efficiently implemented. The*member ship test* associated with each partition-element can be represented by:

(1) a thunk of code which, given a tuple of subscript values, evaluates whether the element indexed by that subscript tuple is logically present in the partition element, or

(2) a boolean structure which has the same shape and index range(s) as the partitioned structure, where each element of the boolean structure is `.TRUE.` iff the corresponding element of the partitioned structure is a member of this partition element (and is `.FALSE.` otherwise).

Neither of these implementation techniques requires the members of a partition element to be contiguous in memory or subscript-tuple address space, nor is it required that each partition element consist of members selected from the original structure by a linear formula.

To demonstrate these implementation techniques, suppose that `ISPRIME` is a function which returns `.TRUE.` iff its argument value is a prime number and that the following `PARTITION` statement is given:

```
REAL A(6)
 . . .
PARTITION(A(I), PRIME(ISPRIME(I)), NOTPRIME)
```

2.3.1. Thunk-Based Implementation

Using the thunk-based implementation of the membership function, this becomes:

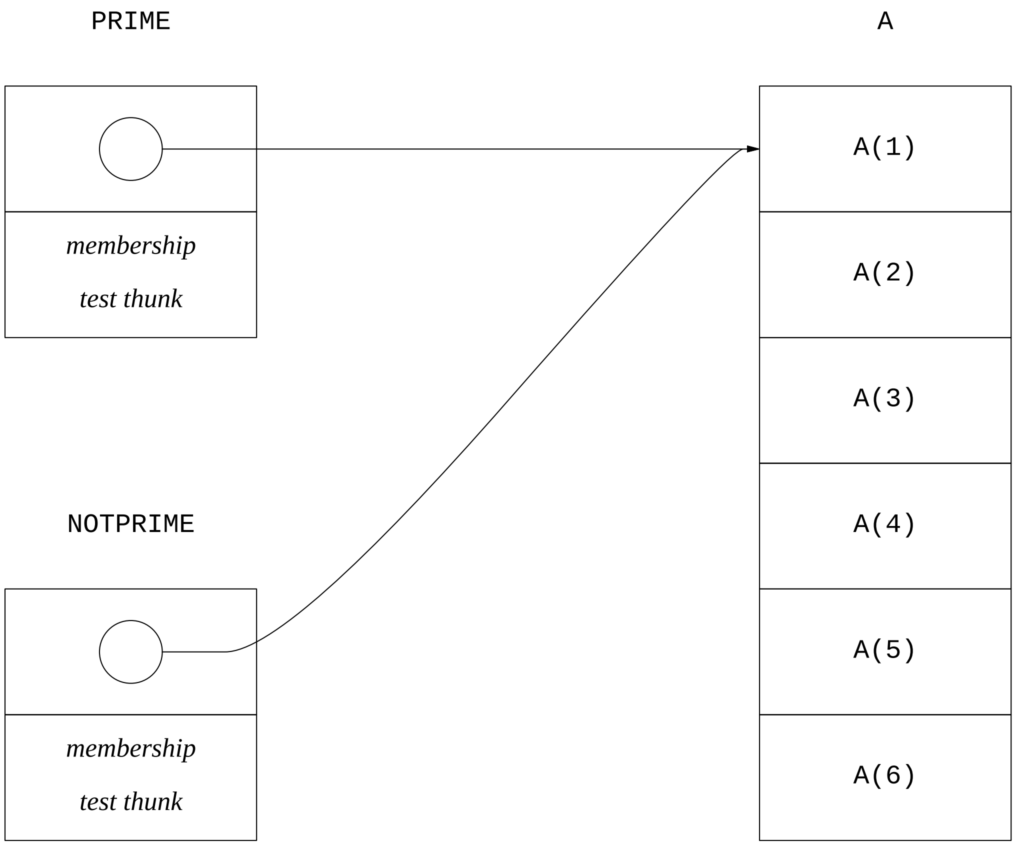


Figure 3:1: Non-Linear Static Partitioning Using Thunks

where the *membership test thunk*s are respectively:

```
((subscript .GE. 1) .AND. (subscript .LE. 6)) .AND.
ISPRIME(subscript)
```

and:

```
((subscript .GE. 1) .AND. (subscript .LE. 6)) .AND.
(.NOT. ISPRIME(subscript))
```

The original array, A, actually had the membership test:

```
(subscript .GE. 1) .AND. (subscript .LE. 6)
```

associated with it, and the partitioning conditions are simply additional constraints on this original membership test. Likewise, for example, if PRIME were further partitioned into the partition elements B and C such that:

```
        PARTITION(PRIME(I), B(I .LT. 4), C)
```

the *membership test thunk*s for B and C would be:

```
((subscript .GE. 1) .AND. (subscript .LE. 3)) .AND.
ISPRIME(subscript)
```

and:

```
((subscript .GE. 4) .AND. (subscript .LE. 6)) .AND.
ISPRIME(subscript)
```

Notice that the membership test did not become more complex in this sub-partitioning because, if we assume that the compiler could determine the exact sequence of subpartitionings in the program, the compiler can eliminate common subexpressions and redundancies within the tests. If this is not done, the test expressions for sub-partitioned partition-elements form chains of thunks.

For obvious reasons, this implementation scheme makes partitioning a zero-cost operation, but makes membership testing moderately expensive. It is important to recognize, however, that the membership tests would typically be executed within parallelized code.

2.3.2. Boolean Structure-Based Implementation

As mentioned earlier, an alternative implementation of static partitioning membership formulas employs a boolean structure of the same shape and index range(s) as the original structure. The partitioning of `A` into `PRIME` and `NOTPRIME`, as given in section 3:2.3.1, can also be implemented by:

PRIME	NOTPRIME	A
.TRUE.	.FALSE.	A(1)
.TRUE.	.FALSE.	A(2)
.TRUE.	.FALSE.	A(3)
.FALSE.	.TRUE.	A(4)
.TRUE.	.FALSE.	A(5)
.FALSE.	.TRUE.	A(6)

Figure 3:2: Non-Linear Static Partitioning Using Booleans

with very similar properties. For example, sub-partitioning the partition-element `PRIME` simply causes the boolean structure representing the members of `PRIME` to be `.AND.`ed with the additional membership constraints.

Using this implementation scheme, partitioning is a relatively costly operation, but membership testing is very inexpensive. However, the construction of boolean structures is also typically able to be parallelized. Further, the close resemblance between these structures and SIMD[21] processing-element enable bit

21. SIMD architecture is discussed in section 6:2.

vectors makes this a particularly good implementation strategy for this class of computer.

2.4. Conclusions

In sections 3:1 and 3:2, we have given a detailed presentation of the application of the language-refinement methodology to FORTRAN and a definition of the resulting language, RF.

Throughout the modifications of section 3:2, the FORTRAN flavor of the language has been maintained and no particular view of parallel processing has been imposed. However, the language no longer prevents the programmer from writing programs so that they can be understood and parallelized by the compiler — using the techniques outlined in Part 2 — into efficient code for nearly any kind of parallel computer.[22] Further, since the refinements aid *any* compiler in building a more accurate flow-graph, RF is completely compatible with, and efficiently usable by, compilers for single-processor machines. In fact, by applying conventional optimization techniques to the better quality flow graph, RF may actually be a more efficient language for SISD machines than ANSI FORTRAN.

3. A Second Case Study: Refined C

Refined C (RC) is a sequential programming language based on the syntax and semantics of the proposed ANSI C standard, but incorporating constructs which provide a compiler with detailed information about *data-access flow*. This additional information is used by the compiler to greatly improve its ability to detect parallelism and, incidentally, to enhance the compiler's general ability to optimize code. Since RC differs only slightly from C, a programmer who is comfortable using C can easily write RC code. RC therefore allows a programmer to write code in a familiar sequential syntax and style, yet it insures that a compiler will recognize the parallelism of each algorithm and will produce good machine-dependent parallel code.

This section presents both a brief review of the problems encountered when flow analysis is applied to unmodified C code (toward concurrency detection) and the specification of the RC programming language. Since section 3:1 discusses the difficulties encountered in analysis of unmodified FORTRAN code, only those problems which occur in C and not in FORTRAN are discussed here.

Every aspect of FORTRAN which causes concurrency detection to falter also occurs, in somewhat modified form, in C. Relative to a FORTRAN program, however, a C program characteristically demonstrates far more complex run-time behavior. As might be expected, the substantially different natures of C and FORTRAN are reflected in significantly different-looking refinements being made to C as compared to FORTRAN. However, the principles are the same. Section 3:3.1 discusses the problem arising from the generality of C control structures. Sections 3:3.2, 3:3.3, and 3:3.4 discuss the RC constructs called **func-**

22. Of course, to obtain peak efficiency the algorithm may have to be changed more drastically than current compilers are able. This is like the argument that everyone should program conventional computers directly in assembly code; it is hard to argue that it wouldn't be more efficient to execute than code written in an HLL, but it's even harder to imagine that the difference is worth the effort.

tion prototypes, **partitions**, and **paramtypes**[23].

3.1. Control Constructs In C

C control structures are richer than those provided by many other languages. One example is the `for` loop construct. A C `for` loop is equivalent to a `while` loop, and any C `while` loop can be expressed in terms of the C macro:

```
#define while(e) for(;(e);)
```

Whereas the keyword `DO` in a FORTRAN program tells the compiler where to find the loop index variable and insures that this variable isn't modified within the loop except to follow the progression specified in the `DO` statement,[24] observing the keyword `for` within a C program provides none of this information. The fact that the keyword `for` appears in the following example is of little help in determining that the loop is executed `for i = 0 to 19` with an increment of `1`:

```
i = 0;
j = 5;
for (k = 20; k > i; ++j) {
        a[i] = b[j] * c[k];
        ++i;
}
```

By providing the programmer with more flexible control constructs, C forces the compiler to use flow analysis to understand them.

C's control flexibility makes programming easier [Ker81], yet the recognition of constructs based on graph analysis makes more parallelism visible to the compiler[25] and incurs only minor additional effort in analysis.

The cost of the recognition process is typically small because, in any event, the graph must be constructed and scanned in order to transform the structure for parallel execution. The benefits are substantial because, using graph analysis instead of keyword matching to understand (parallelizable) loops, loops are recognized even if they were constructed using `goto` statements.

23. The first complete RC compiler implements only two of these three constructs: paramtypes have been *temporarily* omitted. Paramtypes are not needed for the purpose of concurrency detection, but serve primarily to minimize the loss of expressive power associated with the imposition of strict typing on C.

24. Explicit assignment of a value to a loop variable is illegal in FORTRAN.

25. Even with FORTRAN, parallelizing compilers which use graph analysis to identify loops consistently outperform those which use keyword matching [Con85].

Although current programming styles tend to produce code whose structure can be recognized by matching keywords, many important existing programs, and most programs automatically generated by software tools such as YACC [Joh75], depend on use of `goto`-like constructs. This is an additional strong incentive for maintaining the C control constructs.

In summary, control flexibility is a feature, not a bug. RC should, and does, preserve it. The only complication this introduces is that an RC compiler *must* perform flow analysis — keyword-based parallelization techniques would find almost no parallelism in typical RC code.

3.2. Function Prototypes

Function prototypes are the mechanism used to specify data-access flow among the various functions which constitute a RC program.

In C, flow of access permissions to variables between modules can only occur by means of function argument passing (using "pointer" arguments) or references to globals[26]. A function may have any of the following four kinds of access permission to a variable:

`<`	Read-only access permission.
`>`	Write-then-read permission. Permission to write and to read, but to read only after having written. In other words, the function may store a new value in the variable, and it may read the value it has stored, but it may not read the value which the variable had upon entry to the function. This is useful because two regions of code which refer to the same variable, one creating and operating on an old value and the other creating the new value, can be made to run in parallel by allocating an appropriate chunk of uninitialized memory to the process which creates the new value — even though it is writing, it need not depend on completion of the previous write.
`^`	Modify permission. Permission to read and/or write, with these operations occurring in any order.
`-`	No permission.[27]

26. Currently, RC does not permit **upward-exposed** static variables. In other words, static local variables are not permitted to carry values across invocations of the function which contains their definition. This is easily checked at the time the function is compiled: if there is a flow path from the start of the function where it is possible that a static local variable would be read before it is written, the variable is used illegally. It is also illegal to return a pointer pointing to a variable allocated locally, even if it is declared as static.

27. Either the variable's value may be neither read nor written or, in the case of non-pointer arguments, no permission applies.

3.2.1. Permissions To Arguments

Unlike FORTRAN, which uses call-by-address, C uses call-by-value. This means that argument values are copied and only the copies are accessible to the called function. (Only the called function can reference copied arguments, hence there is no aliasing problem regardless of the argument accesses made by the called function.) The only exception occurs when an argument of a pointer type is given, in which case call-by-value operates exactly like call-by-address.

When a pointer-typed expression is an argument to a function, the *object of the pointer* is the item for which permissions must be stated. A simple example of this is:

```
        /* assume there are no globals */
        auto int i, j, k, *p;
         . . .
        /* assume p doesn’t point at i, j, or k;
           likewise, l10 & l20 are not referenced.
        */
l10:    i = j * k;
l20:    subr(j, p, &k);
         . . .
```

Even though `i`, `j`, `k`, and `p` are local variables and C always uses call-by-value. For a compiler to determine if it would be safe to execute statement l10 in parallel with statement l20, it must know what access permissions are passed to `subr()` via:

- the pointer `p` and
- the pointer whose value is the address of `k`.

It is not necessary for the compiler to know how `subr()` will read or write its first argument, since that argument is simply a copied value and, therefore, it is effectively a new variable local to `subr()`.

Suppose the definition of `subr()` appears in a separate file and is (in pre-ANSI “old” C):

```
void
subr(a, b, c)
int a, *b, *c;
{
l30:    *b = *c;  /* write *b, read *c */
l40:    c = &a;
l50:    *b += *c; /* read *b, write *b */
}
```

Since the first parameter is not pointer-typed, it needs no access specification in the function prototype. The second parameter is pointer-typed and its object is unconditionally written in line l30. This value is then read in line l50 and a new value is written. Since a write always precedes the first read, the second parameter must carry at least write-then-read permission to its object. In line l30, the value pointed-to by the third parameter is read; this is the only use of the third parameter's object. The reference to `c` which appears in line l40, and which admittedly exhibits bad programming style, unconditionally makes the object referenced by the original pointer inaccessible — it doesn't constitute a reference to the external object and neither does l50.

If the definition of `subr()` appeared in a separately-compiled file in ANSI C, none of this vital information could be determined without expensive analysis. An RC function prototype to specify the required permissions would be as follows:

```
void
subr(- int a, > int *b, < int *c);
```

If the full generality of C pointers is to be efficiently dealt with, then a language construct for specifying all possible aliases of indirection on a pointer may also be added to the language. Such a construct is implemented in [Ste86], although RC does not strictly require it, since RC partitions can simulate C pointer operations.

3.2.2. Permissions To Global Data

As in RF, it is necessary that the compiler be able not only to determine the permissions passed using call-by-address, but also the side-effects that each function has on global variables. In RF, there are two separate mechanisms used to specify this flow: the subprogram `ARGUMENT` declaration and the `IN`, `OUT`, and `IN OUT` declarations for global data. The original proposal for refined C [DiK84] also incorporated different mechanisms for these separate purposes. However, the ANSI X3J11 committee is almost certain to adopt a function prototype declaration syntax for ANSI C — for the sake of compatibility, the new definition of RC expands this syntax to permit specification of all inter-procedural flow information.

In terms of the refined-language methodology, it does not matter which syntax is used. Both provide means for specifying the same information and both make that inter-procedural information available to the compiler so that analysis of each module can be performed separately from analysis of all others. The choice is largely a matter of personal preference, although the ease of updating the specification to reflect changes in a program and the ease with which the compiler can confirm the accuracy of the specifications are also important concerns. The old style was slightly harder to update; the new style is slightly harder for the compiler to check. We would not have changed style were it not for ANSI X3J11.

Section 3:3.2.1 demonstrated the syntax for specifying argument data-access permissions; the access permissions to each global variable referenced within the function are specified at the beginning of each function prototype:

```
int x, y, z = 20;
{ < x; } int f0(< char *p);
{ ˆ x, y; > z; } void f1(ˆ int *p);
```

Specifies that:

- x, y, and z are global integer variables (z’s initial value is set to 20),
- f0 is a function which can read the global x, returns a value of type int, and takes one argument named p which is of type read-only pointer to char, and
- f1 is a function which can modify the globals x and y and can write-then-read the global z, it has no return value (returns type void), and has modify permission to the object of its argument int pointer named p.

A refined C compiler can detect, and report as an error, any attempt for a function to exceed its data-access rights. For example, a function cannot call another function unless the caller has at least the same access-rights to all globals that the callee has: f1 could call f0, but f0 cannot call f1.

The syntax for a function prototype is:[28]

28. The BNF-like notation presented is taken from the documentation for the first complete refined C compiler. In it, { x } means 0 or more occurrences of x, where x is an arbitrary string of terminals and non-terminals, and [x] means 0 or 1 occurrences of x.

function-prototype ::=

  [*permlst*] *type declarator* ’`(`’ *arglst* ’`)`’ ’`;`’

permlst ::= ’`{`’ *perm* {’`;`’ *perm*} ’`;`’ ’`}`’

perm ::= *access id_lst*

id_lst ::= *id_name* {’`,`’ *id_name*}

arglst ::= [*arg* {’`,`’ *arg*}]

arg ::= [*paccess*] [*type*] [*declarator*]

paccess ::= *access*

  | ’`-`’

access ::= <

  | ’>’

  | ’`^`’

RC provides type checking of actual and formal parameters (if the type of an argument is omitted in the function prototype, it is assumed to be `int`). If the*declar ator* is omitted from an *arg*, then the access specification is applied to the corresponding parameter in the function definition. Any information specified in a function prototype need not be repeated in the function definition.

3.3. Partitioning With Dynamic Indexing

Dynamic partitioning of an array by an arbitrary formula specifying which partition-element each datum belongs to can produce partition-elements which *are not the same size or shape as the original*. In the case of a multidimensional array, a partition element need not even be rectangular. For example, each “row” might be of a different length. Partitioning with *dynamic indexing* permits the shape and indexing structure may change dynamically according to the partitioning specification.

For RC, both partitioning with *static* and *dynamic indexing* are reasonable; however, in the interest of simplicity, the current definition of RC supports only *dynamic indexing*. There are three new built-in functions in RC which support dynamic partitioning: `part()`, `fixpart()`, and `count()`.

The following RC code defines a 101-element array, `grades[]`, and then partitions it into two parts, `odd[]` and `even[]`:

```
float   grades[101], *odd, *even;
part(grades[n], odd,(n & 1), even);
```

such that `odd[]` consists of the odd-numbered elements of `grades[]` and `even[]` consists of the rest. Note that the selected elements were not originally contiguous — `odd[0]` is `grades[1]` and `odd[1]` is `grades[3]`. The count of items within`odd[]` is `count(odd)`, 50 in this e xample.

The partition-element names remain bound until the next reference to the original array appears, at which time the entire array is conceptually restored and the partition-element names become unbound. If `part` was used, the array elements are restored to their original sequence; if `fixpart` was used, the array's elements are re-arranged into the order in which they appeared in the partition-elements. Had the above example been done using `fixpart` instead of `part`, the `odd[]` and `even[]` would have been the same, but `grades[]` elements would have been inverse-shuffled afterward.

In general, a `part` statement can partition data into an arbitrary number of *partition-element*s. For example:

```
float   grades[101], *p1, *p2, *p3;
part(grades[n], p1,(n<25), p2,(n<75), p3);
```

causes `p1[]` to contain `grades[0..24]`, `p2[]` `grades[25..74]`, and `p3[]` `grades[75..100]`.

The syntax for the partitioning constructs is:

statement ::=
    `part` ’`(`’ *expression* ’`[`’ *identifier* ’`]`’
        ’`,`’ *partels* ’`,`’ *identifier* ’`)`’ ’`;`’

| `fixpart` ’`(`’ *expression* ’`[`’ *identifier* ’`]`’
        ’`,`’ *partels* ’`,`’ *identifier* ’`)`’ ’`;`’

partels ::= *partspec* {’`,`’ *partspec*}

parspec ::= *identifier* ’`,`’ *expression*

Partitioning a non-homogeneous `struct` or `union` instance by member name does not require a `part` statement:

```
struct example {
        int an_int;
        float a_float;
} a[100];
```

`a.an_int` is a *partition-element* holding `count(a.an_int)`, or 100, `int`s; `a.a_float` is a *partition-element* holding `count(a.a_float)`, or 100, `float`s. This differs in that in standard C, `a.an_int` would be the same as `a[0].an_int` and `a.a_float` would be the same as `a[0].a_float`.

The generality of the semantics of partitioning with dynamic indexing, as described above, leads one to believe that there is no efficient implementation. However, there are several viable alternatives:

(1) a partition-element can be instantiated by copying the appropriate set of members from the original structure,

(2) a partition-element can be represented using pointers into a *map* array, thereby re-arranging/grouping members of the structure without copying them, or

(3) a partition-element can be represented using a dope vector which gives a new indexing formula for the partition-element’s members, but this works *iff*:

    (a) the partitioning formulas result in partition-elements whose members are placed at addresses within the original structure which can be selected by a linear memory addressing formula and

    (b) `fixpart` is not supported.

Of these, (1) was used in the first RC compiler, (2) is the preferred implementation for most MIMD computers, and (3) is a limited, but very efficient, implementation that was first proposed as an extension of vector notation.

3.3.1. Copy-Based Implementation

There is really very little to be said about the operation of the copy-based implementation of dynamic partitioning; each partition-element is allocated enough space to hold the entire original structure[29], and the members of the original structure which satisfy the partitioning formula are copied into this space.

At the time of re-combination of access rights, the members of the partition-elements are copied back into the original structure.

For example, given a function `isprime` which returns a *true* (non-zero) value if its argument is prime, the following partitioning:

```
float a[6], *prime, *notprime;
int i;
 . . .
part(a[i], prime,isprime(i), notprime);
```

results in copy-implemented partition-elements of:

29. Some functional/single-assignment languages, such as Sisal, also support this kind of operation; however, the lack of structure size information in such languages makes it very difficult to determine how large an allocated space is big enough.

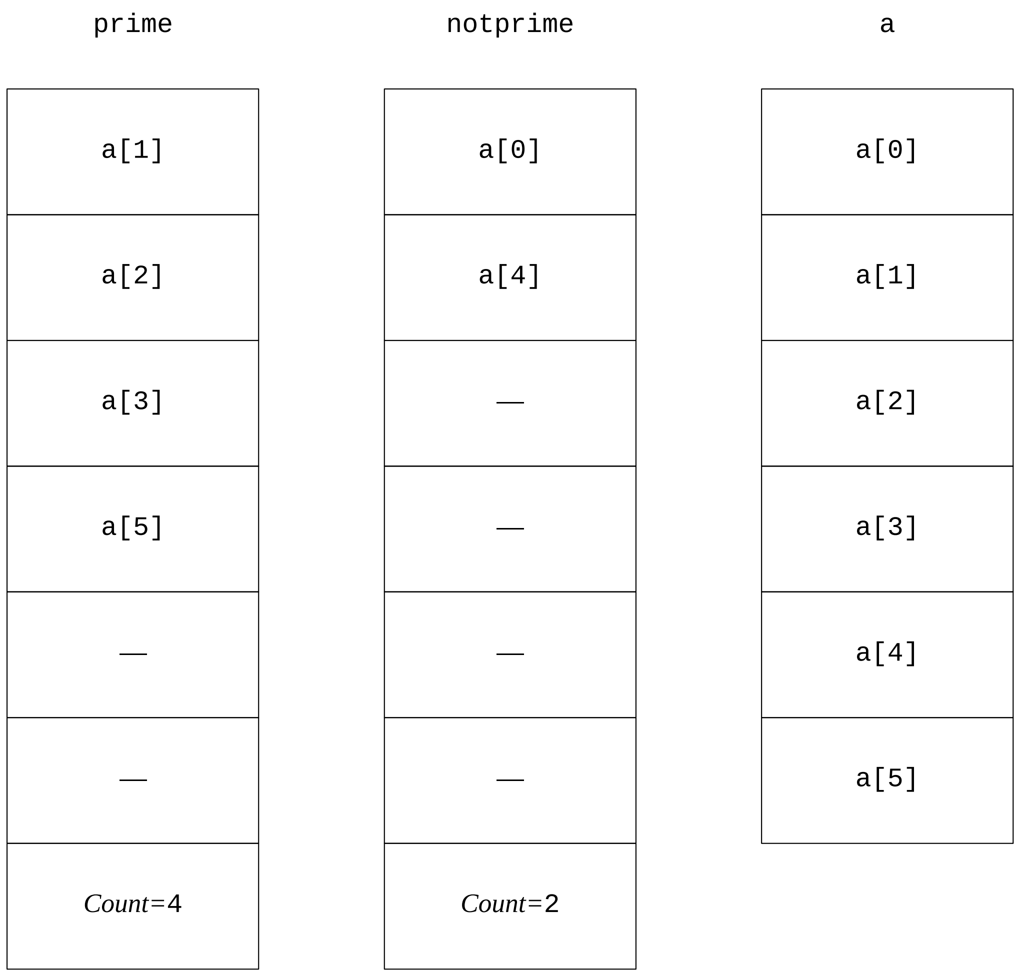


Figure 3:3: Non-Linear Dynamic Partitioning By Copying

3.3.2. Map-Based Implementation

The map-based implementation of dynamic partitioning is, in many respects, very similar to the copy-based implementation. Each partitionable structure has, associated with it, an array of pointers to its elements called the **map**. Normally, a map is allocated at compile-time for each partitionable structure. The act of partitioning is simply a re-ordering (copying) of pointers in the map to form contiguous pointer arrays for each *partition-element*.

However, unlike the copy-based scheme, this often requires only O(*constant*) effort, because the members of the *partition-element*s are either naturally contiguous or they are skewed in a way which can be generated at compile-time. The example used to demonstrate the copy-based implementation appears as follows when the map-based scheme is used:

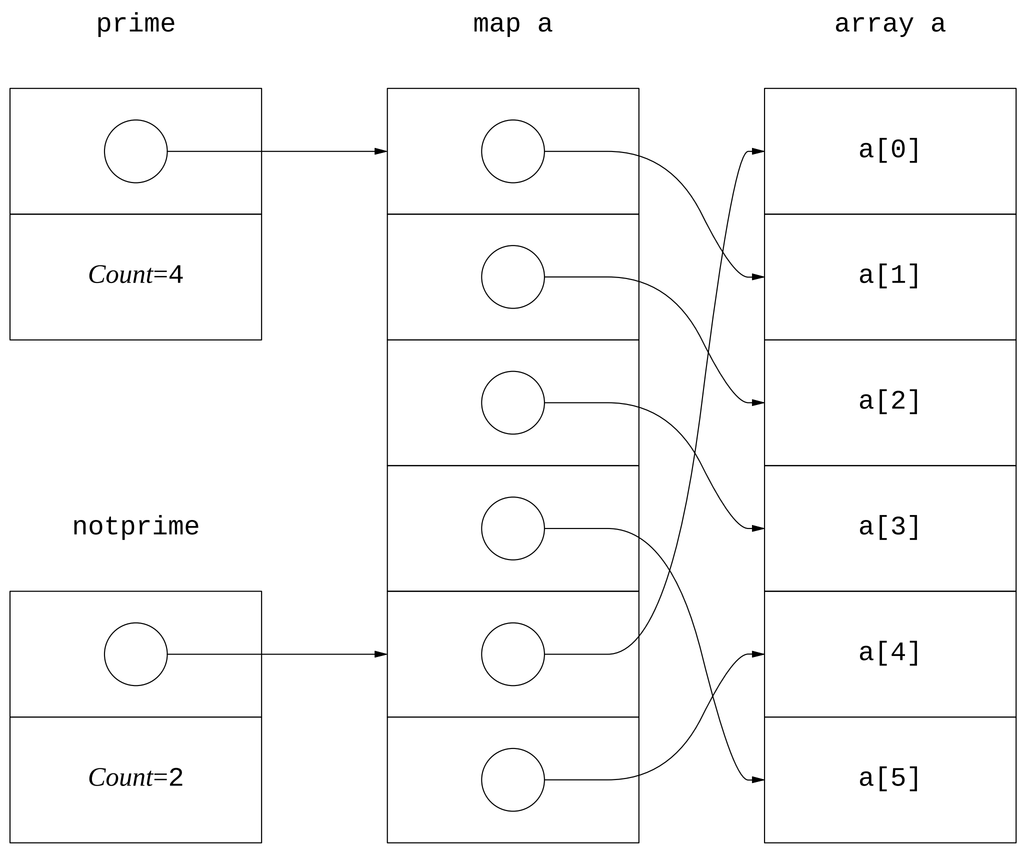


Figure 3:4: Non-Linear Dynamic Partitioning By Mapping

If a partitioned data structure is further partitioned, the map-based technique presented above will always result in a double indirection to fetch data, *no matter how many levels of partitioning intervene*. This overhead is relatively small — about the same as the overhead involved in array subscript checking in a language like Pascal. It is true that the potential savings in partitioning overhead is balanced by the cost of an additional indirection (through an element of the map array) on every reference, however, there are other benefits in using the map implementation.

Using the map technique, it is easy to implement a simple extension of the concept of a partition: the idea of an **infinite partition**. An infinite partition is a partition which can increase its size dynamically, yet always appear to be linearly addressed. This is essentially the mechanism needed to support parallelized file I/O and dynamic memory allocation[30].

An infinite partition is the same as a partition-element, except in that the members of an infinite partition may, or may not, be present. If a member of an infinite partition of type `file` is to be written, and that member is not present (has a `nil` map entry), then a new disk block is allocated and entered in the map. A store into a `nil` member of an infinite partition of any other data type invokes the storage allocator to create an item of the appropriate type and places its address in the map for that member.

For example, if functions `f` and `g` are to simultaneously create substructures of a complex data structure, each allocating its own memory dynamically and returning the allocated structures, how can this be expressed without violating the refined-language methodology rule that no called routine has access privileges superior to those of its caller? Assuming that `@` instead of `*` distinguishes an infinite partition, this can be done by:

```
float @mempool, *a, *b;
int i;
 . . .
/* allow space for up to 100 elements in a,
   and the rest for b.
*/
part(mempool[i], a,(i<100), b);
f(&a);
g(&b);
```

in which no space is allocated for `mempool` until `f` and `g` attempt to store in its members (via the members of the partition-elements `a` and `b`). The partition-element`a` can hold up to 100 members; `b` can hold up to the maximum allocation — 100 members. This does not violate the access flow principles because, although the data was not created in the code calling `f` and `g`, the access rights were, and these rights were passed down to `f` and `g`, and finally returned by them.

30. Despite this, RC does not directly support infinite partitions. This decision was made to maintain the style of C programming in RC; in C, file I/O and dynamic memory allocation have never been built-in features.

The usual map-based double indirection enables the non-consecutive blocks of an infinite partition to appear as a linear array — and even to be partitioned again. In fact, since the map itself is referenced through a pointer, the maximum number of members can be changed simply by allocating a new map and changing the pointer.

Since the implementation of RC partitions in [Ste86] is copy-based, rather than map-based, infinite partitioning is not currently supported.

3.3.3. Dope Vector-Based Implementation

Unlike the copy-based and map-based implementations of dynamic partitioning, the dope vector-based implementation does not involve any changes to a data structure; rather, it involves creation of new linear indexing formulas which refer directly to the original data structure.

This concept is essentially the same idea used to implement index sets in [Per79] and [LuB80]. As discussed in section 3:2.3, it is possible for the compiler to mechanically determine whether linear indexing formulas overlap; hence, the compiler would perform this analysis and generate an error if the linear indexing formulas overlap (since the dope vector-based implementation cannot represent the non-linear indexing formula which would be required).

For example, partitioning into `odd` and `even` partition elements given by:

```
float a[6], *odd, *even;
int i;
 . . .
part(a[i], odd,((i % 2)==1), even);
```

would produce the dope vector-based implementation:

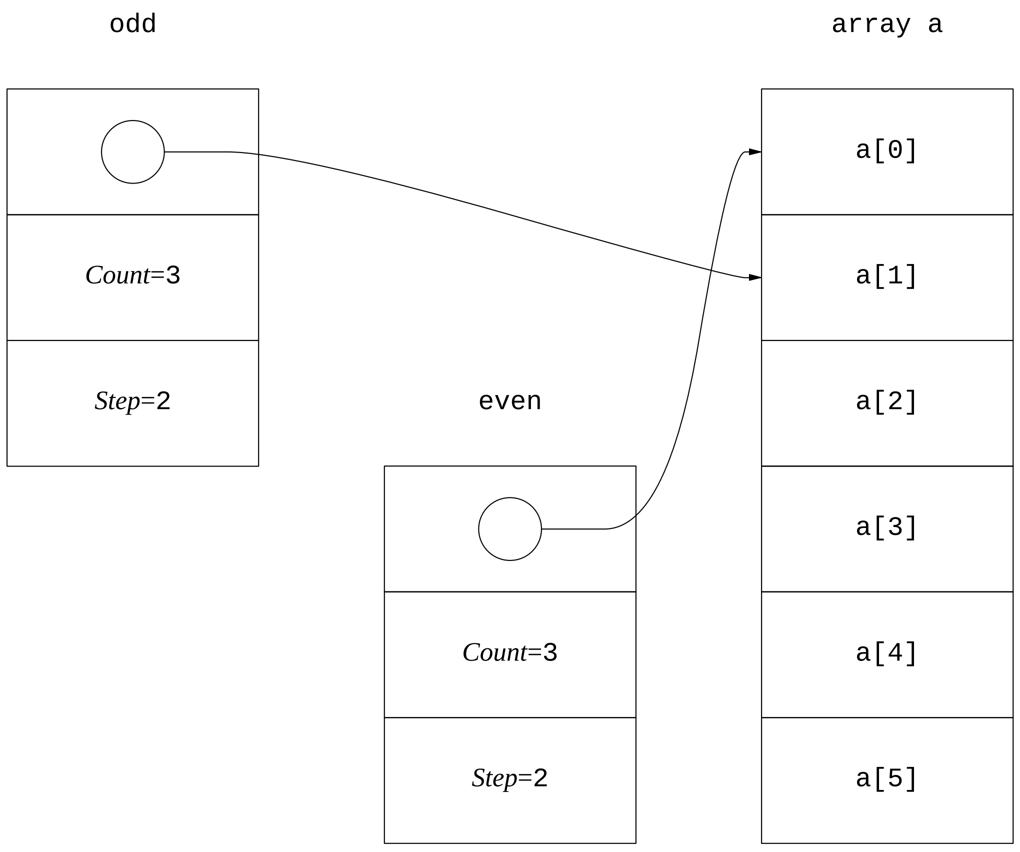


Figure 3:5: Discontiguous Linear Partitioning By Dope Vector

To reference the *Nth* member of `odd` in the example above, one would simply multiply *N* by the step size for `odd`, which is 2 (really 2 * `sizeof(a[i])`), and add that to the base pointer of `odd`, which is the address of `a[1]`. Hence, the reference looks like a conventional array reference, except in that the step between elements does not necessarily equal the size of an element and the base address of the array may have shifted.

Other commonly used linear partitioning formulas would typically partition an array into *N* contiguous parts, as in:

```
float a[6], *a1, *a2, *a3;
int i;
 . . .
part(a[i], a1,(i < t1), a2,(i < t2), a3)
```

which partitions `a` into three partition elements whose members are separated by a step size equal to that of `a`, but whose bases differ (`a1` is based at `&a[0]`, `a2` is based at `&a[t1]`, and `a3` is based at `&a[t2]`); or into *N* non-contiguous parts, as in:

```
float a[6], *a1, *a2, *a3;
int i;
 . . .
part(a[i], a1,((i % 3)==1), a2,((i % 3)==2), a3)
```

which partitions `a` into three partition elements whose members have step sizes equal to three times the step size of `a`, as well as having different base addresses (`a1` is based at `&a[1]`, `a2` is based at `&a[2]`, and `a3` is based at `&a[0]`).

Notice that partitioning a partition element is not a problem — the exact same rules apply.

3.3.4. Dynamic Partitioning Summary

In the worst case, using any of the above implementations, partitioning involves O(*N*) effort[31] — but compile-time optimizations can remove much of the overhead and, in many cases, the operations can be done fully in parallel, so the time taken is often between O(1) and O(log *N*), depending on details of the architecture.

Even if the time taken is O(*N*), this effort does not constitute useless work done merely to support a notation, rather, it was invested in simplifying the indexing computations that would have to be done no matter how the program is written. This is because *partitioning by a "messy" formula would not be specified by a programmer unless he wanted to be able to access most of the data according to that formula*. If a programmer needs only to reference the 14th odd-indexed element of an array `a[]` with 1000 elements, he can refer to it using the usual indexing (2 * 14 - 1, simply `a[27]`) rather than by partitioning and indexing the partition-element.

31. This excludes the cost of evaluating the partitioning conditional expressions, since this cost is not, in general, related to *N*.

3.4. Paramtypes

Paramtype is an extension of the concept of *type* designed to compensate for the expressive power lost by imposing strict typing. Much of the power of C derives from the fact that, while it supports a sophisticated type system, it also provides several ways to override types [Ker81]. Compared to most other languages, C is weakly typed.

Unfortunately, relatively strong typing must be imposed if side-effects are to be understood by the compiler — and that is the primary concern in automatic parallelization.

3.4.1. Strong Typing

An excellent, and well-known, example of the expressive power deriving from weak typing is often given as an argument against Pascal's strong typing.

The declaration of a run-time variable-size array (most commonly, an array of characters) is impossible in standard Pascal. The problem stems from the fact that *the size of an array is part of the array's type*, but the size must be a compile-time constant. If one array is declared to hold 5 characters, and another is declared to hold 6, these two arrays are of different and incompatible types. In Pascal, therefore, it is not possible to write a function which accepts as an argument an array whose length is known only at run time — there is no way to declare the argument's type. Further, there is no way to create an object of arbitrary size at run time — because dynamic memory allocation only knows how to create an instance of a declared type, each of which has a size fixed at compile time.

Although the C's definition hides the fact, the size of a C array is also part of its type and must be a compile-time constant. Consider, for example:

```
int a, b[8];
```

and suppose that `sizeof(a)` is 4. Then `sizeof(b[0])` must also be 4, but `sizeof(b)` is 8*4, or 32. As in Pascal, it is impossible in C to declare a run-time variable-size structure; in C, though, it is possible to get around this restriction. Consider:

```
int *c, d, e;
 . . .
c = (int *) malloc(sizeof(*c) * d);
c[e] = . . .
 . . . = . . . c[e] . . .
```

in which `c` is effectively an array whose size is the value of the run time variable `d`. C allows this because the action of `malloc()` is undecipherable by the compiler, but it is also prone to human error. Weak typing also implies that a pointer to integer might point at any type of datum, whereas strong typing would at least guarantee that non-integer typed data could not be referenced through the pointer under any circumstances.

The differences are that:

- C permits a type to be overridden using a *type cast*, hence there are numerous acceptable “lies” about the type of a structure (even if relatively strong type-checking is done) and
- the conventions of C pointer arithmetic provide a consistent way to address beyond the declared bounds of a structure.

Together, these differences enable C to escape the problem much as FORTRAN’s pseudo-array-dimensioning does, except that the C mechanism does not need to use a subroutine call to evade type checking. It can be escaped by using either a *type cast* or an extra level of indirection (a pointer variable).

Paramtypes repair a logical inconsistency in the type system of C — they make it possible to declare data structures such as a variable-length array of characters, which C programmers commonly use, but have never been able to specify in such a way as to withstand strong type-checking. Paramtypes also permit RC to be used to directly encode algorithms which were designed for weakly-typed functional languages — a useful consequence because many algorithms have been formulated to use that model of computation.

3.4.2. The Paramtype Mechanism

A **paramtype** is a variable-size parametric type, each instance of which is tagged with its dimensions (which may be examined as though they were members of the structure). The notion of a *paramtype* almost occurs in some conventional languages: pseudo-array-dimensioning in FORTRAN does not define a type, but it is used in a way very similar to a *paramtype*. *Paramtype*s allow any data type to parameterized:

```
struct float_array(n) {
        float elements[n];
};
```

or:

```
struct family(length, children) {
        char lastname[length];
        int kids_ages[children];
};
```

Variable-size items would be declared as being of the type specified by a *paramtype* name without parameters — `struct float_array()` or `struct family()` in the above examples. New items of particular sizes can be declared as being of a *paramtype* name with integer-valued expressions for each parameter — such as `struct float_array(10)` or `struct family(16, people-2)`. The value of the parameters in an item can be found by treating the parameters as though they were `struct` members — if`jones` is of *par amtype* `struct family()`, `jones.children` would be the number of `children` in that `family()`.

The implementation of this concept can make use of a single contiguous block (which must be dynamically allocated in some cases) for storage of each *paramtype* instance (as opposed to linked-lists for each variable field, as others have proposed). Formulas for indexing such a structure are well known and differ from techniques for access of `struct` members only in that the expressions might not involve only constants. For various reasons, a *paramtype* instance is generally represented as a pointer to the block which contains the appropriate structure — useful because we may be able to save making copies of structures in some cases (assignments where the original value is never referred to again, like the return value of a function).

3.5. Sample Program

In order to demonstrate the power and ease of use of the RC constructs, we present three versions of quicksort modeled after the Pascal version which appears in [Wir76]. The first version is written in ANSI C (see Listing 3:1), and the second and third versions are written in RC (see Listings 3:2 and 3:3). Quicksort was chosen partly because it does provide a good demonstration of the benefits of RC over C and, more importantly, because quicksort is the de-facto standard example for functional and single-assignment languages such a ID [ArG78] and SISAL [McS85][32]. We encourage comparison with these other examples.

The ANSI C version of quicksort is:

```
/* Function prototype */
void sort(int *; int; int);

/* Function definition */
void
sort(int *a; int i, r)
{
  register int i, j, x, w;

  i = l; j = r;
  x = a[(l + r) / 2];
  do {
    while (a[i] < x) ++i;
    while (x < a[j]) --j;
    if (i <= j) {
      w = a[i]; a[i] = a[j]; a[j] = w;
      ++i; --j;
    }
  } while (i <= j);

  if (l < j) sort(a, l, j);
  if (i < r) sort(a, i, r);
}
```

Listing 3:1: C Quicksort

where the function prototype would normally be `#include`d from a header file separate from the file containing the function definition.

There is certain to be some fine-grain parallelism available from this coding. There is also the potential to execute the two `while` loops in parallel, since they do not interfere with each other. However, neither of these is the major source of parallelism in quicksort.

32. This is somewhat strange, since quicksort is supposed to be a sort in-place, yet functional and single-assignment languages do not permit operations to occur in-place.

In this code, although it is obvious that the two recursive calls do not interfere with each other (because we know how quicksort works), it may be theoretically, as well as practically, impossible for a compiler to discover this fact. The compiler would have to examine all the code which called `sort()`, and attempt to prove relationships involving the argument values, some of which may be derived from run-time input. The major source of parallelism for most machines, parallel execution of the two recursive calls, would not be recognized.

In contrast to the ANSI C version, the RC version is trivially analyzed:

```
/* Function prototype */
void sort(ˆ int *a);

/* Function definition */
void
sort(a)
{
  register int i, j, x, w;
  register int *below, *mid, *above;

  i = 0; j = count(a)-1;
  x = a[count(a) / 2];
  do {
    while (a[i] < x) ++i;
    while (x < a[j]) --j;
    if (i <= j) {
      w = a[i]; a[i] = a[j]; a[j] = w;
      ++i; --j;
    }
  } while (i <= j);

  part(a[w], below,(w<=j), mid,(w<i), above);
  if (count(below) > 1) sort(below);
  if (count(above) > 1) sort(above);
}
```

Listing 3:2: RC Quicksort

although the differences between the two versions are indeed small. Chapter 8 discusses the parallelization of the above RC quicksort in some detail.

However, the above RC quicksort merely demonstrated that writing good RC code to duplicate the function of C code does not require major changes in programming style nor notation. Many algorithms (for example, all divide-and-conquer algorithms), incorporate a partitioning as a basic operation and can therefore be expressed more efficiently in a language which directly supports such an operation. The following RC quicksort, using `fixpart`, takes advantage of the fact that partitioning is a fundamental component of quicksort:

```
/* Function prototype */
void sort(^ int *a);

/* Function definition */
void
sort(a)
{
  register int w, *below, *above;

  /* permanently partition (re-order) the array */
  fixpart(a[w], below,(a[w]<a[count(a)/2]), above);
  if (count(below) > 1) sort(below);
  if (count(above) > 1) sort(above);
}
```

Listing 3:3: “Clever” RC Quicksort

Notice that, unlike most “clever” improvements to code in conventional languages, the above coding is not only more efficient — it is easier to understand. (Of course, this “clever” version is only allowable using dynamic partitioning implemented by copy or map operations.)

4. Refining Other Languages

In the preceding sections, we have shown the application of the refined language methodology to two very different languages, FORTRAN and C. Although the methodology calls for careful study of each language to determine its particular analysis-inhibiting features, most languages in popular use today can be refined in the same general way as FORTRAN or C.

As a partial list, we believe that the following languages can be refined using little beyond the modifications discussed above:

> Ada, Algol, APL, Assembly Language (for most machines), Atlas, Awk, BASIC, BCPL, CoBOL, Forth, Modula 2, Pascal, PL/I, PL/M, RatFor

But this is not true for all languages.

In sections 3:4.1 and 3:4.2, we consider the two largest classes of language which cannot be refined as discussed above: languages with basic data structures which inhibit parallelism (e.g. Lisp and Prolog) and languages which are characterized by their explicitly parallel constructs (such as CSP and OCCAM).

4.1. Toward Refined Lisp

The problems involved in refining Lisp are:

- *Dynamic binding.* Variables, and functions, in a Lisp program are bound to names via *most recent binding*, whereas most languages (and, recently, also many dialects of Lisp, such as Scheme) bind names based on their lexical, compile-time, scope of definition. Many Lisp programmers, especially those working in artificial intelligence, have made this characteristic of Lisp a vital element of their programs, using it to write unconstrained self-modifying code. Such code utterly defies compile-time analysis.
- *Ordered lists.* The primary data structure of Lisp is the list. *Or dered* lists are inherently sequential structures; little parallelism is to be found in operations on a list (except, perhaps, using techniques such as that in section 7:1.3.2). The problem is simply that, even if a particular list is being operated-on in ways which do not depend on the existence of a particular ordering in the list, there is no way to guarantee that the ordering will not be examined later and hence be significant. A number of researchers [SoD85] [TaH86] have proposed specialized data structures which partially linearize access to linked lists, thereby allowing some parallelism, but these are very weak “patches” because they create potential for parallelism in unpredictable ways, rather than reliably enabling parallel execution where the programmer’s algorithm naturally permits it.

These two major difficulties can be remedied by addition of a single construct. We call this construct an **environment**[33], after the same construct in PILE [DiG83], an advanced computer-aided-instruction language. Similar constructs are Snobol’s *table* and Awk’s *associative array* [AhK].

4.1.1. Property Lists

It seems to be the case that the main reason for writing self-modifying Lisp code is that *property lists* operate too slowly. For example, if one is building an expert system which may dynamically add new rules, a common approach is to create, at runtime, a new function to evaluate each new condition. The name of this function belongs to the same namespace as the functions already occurring in the program; hopefully, it will not accidentally be the same as one of them, because, if it is, it will replace that function (thanks to most recent binding). The benefit is that testing the condition will be extremely quick, since Lisp maintains

33. Although it was not discussed in the section 3:3, the original proposal for refined C [DiK84], which was drafted before information about the new ANSI C standard was available, included the environment structure. It was dropped from RC because the new ANSI C standard includes a function prototype syntax which, in conjunction with the existing data declaration syntax, is easily expanded to be sufficient for RC.

a global name table and the binding of the condition function to its name is handled by direct lookup.

The conventional Lisp mechanism intended for this type of use is called a *property list*. Using`getprop` and `putprop`, a *property list* appears to be very much like a local name table, which may be randomly accessed. Speeding-up operations on property lists is difficult, however, because they are, in fact, ordinary linked lists and are occasionally operated on as such — if only `getprop` and `putprop` were used to operate on them, and the compiler could be certain that this would always be the case, then there would be no problem.

4.1.2. Environments

The new refined Lisp construct is the **environment**: a named, unordered, name table — a named "scope."

Although names may be dynamically created and destroyed within an environment — as they may in conventional Lisp's global environment — the data-access flow properties of each item in an environment *may not exceed* the properties of the environment as a whole. Therefore, in addition to providing a non-serial data structure, an environment is a construct for associating routines with access rights to sets of data within explicitly named scopes — a way of specifying exactly "how global" each piece of global data really is.

Each environment is declared as having a name and a list of names in the global environment which may be referenced from within the environment. For example:

```
(environment ’wm)
(environment ’pm
    (canmodify ’wm)
)
(environment ’peekatwm
    (canread ’wm)
)
```

specifies that `wm` is an environment; `pm` is another environment such that any functions whose code is placed in `pm` can only modify the associations of names within the `wm` environment; and `peekatwm` is an environment whose functions can only read `wm`.

A function can only be called if the set of access rights it has to data in various *environment*s does not exceed the caller's rights; hence, in the example above, a function which is placed in the pm environment could call one which resided in the peekatwm environment, but not vice versa. In order to avoid subtle problems involving access permission ambiguities arising from copying of function code into the main environment for execution, *only functions within a named environment may be created at runtime and they must be executed directly from the environment*.

Although we have not yet devised a specification of the syntax of refined Lisp (RL), it is hoped that the following examples, using a somewhat arbitrary syntax, will demonstrate the underlying concepts. References to environments use `{` and `}` so that they can be distinguished from both the global namespace and ordinary (ordered) Lisp lists. A name `this` within the environment `that` would be referenced as:

```
{that ‘this}
```

and its value could be set to `something` by:

```
{that ‘this ‘something}
```

It is very much like having `{` and `}` imply, depending on the number of arguments, either a special kind of `getprop` or `putprop`. Referencing:

```
{that}
```

Also has a special meaning: it returns an ordinary Lisp property list containing the elements of `that` in an unspecified order.

There are, of course, other changes which must be made to Lisp, but they are similar to those discussed for FORTRAN and C.

4.2. Refining Explicitly-Parallel Languages

If one wishes to obtain the benefits of transformability and debuggability associated with refined languages, but the base language is characteristically explicitly parallel, it becomes necessary to consider refinements that can be made without eliminating all explicit parallelism.

Before discussing these refinements, it is useful to define what it means for a language construct to be explicitly-parallel:

> A construct is **explicitly-parallel** if and only if it specifies parallel control flow through code in such a way as to prohibit mechanical transformation, at compile time, into a functionally-equivalent sequential control flow through the code.

For example, in most languages which support it, vector notation is *not* explicitly parallel. Given:

```
A[L .. U] := B[L .. U] + C[L .. U];
```

It is always valid to generate:

```
FOR I := L TO U DO
        T[I] := B[I] + C[I];
FOR I := L TO U DO
        A[I] := T[I];
```

Any vector expression has a well-defined (although not necessarily straightforward) sequential representation which is functionally equivalent.

For example, `DOALL` is a perfectly harmless construct if the language specifies that execution of a `DOALL` must duplicate the result computed when a specific sequential interpretation is used. If the language does not specify this, communication between parallel processes can cause interesting results:

```
DOALLI IN (L+1 .. U)
    A[I] := A[I-1];
```

could copy the value of `A[L]` into all the elements of `A[L+1 .. U]`, or it could shift the values of `A[L .. U-1]`into `A[L+1 .. U]`, or it could do something mixing these interpretations. If code lik e the above is permitted within a `DOALL`, and a *sequential functionally-equivalent* ordering is not specified as part of the language definition, there is no way to determine the intended meaning of the construct. If the intended meaning cannot be determined, it is clearly impossible to transform the construct and guarantee that the meaning is preserved.

The same is true of various other specialized “parallel” constructs which are found in ANSI FORTRAN 8x and similar languages — these constructs are parallel only if one wishes to view them as such.

The kind of control construct which is unavoidably, explicitly, parallel is the kind of construct which permits communication between arbitrary processes at other than process creation/termination. These constructs are not, in general, able to be transformed into sequential equivalents: multiple process states must be maintained. Of course, the parallelism can be simulated, but this is neither efficient nor helpful in making the program debuggable.

Message-passing and other general parallel control constructs cause complexities which include:

- *It may not be possible to determine what code will receive a message until the message is actually sent at run time.* Being unable to determine the destination of the message makes the compiler unable to determine the effects the message will have — it thus prevents the compiler both from restructuring the parallelism and from aiding in debugging the program.
- *Timing relationships may exist dependent on a particular scheduling trace of the program, but the trace is not specified by the program.* An apparently bug-free program actually might be infested with race conditions and deadlocks. Transforming the program (or the target machine) in any way might cause these errors to surface. To preserve equivalence, the compiler may have to generate code which carefully models the original machine environment, *including* relative timing of all instructions.

To refine an explicitly parallel language *without* removing the parallel construct, the approach is very similar to that taken to refine sequential languages, but the explicitly-parallel constructs remain in the language. This results in a useful tool for*detecting possible r ace conditions* in explicitly parallel code and, if the program is correct, it permits the parallelism to be safely increased (by concurrency detection within sequential chunks of the parallel program).

Ideally, the same facilities for specifying data-access flow will be inserted in the language, but a little more information is desired:

- In a situation where it cannot be determined at compile time what code will receive a message, the refinement is to state the set of functions (assuming each process executes the code for a function) which may receive that message.
- Where a timing relationship implicitly resolves a race condition, the *time constraint* should be able to be specified. This would, incidentally, solve the problem of writing real-time control software which must meet critical timing requirements. Assertions can be made about when the program's execution will have reached one point relative to when it reaches another.

These last two points are currently just interesting ideas — further research is needed to determine if and how these modifications would be worthwhile.

This page is intentionally blank.

Compilation
Of Refined Languages

A refined language program is a sequential program. It differs from an equivalent base-language program only in minor ways, as outlined above. By design, most of the differences do not require that the basic compilation technique be altered. The only exceptions are that in compiling refined language programs:

- Separate compilation may be easier to support (depending on the target language),
- Interprocedural information is much easier to collect (since it is, in effect, directly available in the function prototypes, etc.), and
- More precise aliasing information is readily available.

For this reason, the problem of compiling a refined language program into efficient parallel code for a particular target machine can be considered as *identical to* the problem for an equivalent base-language (dusty-deck) program into efficient parallel code for the same machine.

Any of the techniques of [KuM72] [AlK82] [Ell85] [Vei85] [KuS84] [Nic85] [KAI85] [ScK86], all of which were developed to process “dusty deck” FORTRAN code, could be used on refined language code with equal or greater success. Greater success would be very likely, since a refined language program contains more parallelism-relevant information than does a dusty deck. In each case, the resulting refined language compiler would be virtually identical to the dusty deck compiler, save for the minor differences which occur mostly in the lexical analyzer and parser.

Beyond the benefit accrued from the added information in the code, a refined language compiler provides additional benefit in that it performs inter-procedural parallelization, and uses a unique method to tailor code to the target architecture.

The rest of this chapter presents an overview of the complete process of compiling a program written in a refined language. Contributions of this material include the realization that flow analysis is flow analysis (whether used for optimization or parallelization) and the general organization of the compiler, which distinguishes between concurrency detection and process packaging.

1. Overview

As will become clear in the following sections, the compilation of a sequential program into parallel code is nothing more complex than conventional compiler analysis and code improvement; however, the *penalty for neglecting an improvement* is far greater when a parallel machine is the target.

Toward insuring that the target machine is well utilized, our refined language compilers incorporate a technique for non-deterministic process packaging — but this does not complicate the compiler writer's task. In fact, the use of a non-deterministic packaging technique makes writing the compiler slightly easier.

Taking this concept a bit further, it can be argued that conventional compiler analysis and code improvement *are* mostly concerned with a particular kind of parallelism — a parallelizing compiler merely generalizes this. For example, register assignment is a special case of the general problem of assigning data to local memories of multiple processor machines; instruction scheduling to keep pipelines busy within a single processor is very similar to instruction scheduling to keep an entire VLIW supercomputer busy; code motions used in conventional optimizing compilers might not be profitable on parallel machines, but either the code motion or its inverse usually is.

In summary, the "big picture" of a parallelizing compiler is that of a conventional optimizing compiler, but with a number of things re-named:

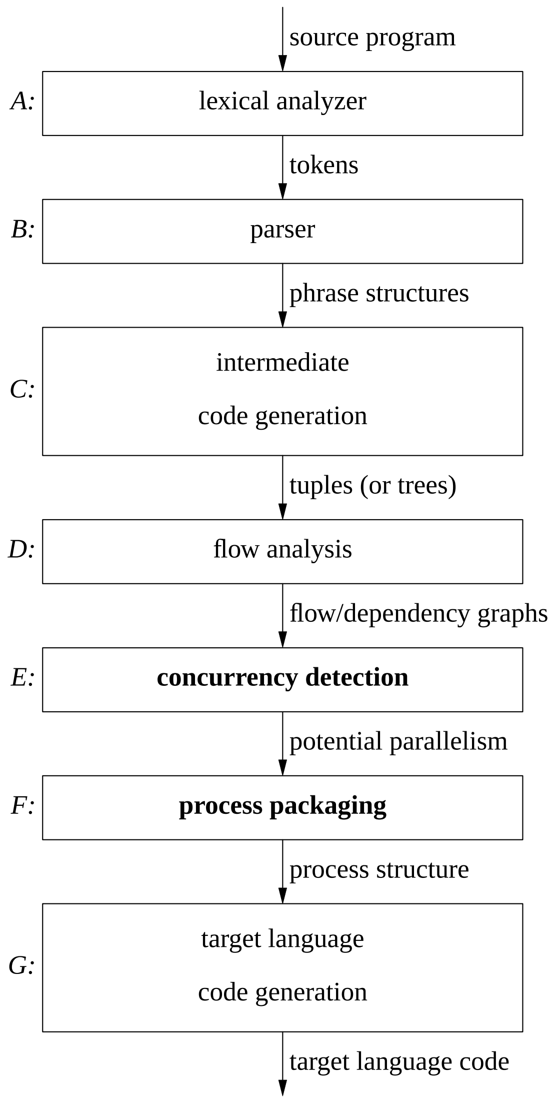


Figure 4:1: Overview Of Refined Language Compiler Organization

2. *A:* Lexical Analysis

The structure of the lexical analyzer of a refined-language compiler is, in every respect, as it would be for the corresponding base-language compiler. There may, however, be a few new keywords and/or symbols — words like `PARTITION`, `IN`, `OUT`, etc.

Since a refined language is supposed to function in as similar a way as possible to its base language, lexation rules are consistent with the base language. For example, if base-language keywords are reserved (may not be used as names for programmer objects), then the refined-language keywords are also reserved.

3. *B:* Syntax Analysis

Syntax analysis, or parsing, of a refined-language program is performed in exactly the same way used to recognize the syntactic structure of a base-language program. There simply are a few new constructs in the refined language.

These new constructs necessitate minor re-design of the compiler's symbol table. This is partly due to the fact that high-level computer languages are actually context sensitive; context-free parsers are constructed to use symbol table information to (semantically) resolve the context-sensitive grammatical structures. For example, the difference between a subscripted array reference and a function call in FORTRAN is simply that an array name would have to have been declared as such; the refined-language compiler may have to distinguish these from a reference to a partition element. A second reason the symbol table must be modified is that information regarding explicitly stated data access flow must be preserved for intermediate code generation and error checking.

4. *C:* Intermediate Code Generation

Starting with the generation of intermediate code, there are significant differences between a sequential-code-generating optimizing compiler for a base language and a parallelizing compiler for a refined version of that language. More precisely, there are important parallelism-related target machine dependencies. For example, consider the assignment statement:

```
A = B
```

Is there any parallelism in this? At first glance, the answer is no: "A is given a copy of B's value." An intermediate form in which variable assignment is a primitive operation would also evidence no parallelism.

However, suppose the target machine is capable of using finer-grained parallelism (e.g. it is a VLIW). Representing intermediate operations at a finer-grained level, the single statement is readily expressed as:

```
t1 ← Address(A)
t2 ← Address(B)
t3 ← Memory(t2)
Memory(t1) ← t3
```

which is parallel-executable: `t1 ← Address(A)` may be executed in parallel with either `t2 ← Address(B)` or `t3 ← Memory(t2)`. The level at which an algorithm is specified in the internal form is the finest grain parallelism for which parallel code could be generated.

It can be argued that the lowest-level intermediate form would suffice for any target machine, but the time required to analyze and re-combine too fine-grained intermediate operations is prohibitive for many large-grained target machines. The complexity of the compiler's target code generation is also magnified by a mismatch of intermediate form and target machine.

In addition, the lowest-level intermediate form depends, to some extent, on the target machine model — there isn't a single lowest level, but many which differ because of machine features such as the memory hierarchy, choice of storage management technique, etc. In the above example, which appears to be at a very low level, it is still possible that the address of a variable cannot be computed using just one intermediate code instruction. It may, for instance, be necessary to obtain a variable's address by adding an offset to the value of a frame pointer.

5. *D:* Flow Analysis

Throughout the literature on dusty-deck parallelization, most discussions of flow analysis in support of concurrency detection have employed non-standard terminology, primarily as a consequence of describing flow analysis on HLL constructs rather than on an intermediate form (for example, [Kuc78] describes the analysis on HLL code). The standard terminology used to describe flow analysis for the purpose of conventional optimization was largely ignored. In any case, the differences in terminology are unimportant; there is *no inherent difference* between flow analysis used for conventional code improvement and that used for concurrency detection.

It is worth noting that it isn't always necessary that complete flow analysis be applied — in fact, it is very rarely useful to perform complete flow analysis. Flow analysis is not an end in itself; analysis need be done only if the information it produces is vital to generating efficient code for the target machine. For example, the analysis used for trace scheduling of VLIWs [Fis84] is essentially the basic-block-oriented value-numbering scheme proposed in [CoS70] and repeated in [AhU77]. The model of parallelism used by

[Fis84] [Ell85] cannot execute control flow statements in parallel, hence more powerful flow analysis (that considered the control flow structure in general) would not result in significantly better parallel code — it would be a waste of time.

When code generation requires flow information spanning larger regions of the program than a basic block (or trace extension thereof), results are not particularly sensitive to the choice of techniques and, if minor variations in the worst-case performances of the various techniques are ignored, any conventional flow analysis technique may be used to produce the needed information. The most commonly discussed techniques are:

- *Linear Nested Region analysis.* This is a straightforw ard extension of the basic block value numbering scheme, which traces the flow of computed values across basic block boundaries. Certain branching structures will cause the algorithm to fail to detect some information, but such a failure is never fatal. This technique was proposed in [CoS70].
- *Set Based analysis.* Aho and Ullman observ ed that a value can be carried across a control structure only if it is stored into a variable. Using this insight, they proposed (in [AhU77]) that only flow of variables, not computed values, should be traced across control structures and they developed a technique for iterating over all paths in a program graph until the set of variables going into (IN) and out from (OUT) each basic block have stabilized. Unlike Linear Nested Region analysis, this technique results in theoretically perfect answers, but it is blind to duplicated intermediate computations because they never are associated with variables. It is clear that by creating “pseudo-variables” to hold each intermediate value this flaw can be eliminated; see [AnC82]. The first RC compiler uses this technique [Ste86].
- *Structured Control analysis.* The rules upon which Linear Nested Re gion analysis is based can be reduced to a purely syntactic form, provided that the language uses only structured control constructs. This analysis [Wed83] is far better known than either of the above two techniques and has been widely published. Recently, specification of flow analyzing compilers using attributed grammar compiler compilers has become a “hot” research topic: virtually all of these compiler-compiler generated flow analyzing compilers perform the analysis by this technique. Flow analysis of single-assignment (restricted side-effect) languages is most often performed in this way. The first C version of PREFINE uses this technique.

Whichever technique is used, it is usually beneficial to view flow analysis as a two-step procedure:

(1) Perform analysis incrementally on the elements of each basic block and

(2) Determine what information crosses each basic block boundary.

Since the techniques and terminology are well known, we will discuss flow analysis no further — except to apply its results. In what follows, we will use the terminology of [AhU77] to describe the results of flow analysis.[34]

It is important to mention that we consider the concept of **dependence** to be a refinement of conventional flow analysis, essentially making flow information track portions of compound-typed variables[35] (sets of array elements as opposed to the array as a whole). A very crude implementation of this concept appears in [CoS70], pages 324-325; more recently, much progress has been made in improving the precision with which references to portions of data structures are recognized and in solving linear recurrences which often result from indexed array references within loops [All83] [Bur84] [BuC86] [Ban86] [All86]. Dependence analysis also provides terminology for discussing data references within a loop where iterations interact with each other; this terminology is particularly useful in discussing parallelization of FORTRAN `DO` loops. In summary, while dependence analysis has become a field unto itself, the current work treats it as *an integral part of flow analysis* and avoids relying on its specialized terminology.

6. *E:* Concurrency Detection

Concurrency detection, whose fundamental principles are discussed in Chapter 5, is the first logical step in parallelization of code. It determines what *could be executed in parallel* and what kinds of parallelism are present. To generate good parallel code for a target machine, however, the compiler must go a step further: it must determine what is *worth executing in parallel* and it must pick the particular process structure which will be used to achieve that parallelism.

Notice that concurrency detection is fundamentally machine independent.

7. *F:* Process Packaging

Process packaging is the procedure by which a graph resulting from flow analysis is partitioned, according to the results of concurrency detection, into the specific regions that will be coded so that they may be executed in parallel on the target machine. Unlike concurrency detection, process packaging is a highly machine-dependent procedure.

Process packaging is a better term than “graph partitioning” because it leaves open the possibility that constructs might be added to a partitioned graph in order to encapsulate each process. Typically, code

34. Surprisingly, although the methods differ, nearly all compiler textbooks agree on the basic terminology used to describe flow analysis for the purpose of code improvement/optimization.

35. Typically, this is not a primary concern in a refined-language compiler, since the *programmer* can explicitly partition a compound-typed variable to create names for any portions of interest. It is critical to good parallelization of “dusty deck” FORTRAN code.

for each process is packaged with the operations which support process creation, termination, synchronization, and communication.

In most cases, very many very small processes could be executed in parallel. However, it is very rare that the target architecture can make effective use of all the potential parallelism in a program. For most target machines, groups of potentially-parallel processes must be packaged together to make each generated process an appropriate size, and type, for the target machine environment.

There are two basic approaches to process packaging: deterministic and non-deterministic. Because work on the first refined-language compilers has dealt with code generation for MIMD computers, the current work refers to a stream of instructions which are to be executed in a specified sequence as a *process* no matter what the target machine's organization is.

7.1. Deterministic Process Packaging

The earliest, and most commonly used, approach to process packaging is based on a deterministic search for any of a set of pre-defined preferred parallel-executable forms. The forms are searched in order of decreasing "value," hence, the first applicable transformation is presumably the best.

For example, most of the work presented in [KuM72] [AlK82] [KuS84] [KAI85] focuses on transformation of dusty deck FORTRAN programs into a vector-oriented extension of FORTRAN. When a`DO` loop is encountered in the source code, a specialized (target machine dependent) kind of concurrency detection is performed on the loop body. This results in classification of the loop based on the relationship of one iteration of the loop to the next (loop-carried dependences) and on the interrelationships of statements within the loop (true dependences). Given the classification of the loop and its contents, the corresponding vector operation(s) are selected.

Since all mappings of loop classification into vector operation(s) are known to preserve the meaning of the original construct, correctness is implicit in the set of transformations considered. The preferability of using the vector operation(s) is simply assumed in most cases[36].

Because only a relatively small number of specific forms of parallelization are sought and only within `DO` loop bodies, reasonably good code is generated with relatively little effort — relative to the effort required to generate good code for more flexible parallel machines such as MIMDs. This permits more compile time to be spent on sophisticated encoding of vector operations. For example, Mini-KAP AF

36. [ScK86] and [Wol86] are exceptions: several vectorized forms are evaluated and the best of those is generated.

[KAI85] will recognize when a `DO` loop is finding the maximum or minimum of an array, and will transform such a loop into a vector maximum or vector minimum operation.

Finer-grained parallelism can also be supported deterministically: good examples are the tree-height reduction techniques given in [Kuc78].

In general, deterministic packaging works best for machines that can only execute highly symmetric and relatively static parallel structures, such as SIMDs, vector processors, and systolic arrays (discussed in Chapter 6). For these machines, there are usually only a few possible parallel-executable forms, hence the relative costs and benefits of the different forms can be cast into a fixed search order for parallelizations, which checks the best first and accepts the first match.

7.2. Non-Deterministic Process Packaging

An alternative form of process packaging, one more suited to architectures which can exploit more complex forms of parallelism, is non-deterministic process packaging — a heavily pruned search of the space of all possible packagings. (Chapter 8 discusses our contribution to this technology.)

Non-deterministic packaging is desirable only when the number of viable alternative parallel-executable forms is large and/or the relative merits (costs) of different parallel forms cannot be estimated with reasonable accuracy unless the features of the parallel code are carefully evaluated for each alternative. Dynamic parallel machines (MIMDs and dataflow machines) and fine-grain machines which do not require large-scale symmetry (VLIWs, pipelined computers, and some array processors) are good targets for non-deterministic packaging. (Further discussion of these machines appears in Chapter 6.)

Those target machine characteristics which make non-deterministic search attractive are such that the relationship between specific source code constructs and the parallel forms they are transformed into is complex. Non-deterministic process packaging cannot save time in concurrency detection by ignoring a portion of the source program, because it wouldn't know what to ignore. Hence, analysis must be *performed on the entire program*.

However, compilation using non-deterministic packaging can be sped up by performing hierarchical analysis [DiK84] [Nic85] [SkG85] [Vei85] [SaH86]. Initially, relatively large-grain parallelism is sought and then parallelism at ever-finer-grain levels is considered. This limits the search sufficiently to make compile-times comparable to those obtained with deterministic vectorizers [Ste86].

On machines which are entirely fine-grain, such as VLIWs, this doesn’t quite prune the analysis enough, but artificial constraints can be imposed on the fine-grain re-arrangement of the code so that the search space is narrowed. **Trace scheduling** [Fis84] employs just such a constraint: only code on the most frequently executed traces through the program is examined in complete detail, while lower-frequency paths may not be analyzed at all.

Process packaging is just beginning to be perceived as a fundamental problem in automatic parallelization; when nearly all parallel machines were vector or array machines, only a few parallel forms were of interest, and picking the right form was far less difficult.

8. *G:* Target Language Code Generation

Fortunately, once process-packaging has been performed, the remaining machine dependent problems (at the level of generating code for each process) are identical to those encountered in sequential machines. This phase of compilation is the same as in a compiler for a sequential machine.

The only difference, and it is not universal, is that memory allocation and executable code module creation can be more complex than for a sequential machine. This is due to the need, in some machines, to assign data and code to positions within explicitly segmented memory systems — binding processes to (virtual) processors at compile time.

This page is intentionally blank.

Concurrency Detection

Concurrency detection is the method by which a compiler can determine what portions of a program may be executed simultaneously without the possibility of changing the meaning of the program. This determination can be made whether the source program uses sequential or explicitly parallel control flow constructs. If the source program uses sequential control, concurrency detection determines how the code's execution could be parallelized without changing the functionality of the program. If the source program uses parallel control, concurrency detection determines both how the code's parallel structure can be safely transformed (into a parallel execution structure other than that given explicitly) and where race conditions may arise within the unmodified source program's structure.

In this chapter, the machine-independent theoretical foundations of concurrency detection are described in terms of conventional, uniprocessor-oriented, compiler flow analysis and optimization. This description concentrates on the analysis of **irregular code** — code which contains arbitrary side effects and control constructs — largely ignoring parallelizations of the iterations of loops. Loop iteration parallelizations are discussed briefly in Chapter 6 and in-depth in Chapter 7, rather than included here, because the fundamental approach to loop parallelization is not consistent across various target architectures. Since the discussion of loops is postponed, the reader may wish to consider source program loops as being completely unwound prior to the analysis in this chapter (although this is rarely done in practice).

Many other researchers have expressed rules for concurrency detection [Kuc78] [Wed83] [Omo84] [Vei85] [AlB86], however, we believe the current presentation to be unique in its use of conventional terminology, separation of machine-independent from machine-dependent analysis, incorporation of analysis of code using explicitly-parallel control constructs, and in its generality (the rules given apply to parallelism at any granularity[37] and completely specify the mechanism whereby the granularity of analysis may be changed).

37. Granularity refers to the "computational content" of each atom (graph node) in the analysis. Fine grain analysis would find potential parallelism among individual intermediate-code instructions, whereas large grain analysis might find parallelism only among atoms representing entire subprograms.

1. Terminology And Notation

In order to put the notions of concurrency detection and process packaging on a more precise foundation, it is useful to define the transformed representation of the source program to which the analysis is applied.

The representation used is actually a hybrid graph representing control flow between nodes, where each node represents a basic block given as a DAG, as used in conventional optimizing compilers [AhU77] [AhS86]. However, the graph structure itself is not vital in concurrency detection — rather, concurrency detection operates on regions of intermediate code, and flow analysis summary information pertaining to them, which are derived from the hybrid graph. For this reason, detailled discussion of the hybrid graph structure is avoided. Instead, concurrency detection is described in terms of detecting potential for race-free parallel execution of **regions**[38] (sets of intermediate code instructions) annotated with flow analysis summary information.

In this section, the intermediate form and relevant terminology are introduced.

1.1. Intermediate Form

For this presentation, each instruction in the intermediate form is expressed using a tuple notation similar to that of [AnC82]. Consider the following fragments of a C program:

38. A precise definition of region appears later in this chapter.

```
        . . .
        a = b * c;
        if (d) {
                b = b * c;
        } else {
                a = f() + b;
        }
        c = a + b;
        . . .

f()
{
        g = a + e;
        return(g);
}
```

Listing 5:1: C Code For Concurrency Detection Example

After parsing and translation into intermediate code this might become:

```
     . . .
  1: Load(#b)
  2: Load(#c)
  3: Mul(1, 2)
  4: Store(#a, 3)
  5: Load(#d)

  6: Load(#b)
  7: Load(#c)
  8: Mul(6, 7)
  9: Store(#b, 8)

 10: Call(#f)
 11: Load(#b)
 12: Add(10, 11)
 13: Store(#a, 12)

 14: Load(#a)
 15: Load(#b)
 16: Add(14, 15)
 17: Store(#c, 16)
     . . .

100: Load(#a)
101: Load(#e)
102: Add(100, 101)
103: Store(#g, 102)
104: Load(#g)
105: Return(104)
```

Listing 5:2: Tuple Code For Concurrency Detection Example

where the number which precedes each operation is used only to refer to the tuple — these numbers do not imply execution order. Blank lines separate the basic blocks (whose DAGs are expressed in the pattern of references to tuple numbers as arguments of each tuple within each block):

Basic Blocks For Concurrency Detection Example

Basic Block	Tuple Numbers	Exit Arc(s)
A	{1, 2, 3, 4, 5}	*True(5)* → B, *else* C
B	{6, 7, 8, 9}	D
C	{10, 11, 12, 13}	D
D	{14, 15, 16, 17}	. . .
E	{100, 101, 102, 103, 104}	*Return*

The corresponding control flow graph is:

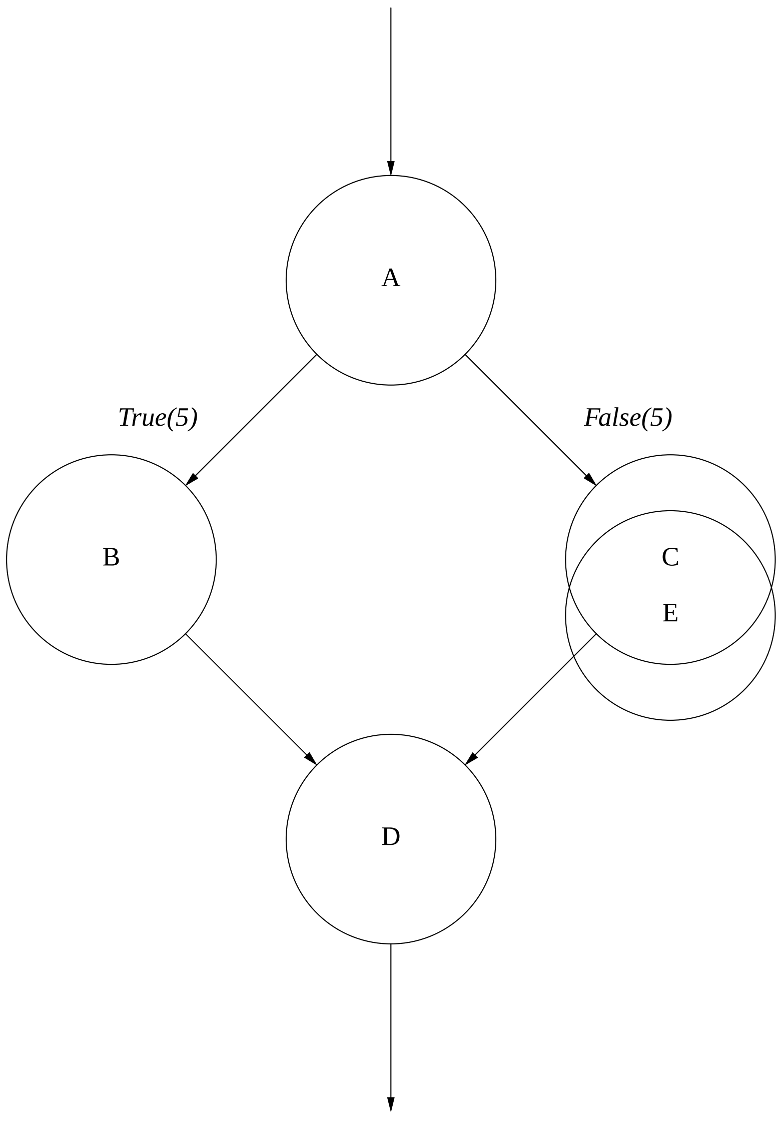


Figure 5:1: Control Flow For Concurrency Detection Example

where block *E* is drawn within block *C* to stress the fact that its contents are conceptually, if not actually, contained within block *C*.

1.2. Variables

To simplify the discussion, we refer to *an atomic named storage cell* as a **variable**. In fact, if operations on individual members of a variable of a compound type (an `array`, `struct`, or `record`) appear in the code, it is usually desirable to treat each member of the compound-typed variable as an independent "variable." (Each result from an intermediate computation is also typically best treated as an independent "variable.")

The complications introduced by this consideration relative to subscripted array references are well-known and have been heavily researched. Dependence analysis [All83] [Bur84] [BuC86] [Ban86] [All86] distinguishes sets of members (array elements) based on recognition of an indexing pattern; in the current work, it is assumed that such techniques will be used where appropriate[39]. Similar complications arise in aliasing of unrestricted multi-level pointer operations [AhU77] [AnC82] [AhS86] [Ste86]. The problem of disambiguating references to members of `struct`s is trivially solved.

Although, for these reasons, it may be necessary to apply any of several sophisticated algorithms to derive the *optimal* set of "variables" within a program, it is always possible to define an acceptable set of "variables". For example, if member references are all grouped together as references to the complete structure, the concurrency detection algorithms given here will function properly — but some parallelism may not be recognized.

In any case, once a set of variables has been defined, concurrency detection can be performed in exactly the same way, no matter how this set was derived. In the example given in Listing 5:1, since only variables of simple types are used, choosing a set of "variables" is easy —`a`, `b`, `c`, `d`, `e`, and `g`.

1.3. Flow Analysis Terminology

Since it is relatively convenient to do so, this discussion employs the terminology of conventional, sequential, compiler flow analysis, as per [AhU77] [AhS86]. These terms are **use**, **def**, **D-U chain**, and **U-D chain**. Two new terms, **def closure** and **use closure**, are also defined — but these are simply extensions of the concepts of D-U chains and U-D chains, respectively.

39. Since refined languages permit the programmer to explicitly create such sets (via partitioning), dependence analysis is not as important as it would be if conventional language programs were the input to the compiler. We believe that dependence analysis is best viewed as a complementary technology to language refinement — it provides a mechanism for mechanically converting "old" code into its refined-language equivalent.

The two most basic terms are def and use:

Definition 1: Def

A **def** (definition point) is an operation which temporarily binds a value *v* to a name *n*.

Definition 2: Use

A **use** is an operation obtaining the value *v* which is bound (at that point in execution) to a particular name *n*.

In terms of the tuple code in Listing 5:2, a line of code of the form:

number`:` *operation*`(`*arguments*`)`

where *operation* is neither `store` nor `call`, constitutes a use of the value associated with each name in the list of *arguments* and a def binding the computed value to the temporary name which identifies this tuple. A tuple like:

number`: Store(`*place*`,` *value*`)`

constitutes a use of the value associated with *value* and a def binding the name *place* and the value *value*. Finally, a tuple like:

number`: Call(`*arguments*`)`

represents the summary of all uses and definitions occurring within the subprogram being called.

It is important to note, however, that the above definitions specify that a use is really *a use of a def or set of defs* — not a use of a name. In other words, a particular use, *u*, of the value bound to the name *n* does not necessarily have any relationship to the value bound to *n* by a specific def; only defs which could be the last def of *n* executed before executing the use *u* are significant.

This distinction is made because, for example, when collecting summary flow information for a region of code (such as a function body), a use of variable *n* is a use in the summary flow information iff the use may be a use of a def appearing outside of the region. Without this distinction substantial parallelism may be lost.

The concepts of D-U and U-D chains embody the information directly relating defs and uses:

Definition 3: D-U Chain

The D-U Chain of a particular def δ which establishes the binding *n*-*v* consists of a set containing the def δ and all uses of the value bound to *n* where, for each use *u*, there exists at least one control flow path from δ to u such that no other defs of n are encountered along that flow path (δ *reaches* u).

Definition 4: U-D Chain

The U-D Chain of a particular use u of the value bound to n consists of a set containing the use u and all defs of n such that for each def δ, u is in the D-U chain of δ.

Hence, for example, the D-U chain of the def of a in tuple number 4 of Listing 5:2 would include both the use of a which is summarized within tuple 10 (which is actually the use in tuple 100) and the use of a in tuple 14. The U-D chain of the use of a in tuple 14 would include both the def of a in tuple number 4 and the def in tuple 13.

Notice that there are two separate reasons that the U-D chain of a particular use might hold more than one definition, although only the first applies in the above example:

(1) The use may follow a conditionally executed definition of the variable in question or

(2) The use may refer to a variable whose identity is not precisely known at compile-time; for example, if the use obtains a value through a pointer reference using the pointer variable p and p may be pointing to either variable a or b, then all definitions of a, b, or *p[40] that *reach* the use would be in the U-D chain of the use. In general, aliasing ambiguities make it necessary to treat each def (or use) as a set of possible defs (or uses), one for each of the possibly-aliased names.

In addition to these traditional terms, it is profitable to define extended versions of D-U and U-D chains:

Definition 5: Def Closure (D*)

The **def closure**, denoted D*, of a particular def δ consists of the D-U chain of $\delta \cup$ the D-U chain of each def which establishes a binding where the value used in the binding was either in D* of δ or resulted from a computation involving at least one use that was a member of D* of δ. Informally, D* of δ consists of the def δ and the set of all subsequent uses and defs which may depend on prior execution of the def δ.

Definition 6: Use Closure (U*)

The **use closure**, denoted U*, of a particular use u consists of the U-D chain of $u \cup$ the U-D chain of each use which is either used to produce the value for a def which is an element of U* or is involved in a computation whose result is used to produce the value for a def which is an element of U*. Additionally, each use which is involved in selecting a flow path applied by D-U chains within U* of u is also a member of U* of u. Informally, U* of u consists of the use u and the set of all uses and defs which may have to be executed to produce the value which is used in use u.

these essentially embody the concept of closure of a D-U chain (for D*) and of closure of a U-D chain (for U*). Hence, for example, D* of the def of a in tuple number 4 of Listing 5:2 would include tuples 4, 10

40. The notation is borrowed from the C language and represents indirection through the pointer p.

(summarizing tuples 100, 102, 103, 104, and 105), 12, 13, 14, 16, and 17. U* of the use of `a` in tuple 14 would include tuples 13, 12, 11, 10 (summarizing tuples 105, 104, 103, 102, 101, and 100), 5 (for control path selection), 4, 3, 2, and 1.

1.4. Regions

Individual tuples (and the uses/defs they imply) can be considered to be the “steps” of the algorithm which actually impose constraints on parallel ordering. However, as mentioned in the beginning of this section, it generally is not desirable to parallelize individual steps, but rather to parallelize larger regions.

Let:

$\mathbf{R} = \{ \mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_P \}$

be a partition of all intermediate code tuples which contains P partition-elements, henceforth called **regions**.

Any partitioning which separates single-control-entry single-control-exit subgraphs is valid, as is any partitioning which can be described in such a way if the graph is modified by a sequence of valid code motions (as used in conventional optimizing compilers [AhU77] [AhS86] [Die84]). This graph-based constraint on validity of a partitioning is essentially requiring that *no region can contain tuples from within a graph cycle (loop) unless either the cycle is completely contained within the region or the region is completely contained within the cycle*.

It is possible to specify the characteristics of each partion-element using only information summarizing the uses and definitions within each region $\mathbf{r}_i$, rather than directly referencing all the individual components of $\mathbf{r}_i$: a particularly useful technique when analyzing regions which contain function or procedure invocations. Summary information for the region $\mathbf{r}_i$ is the set of uses and defs which are needed to describe $\mathbf{r}_i$'s interaction with other regions — exactly as discussed for the summary information about a `Call` tuple. An appropriate set can be found by initially including all uses and defs within the region, and then eliminating those which are not pertinent, as:

- A use which appears within a particular region should only be represented in the summary information for that region if it is a member of a D-U chain whose definition is not also within that region.

- At most, only one definition and one use per variable need be represented in a particular region's summary information.

Only information about *broken D-U chains* is needed.

Hence, if one wished to create a region which incorporated the entire body of the function `f` from Listing 5:1:

```
g = a + e;
return(g);
```

although this code contains a use of the variable `g`, the summary information would exclude that use. The def which is used in the use of `g` is a definition which also appears within this region. Because the summary information depends on knowledge of D-U and U-D chains, it is flow-dependent.[41]

To express constraints on execution order of regions, we define a partial ordering, denoted "<" on R as follows:

$\mathbf{r}_i < \mathbf{r}_j$

Iff $\mathbf{r}_i$ must be executed before $\mathbf{r}_j$

In addition, $\mathbf{r}_i < \mathbf{r}_j$ implies $\mathbf{r}_j > \mathbf{r}_i$.

2. The Sequential Rules

The rules for detecting potential parallel execution among program regions (r's) are straightforward and few in number. Of course, the amount of detectable parallelism is highly dependent on choice of the set of regions, created by partitioning, upon which concurrency detection is performed — but that is a separate issue, relating mostly to the parallelism available in the target machine.

2.1. D-U Chaining

41. Other research efforts (for example, [BuC86] [TrI86]) have proposed flow-independent forms of summary information — but such an approximation clearly yields inferior results. Extending flow-independent summary information to be flow-sensitive in the above sense is relatively simple.

Rule 1:

For any $\mathbf{r}_i$ and $\mathbf{r}_j$,

if there exists a *def* δ of the variable n such that

$\delta \in \mathbf{r}_i$ and $\mathbf{r}_j$ contains a *use* $u \in$ D-U chain of δ,

then $\mathbf{r}_i < \mathbf{r}_j$.

This rule is all that is needed if the mapping of values onto names is one-to-one. For concurrency detection within a basic block, because it is usually easy to optimize a basic block so that the value to name mapping is one-to-one, this rule is often sufficient. In functional or single-assignment languages, the same property is extended to encompass entire functions, hence concurrency detection at the function level is driven by this rule. Using the dataflow computation model, this rule governs concurrency detection throughout the entire program.

2.2. Killed Definitions

Rule 2a:

For any $\mathbf{r}_i$ and $\mathbf{r}_j$,

if there exists a *def* δ_i of the variable n such that $\delta_i \in \mathbf{r}_i$ and

there exists a *def* δ_j of the variable n such that $\delta_j \in \mathbf{r}_j$,

then either $\mathbf{r}_i < \mathbf{r}_j$ or $\mathbf{r}_j < \mathbf{r}_i$.

Rule 2b:

If *Rule 2a* applied and

there exists a *use* $\mathbf{u}_i$ such that

$\mathbf{u}_i \in$ D-U chain of δ_i and

$\mathbf{u}_i \in \mathbf{r}_i$, or,

there exists a *use* $\mathbf{u}_j$ such that

$\mathbf{u}_j \in$ D-U chain of δ_j and

$\mathbf{u}_j \in \mathbf{r}_j$, then if

$((\mathbf{u}_j \in D^* \text{ of } \delta_i) \text{ or } (\mathbf{u}_i \in U^* \text{ of } \delta_j))$,

then $\mathbf{r}_i < \mathbf{r}_j$.

This two-part rule is the key to concurrency detection across basic block boundaries in conventional languages.

Further, if only the first part of this rule applies and it provides the *only* constraint on parallel execution of the regions in question, parallel execution may be obtained by allocating two separate, co-existing, areas for the variable *n*. This is not always beneficial, however, because the new names, n_1 and n_2, will impose that any convergence of flow which causes the two defs to meet must be modified to create a copy of one name's value into the other, so that a single name can be used to refer to the value regardless of where it was computed. The cost of this copy-making may exceed the improvement using parallelism.

2.3. Synthesis

Detection of potential parallelism within sequential code amounts to computing the relation "<"; any two regions $\mathbf{r}_j$ and $\mathbf{r}_k$, for which neither $\mathbf{r}_j < \mathbf{r}_k$ nor $\mathbf{r}_k < \mathbf{r}_j$, can be executed in parallel.

If the relationship between $\mathbf{r}_j$ and $\mathbf{r}_k$ is permitted to be either $\mathbf{r}_j < \mathbf{r}_k$ or $\mathbf{r}_j > \mathbf{r}_k$ with no other constraint (as might be the case of rule 2a applied), the compiler may arbitrarily choose either execution order.

If either $\mathbf{r}_j < \mathbf{r}_k$ or $\mathbf{r}_k < \mathbf{r}_j$, executing the regions $\mathbf{r}_j$ and $\mathbf{r}_k$ without synchronization (without imposing a sequential order) could result in races.

3. The Parallel Rules

There are two separate reasons for concurrency detection to be applied to explicitly-parallel code: to obtain greater parallelism or to confirm that the existing parallel structure is race-free.

3.1. To Expose Parallelism

The rules for concurrency detection in sequential code can be applied to expose additional parallelism within explicitly-parallel code. Additional parallelism can often be found because explicitly-parallel languages tend to favor coarse-grained parallelism, which leaves large chunks of sequential code between explicitly-parallel operations. Parallelism within these chunks may be detected using the technique described above.

The only clarification necessary is that the partitioning of an explicitly-parallel program's graph into regions must insure that each region does not contain *internal* synchronization operations.[42] For purposes of

42. The exception is that it is permissible to synchronize with a child process whose lifetime is contained within that of the parent process. Essentially, this is because the child process can have its effects summarized much as we did for the call in Listing 5:2.

this analysis, *regions can synchronize only on their edges: creation and termination*.

3.2. To Detect Races

To determine the *safety* of parallel execution according to an explicitly-given structure, the same rules are applied. However, instead of finding a partial ordering, we are concerned with finding *violations* of the partial ordering.

Given a graph generated from an explicitly-parallel program, an additional possibility arises in the second part of the second sequential rule:

Rule 2c:

If *Rule 2a* applied, but the language specified that regions $\mathbf{r}_i$ and $\mathbf{r}_k$ were to be executed simultaneously (in which case the concepts of D* and U* used in *Rule 2b* are not readily applied), a potential race condition exists for the variable *n*.

Notice that an “explicitly-parallel” construct which has a well-defined sequential interpretation cannot cause this difficulty.

4. Summary

The discussion of concurrency detection given above is based entirely on D-U and U-D chaining; as such, we believe it is unique. The new concepts of D* and U*, briefly introduced here, are also fundamental to more complex program transformations, as in section 7:1.3.2. Further, we believe that the concept of parallelizing regions, rather than individual instructions or properly-nested subgraphs, is a significant contribution.

In this chapter, however, we have given very little direction as to how region groupings should be chosen: the choice of regions is highly machine dependent. Chapters 6, 7, and 8 provide insights into this machine dependent task.

This page is intentionally blank.

Computer Architecture

Throughout the current work, many references are made to the strong relationship between compiler optimizations and the structure of a target machine. In this chapter, we present brief descriptions of some of the more popular general classes of computer architectures. Each architecture is first defined, then its impact on the machine-dependent design of an optimizing/parallelizing compiler is discussed.

Experience in implementing parallelizing compilers has shown that parallelization is machine dependent in surprisingly drastic ways. Ideally, we wish to create a kind of "expert optimizer/parallelizer-generator" which would accept an architectural description (including implementation-dependent information such as instruction timing) and would generate an optimizer/parallelizer appropriate to that model. Failing this, the current work presents our basic insights as to what the key machine-dependent concerns might be.

The architectural features which demand the most complex machine-dependent compilation techniques are those relevant to machine parallelism. For example, instruction set variations like stack code; 1, 2, or 3-address code; accumulator, stack, register, or memory-based instruction sets; HLL-oriented code; and so forth have little impact on the compiler's basic structure because most instruction set variations are reasonably-efficiently coded *without* resorting to complex flow analysis and large-scale code transformation [Die83]. Even where such sophisticated measures are taken, it has been possible to solve many of the problems in relatively portable ways: [LeC79], [Gan80], and [Cho83] discuss the construction of portable code generators/optimizers.

This is not meant to imply that sequential code optimization is by any measure a solved problem. Rather, it is highlighting the fundamental difference between optimization and parallelization:

- In optimization, any little insight can modestly improve the code, but
- In parallelization, unless the insight encompasses at least a couple of regions in the code, no speedup can be obtained.

For example, suppose a compiler is examining a region of code which contains a reference through a pointer, p, which, at first glance, may be pointing at any of a number of variables including a and b. Further suppose there is another region of code which references both a and b. Simply determining thatp could not be pointing at a provides an opportunity for optimization (propagation of a's value across the pointer reference, placement of a in a register, etc.). However, unless the compiler could also prove that indirection through p does not reference b, the two regions could not be parallelized. This example used insights based on concurrency detection, but the same importance is associated with the highly machine-dependent insights involved in automatic parallelization (parallel machine ECFs).

Parallelization is an all-or-nothing-at-all game.

The parallel architectures which we describe are: SISD, SIMD (including Vector and Array), MIMD (including Dynamically-Scheduled Shared-Memory MIMD, Hypercubes and Non-Shared-Memory MIMD, VLIW, and Dataflow), and Pipelined and Systolic processors. This is not meant to be a comprehensive set. It is impossible to specify the precise impact of all architectures on compiler optimization/parallelization design, but the discussion loosely covers the majority of common machine structures.

It is important to note that we are not criticizing or trying to improve the architectures in the following sections, but merely describing how the best performance can be obtained for each type.

1. SISD

SISD means Single Instruction stream, Single Data stream. In other words, a SISD is a conventional, sequential, computer:

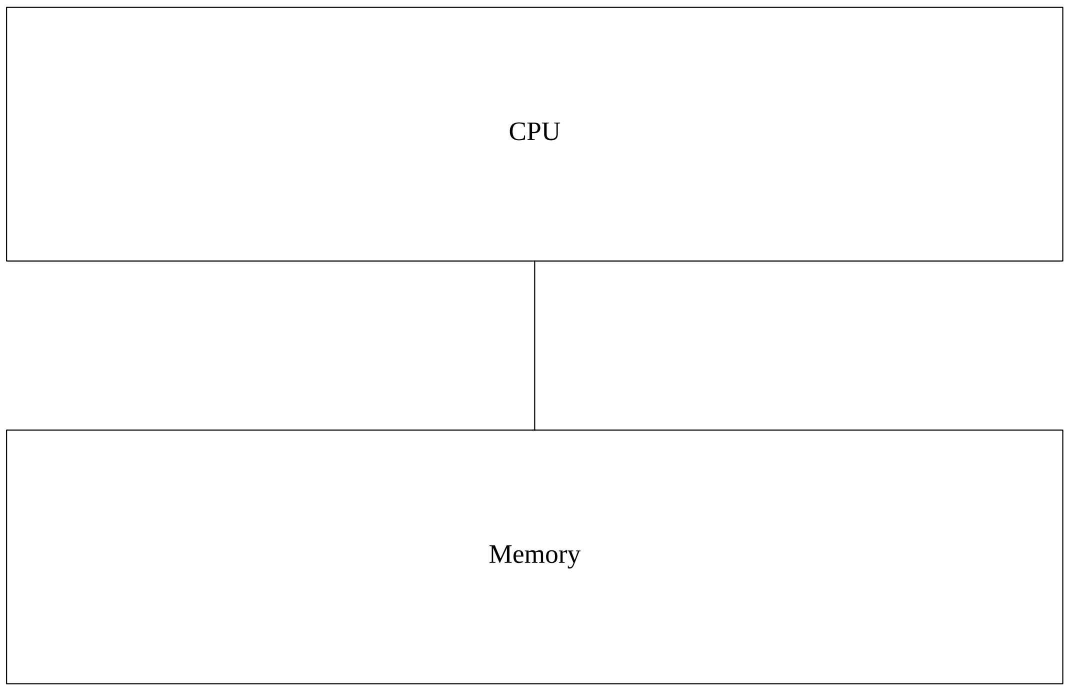


Figure 6:1: SISD Architecture

CPU stands for Central Processing Unit — in the diagrams of multiprocessors, there is more than one such unit, hence each is referred to as a PE, or Processing Element (although each PE might be identical to the hardware labeled CPU in the diagram above).

The line which connects the CPU and Memory has become commonly known as the “Von Neuman Bottleneck,” since all instructions and data must pass through it to be acted upon. This structure makes a number of optimizations particularly important:

- *Optimal register usage, value propagation.* Since registers (and caches) provide a mechanism for avoiding the bottleneck in referencing memory, a primary concern in generating code for a SISD is allocation of as much data as possible into registers (and placement of code to create as high a hit ratio as possible for caches). Value propagation permits the compiler to recognize when a value can be obtained from a register instead of fetched from memory.
- *Common subexpression and dead operation elimination, constant folding, code motions which reduce the expected execution frequency of instructions (such as removing code from loops).* These optimizations reduce the total number of operations which must be executed, hence, the program will execute faster.

There are certainly many other optimizations which can play a major role in generation of efficient SISD code, but most of them are easily managed without flow analysis. For example, the choice of addressing modes to be used within instructions is often important, but it is typically very machine-dependent, very localized in the code generator, and not of particular interest to us since it has little to do with the issues which are important in terms of sequential vs. parallel execution.

2. SIMD

SIMD means Single Instruction stream, Multiple Data stream. Such a machine can execute only a single instruction sequence, but the operations can be performed on many separate pieces of data simultaneously. A block diagram of a typical SIMD machine looks like:

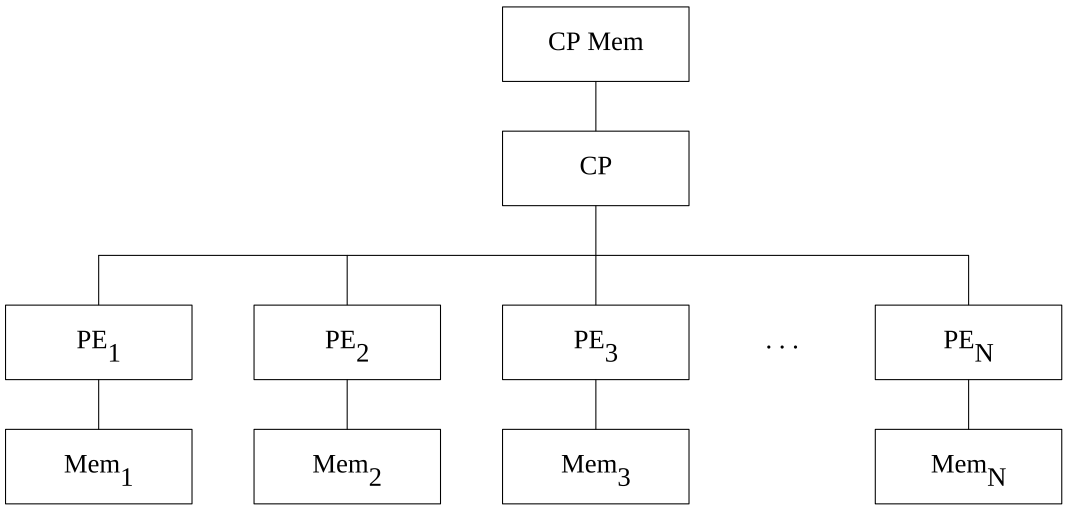


Figure 6:2: SIMD Architecture

In operation, the CP fetches and decodes instructions, one at a time, from its memory. As each instruction is decoded, it broadcasts the operation to the PEs. Simultaneously, every *enabled* PE executes that operation on data in its local memory or registers. PEs which are*disabled* for a sequence of instructions remain idle (ignore the CP’s commands), until they are enabled.

Because there is a single point of control for the entire system at the instruction level, synchronization overhead is very small; *SIMD machines can efficiently use instruction-level (fine grain) parallelism*.

Communication between the local memories of PEs is usually either supported by connections to a small number of PE neighbors or must be accomplished via the CP. If the PEs are interconnected, then motions of data according to the interconnection paths are reasonably efficient: for example, if PE_i is connected to $PE_{(i+1) \bmod N}$ for all $i \in [1,N]$, then a single instruction probably can cause *all* PEs to send an item to their neighbor to the right. Using shuffle/inverse-shuffle connections, any PE can communicate with any other in at most Log_N cycles and, since the shuffle duplicates the data flow of a reduction operation, parallel reductions can be efficiently performed. Using the path to the CP, information easily can be broadcast from one PE to all others.

2.1. Variations On SIMD Machines

Both Array and Vector processors are SIMD-type machines in that they can execute a single instruction on a set of data. The distinguishing features are:

2.1.1. Array Processor

This term has no one definition, but has been used to describe many different architectures which share the ability to perform operations on an array of data at a time. Traditionally, array processors are machines which follow the above definition of SIMD very closely. More recently, the term has been used to describe deeply pipelined machines which place an entire array of data into the pipeline, perform a sequence of operations within the pipeline, and output a stream of results. The second kind of array processor is best understood as a hybrid between SIMD and pipelined execution. Examples of SIMD array processors include the ILLIAC IV [Kuc68] and the Connection Machine [Thi86]. Most of the examples of the pipelined type are attached processors marketed by numerous companies including CSPI and FPS.

2.1.2. Vector Processor

Vector processors typically differ from the above SIMD model in that the PEs do not have local memories. Operations on an entire vector are performed by loading the data from CP memory into “vector registers” (the registers of each PE), operating on it there, and finally storing the vector back into the CP memory. This makes register allocation far more critical in achieving good performance using a vector processor than it is using SIMD-type machines in general. In addition, the cost of initiating parallel execution is higher (hence, slightly larger grain parallelism) due to the cost of loading a vector register from the CP memory. The best known examples of this are the various Cray machines and the Convex C1 [Con85a].

2.2. Optimization/Parallelization Concerns

In terms of code optimization/parallelization, there are just two key points: layout of data in separate memories and equivalence of multiple instruction streams.

2.2.1. Data Layout

Since each PE has only a local memory, sharing data across several PEs is difficult; therefore, the compiler must attempt to minimize the need to communicate computed values between PEs. Various skewing techniques have been developed toward this goal [HaJ86].

Since Vector machines do not have a separate local memory for each PE, skewing is less of a problem on Vector machines than it is for SIMDs in general. However, many Vector machines require that vector registers be loaded from contiguous memory locations, which demands nearly the same concern as having separate local memories.

2.2.2. Equivalence Of Instruction Streams

All PEs can perform the same operation simultaneously, hence optimization for a SIMD will hinge upon the ability to use *exactly the same sequence of instructions* for up to N "processes" occurring in parallel. While most earlier attempts to generate code for SIMD machines take this to mean that non-looping/non-vector code cannot be parallelized, such a drastic measure isn't necessary.

A conceptually simple (but difficult to implement) example is that the two assignments in:

```
A = B * C;
D = E * F;
```

can be executed in parallel, provided the appropriate data, or pointers to them, can be loaded into places addressed in the same way within two different PE's local memories. For example, if the values of `B`, `C`, and `A` are in registers 3, 4, and 5 of PE 2 and the values of `E`, `F`, and `D` are in the corresponding registers of PE 7, it is possible to compute both `A` and `D` in a single "vector" operation.

Any regions of a program which concurrency-detection finds to be parallelizable can be executed with some parallelism on a SIMD provided that some portions of them can be coerced into identical form.

This also slightly re-arranges the preferred *order of application of optimizations* within the compiler: common subexpressions shouldn't be eliminated, etc. unless doing so will have no ill effect on symmetry of the program's code. (Normally, such optimizations are applied very early — better SIMD code generally results with late application of SISD-style optimizations.)

Consider:

```
FOR I:=1 TO N DO
    IF A[I] > B[I] THEN
        C[I] = (A[I] * D[I]) + (A[I] * E[I])
    ELSE
        C[I] = B[I] * (D[I] + E[I]);
```

for which a typical vectorizing compiler would generate code approximating:

```
TEST[1:N] = (A[1:N] > B[1:N]);
WHERE TEST[*] DO
    C[1:N] = (A[1:N] * D[1:N]) + (A[1:N] * E[1:N]);
WHERE NOT TEST[*] DO
    C[1:N] = B[1:N] * (D[1:N] + E[1:N]);
```

This parallelizes evaluation of the condition, but takes about twice as long as necessary for the rest of the code. Compare the above to:

```
TEST[1:N] = (A[1:N] > B[1:N]);
WHERE TEST[*] DO
    TMP[1:N] = A[1:N];
WHERE NOT TEST[*] DO
    TMP[1:N] = B[1:N];
C[1:N] = TMP[1:N] * (D[1:N] + E[1:N]);
```

In both cases, the two `WHERE` statements must be executed in sequence on a SIMD. The first version will execute in the time it takes to sequentially execute:

```
TEST = (A > B);
C = (A * D) + (A * E);
C = B * (D + E);
```

whereas the second version merely requires the time to sequentially execute:

```
TEST = (A > B);
TMP = A;
TMP = B;
C = TMP * (D + E);
```

The better vectorization above did not apply any previously unknown transformations; it merely applied the distributive law of multiplication over addition[43]. The transformation is unusual only in that the compiler deliberately created *symmetry in the regions* for both halves of the `IF` statement.

Although this is an obviously beneficial parallelization technique for SIMDs, we have seen no compilers employing it, and we believe this to be the first mention of it as a general principle.

3. MIMD

MIMD stands for Multiple Instruction stream, Multiple Data stream; a computer system which is most like a huge collection of SISD machines. The generic diagram is:

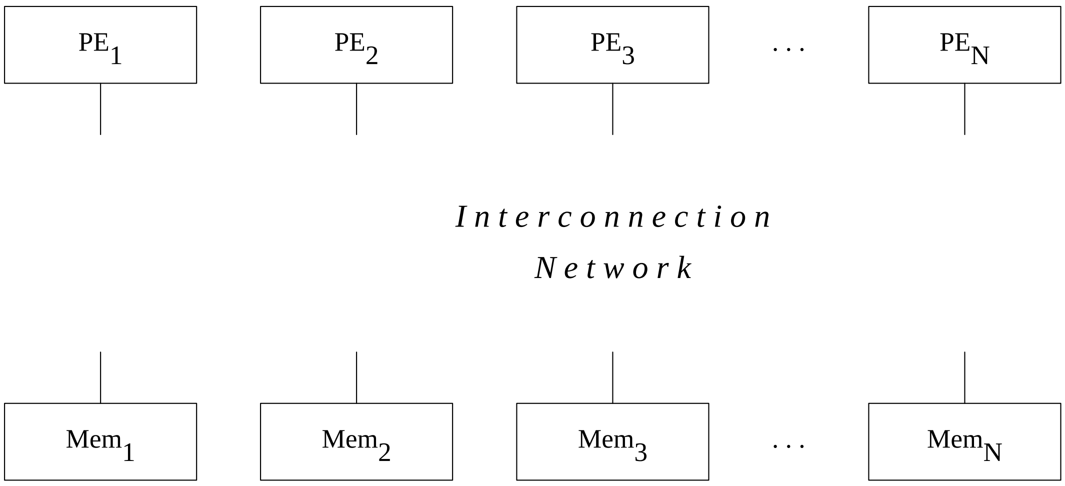


Figure 6:3: MIMD Architecture

In operation, a MIMD closely resembles a collection of SISDs. Each PE fetches instructions and executes them very much as it would if it were the CPU of a SISD. Here, however, the different MIMD-like architectures diverge.

43. As noted in section 7:1.2.2, this rule is only approximately true for finite-precision arithmetic. Here, we assume that this difference is acceptably small.

3.1. Variations On MIMD Machines

Firstly, there are several different organizations of memory: local addressability, global addressability, and local/global addressability. Then there are variations on synchronization technique, ranging from the fine-grained static synchronization of a VLIW [Fis84] to the coarse-grained dynamic parallelism of machines like CHoPP [SuB77], Utracomputer [Sch80] [GoG83] [Got86], and IBM's RP3 [PfB85] [Nor86]. The interconnection topology and technology also vary widely, with very fast shared single-bus systems, low-bandwidth hypercubes, and "smart" medium-speed single or multistage networks.

Rather than defining all the possible combinations of these design features, the current work is restricted to describing the most important aspects of the best known few designs. These are Dynamically-Scheduled Shared-Memory MIMD, Hypercube (Non-Shared-Memory MIMD), VLIW, and Dataflow.

3.1.1. Dynamically-Scheduled Shared-Memory MIMD

For most people, unless specified otherwise, this is the definition of a MIMD: a machine where many PEs share access to a common memory address space but are otherwise completely independent. Each PE operates asynchronously (at the instruction level) of the others — if synchronization is necessary, it may be obtained using any of several different mechanisms. Typically, synchronization is provided by software or hardware semaphores or by process scheduling.

In small machines, with between 2 and 16 PEs, the most common way of connecting processors and memory is to use a single, very fast, shared bus. In order for performance to be reasonable, most memory references *must never reach the bus*; this is typically accomplished by use of "snooping caches" — cache memories associated with each PE which monitor bus activity and update cache entries when needed so that cache consistency is maintained. Some commercially available examples are the Sequent Balance 8000 [Seq84], Areté Series 1000 [Are85], ElXSi System 6400 [ElX85], and the Alliant FX/8 [AbM86]; examples developed in academic settings include MIPS-X [HoC85] [Cho86], SPUR [HiE86], and Cedar [GaL83] [KuD86]. (Some of these machines are also pipelined and/or incorporate vector execution units.)

Larger machines, especially those designed to permit use of hundreds or thousands of PEs, cannot tolerate the delays encountered in using a small number of shared busses and would require too many wires to interconnect fully point-to-point. Most of these larger machines therefore are based on a Log_2 N-stage packet-switched interconnection network which employs "smart" switching nodes. In the CHoPP [SuB77], and in the single-stage network RFM-MIMD [PaK86], these nodes incorporate a special cache mechanism which also manages pre-arrival entries [Kla80] (thus improving response when multiple PEs request a particular memory location either simultaneously, or in sequence before that cache entry is flushed).

In the Ultracomputer [Sch80] [GoG83] [Got86], and RP3 [PfB85] networks, the nodes instead incorporate **fetch-and-add** [Sto84] facilities. Fetch-and-add permits concurrent additions to the same memory cell (as used in counting semaphore operations) to be combined so that a single addition is propagated to memory, hence avoiding contention for that cell [PfN85]. As such, it is particularly useful for parallelizations which require frequent N-way synchronization (SIMD-style parallelization). The more general form of the concept, **fetch-and-op**, may be used with any associative operation to implement parallelized associative reduction operations (see section 7:1.2.2) throughout the network without the need to dimensionally promote variables.

Since the packet-switched interconnection networks all have traversal times of at least a few PE instruction cycles, if most packets actually do traverse the network, each process will execute slower by this factor. One way in which this problem can be minimized is to carefully optimize network caching of each datum.[44] Another way is to incorporate local memory with each processor. In RP3 [PfB85], ELI [Fis84], and [PaK86], local and global memory are one in the same — a reference is local if it happens to be in the requesting PE's memory bank, but memory is globally accessible via the network. This places great importance on the ability to distinguish local and global references and to perform good "register allocation" of the local memory.

3.1.2. Hypercube (Non-Shared-Memory MIMD)

Hypercube-interconnected non-shared-memory machines include the IPSC [Int85], N-Cube [NCu85], and the Ametek [Ame86]. These machines place an asynchronously-operating module containing a PE and a local memory at each vertex of a hypercube. Each hypercube machine also has a link to a conventional machine which acts as its host.

A PE can communicate directly only with its neighbors on the hypercube, which is sufficient for some problems. However, communication through nodes is relatively expensive and this fact imposes severe penalties for picking a suboptimal mapping of processes to PEs or of variables to local memories. How a compiler can solve the layout problem seems to be a research question for which few answers have

44. We have taken this approach in the single-stage machine [PaK86]. Each memory reference is explicitly marked (by the compiler) with a field which determines how and where within the network the item may be cached.

been proposed. Until this problem is answered, the primary concern in automatic parallelization for hypercubes will be finding parallel processes which have very little need for communication.

Another non-shared memory machine is PASM [SiS81] [SiS86], which is interconnected using an extra-stage indirect cube, and has the unusual ability to operate as any collection of smaller SIMD and MIMD computers. The difficulties encountered in utilizing PASM are similar to those found for the hypercube machines, however: (1) the interconnection network makes all processors roughly equidistant for communication, which simplifies the layout task, and (2) because both SIMD and MIMD modes are available, the control of hardware configuration at both compile- and run-time becomes a major concern.

The Flex/32 [Fle85] probably is also best described as a non-shared memory MIMD. Although this machine supports both local and global memories, the global (shared) memory space is quite small, and it is not practicable to ignore the local memories. Global memory would most likely be used for synchronization and communication only.

3.1.3. VLIW

A VLIW, or Very Long Instruction Word, machine is a MIMD in which all processors share a single program counter, but each instruction contains a separate field for every processor[45]. Hence, a VLIW shares the SIMD property of nearly zero-cost synchronization, but, unlike the SIMD, does not need to find or create identical code sequences for all code to be run in parallel. In other words, a VLIW is a statically-scheduled MIMD — all run-time synchronization/communication is completely specified at compile time, all timing relationships throughout the machine are known at compile time.

Programming of an entire VLIW machine is nearly the same problem as microcode programming for conventional machines: many inter-related events may occur in parallel within one instruction, but the synchronization of “processes” on various PEs is accomplished simply by clever compile-time packing of instructions, not by run-time arbitration and scheduling mechanisms.

Although compile time will increase with the irregularities of the VLIW, the PEs of a VLIW will often be of several different types and the interconnection scheme may take almost any form. Some PEs will generally be Floating-Point Units (FPUs), others will be conventional integer-only RISC machines. Compilation is very slow, resulting programs are very large, and the total usable parallelism of a VLIW machine is relatively narrow, but the very fine granularity makes VLIWs consistently able to use between 4

45. From a purely architectural point of view, it is actually a microcoded SISD with a writable control store — but that is also kind of MIMD from the compiler point of view.

and 20 parallelism width — very few sequential HLL programs exhibit less than width 4 at this fine granularity.

Good examples are the ELI and Multiflow computers [RuF83] [Fis84], [ToS86], and [GuF86].

3.1.4. Dataflow

Dataflow machines execute dataflow graphs. Since dataflow graphs are easily parallelized, it has been claimed that they are naturally parallel and the obvious hardware implementation of dataflow is said to be an excellent parallel architecture [Mis75] [AgA82] [WaG82] [Ian82] [DeG84].

Although the diagram usually drawn for a dataflow machine's hardware looks quite different from the MIMD diagram because the MIMD network is bi-directional whereas a dataflow machine is drawn with two unidirectional networks, they are actually strikingly similar [KlL85]. The main difference is that whereas a classical MIMD provides registers and other mechanisms for taking advantage of locality-of-reference, dataflow generally does not. The result is that dataflow machines are usually very fine grain and very dynamic — there is very little potential for static (compile-time) optimization.

Although the standard diagram of a dataflow machine is apparently different from the MIMD in Figure 6:3, it is different only in that it is usually drawn with two unidirectional interconnection networks rather than with one bidirectional network. In execution, each node in a dataflow graph is placed in the *matching store* (memory modules). When a value is computed by any of the PEs, all dataflow nodes awaiting that datum are "matched" with that value. Any nodes which need no more data (all needed inputs have arrived, all graph predecessors have fired) are *fired* by selecting a PE and sending the computation to it.

Optimization for dataflow hinges on depth minimization of the dataflow graph. This includes operations like constant folding, (usually) common subexpression elimination, and several kinds of code motions. Reviews of dataflow optimizations appear in [DaK82] and [BrM]. Because there are no registers, nor memory, most conventional optimizations do not apply.

In some dataflow models, however, large grain "chunks" of conventional control-flow computations are used, in which case significant advantage can be made of locality properties [Bab84].

3.2. Optimization/Parallelization Concerns

On dynamically-scheduled machines, although data sharing is not particularly expensive, creation and scheduling of lots of tiny processes is very expensive. Too few processes leaves PEs idle, too many implies each process is too small and by the time it has been assigned to a PE we could have executed it in sequence. The optimum is therefore to construct a large number of parallel processes, each of which has at least a certain expected execution time.

Where the MIMD has a local memory, optimization will be highly dependent on making good choices about what to have in local memory — which, for optimization purposes, is very much like managing a huge set of PE registers. In addition, there is the problem of avoiding “hot spots” in the layout of data structures [LeK86].

4. Systolic Arrays And Pipelines

A Pipelined machine is a machine where several operations may be in progress simultaneously, but only one operation may be *initiated* in each cycle. A very deep pipelined machine, or one which is multidimensional, is commonly known as a Systolic Array[46]. The generic diagram is:

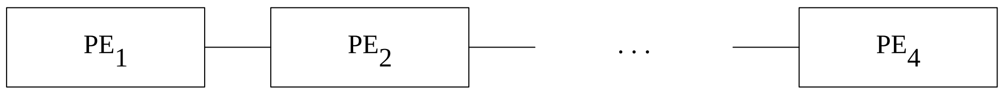


Figure 6:4: Pipeline/Systolic Architecture

Since the 1960’s, most mainframe computers have employed some pipelining of operations within their CPUs: optimization for these is discussed in [CoS70].

Recently, however, the concept of RISC (Reduced Instruction Set Computers) has yielded many more, and deeper, pipelines. For example, RISC-I, RISC-II, SPUR, MIPS, and MIPS-X all rely on compiler techniques to fill pipelines of one or two instructions depth [Gro83]. In GaAs implementations, RISC processor pipelines are often six instructions deep [MiF85].

Systolic Array computers with pipeline depths of ten or more have also been proposed, particularly for specialized applications. Warp [AnA86] is the prime example of a somewhat general-purpose systolic array. In [GrL86], techniques for converting programs written in a special HLL into code for Warp are given, but no discussion of techniques for using systolic arrays as a target of automatic parallelization has

46. Strictly speaking, some systolic arrays may also function as SIMD computers, but the issues for SIMD computation are discussed in section 6:2. The Connection Machine [Thi86] may be viewed as a systolic array using the SIMD, rather than pipeline, model.

yet appeared in the literature.

In either the pipelined or systolic models, the primary concern is simply finding enough instructions to keep the pipeline full. Previous work has failed to keep even four instruction deep pipelines filled [Gro83], yet, as mentioned above, pipelines are often six or more instructions deep. There are two interesting possible solutions:

(1) Perform relatively large grain concurrency detection on each program, then fill the pipeline by interleaving instructions from the “potentially concurrent” regions.

(2) Use the code motions of trace scheduling [Fis84] to fill the pipeline. This has the negative effect of filling the pipeline with some instructions which would not have been executed in the sequential version of the program — instructions which were hoisted from a conditionally-executed basic block which, upon evaluating the condition, would not have to be executed.

either or both should be applied. Currently, compilers attempt to fill pipeline delay slots only with instructions which are from the same conceptual process and which are certain to be executed.

This page is intentionally blank.

Loop Parallelization

In the search for parallelism within a sequential program, the most common sources of operations for large-width parallel execution are looping constructs. Most code has an irregular structure with relatively small-width (often fine-grain) potential parallelism; most iterative or recursive code has the potential for parallelism-width proportional to the *problem size*. In addition, the code resulting from parallelization of a loop's contents, or of recursive calls, usually has a regular structure which makes it possible to:

- Reorganize or repackage the parallelism without actually analyzing each individual potentially-parallel chunk of code — only a single representative chunk need be analyzed.
- Share a single, parameterized, chunk of code for all processes created (this, of course, only works for certain machines, particularly shared-memory dynamically-scheduled MIMDs).

In most languages, loops may be constructed using `WHILE`, `REPEAT`, `FOR` (or `DO`), or `GOTO` — but, as indicated in section 3:3.1, the keywords used need not have any effect on the compiler's perception of the algorithmic structure.

In this chapter, guidelines for loop parallelization, based on flow analysis of algorithmic structure, are presented[47]. Most of these rules have been published by other authors, but there are several refinements which we believe to be stated here for the first time. Our primary goal is simply to state the rules in a way that is consistent with the refined-language approach as it is presented in this document.

1. Types Of Loops

There are many different kinds of loops. All loops, however, have control and a body. If a loop:

(1) has a body which contains no definition points which may be used by the loop's body in a later iteration (no D-U chains linking iterations),

(2) has a body which contains no definition points, or uses of other definitions, which may be killed by the loop's body in a later iteration (no D-U chain in one iteration that uses the same storage cell as a D-U chain in another iteration), and

47. This chapter will not discuss loop unwinding/unraveling, since such techniques reduce loops to structures which are best parallelized by the techniques given in Chapter 8. However, such transformations are important, especially for machines like VLIWs [Fis84].

(3) has loop control which either enumerates the set of iterations or provides a simple *loop invariant* formula to correctly determine whether the N^{th} iteration of the loop will occur and what its parameters will be (note that N may be larger than the number of iterations the loop will actually make)

then the loop is completely parallelizable. All "iterations" of the body could be executed in parallel, or in any sequential/parallel organization.

For example:

```
        DO 10 I=1,J
            A(I) = A(I) * B(I)
 10     CONTINUE
```

If it is known that A and B are independent (non-overlapping) arrays, this loop can be completely parallelized: `A(1) = A(1) * B(1)` may be executed before, after, or in parallel with `A(2) = A(2) * B(2)`, `A(3) = A(3) * B(3)`, and so forth. Since `I` can never equal `I` + (N * 1),[48] the first two rules appear to be satisfied.[49] It can be determined whether the N^{th} iteration will or will not occur by simply testing if $N \leq$ `J`, and the only parameter to the N^{th} execution of the body is N. Therefore, this loop is parallelizable.

1.1. D-U Coupling Of Iterations

Suppose that a loop fulfills the second and third conditions stated above, but violates the first: the body contains a definition point which is used in a later iteration. A good example is:

```
        DO 10 I=2,J
            A(I) = A(I-1) * B(I)
 10     CONTINUE
```

Where the N^{th} iteration uses a definition point occurring in a previous iteration, in this case, the N-1^{th}. `A(`N-1`)` uses the previous iteration's definition of `A(`N`)`.

While it is not generally possible to completely parallelize such a loop, it may be "pipelined" to some extent. The value of `B(`N`)` is available before the N-1^{th} iteration's execution has completed, hence, `B(`N`)` may be fetched by the N^{th} iteration's code in parallel with the N-1^{th} iteration's execution.

48. The formula I + (N * 1) represents the value of I after N iterations of a loop with increment 1. The only way I could equal I + (N * 1) is if N is zero, in which case it is the same iteration.

49. Provision must be made for having multiple simultaneous values associated with the name I, since the D-U chains for I violate the second rule. This is discussed in the section 7:1.2.2.

The maximum speedup which can be obtained in this way is determined by:

- The amount of code within the loop body which can be executed before the first use of a previous iteration's definition. (As is the case in the example.)
- The number of iterations between definition and use; for example, if each iteration used a definition from ten iterations previous, then each set of ten iterations can be executed in parallel.

1.2. Killed Definitions

If only the second condition stated earlier is violated, the loop can be directly "pipelined" in much the same way discussed in section 7:1.1. However, such a loop can be completely parallelized by introducing a new "variable" in the code. This occurs in three different ways: re-binding of a programmer-given name, dimensional promotion of a name, and associative reduction of a name.

1.2.1. Re-Binding Names

Consider:

```
       DO 10 I=1,J
           A(I) = A(I+1) * B(I)
10     CONTINUE
```

At first glance, this seems to be the same as the example where the first rule was violated, but it differs in that the *N+1*th iteration re-defines an element whose *definition prior to entering the loop* is used in an earlier iteration: in this case, the *N*th iteration.

A definition is killed by having its variable's value re-defined: to eliminate the conflict, all that need be done is that two separate variables must be bound to the different definitions. Rewriting the code in this way eliminates the conflict:

```
       DO 10 I=1,J
           NEWA(I) = A(I+1) * B(I)
10     CONTINUE
```

Leaving the array NEWA as the variable to be used in later definitions which refer to the programmer-given name A.

If there is a flow path (sequence of D-U chains) by which a definition of A bound to the array A and a definition of A bound to the array NEWA both *reach* the same point, then a copy is inserted to place the result back in A, giving:

```
         DO 10 I=1,J
             NEWA(I) = A(I+1) * B(I)
 10      CONTINUE
         DO 20 I=1,J
             A(I) = NEWA(I)
 20      CONTINUE
```

which constitutes two completely parallelizable loops. The two different bindings may reach the same point either by:

- Defining only a portion of the data structure within the loop, in which case any *dominating* [AhU77] [AhS86] definition of the data structure's other members is still in effect, or
- Arriving at a point of *convergence of flow* where different bindings were applied to the same name on the different converging paths.

Notice that the copy loop can be hidden if another loop which defines the same variable, over the same range of index values as in the original loop, follows (before the convergence of flow, if that is the cause for the copy operation). One would simply define A from computations on NEWA. For example, if given the code:

```
         DO 10 I=1,J
             A(I) = A(I+1) * B(I)
 10      CONTINUE
         DO 20 I=1,J
             A(I) = A(I) + C(I)
 20      CONTINUE
         D = A(K)
```

Parallelization can proceed as though the following code were input:

```
C
C       Here, a programmed reference
C       to A is bound to the array A
C
        DO 10 I=1,J
            NEWA(I) = A(I+1) * B(I)
 10     CONTINUE
C
C       Here, a programmed reference
C       to A is bound to the array NEWA
C
        DO 20 I=1,J
            A(I) = NEWA(I) + C(I)
 20     CONTINUE
C
C       Here, a programmed reference
C       to A is bound to the array A
C
        D = A(K)
```

A similar transformation can be applied toward elimination of some copy operations induced by functional-language notation [Den86]. The problem in that case is somewhat compounded, however, by the weak typing and inconsistent use of array constructors commonly found in such languages.

1.2.2. Dimensional Promotion

In the example above, the definition being killed was one which existed prior to entering the loop. Instead, the definition being killed could be created and used within each iteration of the body of the loop.

For each of the examples given above, the variable `I` is defined and used in this way. In order to completely parallelize the execution of one of these loops, it is implicit that there would be a vector of bindings for the programmer-given name `I`: one for each concurrently existing definition. Going back to the original example:

```
        DO 10 I=1,J
            A(I) = A(I) * B(I)
 10     CONTINUE
```

Becomes the parallelized execution of up to `J`[50] assignments:

```
A(I_1) = A(I_1) * B(I_1)
A(I_2) = A(I_2) * B(I_2)
A(I_3) = A(I_3) * B(I_3)
 . . .
A(I_J) = A(I_J) * B(I_J)
```

Plus one final assignment, if the last value of I defined in the sequential program was used outside the loop:

```
I = I_J
```

In general, if a variable is used within a loop such that it is defined and used separately for each iteration, to obtain complete parallelization, the data structure bound to that variable must be promoted to hold one more dimension's worth of data. A scalar becomes an array; a *D*-dimensional array becomes a *D+1*-dimensional array.

1.2.3. Associative Reduction

Another kind of loop, which violates the second rule in nearly the same way, is the kind which appears to be performing a completely sequential operation. For example, the following code computes the sum of the points in a 4-point[51] vector:

```
      SUM = 0
      DO 10 I=1,4
             SUM = SUM + A(I)
 10   CONTINUE
```

In fact, this code is "inherently sequential," and cannot be parallelized (other than by "pipelining," as discussed earlier) without undergoing an algorithm change (however minor).

The specified order of the operations directly involving `sum` is:

50. It is necessary only to make the array large enough to hold *as many data as the machine can operate on in parallel* — the variable I didn't necessarily have to become a vector of length J. This problem is discussed further in section 7:2.

51. There is no special significance to the number 4 appearing here; neither is the fact that it is a constant particularly important. It was chosen merely because it simplifies the explanation of the code transformation.

```
SUM = 0
SUM = SUM + A(1)
SUM = SUM + A(2)
SUM = SUM + A(3)
SUM = SUM + A(4)
```

The algorithm change is simply to perform the additions as a tree-structured reduction operation — the usual “divide and conquer” approach. Using the divide and conquer approach, the ordering of necessary operations is:

$\texttt{PARALLEL\{ \{TEMP}_1 \texttt{ = A(1) + A(2)\}, \{TEMP}_2 \texttt{ = A(3) + A(4)\} \}}$
$\texttt{SUM = TEMP}_1 \texttt{ + TEMP}_2$

This different algorithm is well-suited to many parallel machines since it minimizes the sequential depth of the task, however, the original version is typically superior for a strictly sequential target machine because it minimizes the number of intermediate values which must exist simultaneously.

The problem with the new algorithm is that, unlike ordinary arithmetic addition, finite-precision floating-point (or integer) arithmetic addition is not an associative operation. Changing the order of the additions can change the value computed: by round-off in the case of floating-point and by overflow/underflow conditions for both integer and floating-point. The algorithm change, which assumes that the computer’s addition operation is associative, does not insure equivalence of the program before and after the change. Strictly speaking, correctness is*not* preserv ed.

Usually, this problem is not a key concern, since overflow/underflow situations are rare and the difference in round-off error is usually small enough to ignore. For some computer operations, associativity holds, hence correctness would be preserved. The following table outlines the safety of performing associative algorithm changes such as converting the sequential summation into its “divide and conquer” equivalent.

Safety Of Associative Transformation

Operation	Integer	Floating-Point
Or	Safe	Not Defined
And	Safe	Not Defined
Exclusive Or	Safe	Not Defined
Addition	Unsafe: Over/Under	Unsafe: Over/Under, Rounding
Multiplication	Unsafe: Over/Under	Unsafe: Over/Under, Rounding
Minimum	Safe	Safe
Maximum	Safe	Safe

Many optimizing compilers allow the user to select, at compile time, whether the compiler should perform transformations which could change the results computed.

More complex algorithm changes could also be incorporated, but each special case applies to a very small fraction of loops. For example, it is possible to perform associative reduction of exponentiation operations, but the opportunity to apply such a transformation rarely occurs.

1.3. Loop Control Dependency

Given that the first two rules' conditions are met, in order for completely parallel execution, it must be possible for a compiler to determine whether the N^{th} iteration of the loop will occur and what its parameters will be.

In all the example loops given above, the loop conditions were able to be parallelized because of the definition of a `DO` loop; given:

`DO` *label variable*=*first*, *last*, *inc*

FORTRAN permits the compiler to determine whether the N^{th} iteration will occur by a loop invariant test formula (which we call the **iteration decider**) and also provides a loop invariant formula for the loop-body's parameter[52]:

52. Here, we make the simplifying assumption that *inc* is positive. This need not be the case in ANSI FORTRAN '77.

```
IF (N*inc+first .LE. last) DO_BODY(N*inc+first)
```

where `DO_BODY()` executes the loop body for the iteration where the iteration control variable had the value given as the argument. If no such formulas can be found by the compiler, only “pipelining,” not completely parallel execution, is possible. This difference can be effected by a surprisingly small change in the source code. For example, consider:

```
for(i=0; i!=j; ++i) a[i] *= b[i];
```

Which is a C version of the example at the start of section 7:1 — however, there is a problem in this coding.

The obvious transformation of the loop test condition results in an iteration decider which would mistakenly predict that iteration `j + `*K*, where *K* is any counting number, is also part of the loop — to obtain correct results, the loop test in the `for` must be converted from `i != j` to `i < j`. This particular conversion can be performed by a compiler, and embodies the knowledge that repetition of `++i` results in a sequence of monotonically increasing values for `i`.

Not only must the iteration decider be loop invariant, but it must give the proper response for input values of *N* which are *beyond* the loop bounds. When code is parallelized to be executed as *P* processes, it is possible that an iteration number as large as the upper loop bound plus *P-1* will be tested to see if the loop body would actually be executed for that iteration. Consider the C code above being executed on a four-processor MIMD where the value of `j` happens to be 5:

processor 0	*processor 1*	*processor2*	*processor 3*
iteration 0 `i` is 0 `i!=j` true	iteration 1 `i` is 1 `i!=j` true	iteration 2 `i` is 2 `i!=j` true	iteration 3 `i` is 3 `i!=j` true
iteration 4 `i` is 4 `i!=j` true	iteration 5 `i` is 5 `i!=j` false	iteration 6 `i` is 6 `i!=j` true	iteration 7 `i` is 7 `i!=j` true

Figure 7:1: An Incorrect Iteration Decider

here we see that the loop body would incorrectly be executed for values of `I` equal to 6 and 7. In this case, the problem can be corrected simply by converting `!=` into `<`.

The following C code demonstrates a more general kind of violation of the third loop parallelization rule — a loop whose control test expression is not properly defined for potential iterations beyond the end of the loop:

```
for (i=0; a[i]!=j; ++i) ++b[i];
```

There are two approaches to parallelizing this loop. One is pipelining (much as before), the other is what we call **control precomputation**.

1.3.1. Pipelining

The “pipelining” which results would permit concurrent evaluation of `a[i] != j` and addition of one to the value of `b[i]`, but the result cannot be stored into `b[i]` until it has been confirmed that `a[`*K*`]` does not equal `j` for all *K* such that `0` $\le$ *K* $\le$ `i`. If `a[`*K*`]` does equal `j` for some *K* $\le$ `i`, the addition of one to `b[i]` would be wasted effort. This kind of “Pipelining” simply permits operations, except stores, to be executed with some parallelism.

1.3.2. Control Precomputation

Control precomputation is a new[53] kind of loop transformation which attempts to parallelize a portion of a loop body by selectively, deliberately, serializing some operations within the loop. In particular, operations which are involved in computing the expressions which determine when the loop is exited are placed in a separate loop called the **preloop**. The other operations are placed in a parallelizable loop called the **postloop**, and the preloop followed by the postloop are both placed inside the **closure loop**.

The overall effect is to eliminate synchronization overhead by placing the synchronization entirely within a single process — the one which executes the preloop.

For example, the C code:

53. In discussing this with other researchers, we have heard that similar ideas had been proposed before, but we have been unable to locate prior work on the topic. A detailed discussion of control precomputation, and related loop transformations, will appear in [Die87].

```
for (i=0; a[i]!=j; ++i) {
  b[i] = f(i);
}
```

Listing 7:1: C Loop With Control Dependence

where `f(i)` performs a relatively expensive computation with no significant side-effects, becomes something like:

```
int exit, iter, itemp[MAXWIDTH], piter;

i = 0;
/* Closure loop */
exit = FALSE;
do {

  /* Preloop */
  iter = 0;
  do {
    if (!(a[i]!=j)) {
      exit = TRUE;
      break;
    }
    itemp[iter] = i;
    ++i;
  } while (++iter < MAXWIDTH);

  /* Postloop */
  for (piter=0; piter<iter; ++piter) {
    b[ itemp[piter] ] = f( itemp[piter] );
  }

} while (!exit);
```

Listing 7:2: Listing 7:1 Using Control Precomputation

which, although the preloop is still sequentially constrained, permits the postloop to be executed completely in parallel. Although the complete transformation is quite complex and incorporates corrections for multiple-entry multiple-exit loops and conditionals within the loop body, the underlying principles are simple:

(1) Let *P* be the set of all uses which are in the exit condition expression of the loop. For each operation *o* within the loop, if there exists a use $u \in P$ such that $o \in U^*(u)$[54], then *o* is placed in the preloop; otherwise, *o* is placed in the postloop.

54. The use closure of use *u*. See section 5:1.3.

(2) For each def $\delta \in$ preloop, if D*(δ) is not a subset of the preloop (i.e., a def in the preloop is used in the postloop), allocate a vector of temporary storage to buffer the value of this def between the preloop and the postloop. Adjust the references accordingly in the preloop and postloop. In Listing 7:2, `itemp` is created because `i` must be defined in the preloop, but its value is used in the postloop as well.

(3) Construct the final control structure. The closure loop simply permits buffer vectors to be of finite size, in the above example, of size `MAXWIDTH`. Typically, this size would be related to the parallelism width of the hardware.

The main reason that this new transformation is mentioned here is that in analysis of conventional C code, we have found that a very large fraction of all loops in the code are of the types which most other researchers typically classify as “inherently sequential” — unable to be parallelized by any means — yet many of them can be effectively parallelized using the control precomputation transformation (or other related transformations). The following kinds of loops are very common in C code and can all be parallelized in this way:

- a loop which reads and processes input from a single stream (file) until some condition occurs in the stream,
- a loop which traverses a linked list operating on nodes as they are examined, and
- a loop which is essentially a FORTRAN `DO` loop with an alternative premature exit condition.

For example, the loop:

```
while ((c = getchar()) != EOF) {
    checksum += f(c);
}
```

Listing 7:3: An “Inherently Sequential” C Loop

where `f(c)` performs a relatively expensive computation with no significant side-effects, becomes something like:

```
int exit, iter, ctemp[MAXWIDTH], piter;

/* Closure loop */
exit = FALSE;
do {

  /* Preloop */
  iter = 0;
  do {
    if (!((c = getchar()) != EOF)) {
      exit = TRUE;
      break;
    }
    ctemp[iter] = c;
  } while (++iter < MAXWIDTH);

  /* Postloop */
  for (piter=0; piter<iter; ++piter) {
    checksum += f( ctemp[piter] );
  }

} while (!exit);
```

Listing 7:4: Listing 7:3 Using Control Precomputation

which is far better than pipelining its execution, because this reduces the amount of synchronization needed for execution on most machines by a factor of `MAXWIDTH`. It has this effect because it places the synchronization-intensive portion of the loop in a single process/processor. In addition, it is unclear whether it is physically possible for reads from a single file to be spread across multiple processors as pipelining would generally imply.

Notice, however, that parallelization of the above postloop also requires that the postloop be processed as an associative reduction.

1.4. Hybrid/Nested Loops

If a loop contains other loops, these loops may be interchanged [Wol86] so as to maximize useful parallelism. Nesting of loops may be changed only if no dependencies prevent the interchange [Kuc78] [PaK80] [Li85] [MiP86] [PoK86] [AlB86].

If a loop contains code which evidences different kinds of parallelism constraints, the different portions of the loop body can be separated-out into multiple loops with the same bounds. This is the inverse transformation of the conventional optimization technique called **loop jamming** or **loop merging** [AhU77] [AhS86].

2. Packaging Techniques

After discussing the different types of looping structures, and the parallelism opportunities they provide, it becomes necessary to consider the “dirty” details of efficient process packaging for specific machines. There are many ways to present the concerns of machine-specific coding; in the current work, the presentation centers on the fundamental guidelines for large classes of target computer, rather than for a handful of actual machines.

It is useful to create simple examples which expose the problems. A good first example is:

```
      DO 10 I=1, N
             A(I) = B(I) * C(I)
 10   CONTINUE
```

This loop can be transformed into the vector operation:

```
      A(1:N) = B(1:N) * C(1:N)
```

The first observation about this vector encoding is that, since `A(I)` appears in the original program in a scope where `I` can be any value from `1` to `N`, the array `A` must have at least `N` elements; this is how we know that `A(1:N)` is a valid description of a vector. The same principle can be applied to show that `B` and `C` are also referenced in a valid way. The only way in which this parallelization could produce incorrect results is if the original `DO` loop produced incorrect results for that value of `N` as well (i.e. if `N` were greater than an array’s upper bound).

The second insight involves the realization that the vector form written above is only viable as the parallel execution structure if the target machine is able to support at least N “processes.”[55] Suppose the target machine is of parallelism-width W. If N is greater than W, only W operations can be performed in parallel; the other N - W will have to be executed either before or after the first W — to some extent, they must be executed in sequence.

In the following sections, these two observations are expanded into the basic strategy for packaging loop parallelism.

2.1. Maximum Loop Parallelism-Width

In the example above, the transformation into parallel code was quite direct: the loop range was simply pulled inside the array references. No additional modifications had to be made to either control or data structures. Suppose instead that the following code is to be parallelized:

```
       SUM = 0
       DO 10 I=1, N
               SUM = SUM + F(I)
 10    CONTINUE
```

Even assuming that data-access information about F() is available, the parallelization transformation will be considerably more complex.

It is complicated in part because this loop is able to be parallelized only by applying an associative reduction which, for most parallel computers, is commonly accepted to require O(Log N) time and a tree-like control structure. It is also complicated, however, by the fact that SUM is only a single cell, hence it can only hold a single value at a time. The usual solution, discussed in section 7:1.2.3 is to promote SUM from a scalar into a vector, thereby allowing SUM’s elements to be operated-on in parallel.[56] How big should this vector be?

At first glance, the answer appears to be that SUM should be a vector of length N — but N could be arbitrarily large.

It is possible that reasonable bounds on N[57] can be derived at compile time by examining N’s other

55. Pipelined target machines do not violate this claim. The parallelism-width of a pipelined machine is its depth; if the depth of the pipeline is less than N, the pipeline serializes the operations.

56. Machines supporting *fetch-and-op*, in this case *fetch-and-add*, appear to perform this operation in time proportional to N/W and do not require SUM to be dimensionally promoted (an equivalent expansion occurs within the interconnection network’s nodes).

57. Here, we assume that N is an INTEGER; similar rules apply if N is a CHARACTER, REAL,

appearances throughout the program.

In the following rules for determining these constraints, we will refer to the current use of `N` as N_u and to the range (set) of possible values for an item X as Range(X). The intersection ($\cap$) of the ranges by the following constraints constitutes the best estimate of the Range(N_u).

2.1.1. Data Type Constraints

The range of values permitted for a variable of `N`'s declared type form absolute bounds on the value of N_u. If`N` is an `INTEGER` and `INTEGER`s are permitted to hold v alues in the range -32768 through 32767, then it is impossible that N_u's value is less than -32768 or that it exceeds 32767. In other words, Range(N_u) $\subseteq$ {-32768..32767}.

This is a useful constraint mainly in languages which permit definition of enumerated types (Pascal, ANSI C, etc.).

2.1.2. D-U Range Propagation

Range(N_u) cannot be greater than the union of the ranges of values for each definition of `N` which is in the U-D chain of N_u. For example, code such as:

```
N = A
IF (C) N = B
D = Nu
```

Would yield the constraint that Range(N_u) $\subseteq$ (Range(`A`) $\cup$ Range(`B`)).

2.1.3. Array Index Constraint

If there is another use of `N` which is always executed when N_u is executed, and that use appears as a index to an array, then the range of values for N_u cannot be greater than the range of values allowed for that index of the array[58].

Of course, this assumes that the program does not make indexed references which are out of array bounds. (Unfortunately, exceeding the array bounds is a common programming error, and this rule is

etc.

58. The reverse application of this rule has been used to estimate the bounds on arrays from the expressions used to index them [Kle83].

therefore somewhat unreliable.)

An example of this constraint is:

```
int a[100], e[5];
 . . .
if (c) {
  e[N] = f;
} else {
  b = N_u;
  a[N] = d;
}
```

which specifies that Range(N_u) $\subseteq$ {0..99}, since the declaration of a permits index values in {0..99}.

Notice that, although Range(N) for the N which appears in the "then" clause is similarly constrained to be a value in {0..4}, one cannot infer that this range applies for N_u. A condition which does not directly involve N could still represent a constraint on N's value.

2.1.4. Conditional Control Constraint

If there is a use of N which *dominates* N_u and is part of a conditional test which requires Range(N) to be $\subseteq$ a certain range in order for N_u to be executed, then Range(N_u) $\subseteq$ of the tested range. This rule is briefly discussed in [Fis84].

For example:

```
if (N < 601) {
  a = N_u;
}
```

checks that Range(N) is in {$-\infty$..600} and only executes N_u if this is true. Hence, Range(N_u) $\subseteq$ {$-\infty$..600}.

2.1.5. Arithmetic Range Propagation

If N_u's value was originally computed by arithmetic operations on a set of variables and/or constants, range arithmetic may be used to compute Range(N_u) from the ranges of the values used in the arithmetic expression.

For example, suppose Range(`i`) = {5..20} and Range(`j`) = {1..2}. Then Range(`i` — (`j` * 2)) = {1..18}.

Even using the above (relatively expensive) analysis to determine Range($\mathtt{N}_u$) from the program, this range may be too large.

For example, $\mathtt{N}_u$ in the sample loop at the beginning of section 7:2 may be constrained only by the fact that it is a 32-bit integer. If one always allocates the maximum size, this would lead to allocation of space for an array with subscripts ranging from 1 to 2147483647 — it is difficult to imagine that it is worthwhile promoting `SUM` into an array with over *two billion* elements!

2.2. Ideal Parallelism-Width

As discussed at the start of section 7:2, W, which is related to the machine width, forms the true upper bound on the size of a promoted dimension. If `N` > W, then `SUM` would be promoted to a vector of length W.

Let the variable P be the number of parallel processes to be created by the generated code. As a first approximation, P is set as:

If MAX(`N`) < W

P = MAX(`N`*).* If the loop cannot exhibit as much parallelism as the machine, it is conceptually unnecessary to create W-`N` useless processes (the number required to keep all processors busy). In architectures which require a high degree of low-level symmetry, such as SIMDs, one might be required to create these W-`N` “null” processes — the difference is minor and mostly semantic.

If MIN(`N`) > W

P = W. If the loop will always exhibit more parallelism than the machine can use, we need only generate as much as the machine can use.

If MIN(`N`) < W AND MAX(`N`) > W

P = W. In this case it is impossible to decide which is greater, P or W. Because the relationship of`N` and W is not known, it is probably best to minimize the worst case — the worst case occurs when the greatest execution delay could be encountered, which is generally when `N` > W — hence, we pretend that `N` > W. This is the only of these three cases in which it is possible that unnecessary code will be generated and executed.

In fact, while these assignments for P will result in the minimum completion time for the completely-parallelizable portion of a loop, they do not necessarily insure that the minimum number of processes (or resources in general) are used to achieve that minimum time. It has been suggested that the optimum value for P, in any of the above cases where P would have been set to W, is more precisely:

$P = \lceil N / \lceil N / W \rceil \rceil$

The reason for this is that $\lceil N / W \rceil$ is the minimum number of "cycles" in which a machine of width W can complete the computations, and the complete formula therefore gives the smallest integer number of processes for which the same number of cycles will be needed. This formula, as written above, is somewhat awkward to compute, but it can be computed using ordinary integer arithmetic (assuming N and W are integers) by:

D = (N + (W - 1)) / W
P = (N + (D - 1)) / D

where D is the Depth of the parallelization, in cycles. In this form, the formula will typically require integer operations for one decrement, two additions, one shift-right, and one divide. This is cheap enough to be computed at run-time, if other machine characteristics make run-time process structuring desirable.

A good demonstration of the potential savings through use of this formula can be constructed by assuming values for N and W. SupposeN=301 and W=100. This gi ves:

D = (301 + (100 - 1)) / 100
D = 400 / 100
D = 4

P = (301 + (4 - 1)) / 4
P = 304 / 4
P = 76

which states that the fastest a 100-processor machine could execute this code would be 4 cycles, but that only 76 processors are needed in order to insure that no more than 4 cycles are taken.

Not all machines would be able to take advantage of the fact that 24 (or 100 - 76) more processors would be available to do useful work during execution of this code. Multiprogrammed MIMDs are a good example of a class of machines which can benefit greatly in this way. VLIW and SIMD architectures could benefit from this only by finding another loop(s) which could be merged with current loop — an unlikely situation.

Ironically, even if the machine cannot make other use of the processors made idle by the adjustment of P, it may still be desirable to use the adjusted value of P:

- If an associative reduction (such as the summation in the example) will occur, this provides the opportunity to perform the reduction in O(Log P) time rather than O(Log W).
- There may be lower communications cost, since fewer processors need to communicate with each other.
- Machines which require run-time operations to create/terminate processes can benefit by performing W - P fewer such operations.

The reasons not to use the formula are:

- The memory reference pattern may contain fewer conflicts when P = W. Coincidentally, this also leaves open the possibility of minimizing the conflicts by chosing a width between P and W which is relatively-prime to W. (A hardware version of this idea is presented in [LuB80].)
- All the above calculations assume that the loop body has the same cost for any iteration. If this is not true, or if dynamic scheduling imbalances the load, dynamic width adjustment, as described in Chapter 8, might be preferable.

Further, it is not always possible to know at compile time precisely what W of the target machine will be. Although run-time computation of P's value is possible, there is a reasonable approach to guessing W at compile time. A guess is taken based on the probable runtime environment.

For example, if a program is being compiled to be executable on any of a group of machines which are identical except in that each has anywhere from 1 to 32 processors, one would probably compile based on the assumption that W = 32.

If most of the machines had 5 processors, one might instead compile with W = 5; this effectively optimizes execution on the 5-processor machines, but would make 6 to 32 processor machines execute the program no faster than a 5-processor machine.

3. A Complete Example

Consider:

```
int i, n;
double x, sum, h;
 . . .
scanf("%d", &n);
h = 1.0 / ((double) n);
sum = 0;
for (i=1; i<=n; ++i) {
  x = (((double) i) - 0.5) * h;
  sum += 4.0 / (1.0 + x * x);
}
```

Listing 7:5: Looping Example Code

This is a fragment of a program which uses the “rectangle method” to approximate the value of π. It is derived from a program distributed as a quick test of automatic parallelizers.

3.1. Conventional Optimization

When performing automatic parallelization, it is easy to forget that much can be done to speedup programs without using hardware parallelism. The first step is to perform this conventional optimization. (It may be useful to apply various conventional optimizations as part of every step in program transformation.)

The `for` loop is easily restructured using conventional (sequential) compiler code improvements. `i` is an *induction variable*, hence `i` is eliminated and the loop variable becomes `x`. This converts the `for` loop into:

```
for (x=(1.0-0.5)*h; x<=(((double)n)-0.5)*h; x+=h) {
  sum += 4.0 / (1.0 + x * x);
}
```

The expression for the loop limit is *loop invariant*, hence it can be moved out of the loop to a point immediately preceding the loop entry. The final sequential program fragment is:

```
int n;
double x, sum, h;
 . . .
scanf("%d", &n);
h = 1.0 / ((double) n);
sum = 0;
t1 = (((double) n) - 0.5) * h;
for (x=(0.5 * h); x<=t1; x+=h) {
  sum += 4.0 / (1.0 + x * x);
}
```

Listing 7:6: Optimized Looping Example Code

The transformed loop has only about 4/6 the operations that were originally contained within the block inside the inner loop. Presumably, a similar decrease in execution time would ensue (for most target machines, either sequential or parallel).

3.2. Parallelization

Since we have not specified the precise target machine, it is impossible to directly specify the best code to be generated. Instead, this section will present the fundamental parallel structure only loosely targeted to a particular machine; this structure must be encoded in the most appropriate way for each particular target machine, and we will not concern ourselves with determination or production of that encoded form. Abstracting to this parallel structure does not imply that this parallelization is near optimal for all machines, but simply that deriving a near-optimum form for most particular machines is easy given this starting point.

Basically, there are two steps:

(1) Analyze the loop; this includes both concurrency detection analysis by the rules given in this chapter and determination of bounds on iteration. The result is that both parallelization viability and maximum parallelism width are derived.

(2) Construct parallel code for that form.

In step (1), it becomes apparent that the `for` loop can be parallelized using an associative reduction of additions to `sum` and dimensional promotion of the variable `x`. It also becomes clear that the constraints on the value of `t1` are rather wide, and the number of iterations the `for` loop may make is essentially the largest number an `int` can represent — for a 32-bit int, this is 2147483647. Since the number of times the loop will execute could be far larger than the number of processors in the machine, it is probably best to select the parallelism width to be the width of the machine.

In step (2), let us assume that the target machine is a partitionable SIMD, such as PASM [SiS81] [SiS86]. The following support functions and variables will be used:

`pe_avail()`
: A function which returns the maximum number of PEs (SIMD processing elements) which are available to the program at run-time. So that the user program may determine how many of those it wants, this call locks-out all other programs from allocating these PEs until a `pe_alloc(`*N*`)` is executed by this program.

`pe_alloc(`*N*`)`
: A function which allocates up to *N* PEs and returns the number actually allocated. If this call was preceded by a call to `pe_avail()`, and if *N* is less than the value returned by `pe_avail()`, `pe_alloc(`*N*`)` is guaranteed to allocate *N* PEs. For PASM, this is slightly complicated by the fact that SIMD PE groups must contain a multiple of 4 PEs, hence a request of *N* PEs will actually return ((*N* + 3) & (˜3)) PEs.

`pe_number`
: A read-only pe variable which gives the logical number of the pe in which it is executing (a number between 0 and `pe_count` - 1, inclusive).

`treesum(`*pe_v*`, `*min*`, `*max*`)`
: A function which performs a tree summation across PEs logically numbered *min* to *max* of the values of the PE variable *pe_v*, and returns the value of this sum. This takes O(log (1 + *max* - *min*)) time.

The parallelization proceeds as follows. First, any variables which must be dimensionally promoted must be declared:

```
/* Variables from original code */
int n, pes;
double x, sum, h;

/* Declare promoted versions of variables */
pe double pe_x, pe_sum;
```

these promoted variables (`pe_x` and `pe_sum`) are arrays distributed across the pe memories (indicated by the the storage class `pe` in the last declaration).

Next, the ideal parallelism width for the loop can be computed and an appropriate number of processors allocated. The code is:

```
/* n is the number of loop iterations */
scanf("%d", &n);

int pe_count, depth;
pe_count = pe_avail();
depth = (n + (pe_count - 1)) / pe_count;
pe_count = pe_alloc((n + (depth - 1)) / depth);
```

this computation leaves `pe_count` as the ideal number of PEs (rounded up to a multiple of 4) given the number of PEs initially available. Notice that here we are ignoring process granularity issues because we are discussing a SIMD target machine which is capable of using very fine grain parallelism; these issues are discussed in-depth in Chapter 8.

Finally, the code to perform the associative reduction must be created. Since the number of iterations is possibly much larger than the number of processes, each PE may be executing a number of iteration’s worth of additions to `sum`. *It is not necessary to make the summations tree structured within each PE*; only the summations of results in each PE should be tree structured. The resulting parallel code appears in Listing 7:7.

This code also assumes that no loop unrolling/unraveling is done (although such loop transformations are often desirable [Ell85] [Nic85]). If the loop was unrolled/unraveled, the techniques discussed in Chapter 8 would apply.

Note that on some machines, particularly small MIMDs, a linear summation across the PEs would be more efficient for the final collection of `sum` than the tree summation given above. This is due to a higher cost being associated with tree-structured, distributed, synchronization than is associated with centralized synchronization.

```
/* Declare ordinary (CP) variables */
int n, pe_count, depth;
double x, sum, h;

/* Declare promoted versions of variables */
pe double pe_x, pe_sum;
 . . .
scanf("%d", &n);

/* Compute ideal width & allocate PEs */
pe_count = pe_avail();
depth = (n + (pe_count - 1)) / pe_count;
pe_count = pe_alloc((n + (depth - 1)) / depth);

/* Compute h = 1.0 / ((double) n);
t1 = (((double) n) - 0.5) * h;
t2 = 0.5 * h;
t3 = h * pe_count;

/* SIMD code for linear summing in each PE */
pe_sum = 0;
for (pe_x=t2+(pe_number*h); pe_x<=t1; pe_x+=t3) {
  pe_sum += 4.0 / (1.0 + pe_x * pe_x);
}

/* Do the tree additions */
sum = treesum(pe_sum, 0, pe_count);

/* If later code uses x’s value, set it right */
x = t1;
```

Listing 7:7: Parallelized Looping Example Code

This page is intentionally blank.

Irregular Code Parallelization

When concurrency detection is performed, in general, it is not difficult to find huge amounts of parallelism — but most of it is not useful for a given target machine. Hence, irregular code parallelization is primarily concerned with increasing the granularity of processes (increasing the computational content of each process).

This chapter presents rules for obtaining efficient parallelism from irregular code — code which may contain arbitrary control constructs including conditionals, loops, and subroutine calls. These parallelizations are particularly well-suited for dynamically-scheduled MIMD computers. A complete example of irregular code parallelization for such a machine is also given.

1. Process-Packaging Rules

Irregular code includes regions of code which may incorporate loops, conditionals, and subroutine calls; hence, parallelization will depend on the ability to find parallel execution streams that did not all originate in the unwinding of iterations of a loop. This is quite different from parallelization based on traditional vectorization technology, which deeply analyzes only the bodies of loops of a certain kind, namely, those which correspond to FORTRAN `DO` loops.

The most common example of an irregular code parallelization is to execute calls to non-interfering subroutines/functions in parallel [DiK84] [Vei85] [Ban86] [TrI86]; this is essentially the only parallelism used by functional languages. Less obvious irregular code parallelizations involve grouping arbitrary sets of instructions, including conditional and/or looping constructs, into packages which can be executed simultaneously: for example, executing multiple unrelated loops simultaneously.

The theory behind irregular code parallelization is exactly that which is presented in Chapter 5 — concurrency detection. Irregular code parallelization is more than concurrency detection, however, in that for concurrency detection, the set of regions to be parallelized is fixed, whereas irregular code parallelization *must generate the grouping of instructions into regions* which will result in efficient parallel execution. Further, in creating these regions, additional code must be created to encapsulate the code for each region. This is why the refined-language approach distinguishes concurrency detection from process packaging.

While it is true that additional code also must be created to parallelize execution of the body of a loop, the structure of the additional code is very simple and the associated cost of executing that code is typically low. In parallelized irregular code, the costs associated with synchronization and process creation/termination are often quite large, hence these costs play a very significant role in determining the efficiency of the resulting code. However, these costs are not easily obtained by examining the code.

The following sections outline rules for packaging.

1.1. Package Validity

To create packages, the compiler first performs concurrency detection, then re-partitions code into regions of a granularity more appropriate for the target machine, and finally encapsulates each region within additional code for process creation/termination. In re-partitioning the code into regions, there are a few constraints which cannot be violated; in terms of analysis structures which exist before concurrency detection has been performed, these constraints are given in Chapter 5. These constraints may also be expressed using the results of concurrency detection on the original set of regions:

Constraint 1:

Regions $\mathbf{r}_i$ and $\mathbf{r}_j$ can be merged into a new region, $\mathbf{r}_{ij}$, if
there does not exist a region $\mathbf{r}_k$, such that $\mathbf{r}_i < \mathbf{r}_k < \mathbf{r}_j$.

Constraint 2:

If regions $\mathbf{r}_i$ and $\mathbf{r}_j$ are to be merged into a new region, $\mathbf{r}_{ij}$, the merger is legal if
$\mathbf{r}_j < \mathbf{r}_i$ is *false*;
if this is condition is met, $\mathbf{r}_{ij}$ is $\mathbf{r}_i$ followed by $\mathbf{r}_j$.

The first constraint is intended to avoid the problem of generating code for regions with internal synchronization, the second constraint simply insures that the region resulting from the merger has the same functionality as the component regions would have in the sequential program. Since a loop in the code creates a cycle which violates constraint 1, as a special case, loops are treated as “nested” graphs:

- the constraints above apply to code regions contained within a single iteration of a loop (ignoring the backward branch and all regions outside the loop) or
- the constraints above apply to code regions which contain any loop(s) completely within individual regions.

1.2. Package Choice

The constraints given above, when applied to a typical program's regions at the intermediate-code instruction level, generate an incredibly large number of possible parallelizations: the number of different valid sets of regions explodes. In the code a compiler generates, just one of these is used.

There is only one principle to adhere to in chosing the set of regions (processes) for code parallelized for speed-up:

> No code should be spawned as a process *unless* doing so will, with some probability, result in faster completion of the entire program's execution than if the code were executed sequentially appended to another process.

However, this is not easily implemented.

The major complications are computing the execution cost of a region of code and determining which little **potential processes** (initial regions + encapsulation code) should be re-grouped into usably large processes.

1.2.1. Execution Cost Of Code

As any assembly-language programmer knows, it is quite difficult to determine the execution time of a program just by examining the instructions which it contains. This difficulty is rooted in at least several run-time variable quantities: instruction execution time, number of executions of each instruction, and availability of machine resources.

1.2.1.1. Instruction Execution Time

Individual instructions may take different amounts of time to execute depending on their operand values and run-time interference from other devices. For example, instructions such as multiply and divide often will execute much faster or slower depending on the number of `1`-bits in the operand values.

In MIMD computers whose processors communicate with main memory through a shared bus or interconnection network, the time to perform a memory-based operation such as a load or store may vary widely depending on memory traffic from other processors.

Since these effects cannot generally be known at compile-time, it is necessary for the compiler to make reasonable (probabilistic) guesses about them. In addition, since the region re-grouping operates on the intermediate form, rather than on machine instructions, it is convenient for the compiler to use estimates of intermediate instruction costs. These costs also may be approximate if the translation from intermediate code into machine code is complex. For example, the cost of an intermediate code tuple which represents a function call is essentially impossible to compute (especially if the function body is compiled separately); the cost of a tuple for which there are many possible machine code templates (each with a different cost) is also difficult to compute.

A crude, but often acceptable, approximation is to assume that each intermediate instruction executes in 1 unit of time, except for function/subroutine calls, which execute in T units of time where $T >> 1$. This is the approximation used in the first RC compiler [Ste86].

1.2.1.2. Executions Per Instruction

Since irregular code regions may include control structures, some instructions within a region may be executed many times each time the region is entered, whereas others may be executed only once or, under certain conditions, perhaps not at all. It is, therefore, vital that the cost computed for a region incorporate "execution count multipliers" weighting the execution time of each instruction within the region by the number of times it is executed.

In general, the problem of computing these weights is clearly a version of the classic halting problem; hence, it cannot be precisely solved. However, in some cases it can be solved and, in those cases where it cannot be solved, it is possible to substitute a probabilistic value — the *expected number of executions* per region entry.

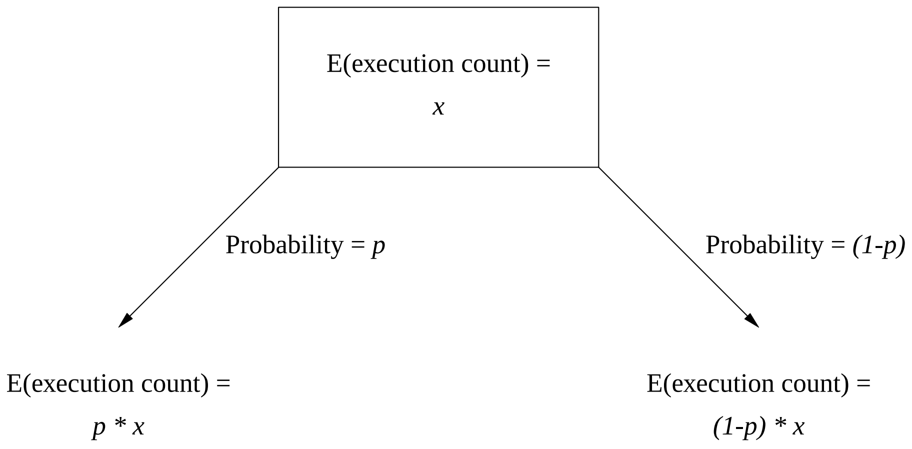
To determine these weights, it is convenient to consider the contents of each region as a control-flow graph[59]. The graph of each region should be straightened: adjacent basic blocks α and β, such that α must be followed by β and no other basic block flows directly into β, should be merged into a single basic block.

All instructions within a basic block in a particular region's straightened control-flow graph will have the same weighting factor, hence, the compiler need only compute weightings for each of the basic blocks. There are just a few cases:

- *Start Basic Block*. The very first (unconditionally executed) basic block in a region is given a weighting of one: it is executed exactly once per region entry.

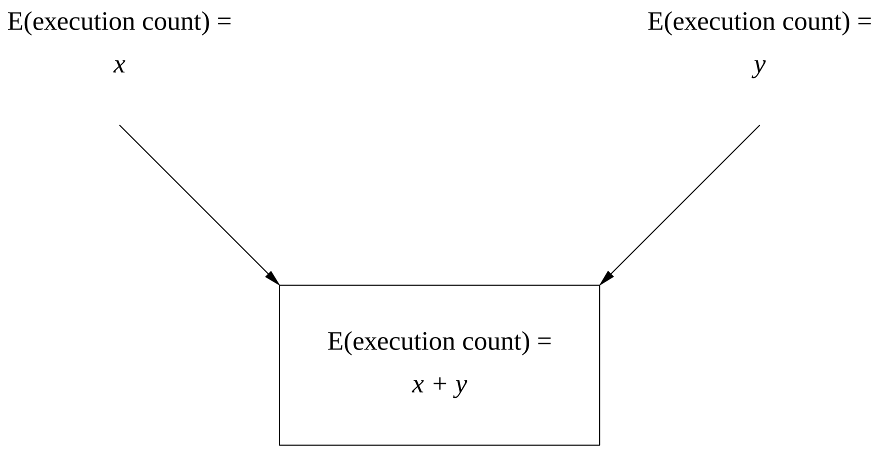
59. This is not, however, the preferred implementation. It is more efficient to determine the weights as the control-flow graph is constructed — each intermediate instruction is associated with a cost (or cost function) prior to concurrency detection. The algorithm is discussed here simply because the results it generates are not used until this stage and there is no conceptual need to compute them earlier.

- *Divergence Of Flow*. Two basic blocks which are, for example, the `THEN` and `ELSE` clauses of an `IF` statement, do not each have the same expected execution count as the basic block which contained the conditional expression of the `IF`. The execution counts for the `THEN` and `ELSE` basic blocks added together must equal the execution count for the `IF` condition's basic block. Branches involved in cycles are treated specially (see below). In general, wherever there is a divergence of control flow, the expected execution counts are related as:

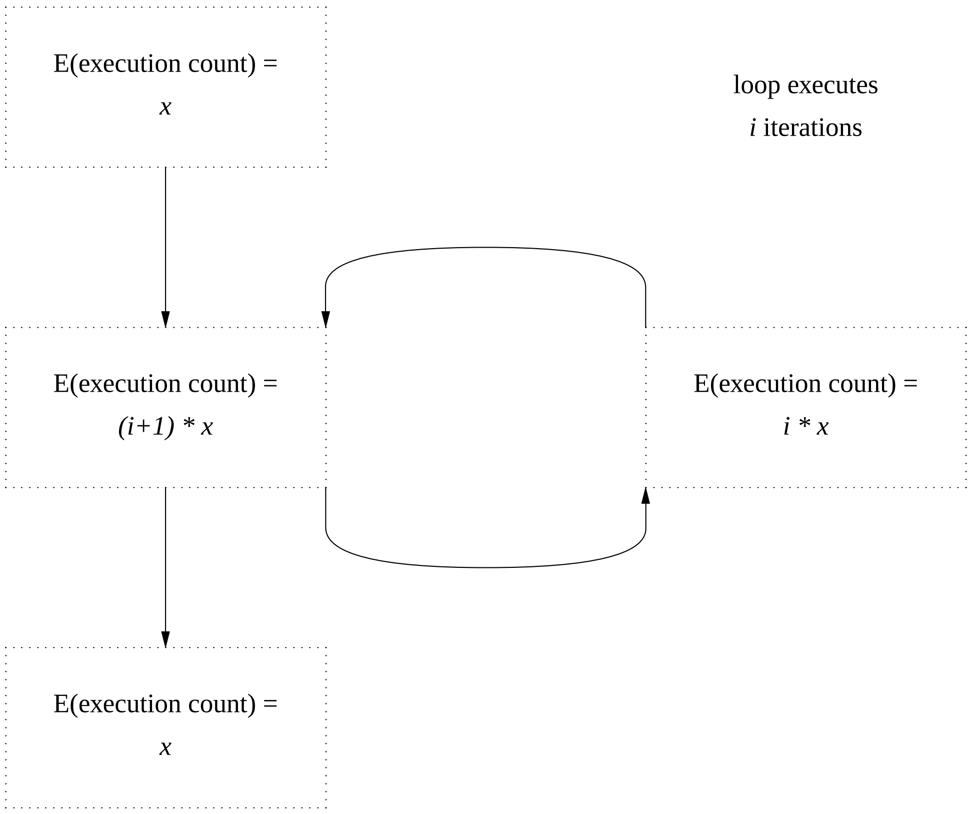


where E(y) is the expected value of y. The same concept can also be applied to arbitrary multi-way branches. Typically, since the branch probability is not known, the compiler would guess that all branch alternatives are equiprobable, with probability 1/n for each arc in an n-way divergence.

- *Convergence Of Flow*. Continuing the previous example, eventually the flow paths through the `THEN` and `ELSE` clauses would once again meet. The expected execution count of a basic block where two or more flow paths converge is the sum of the execution counts on all converging arcs. Branches involved in cycles are treated specially (see below). Convergence of control flow relates the expected execution counts as:



• *Graph Cycle*. Arcs which form a graph cycle are ignored in the previous cases and are processed after the above rules have been applied. Code within a loop is likely to be executed many times, hence, when a cycle in the control-flow graph is located, the weights of all basic blocks within the cycle are multiplied by the number of iterations made by the loop. Code which is within the loop, but is executed once before the loop test, has its execution count multiplied by the the number of iterations + one. Consider the following graph (in which dotted boxes represent entire subgraphs):



Chapter 7 discussed some methods by which the number of loop iterations may be determined. If these methods fail to produce a weight, the compiler can simply assume some number — 2 is a conservative guess.

Better weighting values can be obtained by profiling the code's execution with test data [Fis84], but this is awkward.

These weighting ideas are not new. In some sense, all global optimizations are based on the above weightings. For example, it is clear how register allocation is affected. Another natural optimization possibility is to move code to blocks with lower execution counts or to rewrite the control structure so that the execution counts are lower.

1.2.2. Fundamental Bounds On Package Size

Given the above cost formulation, and information about costs incurred in process creation/termination (which includes communication costs, since the above description of regions insures that regions communicate only at entry/exit), it becomes possible to make some statements about absolute bounds on efficiency of a choice of region groupings:

> Wherever possible, regions should be created large enough so that the cost for the parent process to create/terminate a process to execute the region is less than the expected execution cost of the region.

This guideline states that, if a process would take longer to start/stop as a parallel operation than it would take to complete appended to the current sequential process, then it should be appended to the current sequential process (i.e., be part of that region rather than a separate region).

An interesting, and important, corollary to this is that, regardless of the internal parallelism:

> A subgraph of the control-flow graph which is of cost less than twice the cost to create/terminate a process should be placed in a single region.

since it is impossible to speedup execution by parallelism using only one process and, as discussed above, it does not speedup computation to create a process for a region which will execute sequentially for less time than it takes to create/terminate a process.

1.2.3. Parallelism-Width Adjustment

A secondary, but obviously important, consideration is that enough parallel-executable processes should be made so that the machine's full parallelism can be used to effect speedup. There are two basic approaches to adjusting the parallelism-width of the generated code to match the machine: static adjustment and dynamic adjustment.

Static parallelism-width adjustment is accomplished by the compiler determining, at compile-time, exactly *what* processes *will be* created. For example, code for VLIW, Pipelined, and Systolic computers typically has a parallelism width which is fixed at compile time: the most extreme kind of static adjustment.

A less extreme version of static parallelism-width adjustment would be typical of code generated for MIMD computers by programmers writing in explicitly-parallel languages. Where a parallel process is desired, one is blindly spawned; the availability of another physical (or at least logical) processor is simply assumed. Of course, such blind spawning can easily result in too many processes for the machine, thereby wasting significant amounts of time simulating multiple processors on each physical processor.

Dynamic parallelism-width adjustment can avoid this waste. For MIMD and SIMD machines, instead of blindly spawning processes, code can be generated to check the number of processors available at runtime, and only to spawn a new process if there is a processor free to execute it. If no additional processors were free, the work is appended to one or more existing processes.

The concept of parameterizing code by the machine width is far from new; it has long been the preferred process structure for these target machines. However, there is a significant new variation on the idea [DiK85a]. Rather than merely checking to see if additional processors are available, a parallelizing compiler can generate code to also check runtime conditions which, although undecidable at compile time, can be answered at runtime and can result in increased parallelism.

As others have observed [Ell85], a parallelizing compiler "knows" where it did not have sufficient information available to guarantee that parallel execution of two operations is safe. In fact, *not only does the compiler "know" when it does not "know" enough to permit generation of parallel code, but it has a good estimate of how much it would gain if it did "know"* [DiK84]. For example, suppose that we have the following source code:

```
a[i] = 5;
a[j] = 6;
```

If the compiler cannot determine that `i` is never equal to `j`, the obvious answer is to generate sequential code which looks like:

```
{ a[i] = 5; }
{ a[j] = 6; }
```

The casual observer might even suspect that if static flow analysis proves that `i` will sometimes, but not always, be equal to `j`, the coding given above is the only possible. However, this is not the case.

If the two statements (or regions) were sufficiently expensive so as to be worth computing in parallel (which we do not claim of the example given), we can profit greatly from generating code as:

```
/* Are a[i] and a[j] aliases? */
if (i == j) {
    /* Yes: a[i] = 5; is dead -- don't bother */
    a[j] = 6;
} else {
    /* No: Execute statements simultaneously */
    parallel({ a[i] = 5; }, { a[j] = 6; });
}
```

Because, in general, a region in which a potential conflict for parallel execution arises usually *contains a dead computation in the case where the conflict occurs*. The compiler might "waste" some memory in generating two code streams and a test, but each of the streams is probably faster to execute than the obvious sequential encoding. The parallelized encoding is probably much faster.

2. An Example: RC Quicksort

In order to demonstrate the parallelization of irregular code, this section will examine in-depth the parallelization of an example code: the same RC version of quicksort which appears in Listing 3:2. Quicksort is typical of *recursive* “divide and conquer” algorithms used in languages like Pascal and C.

The target machine for this example is a hypothetical multi-user, large-scale, shared-memory MIMD computer.

The first step in automatically parallelizing the RC code given above is to compile it into intermediate instructions and to construct the control-flow graph. In the interest of simplifying the discussion, the details of the intermediate instructions are ignored in the following exposition; however, it is important to recognize the basic blocks which form the nodes of the control-flow graph. These basic blocks are:

Basic Blocks in RC Quicksort

Code for Basic Block	Name of Block

`i = 0; j = count(a)-1;`
`x = a[count(a) / 2];`
	A

`a[i] < x`	B
`++i;`	C
`x < a[j]`	D
`--j;`	E
`i <= j`	F

`w = a[i]; a[i] = a[j];`
`a[j] = w; ++i; --j;`
	G

`i <= j;`	H

`part(a[w], below,(w<=j), mid,(w<i), above);`
`count(below) > 1`
	I

`sort(below);`	J
`count(above) > 1`	K
`sort(above);`	L

and, when combined with the control-flow information, form the control-flow graph:

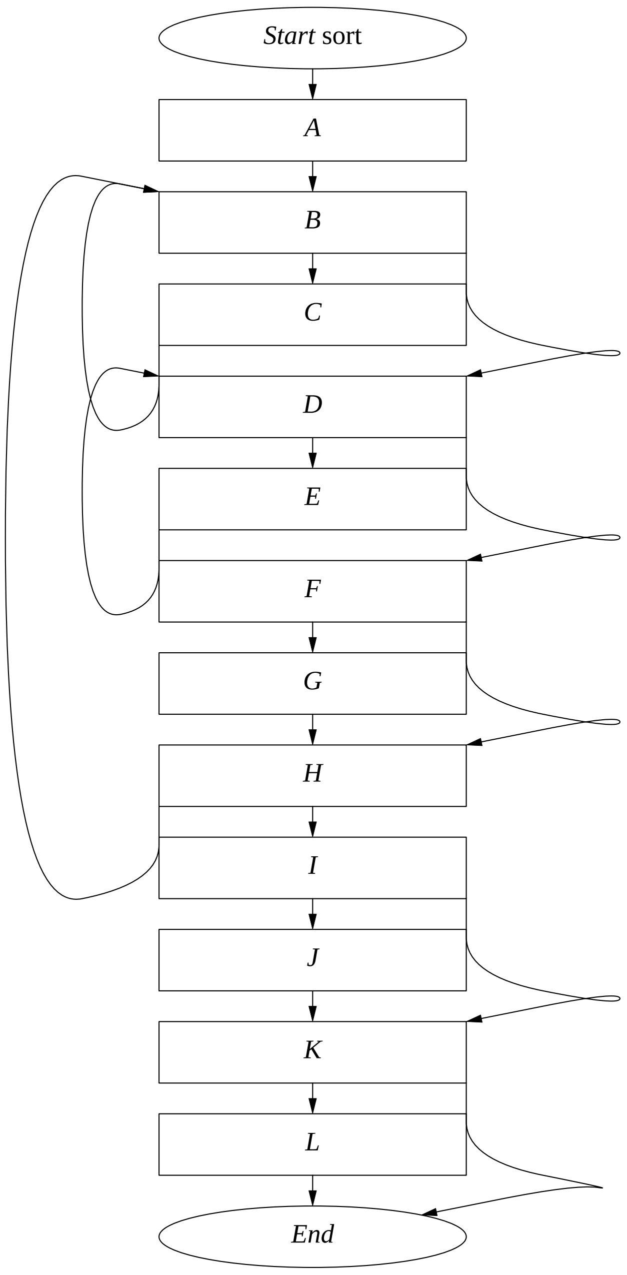


Figure 8:1: Control-Flow Graph Of RC Quicksort

Within this graph, there are three loops:

- A `while` loop with body *B* and *C*

- A `while` loop with body *D* and *E*
- A `do` . . . `while` loop with body *B, C, D, E, F, G,* and *H*

none of which can have execution of its body for efficiently parallelized across iterations. There is parallelism in that the loop with body *B* and *C* reads the values of `x` and `a[]` and modifies the variable `i`, whereas the loop with body *D* and *E* reads `x` and `a[]` and modifies the variable `j`; these two loops can be executed in parallel with each other.

However, the primary source of parallelism in quicksort is not the simultaneous execution of these loops. The two recursive invocations of `sort`, in blocks *J* and *L*, may proceed in parallel since they operate on disjoint portions of `a`. Although this would be extremely difficult (perhaps impossible) for a compiler to recognize through analysis of the C or Pascal version of quicksort, the parallelism of the calls is trivially recognized by an RC compiler given the code in Listing 3:2. The compiler would also recognize that *I* and *J* can be executed in parallel with *K* and *L*.

In addition, there is at least some finer-grained parallelism within basic blocks (particularly in *A* and *G*), although how much depends on the level of granularity at which analysis is performed[60].

Assuming that our target machine is a *dynamicaly-scheduled*, *shared-memory, large-grain MIMD* (as stated earlier), not all of this potential parallelism will speedup the program's execution. The compiler's task is now to determine the packaging of the parallelism which will result in maximum speedup. Toward this goal, it would compute weights (expected execution counts) and costs something like:

60. Although it would be more accurate to present a graph with operations at the level of those used in Chapter 5, if this were done, the resulting graph would span many pages. Such a graph is unusable for a human, but quite convenient for a parallelizing compiler.

Sequential Basic Block Costs in RC Quicksort

Name of Block	**E(execution count)**	**Approx. Cost (in Seq. execution)**
A	1 → 1	12 * 1 → 12
B	3 * 3 → 9	5 * 9 → 45
C	2 * 3 → 6	2 * 6 → 12
D	3 * 3 → 9	5 * 9 → 45
E	2 * 3 → 6	2 * 6 → 12
F	3 → 3	3 * 3 → 9
G	0.5 * 3 → 1.5	14 * 1.5 → 21
H	(0.5 + 0.5) * 3 → 3	3 * 3 → 9
I	1 → 1	18 * 1 → 18
J	0.5 → 0.5	102 * 0.5 → 51
K	(0.5 + 0.5) → 1	5 * 1 → 5
L	0.5 → 0.5	102 * 0.5 → 51

It is important to note that the cost numbers are somewhat arbitrary and that the expected execution counts are listed *relative to the completely sequential execution of the code*. For example, the compiler has guessed that basic block *L* is executed about half the times that `sort` is entered — but the expected execution count of block *L* is 1 if *L* is viewed as a region unto itself (an independent subgraph).

Suppose that the process creation/termination cost is 15 (it might well be a function of memory layout, etc., in a real machine [SaH86]). It is then fairly easy to make the following observations:

- *None of the fine-grain parallelism is useful.* In order to speedup the operation using fine-grain parallelism, it must be possible to create at least two processes of cost (relative to the generated structure) 15 or greater. The largest possible source of fine-grain parallelism is basic block *G* — but, when considered by itself, *G* has an execution cost of only 14, which is less than half what it would need to be for the parallelism to speedup execution.
- *Parallel execution of loops BC and DE is useful.* The e xecution cost of loop BC treated as a region unto itself is (3 * 5) + (2 * 2) = 19, which is greater than 15; the same is true for the cost of DE as a

region unto itself.

- *Parallel execution of IJ and KL is useful.* Excluding the `part` statement (since it cannot be e xecuted in parallel with KL), the execution cost of a region holding IJ is 5 + (0.5 * 102) = 56, which is greater than 15; the same is true for the cost of KL as a region unto itself.

which straightforwardly lead to generation of a parallel code structure like:

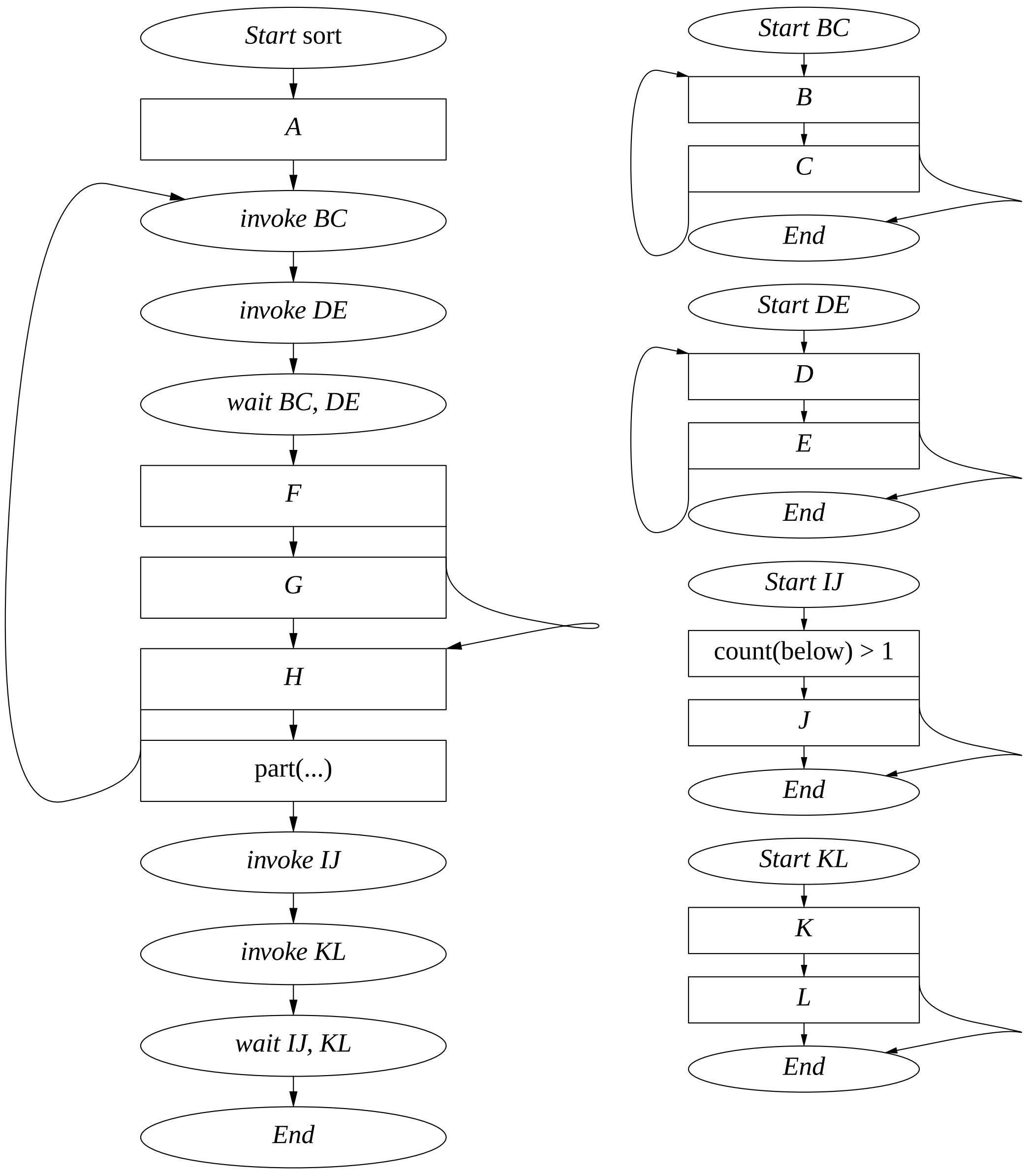


Figure 8:2: “Easy” Parallelization Of RC Quicksort

This parallelized form implements dynamic parallelism-width adjustment using the concept of an **invoke** instruction.

In [DiK85a], we observed that dynamic adjustment can be done very efficiently by careful choice of code sequences for implementing call and spawn — using the same technique for passing argument values to either. It is possible that the difference could be as small as a single instruction, which we call **invoke**. Invoke simply checks to see if another PE is available and either calls or spawns based on the answer.

However, although the above is a valid parallelization and it can be expected to adjust its parallelism width to match machine PE availability and to obtain good speedup, it is not optimal.

Parallel execution of *IJ* and *KL* is useful, and the two regions *contain as many instructions as possible*, but the the inclusion of *I* with *J actually lowers the execution cost of the region*. A reduction in the cost within the parallelized code, without a reduction in the overhead of process creation/termination, lowers the efficiency of the program.

The execution cost of *J* as a region by itself is 102, as compared to a cost of 56 for a region containing *IJ*; the same relationship holds for *L* versus *KL*. In other words, by adding *I* to *J*, the cost of *I*, which is 5, is gained, but *the probability that the process will execute J is reduced from 1 to only 0.5*. The same effect holds for *KL* relative to *L*. In this example, *making some regions contain less code makes them larger grained!*

By this observation, the parallelization becomes:

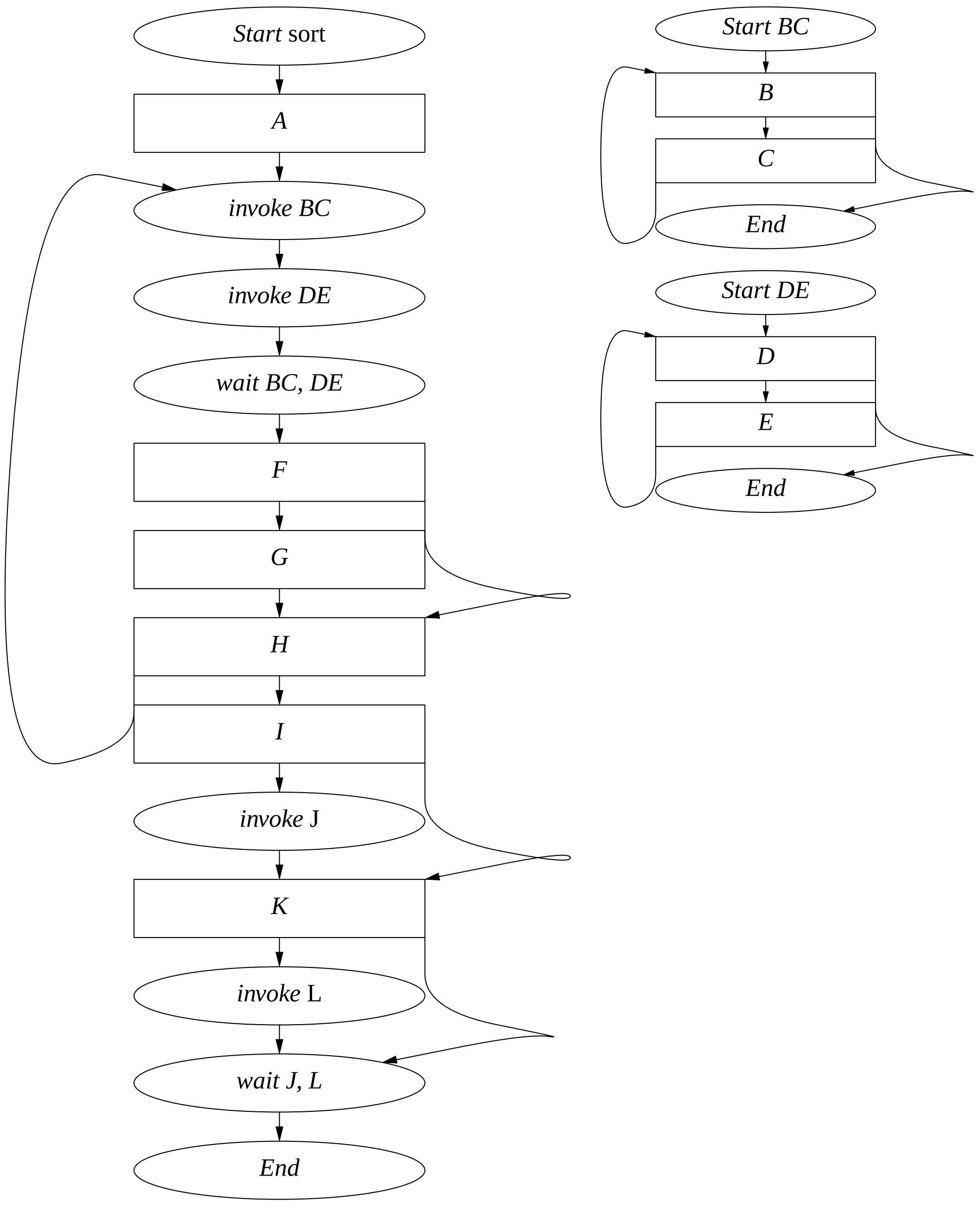


Figure 8:3: Better Parallelization Of RC Quicksort

but even this isn't optimal.

The final step is to truncate the spawning tree. In the above parallelization, when the second process of a pair of parallel processes is spawned, there is nothing left for the parent to do but wait. Instead, to create *N* computing processes from one parent, the compiler should generate code which spawns no more than *N-1* processes:

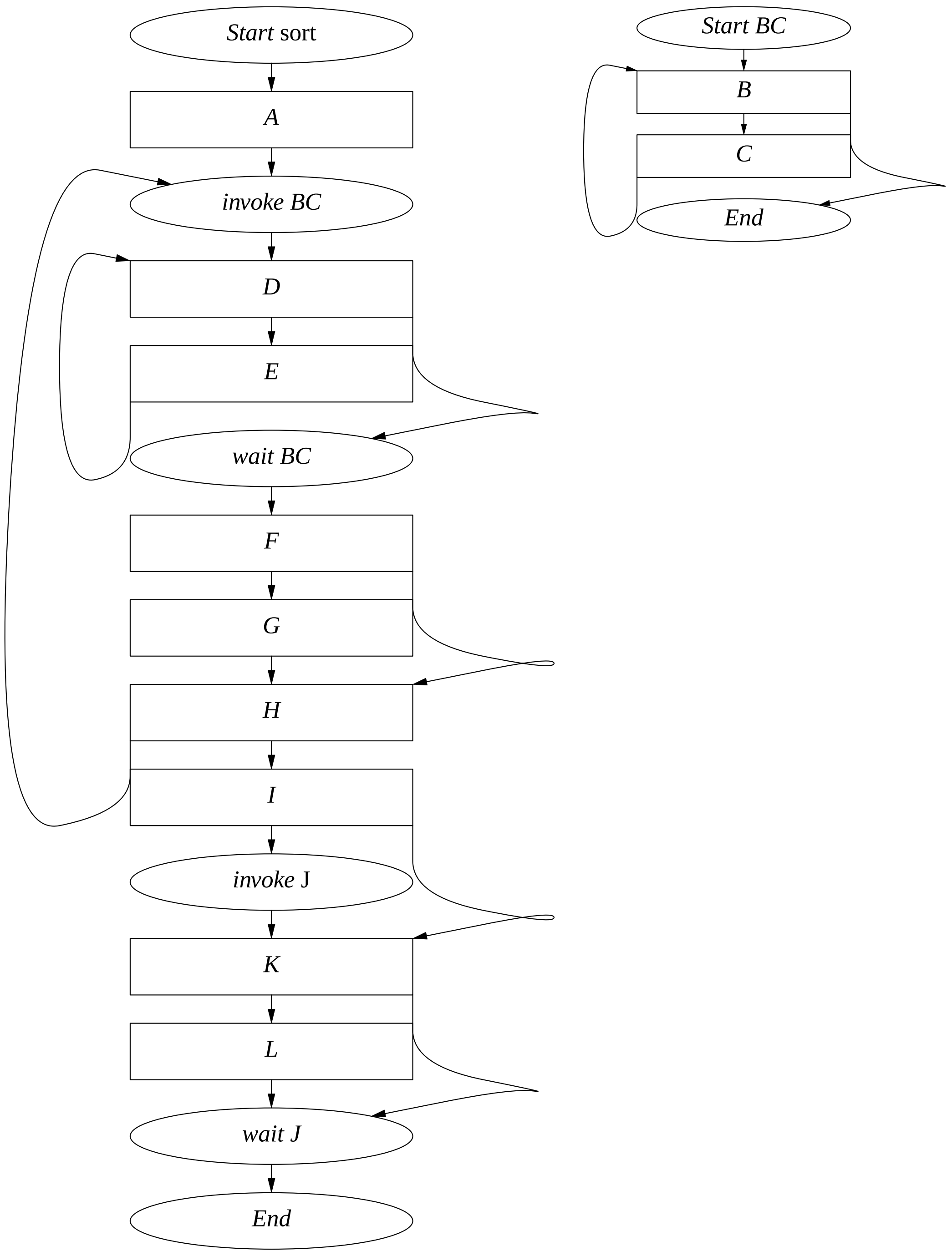


Figure 8:4: “Optimal” MIMD-ization Of RC Quicksort

By the measures given above, this version is probably the best parallelization possible.

Summary

And Conclusions

The refined-language approach to compiling for parallel supercomputers is an holistic approach to solving the problem of compiling, and programming, for speedup-oriented parallel supercomputers. In the current work, this approach was decomposed into two parts:

(1) language design and

(2) automatic parallelization technology.

Numerous small examples of the application of the refined-language approach are given in both of these aspects, and a reasonable theoretical background is given.

However, since the refined-language approach is intended to be a complete methodology independent of choice of base language and target machine, this thesis deliberately avoids measures which are based on specific choices. For example, although we have a working RC compiler (and a number of small test compilers), we quote no speedup benchmark results. Aside from the fact that the approach transcends these implementation details, benchmarks are not reported because:

- Many of the improvements over previous approaches are qualitative, not quantitative. For example, until [Con86], we were the only automatic parallelization group operating on a dialect of the C language, and our compiler is targeted for a MIMD whereas theirs generates vector code. In fact, many of the target machine types which we have considered have *never before been the target of a parallelizing compiler*: a good example is [PaK86], which is a MIMD/VLIW hybrid.
- Even for machines which have been targets of other compilers, we have concentrated on *different parallelizations* — the techniques of, for example, [Kuc78] [AlK82] and [FiE84] are all very well-developed and any of them easily can be incorporated in the framework of our parallelization approach. For this reason, we felt no compulsion to implement parallelization techniques other than our own (although an integrated implementation of these other techniques in our compilers is a long term goal).
- In this field, numbers are very misleading. For example, at one point, we considered using the parallelization of the FORTRAN WHETSTONE benchmark as a test. This code was deliberately written to thwart clever optimizing compilers and, indeed, it does thwart vectorization fairly well. The refined-language MIMD-ization would perform very well on the refined version of this code, by executing each of the seven separate loops in this benchmark as a separate process. However, WHETSTONE is a

program that reads no input — the entire program can be reduced, by a tedious, but straightforward, constant-folding compilation technique, into just a few `WRITE` statements! We also discovered a number of benchmarks which would consistently result in super-linear speedup — mostly due to having more registers and better cache performance in the multiple-processor execution.

That the refined-language language constructs can provide better information for automatic parallelization is not disputed — only the ease/consistency with which *programmers will employ them* has been questioned, and the answer can only be found after there is an extensive user community. Likewise, because our parallelization *is compatible with other techniques currently in use*, the primary concern is not “How do these techniques compete?”; rather, the question is “How much extra parallelism will they find?” It is agreed that the answer is at least some, but quantifying “some” will take a long time.

Research applying the refined-language approach to various combinations of language and target machine is currently underway at both Purdue University in West Lafayette, Indiana, and at the Center for Distributed Processing at Stevens Institute of Technology in Hoboken, New Jersey. Additional refined C and FORTRAN compilers, and several versions of C-REFLEX, targeted for hybrid MIMD/VLIW, MIMD/SIMD, and MIMD/Vector machines are planned. An improved version of the software tool C-PREFINE and FORTRAN-PREFINE are currently under development.

This page is intentionally blank.

Bibliography

[AbM86] W. Abu-Sufah and A. D. Malony, “Vector Processing on the Alliant FX/8 Multiprocessor,” Technical Report, Number CSRD 541, University of Illinois at Urbana-Champaign, 1986.

[Ada80] *Ada Programming Language*, Military Standard, Number MIL-STD-1815 (the green book), Department of Defense, December 1980.

[AgA82] T. Agerwala and Arvind, “Data Flow Systems,” IEEE Computer, February 1982, pages 10-13.

[AhK] A. V. Aho, B. W. Kernighan, and P. J. Weinberger, *Awk — A Pattern Scanning and Processing Language*, UNIX Manual, undated.

[AhS86] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison Wesley, Reading, Massachusetts, 1986.

[AhU77] A. V. Aho and J. D. Ullman, *Principles of Compiler Design*, Addison Wesley, Reading, Massachusetts, 1977.

[AlB86] R. Allen, D. Baumgartner, K. Kennedy, A. Porterfield, “PTOOL: A Semi-Automatic Parallel Programming Assistant,” 1986 International Conference on Parallel Processing, August 1986, pages 164-170.

[AlK82] J. R. Allen, K. Kennedy, “PFC: A Program to Convert Fortran to Parallel Form,” Department of Mathematical Sciences, Rice University, Houston, Report MASC TR 82-6, March 1982.

[All83] J. R. Allen, *Dependence Analysis for Subscripted Variables and its Application to Program Transformations*, Rice University, Ph.D. Thesis, April 1983.

[All86] F. Allen, “The Parallel Translator Project,” NASA / ICASE Parallel Languages and Environments Workshop, November 1986.

[Ame86] *Ametek system 14*, Ametek, Arcadia, California, 1986.

[AnA86] M. Annaratone, E. Arnould, T. Gross, H. T. Kung, M. S. Lam, O. Menzilicioglu, K. Sarocky, and J. A. Webb, “Warp Architecture and Implementation,” Proceedings of the 13th Annual Symposium on Computer Architecture, June 1986, pages 346-356.

[AnC82] P. Anklam, D. Cutler, R. Heinen Jr., and M. D. MacLaren, *Engineering a Compiler: VAX-11 Code Generation and Optimization*, Digital Equipment Corporation, Bedford, Massachusetts, 1982.

[ArC84] Arvind and D. E. Culler, “Why Dataflow Architectures,” CH2022-2/84/0000/0027 IEEE, 1984, pages 27-32.

[ArG78] Arvind, Gostelow, and Plocyfe, "The (Preliminary) ID Report; An Asynchronous Programming Language and Computing Machine," MIT, May 1978.

[Are85] *Areté Series 1000*, Areté Systems Corporation, San Jose, California, 1985.

[Bab84] R. G. Babb II, "Parallel Processing with Large-Grain Data Flow Techniques," IEEE Computer, July 1984, pages 55-61.

[Bac78] J. Backus, "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs," Communications of the ACM, Volume 21, Number 8, August 1978, pages 613-641.

[Ban86] *A Direct Parallelization of Call Statements — A Review*, Technical Report, University of Illinois at Urbana-Champaign, April 1986.

[BrM] J. D. Brock and L. B. Montz, "Translation and Optimization of Data Flow Programs," Technical Report, MIT.

[BuC86] M. Burke, R. Cytron, "Interprocedural Dependence Analysis and Parallelization," SIGPLAN Symposium on Compiler Construction, 1986, pages 162-175.

[Bur84] M. Burke, *An Interval Analysis Approach Toward Interprocedural Data Flow*, IBM, Yorktown Heights, New York, Research Report RC 10640 (#47724), July 1984.

[Cho83] F. C. Chow, "A Portable Machine-Independent Global Optimizer — Design and Measurements," Technical Report, Number 83-254, Stanford University, December 1983.

[Cho86] P. Chow, "MIPS-X Instruction Set and Programer's Manual," Technical Report, Number CSL-86-289, Stanford University, May 1986.

[CoK86] K. D. Cooper, K. Kennedy, L. Torezon, "Interprocedural Optimization: Eliminating Unnecessary Recompilation," SIGPLAN Symposium on Compiler Construction, 1986.

[CoS70] J. Cocke and J. T. Schwartz, *Programming Languages and Their Compilers*, Course Notes, Courant Institute, New York, 1970.

[Con85] *Convex FORTRAN*, Convex Computer Corporation, Richardson, Texas, 1985.

[Con85a] *Convex C-1 Computer System*, Convex Computer Corporation, Richardson, Texas, 1985.

[Con86] *Vectorizing C Compiler*, Convex Computer Corporation, Richardson, Texas, 1986.

[CuW82] Curtis, R. and Wittie, L., "BUGNET: A Debugging System for Parallel Programming Environments," CH1802-8/82/0000/0394 IEEE, 1982, pages 394-399.

[DaK82] A. L. Davis and R. M. Keller, "Data Flow Program Graphs," IEEE Computer, 1982.

[DeG84] J. B. Dennis, G. R. Gao, and K. W. Todd, "Modeling the Weather with a Data Flow Supercomputer," IEEE Transactions on Computers, Volume C-33, Number 7, July 1984, pages 592-603.

[DeS86] N. Delisle and M. Schwartz, "A Programming Environment for CSP," Proceedings of SIGPLAN 1986 Symposium on Language Interfaces and Programming Environments, 1986, pages 34-41.

[Den86] J. Dennis, Personal Communication, SIAM General Conference, 1986.

[DiG83] H. Dietz and I. Gerri, *PILE Reference Manual*, Polytechnic Institute of New York, 1983.

[DiK84] H. Dietz and D. Klappholz, “Refining A Conventional Language for Race-free Specification of Parallel Algorithms,” 1984 International Conference on Parallel Processing, August 1985.

[DiK85] H. G. Dietz and A. D. Klappholz, “RISC CPU Design for MIMDs,” presented at the Second SIAM Conference on Parallel Processing for Scientific Computing, November 20, 1985.

[DiK85a] H. Dietz and D. Klappholz, “Refined C: A Sequential Language for Parallel Programming,” 1985 International Conference on Parallel Processing, August 1985.

[DiK86] H. Dietz and D. Klappholz, “Refined FORTRAN: Another Sequential Language for Parallel Programming,” 1986 International Conference on Parallel Processing, August 1986, pages 184-191.

[DiS85] H. Dietz, K. Stein, and D. Klappholz, “Sequential Languages for Programming Highly-Parallel Computers,” presented at the Second SIAM Conference on Parallel Processing for Scientific Computing, November 21, 1985.

[Die83] H. G. Dietz, *Compiler Design and Construction I*, Graduate Course Notes, Polytechnic Institute of New York, Fall 1983.

[Die84] H. G. Dietz, *Compiler Design and Construction II*, Graduate Course Notes, Polytechnic Institute of New York, Spring 1984.

[Die87] H. G. Dietz, *Loop Parallelization by Selective Serialization*, paper in preparation.

[Dij75] E. W. Dijkstra, “Guarded Commands, Nondeterminacy, an Formal Derivations of Programs,” Communications of the ACM, Volume 18, Number 8, August 1975, pages 453-457.

[ElX85] *System 6400*, ElXSi, San Jose, California, 1985.

[Ell85] J. R. Ellis, *Bulldog: A Compiler for VLIW Architectures*, ACM Doctoral Dissertation Award, MIT Press, 1985.

[FiE84] J. A. Fisher, J. R. Ellis, J. C. Ruttenburg, and A. Nicolau, “Parallel Processing: A Smart Compiler and a Dumb Machine,” Yale University, 1984.

[Fis84] J. A. Fisher, “The VLIW Machine: A Multiprocessor for Compiling Scientific Code,” IEEE Computer, July 1984, pages 45-53.

[Fle85] *The Flex/32 Multicomputer System Overview*, Flexible Computer Corporation, Richardson, Texas, 1985.

[GaL83] D. D. Gajski, D. H. Lawrie, D. J. Kuck, and A. H. Sameh, “Cedar,” Technical Report, University of Illinois at Urbana-Champaign, December 1983.

[Gan80] M. Ganapathi, “Retargetable Code Generation and Optimization Using Attribute Grammars,” Computer Science Technical Report, Number 406, University of Wisconsin-Madison, December 1980.

[GeR85] H. H. Gehani and W. D. Roome, “Concurrent C — An Overview,” 1985 Winter USENIX Conference, 1985.

[Gel86] D. Gelernter, “Portable Parallel Programming Environments: LINDA (real), SYMMETRIC LISP (potential),” Presented at NASA/ICASE Parallel Language and Environment Workshop, November 1986.

[GoG83] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir, “The NYU Ultracomputer — Designing an MIMD Shared Memory Parallel Computer,” IEEE Transactions on Computers, Volume c-32, Number 2, February 1983, pages 175-189.

[Got86] A. Gottlieb, “An Overview of the NYU Ultracomputer Project,” Ultracomputer Note #100, Courrant Institute, New York, July 1986.

[GrL86] T. Gross and M. S. Lam, “Compilation for a High-Performance Systolic Array,” Proceeding of SIGPLAN 1986 Symposium on Compiler Construction, June 1986, pages 27-38.

[Gro83] T. Gross, “Code Optimization of Pipeline Constraints,” Computer Systems Laboratory, Stanford University, Technical Report No. 83-255, December 1983.

[GuF86] A. Gupta, C. Forgy, A. Newell, and R. Wedig, “Parallel Algorithm and Architectures for Rule-Based Systems,” Proceedings of the 13th Symposium on Computer Architecture, June 1986.

[GuW80] J. Gurd, I. Watson and J. Glauert, *A Multilayered Data Flow Computer Architecture,* Technical Report, University of Manchester, March 1980.

[HaJ86] D. T. Harper III and J. R. Jump, “Performance Evaluation of Vector Accesses in Parallel Memories Using a Skewed Storage Scheme,” Proceedings of the 13th Symposium on Computer Architecture, June 1986, pages 324-328.

[HiE86] M. D. Hill, S. J. Eggers, J. R. Larus, G. S. Taylor, G. Adams, B. K. Bose, G. A. Gibson, P. M. Hansen, J. Keller, S. I. Kong, C. G. Lee, D. Lee, J. M. Pendleton, S. A. Ritchie, D. A. Wood, B. G. Zorn, P. N. Hilfinger, D. A. Hodges, R. H. Katz, J. Ousterhout, and D. A. Patterson, *SPUR: A VLSI Multiprocessor Workstation,* Technical Report, Number UCB/CSD 86/273, University of California at Berkeley, 1986.

[HoC85] M. Horowitz and P. Chow, “The MIPS-X Microprocessor,” Proceedings of Wescon 1985, 1985.

[Hoa78] C. A. R. Hoare, “Communicating Sequential Processes,” Communications of the ACM, Volume 21, Number 8, August 1978.

[Hof79] Hoffman, M., “Development of a Voice Funnel System,” Various Quarterly Technical Reports,” Bolt Beranek and Newman Inc., 1979-1983.

[Ian82] R. A. Iannucci, “Implementation Strategies for a Tagged-Token Data Flow Machine,” Computation Structures Group Memo, Number 218, Massachusetts Institule of Technology, June 1982.

[Inm84] INMOS Limited, *OCCAM Programming Manual*, Prentice-Hall, Englewood Cliffs, New Jersey, 1984.

[Int85] “The ISPC”, Intel Corporation, 1985.

[Joh75] *Yacc — Yet Another Compiler Compiler,* Bell Laboratories, New Jersey, Computing Science Technical Report, Number 32, 1975.

[Jor87] *The Force*, Technical Report, University of Colorado, January 1987.

[KAI85] “Mini-KAP/AF,” Kuck and Associates, Inc., new product release, 1985.

[KeS81] R. M. Keller and M. R. Sleep, “Applicative Caching: Programmer Control of Object Sharing and Lifetime in Distributed Implementations of Applicative Languages,” ACM 0-89791-060-5/81-10/0131, 1981, pages 131-140.

[Ker81] B. W. Kernighan, *Why Pascal is Not My Favorite Programming Language*, Computing Science Technical Report Number 100, Bell Laboratories, New Jersey, July 18, 1981.

[KlL85] D. Klappholz, Y. Liao, D. J. Wang, A. Brodsky, and A. Omondi, “Toward a Hybrid Data-Flow/Control-Flow MIMD Architecture,” CH2149-3/85/0000/0010 IEEE, 1985, pages 10-15.

[Kla80] D. Klappholz, “An Improved Design for a Stochastically Conflict-Free Memory/Interconnection System,” Proceedings of the 14th Asilomar Conference on Circuits, Systems, and Computers, November 1980.

[Kle83] M. Klerer, personal communication, The Polytechnic Institute of New York, Brooklyn, New York, 1983.

[KrA82] D. W. Krume and D. H. Ackley, “A Practical Method for Code Generation Based on Exhaustive Search,” Proceedings of the SIGPLAN ’82 Symposium on Compiler Construction, June 1982, pages 185-196.

[KuD86] D. J. Kuck, E. S. Davidson, D. H. Lawrie, and A. H. Sameh, “Parallel Supercomputing Today and the Cedar Approach,” SCIENCE, Volume 231, February 1986, pages 967-974.

[KuM72] D. J. Kuck, Y. Muraoka, and S-C. Chen, “On the Number of Operations Simultaneously Executable in Fortran-Like Programs and Their Resulting Speedup,” IEEE Transactions on Computers, December 1972.

[KuS84] D. J. Kuck, A. H. Sameb, R. Cytron, A. V. Veidenbaum, C. D. Polychronopoulos, G. Lee, T. McDaniel, B. R. Leasure, C. Beckman, J. R. B. Davies, and C. P. Kruskal, “The Effects of Program Restructuring, Algorithm Change, and Architecture Choice on Program Performance,” IEEE Proceedings of the 1984 International Conference on Parallel Processing, August 1984.

[KuS85] J. T. Kuehn and H. J. Siegel, “Extensions to the C Programming Language for SIMD/MIMD Parallelism,” Proceedings of the 1985 International Conference on Parallel Processing, August 1985, pages 232-235.

[Kuc68] D. J. Kuck, “ILLIAC IV Software and Application Programming,” IEEE Transactions on Computers, Volume C-17, Number 8, August 1968, pages 758-770.

[Kuc78] D. J. Kuck, *The Structure of Computers and Computations*, Volume 1, John Wiley and Sons, New York, 1978.

[LaL75] D. H. Lawrie, T. Layman, D. Baer, and J. M. Randal, “The Glypnir Language,” Communications of the ACM, Volume 18, Number 3, March 1975.

[LeC79] B. W. Leverett, R. G. G. Cattell, S. O. Hobbs, J. M. Newcomer, A. H. Reiner, B. R. Schatz, and W. A. Wulf, “An Overview of the Production Quality Compiler-Compiler Project,” Technical Report, Number CMU-CS-79-105, Carnegie-Mellon University, 1979.

[LeK86] G. Lee, C. P. Kruskal, and D. J. Kuck, “The Effectiveness of Computing in Shared Memory Parallel Computers in the Presence of ’Hot Spots’,” Technical Report, Number CSRD 547, University of Illinois at Urbana-Champaign, 1985.

[Li85] Z. Li, *A Technique for Reducing Data Synchronization in Multiprocessed Loops*, MS Thesis, University of Illinois at Urbana-Champaign, May 1985.

[LiS85] K-C. Li, and H. Schwetman, “Vector C — A Vector Processing Language,” presented to ANSI X3J11 Committee, April 1985.

[LuB80] S. F. Lundstrom and G. H. Barnes, “A Controllable MIMD Architecture,” Proceedings of the 1980 International Conference on Parallel Processing, 1980, pages 165-173.

[Mac85] T. MacDonald, “C Vector Syntax,” presented to ANSI X3J11 Committee, April 1985.

[McH86] S. McFarling and J. Hennessy, “Reducing the Cost of Branches,” 13th International Symposium on Computer Architecture, June 1986.

[McS82] J. R. McGraw and S. K. Skedzielewski, “Streams and Iteration in VAL: Additions to a Data Flow Language,” CH1802-2/82/0000/0730 IEEE, 1982, pages 730-739.

[McS85] J. R. McGraw, S. K. Skedzielewski, S. Allan, R. Oldehoeft, J. Glauert, C. Kirkham, B. Noyce, and R. Thomas, *SISAL: Streams and Iteration in a Single Assignment Language, Language Reference Manual, Version 1.2*, Computing Research Group, Lawrence Livermore National Laboratory, March 1985.

[MeV85] P. Mehrotra and J. Van Rosendale, “The Blaze Language: A Parallel Language for Scientific Programming,” NASA Contractor Report, ICASE Report Number 85-29, May 1985.

[MiF85] V. Milutinovic, D. Fura, and W. Helbig, “An Introduction to GaAs Microprocessor Architecture for VLSI,” IEEE Computer, September 1985.

[MiP86] S. P. Midkiff and D. A. Padua, *Compiler Generated Synchronization for DO Loops*, Technical Report, University of Illinois at Urbana-Champaign, Number CSRD 554, 1986.

[Mis75] D. P. Misunas, *A Computer Architecture for Data-Flow Computation,* Technical Report, Number MIT/LCS/TM-100, Massachusetts Institule of Technology, July 1975.

[Moo82] I. W. Moor, “An Applicative Compiler for a Parallel Machine,” ACM 0-89791-074-5/82/006/0284, 1982, pages 284-293.

[NCu85] *NCube Ten*, NCube Corporation, Beaverton, Oregon, 1985.

[Nic85] A. Nicolau, “Uniform Parallelism Exploitation in Ordinary Programs,” 1985 International Conference on Parallel Processing, August 1985, pages 614-618.

[Nor86] A. Norton, “Parallel Programming Models for Highly Parallel Processing on RP3,” Presented at NASA/ICASE Parallel Language and Environment Workshop, November 1986.

[Omo84] A. Omondi, *A Formal Translation Algorithm for a Data Flow Language*, Internal Report, Polytechnic Institute of New York, 1984.

[Ost85] A. Osterhaug, *Guide To Parallel Programming On Sequent Computer Systems*, Sequent Computer Systems, Inc., Beaverton, Oregon, 1985.

[PaK80] D. A. Padua, D. J. Kuck, and D. H. Lawrie, “High-Speed Multiprocessors and Compilation Techniques,” IEEE Transactions on Computers, September 1980.

[PaK86] H-C. Park, D. Klappholz, and H. Dietz, “A Single Stage MIMD Architecture Incorporating Smart Switching Nodes and an MIMD RISC CPU,” Internal Report, Stevens Institute of Technology, 1986.

[Per79] R. H. Perrott, “A Language for Array and Vector Processors,” ACM Transactions on Programming Languages and Systems, Volume 1, Number 2, October 1979, pages 177-195.

[PfB85] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss, “The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture,” Proceedings 1985 International Conference on Parallel Processing, August 1985, pages 764-771.

[PfN85] G. F. Pfister and V. A. Norton, “Hot Spot” Contention and Combining in Multistage Interconnection Networks,” 0190-3918/85/0000/0790 IEEE, 1985, pages 790-797.

[PoK86] C. D. Polychronopoulos, D. J. Kuck, and D. A. Padua, *Execution of Parallel Loops on Parallel Processor Systems*, University of Illinois at Urbana-Champaign, Number CSRD 552, 1986.

[Pra85] T. W. Pratt, “Pisces: An Environment for Parallel Scientific Computation,” IEEE Software, Volume 2, Number 4, July 1985, pages 7-20.

[Pra86] T. W. Pratt, “The PISCES 2 Parallel Programming Environment,” Presented at the NASA/ICASE Parallel Language and Environment Workshop, November 1986.

[Ree84] A. P. Reeves, “Parallel Pascal: An Extended Pascal for Parallel Computers,” Journal of Parallel and Distributed Computing, Volume 1, 1984, pages 64-80.

[RuF83] J. C. Ruttenberg and J. A. Fisher, “Lifting the Restriction of Aggregate Data Motion in Parallel Processing,” CH1879-6/83/0000/0211 IEEE, pages 211-215, 1983.

[SaH86] V. Sarkar, J. Hennessy, “Compile-Time Partitioning and Scheduling of Parallel Programs,” SIGPLAN Symposium on Compiler Construction, 1986.

[ScK86] R. G. Scarborough and H. G. Kolsky, “A Vectorizing Fortran Compiler,” IBM Journal of Research and Development, Volume 30, Number 2, March 1986.

[Sch80] J. T. Schwartz, “Ultracomputers,” ACM Transactions on Programming Languages and Systems, Volume 2, Number 4, October 1980, pages 484-521.

[SeR85] Z. Segall and L. Rudolph, “PIE: A Programming and Instrumentation Environment for Parallel Processing,” IEEE Software, Volume 2, Number 6, November 1985, pages 22-37.

[Seq84] *Balance 8000 System Technical Summary*, Sequent Computer Systems, Beaverton, Oregon, 1984.

[SiS81] H. J. Siegel et al, “PASM: A Partitionable SIMD/MIMD System for Image Processing and Pattern Recognition,” IEEE Trans. Computers, Volume C-30, Number 12, December 1981, pages 934-947.

[SiS86] H. J. Siegel, T. Schwederski, J. T. Kuehn, and N. J. Davis IV, “An Overview of the PASM Parallel Processing System,” in Tutorial: Computer Architecture, IEEE Computer Society Press, Washington DC, 1986, pages 387-407.

[SkG85] S. K. Skedzielewski and J. Glauert, *IF1: An Intermediate Form for Applicative Languages*, Lawrence Livermore National Laboratory, Computing Research Group, July 1985.

[SnS86] L. Snyder and D. Socha, “Poker on the COSMIC Cube: The First Retargetable Parallel Programming Language and Environment,” Proceedings of the 1986 International Conference on Parallel Processing, August 1986, pages 628-635.

[SoD85] G. S. Sohi, E. S. Davidson and J. H. Patel, “An Efficienct Lisp-Execution Architecture with a New Representation for List Structures,” 149-7111/85/0000/0091 IEEE, 1985, pages 91-98.

[Ste86] K. Stein, *Refined C Compiler Status Report*, Internal Report, Stevens Institute of Technology, 1986.

[Sto84] H. S. Stone, “Database Applications of the Fetch-and-Add Instruction,” IEEE Transactions on Computers, Volume C-33 Number 7, July 1984.

[SuB77] H. Sullivan, T. R. Bashkow, D. Klappholz, and L. Cohn, *CHoPP: Interim Status Report 1977*, Internal Report, Columbia University, 1977.

[TaH86] G. S. Taylor, P. N. Hilfinger, J. R. Larus, D. A. Patterson and B. G. Zorn, “Evaluation of the SPUR Lisp Architecture,” Proceedings of the 13th Annual Symposium on Computer Architecture, June 1986, pages 444-452.

[Thi86] *Introduction to Data Level Parallelism*, Techical Report, Number 86.14, Thinking Machines Corporation, April 1986.

[ToS86] S. Tomita, K. Shibayama, T. Nakata, S. Yuasa, and H. Hagiwara, “A Computer with Low-Level Parallelism QA-2 — Its Applications to 3-D Graphics and Prolog/Lisp Machines,” 13th Symposium on Computer Architecture, June 1986.

[TrI86] R. Triolet, F. Irigoin, and P. Feautrier, “Direct Parallelization of Call Statements,” SIGPLAN Symposium on Compiler Construction, 1986.

[Vei85] A. Veidenbaum, *Compiler Optimizations and Architecture Design Issues for Multiprocessors*, Ph.D. Thesis, University of Illinois at Urbana-Champaign, May 1985, pages 96-101.

[WaG82] I. Watson and J. Gurd, “A Practical Data Flow Computer,” IEEE Computer, February 1982, pages 51-57.

[Wed83] R. G. Wedig, “The Detection of Concurrency Using Structured Control Flow,” Carnegie-Mellon University, 1983.

[Wir76] N. Wirth, *Algorithms + Data Structures = Programs*, Prentice-Hall, Englewood Cliffs, New Jersey, 1976, page 79.

[Wol86] M. Wolfe, “Advanced Loop Interchanging,” pre-print extended version of a paper, Proceedings of the 1986 International Conference on Parallel Processing, 1986.

[WuJ75] W. Wulf, R. K. Johnson, C. B. Weinstock, S. O. Hobbs, and C. M. Geschke, *The Design of an Optimizing Compiler*, North Holland, New York, New York, 1975.