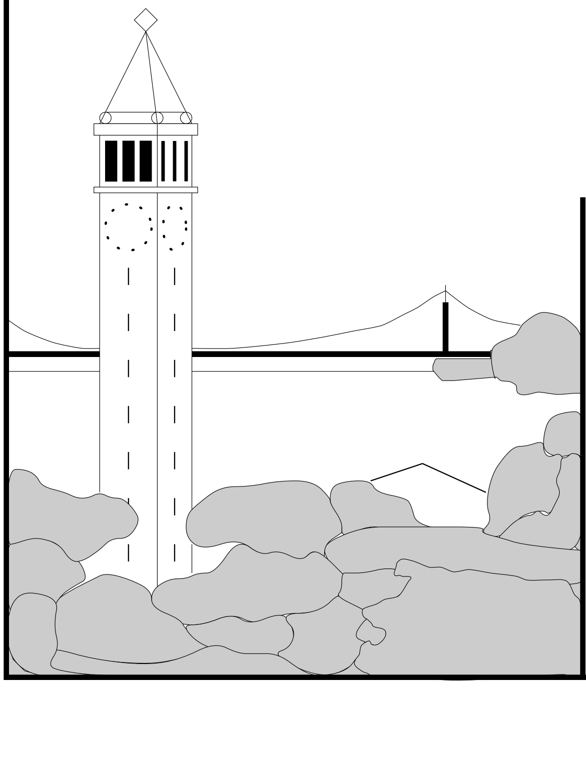
High-Level Abstractions for Symbolic Parallel Programming (Parallel Lisp Hacking Made Easy)

Kinson Ho



Report No. UCB//CSD-94-816

June 1994

Computer Science Division (EECS)
University of California
Berkeley, California 94720

High-Level Abstractions for Symbolic Parallel Programming (Parallel Lisp Hacking Made Easy)

by

Kinson Ho

B.S. (Carnegie-Mellon University) 1985
M.S. (University of California at Berkeley) 1989

A dissertation submitted in partial satisfaction of the requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION

of the

UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:

Professor Paul N. Hilfinger, Chair
Professor David E. Culler
Professor Philip Colella

1994

The dissertation of Kinson Ho is approved:

Chair Date

Date

Date

University of California at Berkeley

1994

High-Level Abstractions

for

Symbolic Parallel Programming

(Parallel Lisp Hacking Made Easy)

by

Kinson Ho

This research was supported in part by NSF Grant CCR-84-51213.

Abstract

High-Level Abstractions
for
Symbolic Parallel Programming
(Parallel Lisp Hacking Made Easy)
by
Kinson Ho
Doctor of Philosophy in Computer Science
University of California at Berkeley
Professor Paul N. Hilfinger, Chair

Symbolic applications are characterized by irregular and data-dependent patterns of control flow and data structure accesses. In this dissertation I propose a toolbox approach for writing parallel symbolic applications, including those with side effects. Programs written using the high-level abstractions in this toolbox may be executed without modification on uniprocessors and multiprocessors. The toolbox is designed and implemented by parallel programming experts, and is intended for novice parallel programmers. Most of the parallel programming issues are hidden from the programmer. Each tool of the toolbox provides a stylized sequential interface, and programs that use these tools may be parallelized by using the parallel implementation of the toolbox. These tools are divided into *parallelism abstractions* and *data-sharing abstractions*. Parallelism abstractions represent common, time-consuming operations that offer good speedup potentials for parallel implementations. Data-sharing abstractions support common side-effecting operations on shared objects, simplifying the coding of a large class of algorithms that modify shared data structures in a parallel implementation. The thesis of this research is that a small toolbox is sufficient for a large class of symbolic applications, and that there are efficient (sequential and parallel) implementations of the toolbox.

To evaluate the feasibility of the toolbox approach for symbolic parallel programming, I constructed a prototype toolbox of commonly used abstractions, and converted two significant applications to use the toolbox. The applications include a

constraint satisfaction system and a type analysis system. These applications have been developed by other authors, and contain approximately 8,000 lines of Common Lisp each. They are much larger than what other researchers have typically used to evaluate their symbolic parallel programming systems.

My experience with these two applications suggests that the toolbox approach for symbolic parallel programming is indeed successful. Many toolbox abstractions are common to the applications, and the toolbox version of either application may be ported between uniprocessors and multiprocessors without modification. Furthermore, the toolbox may be used by novice parallel programmers to parallelize an application relatively easily, since all the low-level parallel programming details are hidden by the parallel toolbox implementation.

An unexpected discovery from this research is an optimistic method for implementing discrete relaxation in parallel, in problem domains where monotonicity is a *necessary* correctness condition. This scheme is developed in the context of parallelizing the constraint application, but is applicable to discrete relaxation in general.

Contents

List of Figures vii

List of Tables ix

1 Introduction 1

I Abstractions 11

2 Parallelism Abstractions 13
- 2.1 Fixed-Point Computations 14
  - 2.1.1 Formulation of Fixed-Point Computation 18
  - 2.1.2 General Use of Fixed-Point 19
  - 2.1.3 Basic Use of Fixed-Point Library 19
  - 2.1.4 Application-Specific Scheduling 21
  - 2.1.5 Summary 23
- 2.2 Traversals of Directed Acyclic Graphs 25
  - 2.2.1 Dag-Traversal 25
  - 2.2.2 Dag-Traversal for Fixed-Point Computation 26
  - 2.2.3 Summary 29
- 2.3 Summary 31

3 Data-Sharing Abstractions 32
- 3.1 Side Effects on Shared Data 32
- 3.2 Writing Parallel Code: Shared Data Accesses 35
- 3.3 Simple Concurrent Datatype 36
  - 3.3.1 Example Simple Concurrent Datatype: Counter 37
  - 3.3.2 Conditional Update for Concurrent Datatypes 37
- 3.4 Dictionary 38
  - 3.4.1 Dictionary Interface 39
  - 3.4.2 Sample Use of Dictionary 40
  - 3.4.3 Split-fn implementation in Common Lisp 41

3.5 Autolock 45
3.5.1 Autolock Interface 46
3.5.2 Sample Use of Autolocks 48
3.6 Optimistic Read-Modify-Write 48
3.6.1 Optimistic Read-Modify-Write Interface 50
3.6.2 Sample Uses of `ORMW` 52
3.7 Per-Thread Location 53
3.7.1 False Sharing 53
3.7.2 False Sharing in Shared Structures 54
3.7.3 Pdefstruct Interface 55
3.7.4 Sample Use of Pdefstruct: A Unique-Id Generator 57
3.8 Summary 58

II Sample Applications 61

4 Application: CONSAT 64
4.1 CONSAT Overview 64
4.2 CONSAT as a Parallel Application 72
4.3 Parallelizing CONSAT 73
4.3.1 Sequential CONSAT 73
4.3.2 Introducing Parallelism 73
4.3.3 Application-Specific Scheduling 75
4.3.4 Shared Data Accesses 75
4.4 Summary 77

5 Optimistic Parallel Discrete Relaxation 81
5.1 Discrete Relaxation 82
5.2 Discrete Relaxation and Fixed-Point Computations 87
5.3 CONSAT as Discrete Relaxation 87
5.4 Summary 88

6 Application: RC 89
6.1 Introduction 89
6.2 RC Overview 89
6.3 RC as a Fixed-Point Computation 93
6.3.1 Notation 93
6.3.2 Fixed-Point Computation 93
6.3.3 Introducing Parallelism 94
6.3.4 RC as a Dag Traversal 95
6.4 Shared Data Accesses 96
6.4.1 P-sets 97

6.4.2 Interning Productions and Nonterminals 99
6.4.3 False Sharing in RC 99
6.5 Summary 100

III Immplementation and Evaluation 103

7 Toolbox Implementation 105
7.1 Multiprocessing SPUR Lisp 105
7.2 CLiP 106
7.3 Parallel Lisp Features Used 106
7.4 Changes made to CLiP 107
7.5 CLiP Allocation Bottlenecks 108
7.6 User-Level Object Allocation 108
7.7 Future Improvements to CLiP 110

8 Performance Tuning 114
8.1 Locating Performance Bottlenecks 114
8.1.1 Why is a Parallel Program Slow? 114
8.1.2 Locating Performance Bottlenecks 115
8.2 Removing Performance Bottlenecks 117
8.3 Speedup and Garbage Collection 119
8.4 Multiprocessor Used 119
8.4.1 Measurement Conditions 119

9 Evaluation 121
9.1 Introduction 121
9.2 The Toolbox Experience 123
9.2.1 CONSAT 123
9.2.2 RC 125
9.3 CONSAT Performance 127
9.3.1 Sample Constraint Network 127
9.3.2 Uniprocessor Performance 127
9.3.3 Multiprocessor Performance 127
9.4 RC Performance 130
9.4.1 Sample Input Program 130
9.4.2 Uniprocessor Performance 130
9.4.3 Multiprocessor Performance 130

10 Related Work 132
10.1 Explicitly Parallel Programming Systems 133
10.2 Parallelism Abstractions 136
10.2.1 Fixed-Point 136

10.2.2 Dag-Traversal 138
10.3 Data-Sharing Abstractions 138
10.3.1 Other Approaches to Concurrent Accesses 138
10.3.2 Per-Component Comparisons 142
10.3.3 Architectural Support for Data-Sharing Abstractions 147
10.4 Asynchronous Iterations 148
10.5 Parallel Constraint Satisfaction 149
10.6 `Dag-Traversal` 150
10.6.1 Circular Grammar Evaluator 150
10.6.2 Hybrid Dataflow Analysis Algorithms 150

11 Conclusion **151**
11.1 Contributions 151
11.2 Future Directions 155

Bibliography **156**

List of Figures

1.1 Parallel Programming: Without Toolbox 4
1.2 Parallel Programming: With Toolbox 5
1.3 Parallelism Abstraction: Sequential & Parallel Implementations . . . 6

2.1 Sequential Connected Component Algorithm 14
2.2 Operation performed at every node 15
2.3 Fixed-Point formulation of Connected Component Algorithm 16
2.4 Fixed-point version of operation performed at every node. 16

3.1 Sequential Dictionary Insert 41
3.2 Simple Parallel Dictionary Insert 42
3.3 Optimized Parallel Dictionary Insert 43
3.4 Insert Using `Conditional-Insert-Dict` 44
3.5 Use of Autolocks 46
3.6 Sequential mutation of `Obj` 48
3.7 Parallel mutation of `Obj` without autolock 49
3.8 Sequential & Parallel mutation of `Obj` with autolock 49
3.9 Conceptual Parallel Implementation of `ORMW` 52
3.10 Semantics of `ORMW` 53
3.11 Linear Time Set Union Algorithm 54
3.12 Unique-Id Generator 60

4.1 Constraint Network A 65
4.2 Constraint Network B 66
4.3 Activations of constraint network A 67
4.4 Activations of constraint network B 68
4.5 Globally Consistent Solutions of Constraint Network A 69
4.6 Sequential CONSAT 73
4.7 Activate-Constraint 74
4.8 Activate-Constraint using Monotonic Asynchronous Iteration 79
4.9 Activate-Constraint using `ORMW` 80

6.1 Sequential RC 93
6.2 `Activate-Node` 94
6.3 RC using `Fixed-Point` 94
6.4 RC using `Dag-Traversal` 96
6.5 Locking details of `Activate-Node` 97
6.6 Locking in `Activate-Node` using `ORMW` 98

List of Tables

3.1 Sizes of Application Programs and Toolbox 63

7.1 Cons Allocation 109
7.2 Array Allocation 110
7.3 Structure Allocation 111

9.1 Performance of CONSAT for Stair(5) using Autolocks 128
9.2 Performance of CONSAT for Stair(5) using `ORMW` 128
9.3 Performance of CONSAT for Stair(5) using `ORMW` 129
9.4 Performance of RC for `(Chainx 20 10)` 131

Acknowledgements

I would like to thank a number of people for making this research possible, and for making my years at Berkeley an enjoyable experience.

My advisor, Paul Hilfinger, gave me complete freedom in pursuing this research, but was always available for consultation when I needed it. Paul emphasized the distinction between an abstraction and its implementation, and insisted over my objection that the toolbox abstractions should have sequential interfaces. This dissertation is proof that he is right. Paul's comments on the drafts of this dissertation have also improved it significantly.

Richard Fateman, Paul Hilfinger, David Culler and Phil Colella served on the qualifying examination committee. Paul Hilfinger, David Culler and Phil Colella served on the thesis committee. I thank David Culler and Phil Colella for giving me an outsider's perspective on this work, and for reading the dissertation with very short notice. David Culler gave me some very thoughtful comments about the methodology of the research, and improved this dissertation significantly.

I am deeply indebted to a few people who helped me enormously in this research. Hans Guesgen gave me the source code of CONSAT, and helped me port the system to Common Lisp. Hans answered my endless questions about CONSAT internals, and gave me very useful and timely comments on drafts of the paper on discrete relaxation. It is a tribute to the Internet that most of this collaboration was performed remotely using electronic mail. Edward Wang took time from his own research to help me with the parallelization of RC. Ed spent countless hours explaining the internals of RC to me, and even modified RC so that the result computed by the parallel implementation could be verified. In addition, Ed gave me some very detailed comments about my formulation of discrete relaxation in Chapter 5. Franz Inc. gave me access to the source code of CLiP. George Jacob deserves special thanks for making the numerous modifications to CLiP I requested, and for answering my tons of questions about the details of the system.

Bert Halstead and Takayasu Ito invited me to the Parallel Symbolic Computing Workshop in Cambridge, and gave me very helpful comments about an earlier paper on data-sharing abstractions. Suresh Krishna performed the impossible task of translating Miellou's paper on chaotic iterations from French into English. Kathy Yelick and Chu-Cheow Lim read drafts of my paper on discrete relaxation. Their comments improved the paper significantly.

Jon Forrest maintained Hermes for the Postgres project, and ensured that my work was not disrupted by the transition from Hermes to Soda. Tom Holub of CSUA provided assistance with matters on Soda. Ed Wang kept our cluster of workstations running flawlessly. Kathryn Crabtree, Liza Gabato and Theresa Lessard-Smith of the Computer Science Division helped me with various matters over the years. My sincere thanks to all of them.

Former and current occupants of 608-1 Evans—James Larus, Ben Zorn, Ken Rimey, Luigi Semenzato, Edward Wang, Allen Downey, Geoff Pike and Doug Hauge—made the office a fun and exciting place to work. My family has been a constant source of love, inspiration and support over the years. Finally, I thank Su for putting up with the lifestyle of a disorganized graduate student, for repeatedly believing my projection of "one more year", and for dragging me away from my work when I needed the distraction. Life would have been a lot less fun without her. This dissertation is dedicated to the memories of my grandmothers, who unfortunately are not around to see the completion of this work.

Chapter 1

Introduction

As uniprocessor architectures begin to approach their practical performance limits, multiprocessing, the use of many processors to solve a problem together, is rapidly becoming a cost-effective way of achieving higher performance. Many commercial multiprocessors are now available – Cray-T3D, Intel i860, NCube, Thinking Machines CM-5, Maspar MP-1/2, Sequent Symmetry, Kendall Square Research KSR-1, and multiprocessor workstations from Silicon Graphics and Sun Microsystems.

The Problem I am interested in the efficient execution of *symbolic* programs on shared-address space (MIMD) multiprocessors. Symbolic applications are characterized by irregular and data-dependent patterns of control flow and data structure accesses. They include optimizing compilers, VLSI CAD programs for logic synthesis and simulation, expert systems, AI programs and rapid prototyping systems. These programs frequently take a substantial amount of time to execute, and can definitely benefit from speedups arising from the use of parallelism.

Shared-address space multiprocessors (a superset of shared-memory multiprocessors) are attractive to application programmers because of the simple and intuitive programming model they support. In addition, recent innovations in multiprocessor design have resulted in shared-address space multiprocessors that are scalable, such as the KSR-1 [15].

Why is it hard to get symbolic applications to run on multiprocessors? Techniques have been developed for the automatic detection of parallelism in FORTRAN programs. Parallelizing FORTRAN compilers usually attempt to schedule different iterations of a loop for execution in parallel on different processors, taking advantage of the regularity of control flow in FORTRAN programs. These techniques are unlikely to have the same success with symbolic applications because of the pervasive use of pointers and indefinite iteration in languages such as Lisp commonly used for symbolic applications [46], and because of the dynamic and data-dependent nature of symbolic applications.

On the other hand, parallelizing a real symbolic application by hand is usually a difficult task. In a sequential program there is a total ordering on all the events (state changes) associated with one execution of the program, and these events are perfectly reproducible. In a parallel program, however, events obey much looser partial orders. Indeterminacy manifests itself in the form of irreproducible program behavior, making parallel programs difficult to debug. In some cases seemingly innocuous changes, such as the addition of print statements for debugging purposes, may also affect the execution of the program. Issues such as race, deadlock avoidance, communication and synchronization are completely foreign concepts to an inexperienced parallel programmer. To make matters worse, techniques for writing efficient sequential programs may be very different from techniques for writing efficient parallel programs. (For example, efficient sequential code tends to reuse results to avoid recomputation, while efficient parallel code may recompute results to avoid communication or contention.) Finally, certain features of sequential code, such as the use of global variables to pass data, may have unintended effects if used in parallel programs. As a result of these difficulties with parallel programming, symbolic applications are usually ported to parallel machines in research environments only. They are not ported to parallel machines in production environments because of a lack of experienced parallel programmers, and because of the large programmer effort required.

To complicate matters further, the source code for the sequential and parallel implementations of the same algorithm are often very different. This creates a program maintenance nightmare, especially if the sequential implementation is still needed for portability and other reasons. For large applications this problem may be serious enough to justify *not* pursuing a parallel implementation, even when substantial performance improvements are expected.

Parallel Programming Toolbox In this dissertation I describe a toolbox of high-level abstractions (or libraries) for writing symbolic applications, including those with side effects. Programs written using this toolbox may be executed without modification on uniprocessors and multiprocessors. The toolbox is designed and implemented by parallel programming experts, and is intended for programmers who need to use their multiprocessing hardware effectively—but are not interested or experienced in parallel programming. It is not intended for applications needing optimal performance on a given platform. Most of the parallel programming issues are hidden from the programmer. This toolbox approach is analogous to the performance-simplicity tradeoff between standard cell and full-custom styles of VLSI design. The abstractions of the toolbox may be divided into *parallelism abstractions* and *data-sharing abstractions*.

Parallelism Abstractions Many algorithms specify a set of computations that may be performed in any order. In a typical sequential implementation without using the toolbox, these computations are performed in some arbitrary but fixed order. As an example, the breadth-first traversal of a tree may visit the immediate successor nodes of a given node in any order. In general, a set of unordered computations may be executed concurrently if they are *linearizable* [27]. The programmer must determine that the operations are indeed linearizable by using his high-level knowledge about the algorithm. The toolbox does not provide any support for this step.

The sources of parallelism in a set of unordered computations may be captured by a parallelism abstraction. A parallelism abstraction represents a common, high-level and time-consuming operation that offers good speedup potentials for a parallel implementation. For example, the *dag-traversal* abstraction visits all the nodes of a dag such that a node is visited only if all its parents have been visited. In a sequential implementation the nodes are visited in some topological-sort order. In a

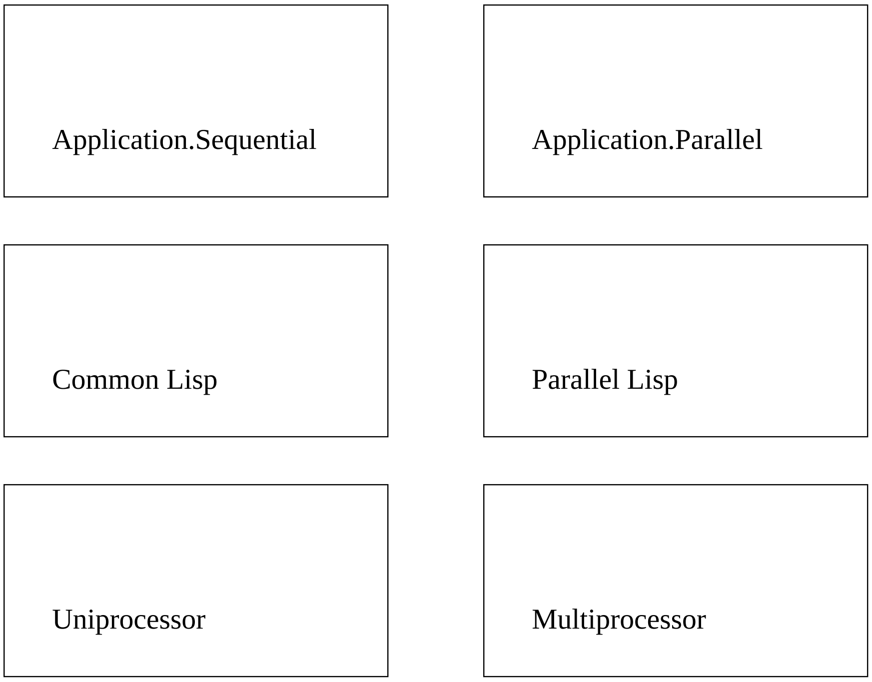


Figure 1.1: Parallel Programming: Without Toolbox

parallel implementation multiple nodes may be visited concurrently. Other parallelism abstractions may identify the sources of parallelism in heuristic searches or work queue-style computations.

A parallelism abstraction has a sequential (or single-threaded), parameterizable interface. The programmer uses this interface to *specify* all the sources of parallelism in an algorithm. In a parallel implementation of the abstraction these sources of parallelism may be exploited internally (Figure 1.3). However, all the low-level parallel programming details (including the creation of threads and termination detection) are hidden from the programmer.

A parallelism abstraction has identical semantics for its sequential and parallel implementations. To provide maximum flexibility for the implementations, nondeterministic results may be specified. For example, a search abstraction may return *any* valid solution, and consequently the sequential and parallel solutions may be different.

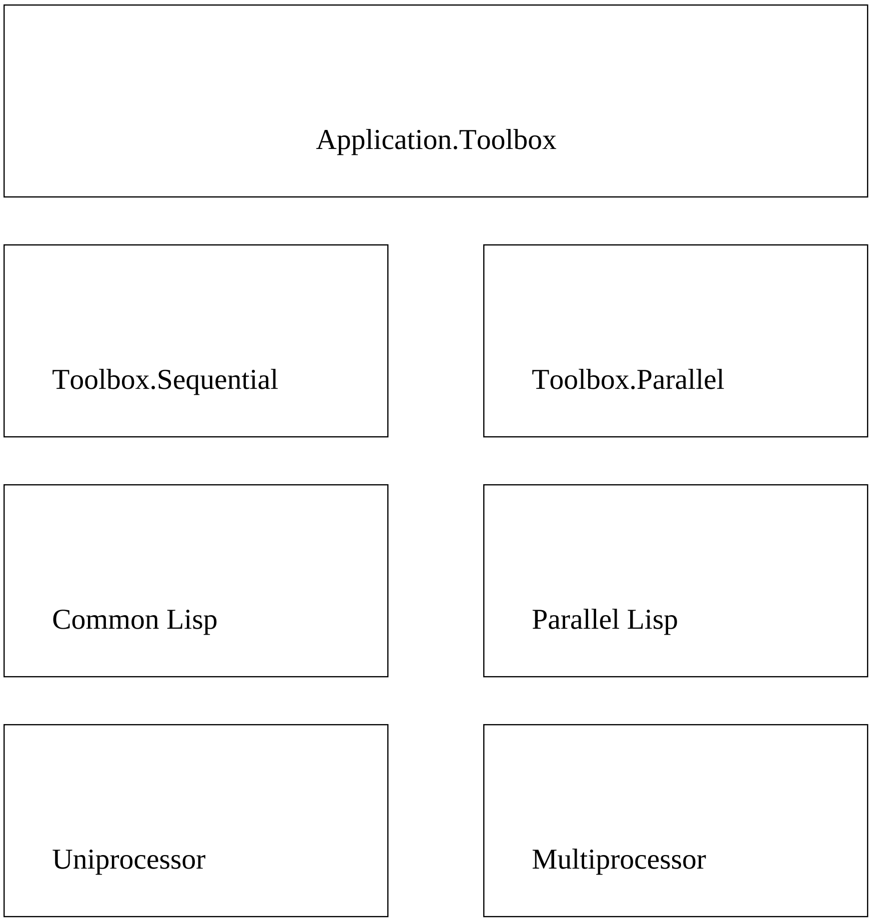


Figure 1.2: Parallel Programming: With Toolbox

Data-Sharing Abstractions In the parallel implementation of a parallelism abstraction, concurrent operations may access and modify shared data structures of the application. In general, the programmer has to use his high-level understanding of the application to determine if the interleaving of the concurrent operations (with side effects) produces the desired result. While this question is difficult to answer in the general case, serialization of accesses to shared data objects is a sufficient correctness condition in the simple (but common) case. (Serialization of shared data accesses does not necessarily guarantee correctness in all cases. For example, concurrent operations on a dictionary that supports atomic lookup and insert operations may still lead to races because the lookup and insert operations are not performed in a single critical section.)

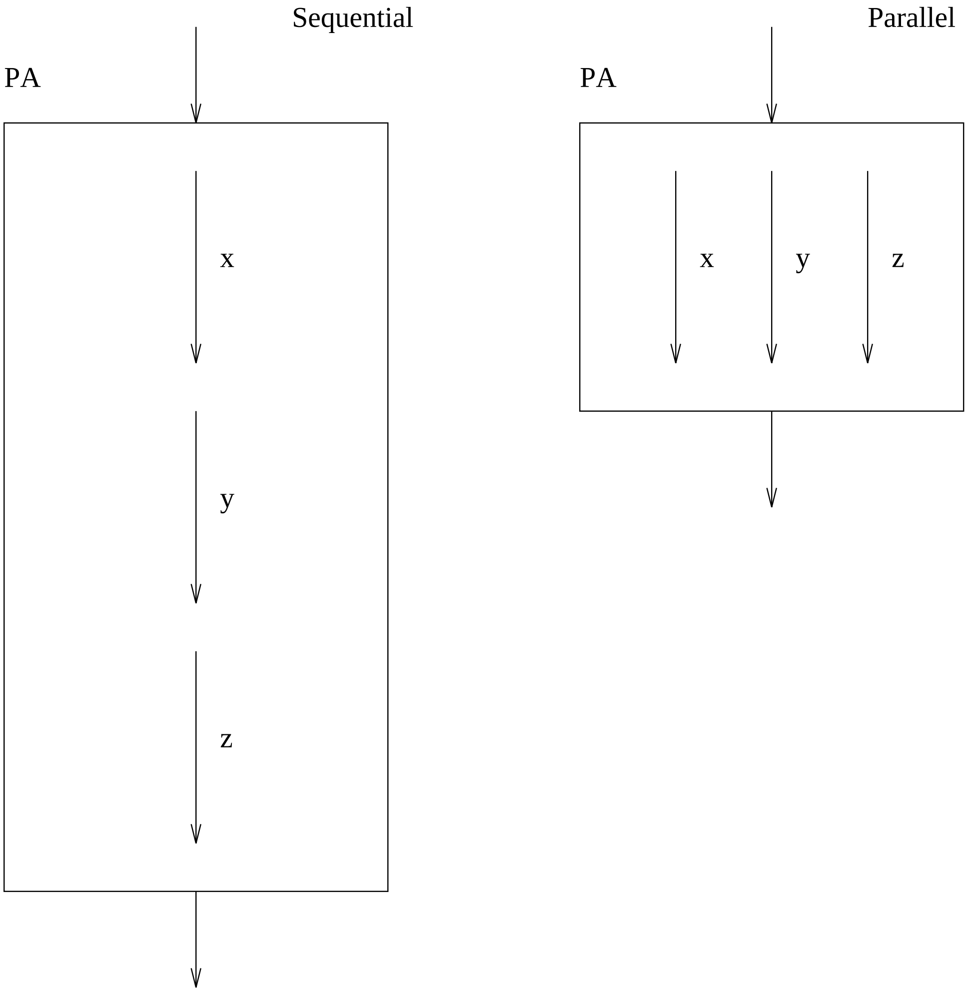


Figure 1.3: Parallelism Abstraction: Sequential & Parallel Implementations. `x`, `y`, `z` are the sources of parallelism expressed using the `PA`.

Data-sharing abstractions support common side-effecting operations on shared objects, simplifying the coding of a large class of algorithms that modify shared data structures in a parallel implementation. In the simplest case a data-sharing abstraction may be an abstract datatype whose operations are implicitly synchronized. Another abstraction provides a mechanism for associating a mutex lock with a shared object transparently without knowing or altering the representation of the object. If contention for the object is significant, the mutex lock may sometimes be replaced an abstraction that supports optimistic concurrency control for improved performance. Finally, *false sharing* of structure fields is eliminated by an abstraction that provides one private field for each thread that falsely shares the field.

Operations on a data-sharing abstraction associated with a shared object are invoked in the same way as operations on sequential abstract datatypes. In a parallel implementation of the abstraction all the low-level synchronization details (such as retries in optimistic schemes) are hidden from the programmer.

Using the Toolbox To use the toolbox the programmer first identifies the sources of parallelism in an application, and expresses them using one or more parallelism abstractions. Accesses to shared data structures are also identified, and are modified to use the data-sharing abstractions. In this way, all the details of parallel programming are hidden by the toolbox implementation. As this approach requires the programmer to rewrite an application to use the toolbox abstractions, it is not applicable to unmodified dusty deck programs.

The toolbox approach for parallel programming is not *universal* in that a given toolbox is not guaranteed to contain all the abstractions necessary for a given application. If the sources of parallelism or pattern of data accesses of an application do not fit the model provided by any of the toolbox abstractions, the toolbox does not simplify the parallelization of the application. If this happens the application programmer may either rewrite the application using an explicitly parallel programming language, or requests that the necessary abstractions be added to the toolbox by a parallel programming expert. An abstraction should be added to the toolbox if it represents a general, common style of computation or shared data access, and has an interface that is meaningful to novice parallel programmers. The toolbox approach for symbolic parallel programming is cost-effective if a small number of toolbox abstractions may be used for a large number of applications.

Toolbox and Parallel Programming Experts The toolbox may also be used by an expert parallel programmer as a rapid-prototyping tool. An existing sequential application may be converted to run on a parallel machine relatively easily by rewriting it to use the toolbox. The programmer may then concentrate his efforts on locating and removing the performance bottlenecks of the toolbox version of the application. To simplify performance tuning, many toolbox abstractions provide performance hooks to the programmer, who may use application-specific knowledge to optimize the performance of the application. For example, a parallelism abstraction may allow the programmer to provide a scheduler module, while a data-sharing abstraction may allow the programmer to specify the granularity of locking. If the performance of the toolbox version of the application is still unsatisfactory, the expert may then concentrate his efforts on rewriting (only) those sections of code that are crucial for performance as explicitly parallel code. This is somewhat analogous to the limited use of assembly language routines in a program written in a high-level language.

Toolbox Implementation The toolbox is implemented using CLiP [17], a superset of Common Lisp with multiprocessing extensions from SPUR Lisp [76], and currently runs on Sequent Symmetry shared-memory multiprocessors. Lisp is chosen because it provides an easy environment for experiments. However, the ideas developed here are not specific to Lisp, and are directly applicable to other languages.

Thesis The thesis of this research is that a small toolbox is sufficient for a large class of symbolic applications, and that there are efficient sequential and parallel implementations of the toolbox.

Contributions In this dissertation I proposed a toolbox approach for symbolic parallel programming. To evaluate the feasibility of the toolbox approach, I have identified a number of parallelism abstractions and data-sharing abstractions common to symbolic applications, and constructed a prototype toolbox of these abstractions. I converted two significant applications to use the toolbox. The applications include a constraint satisfaction system and a type analysis system. These applications have been developed by other authors, and contain approximately 8,000 lines of Common Lisp each. They are much larger than what other researchers have typically used to evaluate their symbolic parallel programming systems.

My experience with these two applications suggests that the toolbox approach for symbolic parallel programming is indeed successful. In particular, many toolbox abstractions are common to the two applications. In addition, the toolbox version of either application may be ported between uniprocessors and multiprocessors by selecting the appropriate toolbox implementation. No change to the source code of the application is required. Furthermore, the toolbox may be used by novice parallel programmers to parallelize an application relatively easily, as all the low-level parallel programming details are hidden by the parallel toolbox implementation.

An unexpected discovery from this research is an optimistic method for implementing discrete relaxation ($X \leftarrow f(X)$) in parallel, in problem domains where monotonicity is a *necessary* correctness condition. This scheme is developed in the context of parallelizing a constraint satisfaction system, but is applicable to discrete relaxation in general.

Dissertation Overview Chapter 2 defines two generally useful parallelism abstractions, `Fixed-Point` and `Dag-Traversal`. `Fixed-Point` is a work queue-style parallelism abstraction that may be used for a large class of applications. `Dag-Traversal` is a parallelism abstraction that may be used to traverse multiple nodes of a directed acyclic graph concurrently, and abstracts the computation performed by many graph algorithms. Data-sharing abstractions simplify the parallelization of applications with side effects, and are discussed in detail in Chapter 3. CONSAT and RC, the two applications that are used as test cases for the toolbox, are described in Chapters 4 and 6 respectively. CONSAT is a generalized constraint satisfaction system that computes globally consistent solutions [22, 21], and RC is a system that performs conservative type determination of Lisp programs using dataflow analysis that is commonly done by optimizing compilers [70]. Chapter 5 introduces *monotonic asynchronous iteration*, an optimistic technique for performing parallel discrete relaxation efficiently, and illustrates the use of this technique in the parallel toolbox version of CONSAT. Chapter 7 presents a short overview of the implementation of the toolbox, including a short description of CLiP, the parallel Lisp system used to implement the toolbox. The procedure for tuning the performance of a toolbox-based application is described in Chapter 8. In Chapter 9 I reflect on my experience of converting CONSAT and RC to use the toolbox, and extrapolate this experience to the feasibility of the toolbox approach for symbolic parallel programming by novice parallel programmers in general. Performance numbers for the toolbox versions of CONSAT and RC will also be given. Chapter 10 compares the toolbox approach for symbolic parallel programming to other symbolic parallel programming systems, and discusses systems that are similar to CONSAT and RC. Finally, I will summarize this research in Chapter 11, and close with some directions for future research.

Part I

Abstractions

Chapter 2

Parallelism Abstractions

In this chapter I introduce two generally useful parallelism abstractions, `Fixed-Point` and `Dag-Traversal`. Other parallelism abstractions may be designed for heuristic searches, discrete-event simulations and simulated annealing. `Fixed-Point` captures the potential parallelism in work queue-style computations, and may be used with an application-specific scheduler for improved performance. `Dag-Traversal` traverses multiple nodes of a directed acyclic graph concurrently while respecting certain precedence constraints, and abstracts the computation performed by many graph algorithms.

2.1 Fixed-Point Computations

Iterative algorithms for finding the fixed point of a computation appear in many different applications, including constraint-satisfaction systems, program-dataflow analysis, and certain graph algorithms. These algorithms are all instances of the following abstract algorithm: "$X \leftarrow X_0$; Repeat $X \leftarrow f(X)$ until no change in X". As an example of a fixed-point computation, consider an algorithm for finding all the nodes in a connected component of an undirected graph, starting from a node `Root`.

```
function Connected-Component (Root)
    ;; returns a set of all the nodes reachable from Root
    Reachable-Nodes := Make-Set()
    ;; Invariant: Reachable-Nodes is a subset of the nodes
    ;; reachable from Root

    Nodes-to-Visit := Make-PQ()
    ;; Invariant: All reachable nodes are in Reachable-Nodes,
    ;; or are reachable from some member of Nodes-to-Visit
    ;; without traversing Reachable-Nodes

    PQ-Insert(Root, Nodes-to-Visit)

    while not PQ-Empty-p(Nodes-to-Visit) do
        Node := PQ-Delete(Nodes-to-Visit)
        Visit-Node(Node)

    return Reachable-Nodes
```

Figure 2.1: Sequential Connected Component Algorithm

Figures 2.1 and 2.2 show a typical sequential implementation of the connected component algorithm. The state X of the abstract fixed-point computation corre-

```
function Visit-Node (Node)
    if Add-Element(Node, Reachable-Nodes)
    then
        foreach Adj-Node in Neighbors-Of(Node)
            if not Is-Element-p(Adj-Node, Reachable-Nodes)
            then
                PQ-Insert(Adj-Node, Nodes-to-Visit)
```

Figure 2.2: Operation performed at every node: Add-Element does not add duplicates to Reachable-Nodes, and returns t iff a new element is added. The call to Is-Element-p is merely a *performance* optimization.

sponds to `Reachable-Nodes` and `Nodes-to-Visit`. The fixed point is reached when no further nodes can be added to `Reachable-Nodes`, the set of all nodes in the connected component. During execution, all the nodes of the connected component are either in `Reachable-Nodes` or are reachable from some member of `Nodes-to-Visit` without traversing `Reachable-Nodes`. Thus, `Nodes-to-Visit` represents the outstanding computations to be performed, and corresponds to the *work queue* commonly found in the implementation of fixed point algorithms. `Nodes-to-Visit` becomes empty when all the nodes of the connected component are in `Reachable-Nodes`. In general, a fixed point computation is usually organized such that when the work queue becomes empty, the fixed point has been reached.

`Visit-Node` invokes `Add-Element` to add the node being visited (`Node`) to `Reachable-Nodes`, the set of nodes currently known to be reachable from `Root`. `Add-Element` adds an element to a set, returning `t` if it succeeds. If it fails (because the element was already in the set), `nil` is returned. If the call to `Add-Element` in `Visit-Node` returns `t` (i.e., `Node` was not previously known to be reachable), `Visit-Node` is applied transitively to all the adjacent nodes of `Node` by adding them to `Nodes-to-Visit`, the set of nodes yet to be visited. As a purely *performance* optimization, the call to `Is-Element-p` identifies all the nodes that are known to be reachable (and have therefore been visited). These nodes are not added to `Nodes-to-Visit`, and will not be visited again. If the call to `Is-Element-p` were absent (i.e., all the adjacent nodes of `Node` were added to `Nodes-to-Visit`), an adjacent node `Adj-Node` that is in `Reachable-Nodes` may be added to `Nodes-to-Visit`. When `Adj-Node` is visited by `Visit-Node` again, the `Add-Element` operation will fail, and the traversal from `Adj-Node` will not be repeated.

```
function Connected-Component (Root)
    Reachable-Nodes := Make-Set()
    ;; Invariant: Reachable-Nodes is a subset of the nodes
    ;; reachable from Root

    Fixed-Point(Visit-Node, Lifo, Root)

    return Reachable-Nodes
```

Figure 2.3: Fixed-Point formulation of Connected Component Algorithm

```
function Visit-Node (Node)
    if Add-Element(Node, Reachable-Nodes)
    then
        foreach Adj-Node in Neighbors-Of(Node)
            if not Is-Element-p(Adj-Node, Reachable-Nodes)
            then
                FP-Insert-fn(Adj-Node)
```

Figure 2.4: Fixed-point version of operation performed at every node. The call to Is-Element-p is merely a *performance* optimization.

Figures 2.3 and 2.4 show an alternative formulation of the same connected-component algorithm, written using a library for fixed-point computations

I have developed. The explicit loop of the sequential implementation in Figure 2.1 has been replaced by the construct `Fixed-Point`. The implementation of `Fixed-Point` creates the work queue (which is now invisible), and enqueues the initial element `Root` into this work queue. The `Lifo` argument selects a scheduling strategy for the work queue. `Visit-Node`, which is invoked for each element removed from the work queue, has side effects on `Reachable-Nodes`. The call `PQ-Insert(Adj-Node, Nodes-to-Visit)` in function `Visit-Node` of Figure 2.2 is now replaced by the call `FP-Insert-fn(Adj-Node)`. `FP-Insert-fn` is a function defined by `Fixed-Point`, and is used to add new elements to the work queue. `Fixed-Point` returns when the work queue becomes empty. The detailed interface of `Fixed-Point` is given in Section 2.1.3.

The correctness of the connected component algorithm does not depend on the order of visiting the nodes of the graph (the nodes in `Nodes-to-Visit` in Figure 2.1). If accesses to global data structures (`Reachable-Nodes` and the unnamed work queue created by `Lifo`) are synchronized properly, multiple instances of `Visit-Node` can operate on different nodes of `Nodes-to-Visit` concurrently. By providing sequential and parallel implementations of `Fixed-Point`, the connected component algorithm in Figures 2.3 and 2.4 can be executed on sequential and parallel machines without modification. All the low-level parallel programming details concerning process creation, synchronization, scheduling and termination detection are handled by the implementation of `Fixed-Point`.

`Reachable-Nodes`, the (shared) set of nodes currently known to be reachable, is implemented using a set datatype (data-sharing abstraction) from the toolbox. Concurrent updates of `Reachable-Nodes` are simplified because the datatype is implemented such that concurrent calls to `Add-Element` are atomic, and duplicate elements are not added to the set.

The connected component algorithm typifies a large class of algorithms whose sources of potential parallelism may be expressed using `Fixed-Point`. `Fixed-Point` has a sequential interface, and may be used by programmers not experienced in parallel programming. Accesses to data structures that are shared in a parallel implementation of `Fixed-Point` are simplified by using the data-sharing abstractions of the toolbox, which provide implicitly synchronized operations. An application-specific scheduler may also be used with a program written using `Fixed-Point`. The interface of `Fixed-Point` allows the programmer to enforce arbitrary dependencies between the (potentially parallel) computations in the work queue for correctness or performance, but hides all the low-level parallel programming details used by the implementation of `Fixed-Point`. This chapter describes the interface of `Fixed-Point`, including the use of application-specific schedulers, in detail. The use of data-sharing abstractions to simplify concurrent accesses to shared data will be discussed in Chapter 3. The use of `Fixed-Point` in CONSAT and RC will be discussed in Chapters 4 and 6 respectively.

2.1.1 Formulation of Fixed-Point Computation

In this section I give an abstract description of the sources of parallelism in the use of discrete relaxation (or iteration) to compute the fixed-point of a system using `Fixed-Point`. I will show abstractly how a fixed-point computation may be parallelized if the function f and its domain, D, of a fixed-point computation $X \leftarrow f(X)$ satisfies the following properties. A detailed discussion of parallel discrete relaxation is given in Chapter 5.

- Values in D are structured. If $X \in D$, the parts of X may be designated with subscripts, as in X_c, where c is a subset of some index set I. This is intended to be a very general description. For example, X may be an array, and c a singleton containing an integer index, indicating a single element of X, or a set of integers, indicating a subset of the elements of X. Likewise, X may be a graph structure, and c may be a subset of its vertices.

- f is a non-deterministic function (a relation, in other words) that is decomposable into component functions f_c (for certain subsets $c \subseteq I$) such that $f_c(X)$ differs from X only at the indices $i \in c$ (that is, $X_j = f_c(X)_j$ if $j \notin c$). A value of $f(X)$ is a value of one of the $f_c(X)$ for some $c \subseteq I$.

If f were an arbitrary non-deterministic function, of course, the fixed point computed by the abstract algorithm might not have a unique solution. It is assumed the programmer has determined that f has the necessary confluence properties. (Such a function possesses what Barth calls *allowable indeterminacy* [6].) To take advantage of parallelism, some additional properties are needed:

- The value of $f_c(X)$ typically depends only on some part, X_r, of X, where $r = r(c) \subseteq I$ depends only on c. That is, $f_c(X) = f_c(X_{r(c)})$.

- If f_c and f_d are components of f, c and d do not intersect, c and $r(d)$ do not intersect, and d and $r(c)$ do not intersect, then $f_{c \cup d}$ is a component of f, $f_{c \cup d}(X)_c$ is a value of $f_c(X)_c$, and $f_{c \cup d}(X)_d$ is a value of $f_d(X)_d$. In other words, it is valid to compute values of separate, *non-interfering* parts of f concurrently. (In addition, application-specific knowledge may be used to determine if potentially interfering component functions of f may be computed concurrently.)

As an example, consider a forward dataflow analysis problem where I is the set of nodes of the flow graph. X denotes some quantity Q (say constant propagation information) for every node of the flow graph, while $X_{\{i\}}$ denotes Q for one particular node i. f computes Q for every node, while $f_{\{i\}}$ computes Q for one particular node i. If the input value of a node has not changed, its output value will also remain unchanged. Hence, a function $f_{\{i\}}$ is enqueued for re-computation only if the corresponding $X_{\{i\}}$ has been modified by one of the predecessors of i.

2.1.2 General Use of Fixed-Point

Parallel discrete relaxation is not the only use of `Fixed-Point`. The abstraction may also be used for general work queue-style computations that are usually not considered as fixed-point computations.

2.1.3 Basic Use of Fixed-Point Library

A typical invocation of `Fixed-Point` has the form:

```
Fixed-Point(Op, Sch-Strategy, Initial-WQ-Elts)
```

Fixed-Point creates a priority queue of outstanding computations, `WQ`, which is hidden, initializes `WQ` with the elements of the list `Initial-WQ-Elts`, and repeatedly invokes `Op` on elements removed from `WQ` until `WQ` becomes empty. (Thus, the parallel implementation returns when `WQ` is empty *and* no call to `Op` is in progress.) Elements of `WQ` are scheduled according to `Sch-Strategy`.

Op `Op(Elt)` is invoked for every element `Elt` removed from `WQ` by `Fixed-Point`. `Op` may have side effects on shared data structures, and may also add new elements to `WQ`. In a parallel implementation there may be concurrent calls to `Op` for different elements removed from `WQ`. The side effects of concurrent calls to `Op` on shared data structures are simplified by the use of data-sharing abstractions (Chapter 3).

Sch-Strategy The implementation of `Fixed-Point` provides common scheduling strategies for `WQ` (e.g., Lifo or Fifo). Section 2.1.4 describes the use of `WQ`s with application-specific scheduling strategies with `Fixed-Point`.

Initial-WQ-Elts is a list of initial elements of `WQ`.

`Fixed-Point` defines the following functions that may be called by user code (`Op`):

FP-Insert-fn `FP-Insert-fn(Elt)` is called by `Op` to add new elements to `WQ`.

FP-Abort-fn may be called by `Op` to terminate `Fixed-Point` before `WQ` becomes empty.

The following scheduling strategies for `WQ` are predefined by `Fixed-Point`:

Lifo Elements of `WQ` are processed by `Op` in last-in-first-out order.

Fifo Elements of `WQ` are processed by `Op` in first-in-first-out order.

2.1.4 Application-Specific Scheduling

In many fixed-point computations the programmer may want to use application-specific knowledge to schedule outstanding computations for correctness or performance reasons. As an example, consider a constraint satisfaction system expressed as a fixed-point computation. The correctness of the parallel implementation may require that any two constraints with common variables are not activated concurrently. For performance reasons (faster convergence to fixed point), it may be desirable to activate all constraints at least once before re-activating any constraint. These requirements are not satisfied by any of the standard scheduling strategies provided by `Fixed-Point`.

To implement scheduling strategies that may be dependent on the total history of the computations (calls to `Op`) performed so far, elements (computations) in `WQ` are partitioned into one of three conceptual states: *active*, *ready* and *blocked*. An element `Elt` is active if a call `Op(Elt)` is in progress. An element is ready if it may be activated (by `Op`) immediately. An element is blocked if it may not be activated until some other element has been activated. These states are internal to the implementation of `WQ`, and are not visible to the user of `Fixed-Point`.

A typical invocation of `Fixed-Point` with an application-specific work queue has the following form:

```
Fixed-Point(Op, Make-Priority-Queue, Initial-WQ-Elts)
```

Fixed-Point `Fixed-Point` invokes `Make-Priority-Queue()` to create a work queue of outstanding computations, `WQ`, which is hidden, initializes `WQ` with the elements of `Initial-WQ-Elts`, and repeatedly invokes `Op` on elements removed from `WQ` until `WQ` becomes empty. This interface between `Fixed-Point` and `WQ` allows the programmer to provide an application-specific scheduling strategy and allows `Fixed-Point` to keep track of the number of outstanding computations in `WQ` at any time.

Op `Op(Elt)` is invoked for every element `Elt` removed from `WQ` by `Fixed-Point` (using `WQ(Dequeue)`). `Op` may have side effects on shared data structures, and may also add new elements to `WQ`. In a parallel implementation there may be concurrent calls to `Op` for different elements removed from `WQ`. `Op` must not invoke the function `WQ` explicitly.

Make-Priority-Queue `Make-Priority-Queue()` returns a priority queue of outstanding computations (`WQ`, which is hidden). `WQ` is a *dispatch* function that accepts four standard messages (`Dequeue, Empty-p, After-Op, Enqueue`) from `Fixed-Point` for performing various operations on the work queue. An application-specific scheduling strategy may be provided by defining the appropriate `Make-Priority-Queue` function that meets the requirements specified in this section. (Strictly speaking, `Lifo` in Sections 2.1 and 2.1.3 is a short form for `Make-Lifo()`.) The work queue may use runtime information—including the state of global data and the history of the computations—in its scheduling decisions, and allows arbitrary dependencies between elements of the work queue to be enforced. The clean interface between `Fixed-Point` and the work queue encourages the programmer to separate the basic algorithm from scheduling decisions, so that performance-related scheduling strategies may be experimented with by modifying a restricted portion of the code.

Initial-WQ-Elts is a list of initial elements of `WQ`.

`WQ` is a dispatch function for four different operations on the work queue. `Fixed-Point` interacts with the work queue by invoking `WQ` with the following four messages (and other arguments as appropriate):

Dequeue `WQ(Dequeue)` returns a ready element removed from `WQ`, and changes the state of the element to active. Blocked elements in `WQ` are never returned by this function. `Fixed-Point` ensures that `WQ` is non-empty when this function is called.

Empty-p `WQ(Empty-p)` returns t iff `WQ` is empty. It is invoked by the termination detection code of `Fixed-Point` when no call to `Op`, `WQ(Enqueue, *)`, `WQ(Dequeue)` or `WQ(After-Op, *)` is in progress.

After-Op `WQ(After-Op, Elt)` is invoked by `Fixed-Point` after each call of `Op(Elt)`, and is required by some `WQ`s with sophisticated scheduling strategies. This function changes the state of `Elt` from active to *not in* `WQ`, and may also change the state of some (multiple) other `Elt'` in `WQ` from blocked to ready by calling `FP-Insert-fn(Elt')`. For example, this is done if the correctness of an application requires that `Elt` and `Elt'` are not activated concurrently for some `Elt` and `Elt'`.

Enqueue `WQ(Enqueue, Elt)` returns t if `Elt` is added to `WQ` in the ready state, and nil if `Elt` is either not added to `WQ` or is added to `WQ` in the blocked state. This function should *not* be called by `Op`, which should instead call `FP-Insert-fn` defined below.

The following functions are defined by `Fixed-Point` for use by user code (`Op`):

FP-Insert-fn `FP-Insert-fn(Elt)` is called by `Op` to add a new element `Elt` to `WQ`. `Op` calls `FP-Insert-fn` instead of `WQ(Enqueue, *)` so that `Fixed-Point` may keep track of the number of active and ready elements in `WQ`. (`FP-Insert-fn` calls `WQ(Enqueue, *)` internally.) This function may add an element to `WQ` in the blocked state such that this element is not activated until the activation of some other element has completed. (See `After-Op`.)

FP-Abort-fn may be called by `Op` to terminate `Fixed-Point` before `WQ` becomes empty.

2.1.5 Summary

In this section I described fixed-point computations and an abstraction (`Fixed-Point`) for computing the fixed point of a system using iteration. If the system satisfies a given set of conditions, the iterative algorithm for computing its fixed point may be parallelized (by using the parallel implementation of `Fixed-Point`). In addition, `Fixed-Point` may also be used in general work queue-style computations.

`Fixed-Point` is a typical parallelism abstraction with a simple, sequential interface. This interface allows a programmer to identify the sources of parallelism in an application without being burdened by low-level parallel programming details, which are handled by the parallel implementation. Accesses to data structures that are shared in a parallel implementation of `Fixed-Point` are simplified by using the data-sharing abstractions of the toolbox.

The clean and flexible interface between `Fixed-Point` and the work queue allows the programmer to provide an application-specific scheduler to enforce arbitrary dependencies between potentially concurrent computations for correctness or performance. This interface enhances modularity and simplifies performance tuning because all the performance-related scheduling code is neatly captured by the work queue. Consequently, the programmer may experiment with various scheduling strategies without changing other parts of the application program.

`Fixed-Point` does not provide any support for controlling the grainsize of concurrent computations in a parallel implementation. It remains to be seen whether this simple interface provides good parallel performance for a large number of applications.

2.2 Traversals of Directed Acyclic Graphs

Many algorithms may be specified as a set of *unordered* state transitions such that a transition may be taken if the corresponding precondition is satisfied. If such an algorithm is implemented in a sequential language, the transitions are specified in some arbitrary but fixed order. In a parallel implementation multiple transitions can often be taken concurrently, provided accesses to shared data are synchronized properly. As an example, a large class of computations may be abstracted as the traversal of a directed acyclic graph (dag) such that a node is visited only if all its parents have been visited. The nodes of the dag represent units of computation, and the edges represent precedence constraints between the computations. A directed edge from node A to node B specifies that the computation represented by B may only be started after the computation represented by A has terminated. In a sequential implementation the nodes are visited in some topological-sort order. In a parallel implementation the computations represented by the nodes of the dag may be performed concurrently while observing the precedence constraints.

2.2.1 Dag-Traversal

The potential parallelism in a dag traversal may be captured by the parallelism abstraction `Dag-Traversal`. `Dag-Traversal` visits the nodes of a dag in an order that respects the precedence constraints represented by the edges. It has a sequential interface, and may be used by novice parallel programmers easily. The dag is specified by a list of all its nodes and a function that returns all the (immediate) successor nodes of any given node. When a node is visited a programmer-specified function (possibly with side effects) is invoked on the node. The dag must not be modified during the execution of `Dag-Traversal` (say by the function that visits a node). `Dag-Traversal` returns when all the nodes of the dag have been visited. The sequential implementation of `Dag-Traversal` visits the nodes of a dag in some topological-sort order, while the parallel implementation may visit multiple nodes of the dag concurrently subject to the precedence constraints. For any execution of the parallel version of `Dag-Traversal`, the linearized order in which the nodes are visited corresponds to some topological-sort ordering of the nodes of the dag.

A typical invocation of `Dag-Traversal` has the following form:

```
Dag-Traversal(Dag-Nodes, Succ-Nodes-fn, Op)
```

Dag-Nodes is a list of all the nodes in the dag (in any order).

Succ-Nodes-fn `Succ-Nodes-fn(Node)` returns a list of all the nodes that are immediate successors of `Node`, for any `Node` in `Dag-Nodes`.

Op `Op(Node)` is invoked (by the implementation of `Dag-Traversal`) when `Node` is visited. `Op` may have side effects on shared data structures, but may not modify the dag. In a parallel implementation there may be concurrent calls to `Op` for different nodes of the dag.

It is the programmer's responsibility to ensure that an algorithm using `Dag-Traversal` is correct for any topological-sort order of visiting the nodes of the dag. In general, this means that the precedence constraints represented by the dag edges represent the only orderings that are necessary for the correctness of the algorithm being implemented. If accesses to global data structures are synchronized properly, multiple instances of `Op` may be invoked on different nodes concurrently. By providing sequential and parallel implementations of `Dag-Traversal`, the algorithm may be executed on sequential and parallel machines without modifications. All the low-level parallel programming details concerning process creation, synchronization, scheduling and termination detection are handled by the implementation of `Dag-Traversal`.

2.2.2 Dag-Traversal for Fixed-Point Computation

Many fixed-point computations may be expressed as the traversal of an immutable directed graph (digraph) G whose structure is known a priori. A node of G represents a computation (a work queue element), and a directed edge from a node A to a node

B means that a visit of A will result in a subsequent visit of B. (B is scheduled for the visit by adding it to the work queue using `FP-Insert-fn`.) If there are cycles in G, the traversal of G may follow an edge multiple times, so a node may be visited more than once. As an example, consider a dataflow analysis algorithm that computes constant propagation information using a fixed-point computation. The nodes correspond to the basic blocks of the control flow graph, and the edges correspond to the flow of control (and information) between the basic blocks. If a basic block is part of a loop in the program, it may be visited multiple times by the dataflow algorithm before the fixed point is found.

A naive way to compute the fixed point of a system specified as a directed graph is to apply `Fixed-Point` to all the nodes of G directly. This is inefficient because the scheduling strategy may (in general) schedule a node for activation before the fixed points of all its predecessor nodes have been computed. (Nodes in a cycle are not considered predecessors of each other.) A more efficient alternative is to partition G into a dag D of maximal strongly-connected components (SCCs, or simply *components*), and then apply `Fixed-Point` to a component of D if the fixed points of all its predecessor components have been computed. This alternative corresponds to a dag traversal of D in which the fixed-points of the components are computed in some topological-sort order. In a parallel implementation there are two levels of parallelism in the computation, with `Dag-Traversal` coordinating the possible concurrent activations of components that are not predecessors or successors of one another, and with `Fixed-Point` computing the fixed points of the individual nodes within any component.

Directed Graph to Dag

To facilitate the use of `Dag-Traversal` in a fixed-point computation as described above, the parallel programming toolbox provides a library `Digraph-to-Dag` for generating the dag that corresponds to a directed graph. A digraph G is specified by a list of all its nodes and a function that returns all the (immediate) successor nodes of any given node. The corresponding dag D computed by `Digraph-to-Dag` is specified by a list of the components D, a function that returns all (immediate) successor components of any given component, a function that returns all the nodes in G of a component in D, and a function that returns the component (in D) of any node in G.

A typical invocation of `Digraph-to-Dag` has the following form (`:=` denotes a multiple value assignment):

```
Comp-List, Succ-Comps-fn, Comp-to-Nodes-fn, Node-to-Comp-fn
  := DiGraph-to-Dag(Node-List, Succ-Nodes-fn)
```

A directed graph G is specified for `Digraph-to-Dag` as follows:

Node-List is a list of all the nodes of G (in any order).

Succ-Nodes-fn `Succ-Nodes-fn(N)` returns all the immediate successors of any node `N` in `Node-List`.

`Digraph-to-Dag` returns the dag D corresponding to the directed graph G. Dag D is specified as follows:

Comp-List `Comp-List` contains all the components of D (in some unspecified order).

Succ-Comps-fn For any component `C` in `Comp-List`, `Succ-Comps-fn(C)` returns a list of all its immediate successor components.

Comp-to-Nodes-fn `Comp-to-Nodes-fn(C)` returns a list of all the nodes (in `Node-List`) in component `C`.

Node-to-Comp-fn For any node `N` in `Node-List`, `Node-to-Comp-fn(N)` returns the component `C` in `Comp-List` that `N` belongs to. (This function is for debugging use only.)

Combined use of Fixed-Point and Dag-Traversal

Instead of applying `Fixed-Point` to all the nodes of a digraph directly:

```
Fixed-Point(Op, Make-PQ, Node-List)
```

the following more efficient code skeleton that uses `Dag-Traversal` may be used:

```
Comp-List, Succ-Comps-fn, Comp-to-Nodes-fn, Node-to-Comp-fn
  := DiGraph-to-Dag(Node-List, Succ-Nodes-fn)

Dag-Traversal(Comp-List, Succ-Comps-fn, Component-Op)
```

where the function `Component-Op` is used to compute the fixed point of a component:

```
Function Component-Op (C)
  Fixed-Point(Op, Make-PQ(C), Comp-to-Nodes-fn(C))
```

Note that `Make-PQ`, which determines the scheduling policy for computing the fixed point of a component `C`, may be parameterized by `C`.

2.2.3 Summary

Many graph *algorithms* visit the nodes of a dag such that a node is not visited until all its parent nodes have been visited. In a sequential *implementation* of such an algorithm without the toolbox, the nodes of the dag are visited in some topological-sort order. In this section I described a parallelism abstraction `Dag-Traversal` that captures the potential parallelism in such a dag traversal *algorithm*. `Dag-Traversal` visits the nodes of a dag in an order that respects the precedence constraints represented by the edges. A programmer-defined function is invoked on any node of the dag that is visited. `Dag-Traversal` has a sequential interface, and may be used by novice parallel programmers easily. The sequential implementation of `Dag-Traversal` visits the nodes of a dag in some topological-sort order, while the parallel implementation may visit multiple nodes of the dag concurrently subject to the precedence constraints. All the low-level parallel programming details are hidden from the programmer. The linearized order in which the nodes are visited in a parallel implementation corresponds to some topological-sort ordering of the nodes. Accesses to data structures that are shared in the parallel implementation are simplified by using the data-sharing abstractions of the toolbox. A sample use of `Dag-Traversal` will be given in Chapter 6. Additional examples of applications that can be expressed in terms of `Dag-Traversal` are given in Section 10.6.

`Dag-Traversal` may also be used in conjunction with `Fixed-Point` to perform a large class of fixed-point computations efficiently. A computation of this class may be expressed as the traversal of an immutable digraph with a known structure. Instead of applying `Fixed-Point` to the entire digraph G directly, the library `Digraph-to-Dag` may be used to partition G into a dag of strongly-connected components. `Dag-Traversal` is then used to invoke `Fixed-Point` on individual components in an order that observes the precedence constraints of the dag edges, and the fixed points of multiple components may be computed concurrently in a parallel implementation.

2.3 Summary

In this chapter I described two typical parallelism abstractions in the prototype toolbox, `Fixed-Point` and `Dag-Traversal`. A parallelism abstraction may be thought of as a high-order function that may be invoked from the application with application-specific function arguments. The abstraction has a sequential interface that is used by the programmer to identify the potential parallelism in a computation. As all the parallel programming details are hidden by the implementation, a parallelism abstraction may be used by a novice parallel programmer relatively easily.

A parallelism abstraction captures the allowable indeterminacy in a set of computations, and the implementation of the abstraction performs some of these computations concurrently. `Fixed-Point` is an example of an abstraction that may be used for a set of totally unordered computations. On the other hand, the work queue of `Fixed-Point` may be customized to impose any precedence constraint on the set of computations to be performed. In fact, the work queue may be specialized such that `Fixed-Point` behaves like `Dag-Traversal`. `Dag-Traversal` is an example of an abstraction with more structure between the potentially concurrent computations. The interface of `Dag-Traversal` allows the programmer to specify the (restricted class of) precedence constraints between the computations more easily, while the implementation uses more specialized algorithms and data structures for better performance.

An application program may make use of more than one parallelism abstraction. In addition, parallelism abstractions may be composed to express different levels of parallelism in an algorithm. An example of the use of `Dag-Traversal` with `Fixed-Point` will be described in Section 6.3.4.

Chapter 3

Data-Sharing Abstractions

3.1 Side Effects on Shared Data

Most algorithms implemented in an imperative programming language involve the use of operations with side effects on data structures, and a parallel (toolbox) version of any such algorithm will usually involve concurrent accesses to shared data. For example, multiple threads of control (created by a parallelism abstraction) in a parallel unification program may access the unification database concurrently. This may lead to incorrect results if the accesses are not synchronized properly. Ensuring that these concurrent accesses on shared data behave correctly without limiting concurrency is a major problem in the parallelization of a sequential program, and the use of complicated schemes such as optimistic concurrency control for performance is a potential source of errors for parallel programs.

In the toolbox approach for symbolic parallel programming, the programmer uses his high-level understanding of the application to determine if the interleaving of the concurrent operations (with side effects) produces the desired result. While this question is difficult to answer in the general case, serialization of accesses to shared data objects is a sufficient correctness condition in the simple (but common) case.

Data-sharing abstractions support common side-effecting operations on shared

[1] An earlier version of this chapter appeared as a paper co-authored with Hilfinger, "Managing Side Effects on Shared Data" [29].

objects, simplifying the coding of a large class of algorithms that modify shared data structures in a parallel implementation. The abstractions are designed so that the common styles of data accesses in typical sequential code may be parallelized with minimal changes. The programmer selects an appropriate data-sharing abstraction for each shared object, and expresses the accesses on the object in terms of operations provided by the abstraction. To ensure that an abstraction may be used by a novice parallel programmer easily, all the low-level parallel programming details are hidden in the implementation of the abstraction. Some abstractions also provide a set of performance hooks (in the form of optional arguments) that may be used by the programmer to customize or tune the abstractions using application-specific information for improved performance. The abstractions span a spectrum that trades ease of use for performance. An overview of them follows.

Simple concurrent datatypes are atomic implementations of common datatypes such as counters, sets and priority queues. A simple concurrent datatype has the same interface as its sequential counterpart, and concurrent operations on an object of such a datatype behave correctly (e.g., are linearizable). These datatypes simplify the parallelization of a large class of programs for which mutual exclusion for shared data accesses is a sufficient correctness condition. The implementation of such a datatype may make use of the semantics of the datatype to make the operations highly concurrent, so that shared data accesses do not become performance bottlenecks. No change to an existing (sequential) program is needed to convert it to use concurrent datatypes.

A common high-level operation on a compound object is to destructively modify the object if some test succeeds. For example, a program may add an element x to a list L if x is not in L initially. In a sequential program such a high-level operation is usually implemented as two distinct operations, Test and Modify. In a parallel implementation this scheme would fail, even if Test and Modify are individually synchronized. My solution to this problem is to recognize Test and Modify as logical parts of *one* high-level operation, a *conditional update*. The toolbox provides (parallel implementations of) conditional update operations for common concurrent datatypes such as priority queues, thus eliminating a large class of race conditions in parallel programs. If the mutation operations in a sequential program are expressed in terms of conditional update operations, the mutation operations may be parallelized by simply using the parallel implementations of the conditional update operations. A small amount of existing code may need to be rewritten to convert an existing program that uses Test and Modify pairs to use conditional update operations.

A *dictionary* is a mapping from *Keys* to *Values*. The dictionary datatype of the toolbox supports a conditional update operation called *conditional insert*, which adds a (Key, Value) pair to a dictionary D if no pair for Key is in D, and returns the existing value corresponding to Key otherwise. The parallel toolbox implementation uses fine-grain locks to allow multiple concurrent operations on a dictionary, and uses optimistic concurrency control for better parallel performance. The lookup and insert pairs of an existing program have to be replaced by calls to conditional insert to use this datatype.

Concurrent accesses to shared objects not supported by any data-sharing abstraction of the toolbox directly may be synchronized using mutual exclusion (mutex) locks if the objects are not heavily contended. *Autolocks* are mutex locks that may be associated with arbitrary objects without knowing or modifying the internal representation of the objects. They are useful for synchronizing operations with short critical sections on shared objects, and may be retrofitted to a sequential program relatively easily.

Some mutation operations with long critical sections on shared objects (implemented using simple concurrent datatypes or autolocks) may be rewritten in a stylized functional form using *optimistic read-modify-write* (`ORMW`), an optimistic scheme that does not lock an object for the entire duration of the operation. The parallel implementation of `ORMW` uses an application-specific predicate to detect *incompatible* races that may lead to inconsistent results, and retries the operation if such races are detected. `ORMW` is more difficult to use than autolocks because the programmer has to define what constitutes an incompatible race condition, but has the potential of providing much better performance for heavily contended objects. Substantial changes to an existing program may be necessary to convert it to use `ORMW`.

False sharing occurs in a parallel program derived from a sequential counterpart when a static global variable is used to hold data logically private to a particular instantiation of a set of procedures. In this context a *variable* is called a *generalized variable* in Common Lisp, and refers to any container for (a pointer to) another object, including a field of a structure. A global variable is one accessible to any thread. In a parallel implementation multiple threads *appear* to conflict at some field of a shared variable, and consequently the *algorithm* appears to be inherently sequential. For example, in the absence of recursion, a set of procedures may use a global named variable instead of arguments and return values for communication. In a parallel implementation this coding style results in false sharing for the global variable. *Per-thread locations* provide a transparent solution to this problem in the common case where the variable is a field of a structure shared by multiple threads. Through the use of per-thread locations, access and update functions for that field of a shared structure actually refer to different memory locations when invoked from different threads, and only the definition of the structure has to be modified. (If the incorrectly shared variable is a *named* variable at the programming language level, it may simply be replaced by dynamically rebinding the variable in each thread of control in the parallel implementation.)

In the following sections I outline the toolbox approach for handling concurrent data accesses, and examine each data-sharing abstraction for simplifying concurrent accesses.

3.2 Writing Parallel Code: Shared Data Accesses

This section outlines the steps necessary to synchronize shared data accesses in the parallel implementation of an algorithm using the toolbox.

- The algorithm is modified to use a parallelism abstraction such as `Fixed-Point` or `Dag-Traversal` to make the parallelism in the algorithm explicit.
- Accesses to data structures that would be shared in a parallel implementation of the algorithm are identified.

- False sharing is eliminated from a set of procedures that use global, named variables for communication by converting the set of procedures to use arguments and return values for communication. In general, code with false sharing of named variables may be made re-entrant by rebinding these variables dynamically in each thread.

- False sharing of fields in structures is eliminated by using per-thread locations (Section 3.7).

- Concurrent accesses to (truly) shared objects are handled by using the simple concurrent datatypes provided by the toolbox (Section 3.3), and may involve rewriting existing code to use conditional update operations (Section 3.3.2). Shared objects not supported by the toolbox as simple concurrent datatypes may be protected by the use of autolocks (Section 3.5).

- Programs with time-consuming operations on shared objects (protected by autolocks) are rewritten to use optimistic read-modify-write operations to reduce contention (Section 3.6).

- Performance tuning of parallel programs written using the toolbox is discussed in Chapter 9.

3.3 Simple Concurrent Datatype

Simple concurrent datatypes are atomic implementations of common datatypes such as counters, sets and priority queues. A simple concurrent datatype has the same interface as its sequential counterpart, but is implemented to behave correctly (i.e., is linearizable) in the presence of concurrent operations. These datatypes simplify the parallelization of a large class of programs for which mutual exclusion for shared data accesses is a sufficient correctness condition. The implementation of such a datatype may make use of the semantics of the datatype to make the operations highly concurrent, so that shared data accesses do not become performance bottlenecks. No change to an existing (sequential) program is usually needed to convert it to use concurrent datatypes.

3.3.1 Example Simple Concurrent Datatype: Counter

The toolbox supports the *counter* concurrent datatype with two atomic operations, `Counter-Inc` and `Counter-Value`.

Make-Counter `Make-Counter(Init-Value)` returns a counter object initialized to `Init-Value`.

Counter-Inc `Counter-Inc(Counter, Delta)` atomically increments the value of `Counter` by `Delta`, and returns the new value of `Counter`.

Counter-Value `Counter-Value(Counter)` returns the value of `Counter`.

3.3.2 Conditional Update for Concurrent Datatypes

A common high-level operation on an object is to destructively modify the object if some test succeeds. For example, a program may add an element x to a list L if x is not in L initially. In a sequential program, such a high-level operation is usually implemented as two distinct operations, Test and Modify. In a parallel implementation this scheme would fail, even if Test and Modify are individually synchronized. The toolbox solution to this problem is to recognize Test and Modify as logical parts of *one* high-level operation, *conditional update*.

A conditional update operation has a sequential interface, and modifies an object atomically if the existing value of the object satisfies some condition. Two values are returned: the new value of the object and a boolean indicating whether the object has been modified.

As a concrete example, consider a multiset datatype with operations `Make-MultiSet()` and `Add-New-Element(S, Elt)`. `Add-New-Element` is a conditional update operation that adds `Elt` to `S` if `Elt` is not already in `S`. The use of

`Add-New-Element` instead of the usual `Element-of-p` and `Add-Element` pair eliminates a common class of race conditions, and code written in the conditional update style may be parallelized by using a parallel implementation of the multiset datatype from the toolbox. No explicit locking or synchronization by the programmer is required.

The toolbox provides conditional update operations for common concurrent datatypes. If a sequential program is expressed in terms of conditional update operations on these datatypes, it may be parallelized by simply using the parallel implementations of the conditional update operations provided by the toolbox. The implementation of the datatype may use optimistic techniques to achieve good parallel performance without exposing any parallel programming detail to the application programmer. A small amount of existing code may need to be rewritten to convert an existing program to use conditional update operations.

3.4 Dictionary

A dictionary is a mapping from *Keys* to *Values*, and is used in a large number of application programs. For example, a dictionary may be used in a unification program to record the correspondence between ground and non-ground terms. The dictionary datatype is an important example of a concurrent datatype with a conditional update operation called *conditional insert*. Conditional insert does not modify a dictionary D if a (Key, Value) pair with the same Key is already present in D, and adds the pair otherwise. The use of this operation instead of separate lookup and insert operations, where insert does not check for the presence of Key in D, eliminates a common race condition in the use of dictionaries. As a result, sequential code written using this operation can be parallelized by using a parallel implementation of dictionary, and the programmer is not burdened with synchronization details.

One naive way to implement the dictionary datatype on a parallel machine is to use a single mutex lock to synchronize accesses to a hash table representation. This implementation serializes all accesses to a shared dictionary—even for concurrent accesses with different Keys—and leads to poor parallel performance if the dictionary is heavily contended. The toolbox implementation of the dictionary datatype reduces unnecessary serialization for accesses involving different keys by partitioning the key space into disjoint subspaces, each protected by its own mutex lock. (These locks are not manipulated by the programmer.) The toolbox implementation provides a general hash function for partitioning the key space, and the programmer may override this by providing an application-specific hash function for better performance. In this way, concurrent conditional insert operations on a dictionary may proceed in parallel if the hash function distributes the keys for the operations to different key spaces. This dictionary interface provides good parallel performance without exposing the internal representation of the dictionary datatype or the low-level locking details to the programmer.

3.4.1 Dictionary Interface

A dictionary supports the following atomic operations:

Make-Dict `Make-Dict(Test-fn, Split-fn, Buckets)` returns an empty dictionary. `Test-fn(Key1, Key2)` is the equality predicate between keys, and returns `true` iff `Key1` and `Key2` are considered identical. `Split-fn(Key, Buckets)` is an application-specific function that maps `Key` to one of the integers (subspaces) 0..`Buckets-1`. `Buckets` is a positive integer that specifies the number of subspaces in the internal representation of the dictionary. In a parallel implementation concurrent insert operations of (potentially different) keys with the same `Split-fn` value may take longer to complete than concurrent inserts of keys with different `Split-fn` values. The programmer may use high-level knowledge about the application to provide a faster `Split-fn`, or one that distributes keys more evenly among the subspaces. Assuming that `Split-fn` distributes keys from the key space uniformly among 0..`Buckets-1`, a larger value for `Buckets` may be chosen for faster dictionary operations at the cost of additional storage in a parallel implementation. Defaults for `Test-fn`, `Split-fn` and `Buckets` are provided by the toolbox.

Conditional-Insert-Dict `Conditional-Insert-Dict(D, Key, Value)` inserts the pair `(Key, Value)` into `D` if no pair corresponding to `Key` is in `D`, and does not modify `D` otherwise. It returns two values, `Insertedp` and `Value'`. `Insertedp` is `true` if the pair `(Key, Value)` is added to `D` in the current operation, and `nil` otherwise. `Value'` is `Value` if the pair `(Key, Value)` is already in `D`, or is added to `D` in the current operation. `Value'` is *not* `Value` if some pair `(Key, Value')` is already in `D`. `Value` and `Value'` are not tested for equality.

Conditional-Insert-Funcall-Dict If no pair corresponding to `Key` is in `D`, `Conditional-Insert-Funcall-Dict(D, Key, Fn)` invokes `Fn()` and inserts the pair `(Key, Value)` (where `Value` is the object returned by invoking `Fn`) into `D`. If some pair `(Key, Value')` is already in `D`, `Fn` is not invoked, and `D` is not modified. Two values are returned: `Insertedp` and the value corresponding to `Key` in `D`. `Insertedp` is `true` if the pair `(Key, Value)` is added to `D` in the current operation (i.e. `Fn` is invoked), and `nil` otherwise. The value returned is `Value` if no pair corresponding to `Key` is initially in `D`, and `Value'` if `(Key, Value')` is already in `D`. This function is useful if the object returned by `Fn` is expensive to create.

Lookup-Dict `Lookup-Dict(D, Key)` returns `Value` if some pair `(Key, Value)` is in `D` for some `Value`, and `nil` otherwise.

Map-Dict `Map-Dict(Fn, D)` invokes `Fn(Key, Value)` for all `(Key, Value)` pairs in `D` (for side effects). This operation is undefined if `Fn` modifies `D`, or if there are concurrent `Conditional-Insert-Dict` (or `Conditional-Insert-Funcall-Dict`) operations on `D`.

3.4.2 Sample Use of Dictionary

Figure 3.1 shows a sequential code segment for inserting a `(Key, Value)` pair into a dictionary `D` if no pair corresponding to `Key` is initially in `D`. The two return values correspond to those returned by `Conditional-Insert-Dict`. Figure 3.2 shows a simple parallel version of the same computation, and Figure 3.3 is an optimized version that tries not to lock the shared dictionary `D` unless a pair corresponding to `Key` is not found. Note the extra call to `Lookup-Dict` *after* `D` is locked to check for race conditions. Figure 3.4 is a version that uses `Conditional-Insert-Dict`. This version is much simpler than all of the other alternatives, and may be used in sequential and parallel code without modification. (In Figures 3.1, 3.2 and 3.3, `Existing-Value` and `Insertedp` are local or per-thread variables.) The optimized parallel version in Figure 3.3 is also inferior to the version in Figure 3.4 using `Conditional-Insert-Dict` because the toolbox implementation of dictionary provides fine-grain locking without exposing the details of the implementation to the programmer.

```
D := Make-Dict(Test-fn, Split-fn, Buckets)
...
Existing-Value := Lookup-Dict(D, Key)
if (Existing-Value = nil)
    then Insert-Dict(D, Key, Value)
         Insertedp := t
    else
         Insertedp := nil

return(Insertedp, Existing-Value)
```

Figure 3.1: Sequential Dictionary Insert

3.4.3 Split-fn implementation in Common Lisp

The default implementation of Split-fn:

```
(defun Split-fn (Key Buckets)
  (mod (sxhash Key) Buckets))
```

is *not* completely general because the Common Lisp hash function `sxhash` is only guaranteed to return identical values for `equal` objects.

```
D := Make-Dict(Test-fn, Split-fn, Buckets)
...
; Create multiple threads transparently by
; invoking some parallelism abstraction
...
; In each thread of the parallelism abstraction...
; Existing-Value and Insertedp are per-thread variables
Lock(D)
Existing-Value := Lookup-Dict(D, Key)
if (Existing-Value = nil)
    then Insert-Dict(D, Key, Value)
         Insertedp := t
    else
         Insertedp := nil
Unlock(D)

return(Insertedp, Existing-Value)
```

Figure 3.2: Simple Parallel Dictionary Insert

```
D := Make-Dict(Test-fn, Split-fn, Buckets)
...
; Create multiple threads transparently by
; invoking some parallelism abstraction
...
; In each thread of the parallelism abstraction...
; Existing-Value and Insertedp are per-thread variables
Existing-Value := Lookup-Dict(D, Key)
if (Existing-Value = nil)
    then Lock(D)
         Existing-Value := Lookup-Dict(D, Key)
         if (Existing-Value = nil)
             then Insert-Dict(D, Key, Value)
                  Insertedp := t
             else
                  Insertedp := nil
         Unlock(D)
    else
         Insertedp := nil

return(Insertedp, Existing-Value)
```

Figure 3.3: Optimized Parallel Dictionary Insert

```
D := Make-Dict(Test-fn, Split-fn, Buckets)
...
; Create multiple threads transparently by
; invoking some parallelism abstraction
...
; In each thread of the parallelism abstraction...
Conditional-Insert-Dict(D, Key, Value)
```

Figure 3.4: Sequential & Parallel Dictionary Insert using `Conditional-Insert-Dict`

Implementation Notes In the current CLiP-based toolbox implementation the `sxhash` value of a structure is independent of its contents, while the `sxhash` value of a cons, vector or array depends on its contents.

Because of this limitation of `sxhash`, a programmer is not completely free in the choice of Split-fn for a dictionary that may contain mutable keys. In particular, special care must be taken to ensure that the chosen Split-fn will always return the same value for a mutable object, regardless of its contents. There are three solutions:

- Ensure that sxhash will return the same value for the mutable object, regardless of its contents. (For example, structures in CLiP belong to this category.) This is a *non-portable* solution.

- Define Split-fn so that its value is (only) dependent on the value of some immutable part of the mutable object. As an example, the mutable objects may all be assigned unique-id's that are examined by Split-fn. (Assigning unique-id's to objects at creation time is the key behind very efficient Split-fn's.)

- Encapsulate the mutable object in another object, say a cons cell. Split-fn will always return the same value for the (immutable) cons cell.

The last two solutions are portable, but may require minor changes to the application.

Autolock Implementation As a dictionary is used in the implementation of autolocks to maintain the association between objects and mutex locks, a programmer must use one of the above solutions to ensure that autolocks behave correctly.

3.5 Autolock

Concurrent accesses to shared objects may be synchronized by using mutual exclusion (mutex) locks. Mutex locks are easy to use, and are appropriate for protecting shared objects that are not heavily contended. The object-lock association is usually maintained by storing the lock as a component of the object. This approach requires knowledge about the internal representation of the object, and is unsatisfactory for abstraction reasons. If the internal representation of the object is not available, or may not be modified without major changes to the code (e.g., the object is a list) some form of composite object—with *data* and *lock* fields— is needed to maintain the object-lock association. This alternative is also unsatisfactory as it requires changes to the way the object is accessed, and generally involves changing code that is scattered throughout the application.

Autolocks are mutex locks that may be transparently associated with arbitrary objects without knowing or changing the internal representations of the locked objects. They are especially useful for protecting shared data structures with short critical sections that are *not* heavily contended. (Shared data structures that are potential hot-spots should be protected using more sophisticated synchronization protocols to reduce contention.) They are provided for synchronizing accesses to datatypes that are not supported by the toolbox directly as simple concurrent datatypes, and support a programming style of simple critical sections. A trivial change in the source code is required to signify that a section of code modifies a shared object (and hence should be protected by the mutex lock of that object). This is less flexible than the use of explicit lock and unlock operations, but is less prone to programmer errors.

3.5.1 Autolock Interface

A typical use of autolock is shown in Figure 3.5.

```
Autolock-Init(Split-fn, Buckets)
...
; Create multiple threads transparently by
; invoking some parallelism abstraction
...
; In each thread of the parallelism abstraction...
With-Autolock(Obj, ...
                   Mutate(Obj),
                   ...),
...
With-Autolocks((Obj1, Obj2, Obj3),
               #'Obj-<,
               ...
               Mutate*(Obj1, Obj2, Obj3),
               ...)
```

Figure 3.5: Use of Autolocks

Autolock-Init `Autolock-Init(Split-fn, Buckets)` initializes a mapping from objects to mutex locks. (This mapping is not visible to the programmer.) In a typical program `Autolock-Init` is invoked by the *parent* thread before any *child* thread (which may mutate a shared object `Obj` concurrently) is created, and `With-Autolock` is invoked by each child thread before it attempts to modify `Obj`. (See below.)

A mutex lock is automatically associated with an object `Obj` the first time `Obj` is protected by `With-Autolock` or `With-Autolocks`. This lock is recorded by the object-lock mapping created by `Autolock-Init`.

With-Autolock `With-Autolock(Obj, Form*)` associates a new mutex lock for `Obj` if none exists, acquires the lock associated with `Obj`, and evaluates the enclosed expressions `Form*` while holding the lock. The result of the last evaluated expression in `Form*` is returned, and the lock is released.

Mutate `Mutate(Obj)` is any expression that may modify `Obj` destructively.

`With-Autolocks` provides a way to destructively modify multiple objects atomically without the risk of deadlocks:

With-Autolocks `With-Autolocks(Obj-List, Predicate-fn, Form*)` associates mutex locks with objects in `Obj-List` that do not have locks, sorts the objects in `Obj-List` according to the total order defined by `Predicate-fn`, and acquires the locks of objects in `Obj-List` in this sorted order. The enclosed expressions in `Form*` are then evaluated, and the locks are released. The result of the last expression in `Form*` evaluated is returned. `Obj-List` is not destructively modified.

If concurrent invocations of `With-Autolocks` with overlapping `Obj-List`s use the same `Predicate-fn`, the implementation will ensure that no deadlock will occur. `With-Autolocks` is useful for applications that require multiple objects to be mutated atomically.

Mutate* `Mutate*(Obj1, Obj2, ...)` is any expression that may modify the objects `Obj1`, `Obj2`, etc.

`Autolock-Init` accepts two optional arguments `Split-fn` and `Buckets` that may be specified by the programmer to improve the performance of `With-Autolock` or `With-Autolocks` for objects that have not been locked before.

Buckets is a positive integer that determines the number of subspaces in the internal representation of the object-lock mapping created by `Autolock-Init`. A larger value for `Buckets` may lead to better performance at the cost of extra storage in a parallel implementation.

Split-fn `Split-fn(Obj, Buckets)` maps `Obj` to one of the integers (subspaces) in the range 0..`Buckets-1`. In a parallel implementation the performance of concurrent attempts to lock objects with the same `Split-fn` value (i.e., a *collision*) may be degraded if the objects have not been locked before. The programmer may provide an application-specific `Split-fn` that may be computed more efficiently than the general one provided by the toolbox, or one that distributes objects to the subspaces more evenly to minimize the frequency of collisions for better performance. (See Section 3.4.3 for the limitations of the current CLiP-based implementation of autolocks.)

3.5.2 Sample Use of Autolocks

```
Obj := Make-Object(Init-State)
...
Mutate(Obj)
```

Figure 3.6: Sequential mutation of `Obj`

Figure 3.6 shows a sequential code segment that modifies an object `Obj`. In a parallel implementation without autolocks (Figure 3.7), a lock field has to be added to the representation of `Obj`, and code that mutates `Obj` may have to modified as well (e.g., from `Mutate(Obj)` to `Mutate(Obj.data)`). This is not necessary when autolock is used (Figure 3.8).

3.6 Optimistic Read-Modify-Write

In many applications, objects are destructively modified in a sequence of read-modify-write steps. In a read-modify-write operation values are copied from some parts of the object (read), new values for certain parts of the object are computed in a side-effect-free manner (modify) based on the result of read, and the new values are written to some parts of the object destructively (write).

```
Obj := Make-Object(Init-State)
Obj.lock := Make-Lock()
...
; Create multiple threads transparently by
; invoking some parallelism abstraction
...
; In each thread of the parallelism abstraction...
Lock(Obj.lock)
Mutate(Obj.data)
Unlock(Obj.lock)
```

Figure 3.7: Parallel mutation of `Obj` without autolock: `Obj` is now an indirect object with `data` and `lock` fields, where `Obj.data` stores what was formerly in `Obj`. The calling sequence for `Mutate`, which is scattered throughout the code, is also changed, and explicit `Lock` and `Unlock` calls have to be added.

```
Autolock-Init(Obj-Split-fn, 10)
Obj := Make-Object(Init-State)
...
; Create multiple threads transparently by
; invoking some parallelism abstraction
...
; In each thread of the parallelism abstraction...
With-Autolock(Obj, Mutate(Obj))
```

Figure 3.8: Sequential & Parallel mutation of `Obj` with autolock

A simple parallel implementation of a read-modify-write-style operation can be constructed by performing each read-modify-write in a critical section (say using autolock). If a shared object is heavily contended, this will effectively serialize the accesses, leading to poor parallel performance. As an alternative, the toolbox provides an optimistic abstraction, *optimistic read-modify-write* (`ORMW`), for accessing shared objects concurrently. `ORMW` does not lock an object during the (potentially time-consuming) modify phase of the computation, and uses an application-specific predicate to detect *incompatible* changes to the object by other threads (during the modify phase) that may lead to inconsistent results. (Not all changes to an object by other threads while the current thread is in the modify phase are necessarily incompatible.) If an incompatible change to the object is detected, the `ORMW` operation is re-executed. `ORMW` is more difficult to use than autolocks because it requires the programmer to use high-level understanding of the application to define incompatible race conditions, but has the potential of providing much better performance for heavily contended objects. Substantial changes to an existing program may be necessary to convert it to use `ORMW`.

3.6.1 Optimistic Read-Modify-Write Interface

The following is a typical invocation of `ORMW`: (`:=` denotes a multiple value assignment operator.)

```
New-Value, Writtenp :=
  ORMW(Object, Read-fn, Modify-fn, Write-fn, Incompatible-p-fn)
```

`ORMW` changes the state of (some slots of) `Object` as specified by the functions `Read-fn`, `Modify-fn`, `Write-fn` and `Incompatible-p-fn`, and returns two values, `New-Value` and `Writtenp`. `New-Value` is the new value of (the parts of) `Object` that has been changed, and `Writtenp` is a boolean that is `t` iff `Object` has been changed. (These two values correspond to the two values returned by `Modify-fn` below.)

Object is any arbitrary object. Conceptually it may be thought of as a structure in which a few slots are changed in each `ORMW` operation.

Read-fn `Read-fn(Object)` returns the current value of (some slots of) `Object`. The programmer has to lock `Object` explicitly (using `With-Autolock`) if necessary for consistency, based on the semantics of the application and the representation of `Object`. (This may either involve a pointer copy or an object copy, depending on the application.)

Modify-fn `Modify-fn(Old-Value)` is a side-effect-free function that computes the new value of (some slots of) `Object`, `Modify-Value`, based on the value returned by `Read-fn`, `Old-Value`. `Modify-fn` returns two values, `Modify-Value` and `Writep`. If `Writep` is `nil`, `ORMW` is a no-op. `Modify-fn` is invoked by the implementation without locking `Object`.

Write-fn `Write-fn(Object, Modify-Value)` changes the state of (some slots of) `Object` destructively based on `Modify-Value` computed by `Modify-fn`. The implementation invokes this operation in a critical section together with `Incompatible-p-fn`.

Incompatible-p-fn `Incompatible-p-fn(Object, Old-Value, Modify-Value)` returns `t` iff the newly computed (but not written) `Modify-Value` based on `Old-Value` is incompatible with the current value of `Object`, based on the semantics of `Object`. The current value of `Object` may be read using `Read-fn` or simply examined in place. If `Incompatible-p-fn` returns `t`, `Modify-Value` is discarded, and the `ORMW` is re-executed. `Incompatible-p-fn` is not invoked by a sequential implementation of `ORMW` as it corresponds to race conditions that would not occur sequentially, and should return `nil` in a parallel implementation in the absence of concurrent `ORMW` operations on `Object`.

A conceptual parallel implementation of `ORMW` is shown in Figure 3.9. For a properly chosen `Incompatible-p-fn`, `ORMW` is semantically equivalent to the code segment in Figure 3.10.

```
; Old-Value, Modify-Value, Writep, Writtenp are local variables
Writtenp := nil
Old-Value := Read-fn(Object)
Modify-Value, Writep := Modify-fn(Old-Value)

if Writep
then
    With-Autolock(Object,
                  if not Incompatible-p-fn(Object,
                                           Old-Value, Modify-Value)
                  then
                      Write-fn(Object, Modify-Value)
                      Writtenp := t
return(Modify-Value, Writtenp)
```

Figure 3.9: Conceptual Parallel Implementation of `ORMW`

3.6.2 Sample Uses of `ORMW`

`ORMW` has been used successfully in the parallel implementation of two medium-sized applications based on discrete relaxation. These include CONSAT, a general-purpose constraint satisfaction system (Chapters 4 and 5), and RC, a dataflow analysis system (Chapter 6). In each application an application-specific predicate (`Incompatible-p-fn`) ensures that successive states of the system (`Object`) are monotonic with respect to some partial order, as monotonicity is a necessary correctness condition for the discrete relaxation performed by these applications that is not automatically maintained. The performance of the parallel implementation of either application based on `ORMW` is substantially better than the alternative based on simple critical sections.

```
; Modify-Value and Writep are local variables

With-Autolock(Object,
              Modify-Value, Writep := Modify-fn(Read-fn(Object))
              if Writep
              then
                  Write-fn(Object, Modify-Value)
              return(Modify-Value, Writep))
```

Figure 3.10: Semantics of `ORMW` (for a properly chosen `Incompatible-p-fn`)

3.7 Per-Thread Location

3.7.1 False Sharing

In some sequential programs data logically private to a particular instantiation of a set of functions is kept in static global variables. In the absence of recursion in a sequential implementation, this logically private data is indistinguishable from static global data. In a parallel implementation this equivalence between logically private data and static global data breaks down. Multiple threads may *appear* to conflict at some field of a shared variable, and the *algorithm* appears to be inherently sequential. This phenomenon is called false sharing.

One example of false sharing arises when a set of procedures in a program uses a global (i.e., top-level, or named) variable for communication. One way to eliminate false sharing is to convert the code to use arguments and return values instead of global variables for communication. Alternatively, code with false sharing of named variables may be made re-entrant by rebinding these variables dynamically in each thread (e.g., by using Common Lisp *special* variables that are dynamically rebound in each thread) in the parallel implementation. Both changes can usually be made with only minimal changes to the program.

3.7.2 False Sharing in Shared Structures

The above transformations do not cover false sharing of individual structure fields, however. Consider the linear time algorithm for finding the union `C` of two sets `A` and `B` in Figure 3.11, which (only) works for set elements with the property that equivalent elements are identical at the representation level. Each element `x` of either set `A` or `B` has a field `x.Mark` that is used by the union algorithm. The algorithm first generates a new unique symbol `Mark`, and initializes `C` to contain all the elements of `A`. Next the `a.Mark` field of each element `a` of `A` is set to `Mark`. Each element `b` of `B` is then examined, and if `b.Mark` is different from `Mark`, `b` is added to `C`. This algorithm appears to be inherently sequential if an element may belong to multiple sets, as concurrent set unions using this algorithm may conflict at the `Mark` fields and lead to incorrect results. False sharing also arises in graph algorithms that make use of a *mark* field in the nodes of a graph structure, say to record that the current node has been visited. Barth calls operations on such fields *benevolent side effects* [6].

```
Function Union (A, B)
;; Mark and C are local variables

    Mark := Generate-Unique-Mark()
    C := {}

    foreach a ∈ A
        a.Mark := Mark
        C := C ∪ {a}

    foreach b ∈ B
        if b.Mark ≠ Mark
            C := C ∪ {b}

    return C
```

Figure 3.11: Linear Time Set Union Algorithm

Per-thread locations provide a transparent solution for false sharing in the common case where the object is a structure, and multiple threads falsely share a field of the structure. Through the use of structures with per-thread locations, access and update functions for that field of a shared structure actually refer to different memory locations when invoked from different threads, and only the definition of the structure has to be modified. In a parallel implementation of per-thread locations a field is just a name, and may refer to distinct memory locations in different threads. The identity of a thread is an implicit argument to the access and update functions of these fields in a parallel implementation, so that different threads may access the same field of a shared structure as if that field were private to each thread. By using per-thread locations for structure fields that are falsely shared, many sequential algorithms with false sharing may be parallelized with minimal changes. The following section describes the toolbox version of per-thread locations for Common Lisp structure fields.

3.7.3 Pdefstruct Interface

The Common Lisp structure-defining facility `defstruct` is extended so that certain slots (slot names) of a structure may be specified as *per-thread*. The new structure-defining facility is called `pdefstruct`. Conceptually, a per-thread slot of a shared structure corresponds to a different memory location for each thread in the system.

A typical use of `pdefstruct` has the following form:

```
(pdefstruct Element
    Value
    (Mark Default :private)
    ...
)
```

Pdefstruct defines a Common Lisp structure type that may contain per-thread slots. All valid invocations of `defstruct` are also valid invocations of `pdefstruct`. In the following description a structure type `Element` with a per-thread slot `Mark` is used to illustrate the interface of `pdefstruct`. A structure (type) may have more than one per-thread slot. (Thread creation is *not* part of the `pdefstruct` implementation. Threads are usually created transparently by the implementation of a parallelism abstraction, say `Fixed-Point` or `Dag-Traversal`.)

:private A per-thread slot `Mark` is denoted by the slot description `(Mark Default :private)`. `Default` is the expression that evaluates to the default value of (each location of) `Mark`, and is a required argument for the description of per-thread slots. As in Common Lisp `defstruct`, `Default` is evaluated *once* for each instance of type `Element` created, if the corresponding argument is not supplied to the constructor function of the structure type (`Make-Element`) [62]. In a parallel implementation all the per-thread locations corresponding to the slot `Mark` of an instance of type `Element` are initialized to the same initial value, which may come from the argument corresponding to `Mark` in the constructor function or from `Default`. (Different instances of the structure type `Element` may have different initial values for the same per-thread slot `Mark` if `Default` has some internal state.)

Reduce-$\langle StructName \rangle$**-**$\langle SlotName \rangle$

`(Reduce-`$\langle StructName \rangle$`-`$\langle SlotName \rangle$ `Fn Struct Identity-Value)` is a reduction function defined by `pdefstruct` for each per-thread slot of every structure type. For example, `Reduce-Element-Mark` is defined for the structure type `Element` with per-thread slot `Mark`, and reduces the value of *all* the per-thread locations corresponding to `Mark` of a structure instance `Struct` using the binary function `Fn`. `Identity-Value` is the identity value of the reduction operator `Fn`. As the order of applying `Fn` to the per-thread locations corresponding to `Mark` is not specified, `Fn` is usually associative and commutative. The result of the reduction is undefined if the per-thread slot is modified by any thread during the reduction.

A minimal amount of change is necessary to convert existing code to use `pdefstruct`. Structure declarations are converted to use `pdefstruct` instead of `defstruct`, and per-thread fields have the additional field option `:private`. In particular, no change is necessary for code that accesses or updates a private field. Through the use of `pdefstruct`, a large class of sequential algorithms can be parallelized relatively easily.

3.7.4 Sample Use of Pdefstruct: A Unique-Id Generator

The following example illustrates how `pdefstruct` can be used to construct a unique-id (uid) generator of integers that has practically no contention in a parallel implementation, yet does not expose the parallel programming details to the programmer. The generator is a shared structure that consists of a global counter, a subrange size, and a pair of integers for each thread. Each pair of per-thread integers denote a consecutive subrange of uids to be generated by that thread. When the subrange of a thread is exhausted, a new subrange is allocated from the global counter. Uids are generated from each subrange in ascending order.

The structure of the uid generator is defined as follows:

```
(pdefstruct Uid
  Subrange-Step                   ; read-only
  Last-Global-Min                 ; shared atomic counter

  (My-Prev-Uid nil :private)      ; per-thread
  (My-Max-Uid nil :private))      ; per-thread
```

Subrange-Step is the number of uids in each subrange allocated to a thread.

Last-Global-Min is initialized to be an atomic counter shared by all the threads. `((Counter-Value Last-Global-Min) + Subrange-Step - 1)` is the lower bound of the next subrange to be allocated.

My-Prev-Uid is a per-thread value, and is the *previous* uid generated by this thread.

My-Max-Uid is a per-thread value, and is the last uid in the subrange. If `My-Prev-Uid` is equal to `My-Max-Uid` when a uid is requested, the subrange for the thread has been exhausted, and a new subrange must be allocated by incrementing `Last-Global-Min` (atomically).

The following returns a uid-generator with a minimum uid value of `Start` and a subrange step of `Step` for each thread:

```
(Make-Uid :Subrange-Step Step
          :Last-Global-Min (Make-counter (- Start Step)))
```

In Figure 3.7.4 each thread generating unique identifiers in an application takes the *same* (shared) uid-generator `Uid-G`, and invokes `New-Uid` on `Uid-G` to generate unique identifiers. `Last-Global-Min` is an atomic counter, so concurrent accesses (increment operations) are synchronized by the counter implementation. (`(Counter-Inc Counter Inc)` destructively increments the atomic `Counter` by the amount `Inc`.) `My-Prev-Uid` and `My-Max-Uid` are per-thread values, and no locking is required for accessing or updating them.

`Reduce-My-Prev-Uid` (defined by `pdefstruct`) may be used to sum the values of `My-Prev-Uid` across all the threads:

```
(Reduce-My-Prev-Uid #'+ Uid-Generator 0)
```

By using `Reduce-My-Prev-Uid` and `Reduce-My-Max-Uid`, the total number of uids generated by a multi-threaded uid generator can be computed.

3.8 Summary

In the toolbox approach for symbolic parallel programming, the programmer uses his high-level understanding of the underlying algorithm to identify the potential parallelism using a parallelism abstraction. The programmer is also responsible for identifying all the instances of data sharing, and for selecting the appropriate data-sharing abstractions for the shared accesses. The abstractions are designed so that the common styles of data accesses in typical sequential code may be parallelized with minimal changes. To ensure that the abstractions may be used by novice parallel programmers easily, all the low-level parallel programming details are hidden in the implementation. The abstractions cover a spectrum that trades ease of use with performance, and performance hooks are provided by some abstractions for the programmer to customize the abstractions using application-specific information for improved performance. If an application is written using parallelism abstractions and data-sharing abstractions, it may be ported between uniprocessors and multiprocessors without changes to the source code. Only the implementation of the abstractions has to be changed.

The data-sharing abstractions include simple concurrent datatypes (including dictionaries), autolocks, optimistic read-modify-writes and per-thread locations. Simple concurrent datatypes and dictionaries are supported by the toolbox directly, and may be used as drop-in replacements for their sequential counterparts. The use of conditional update operations on these datatypes eliminates a major source of race conditions in parallel programs. Autolocks may be used to synchronize concurrent accesses to arbitrary (lightly contended) objects not supported as concurrent datatypes by the toolbox, and very little modification to the source code is required. Optimistic read-modify-write operations may be used to synchronize concurrent updates to heavily contended objects by making use of the semantics of the datatype to achieve more concurrency. They have the potential of providing significant performance improvements over implementations based on simple critical sections, but require some high-level understanding of the operations being performed. Per-thread locations in structures help eliminate false sharing of structure fields in parallel code, and simplify the parallel port of a class of sequential programs (with benevolent side effects) significantly.

```
(defun New-Uid (Uid-G)
  (let ((Count (Uid-My-Prev-Uid Uid-G))
        (Step 0))
   (cond ((null Count)
          ; get first subrange for current thread
          (setf Step (Uid-Subrange-Step Uid-G))
          (setf Count (Counter-Inc (Uid-Last-Global-Min Uid-G) Step))
          (setf (Uid-My-Prev-Uid Uid-G) Count)
          (setf (Uid-My-Max-Uid Uid-G) (+ Count Step -1)))
         ((< Count (Uid-My-Max-Uid Uid-G))
          ; normal case
          (setf (Uid-My-Prev-Uid Uid-G) (1+ Count)))
         (t
          ; need new subrange for current thread
           (setf Step (Uid-Subrange-Step Uid-G))
          (setf Count (Counter-Inc (Uid-Last-Global-Min Uid-G) Step))
          (setf (Uid-My-Prev-Uid Uid-G) Count)
          (setf (Uid-My-Max-Uid Uid-G) (+ Count Step -1))))
  (values (Uid-My-Prev-Uid Uid-G) Uid-G)))
```

Figure 3.12: Unique-Id Generator: New-Uid is called by each thread using a shared Uid-G to generate globally unique identifiers (integers).

Part II

Sample Applications

In Part I of this dissertation I introduced a collection of parallelism abstractions and data-sharing abstractions for a prototype toolbox for symbolic parallel programming. In the following chapters I shall describe how the use of this toolbox simplifies the parallelization of two existing sequential Common Lisp applications that have been converted to use the toolbox: CONSAT [22, 21], a general-purpose constraint satisfaction system, and RC [70, 69], a dataflow analysis system. These applications have been developed by different authors, and are coded in very different programming styles. They are significantly larger than the applications that other symbolic parallel programming researchers have typically used to evaluate their systems. The sizes of these applications are shown in Table 3.1.

Program	Lines
CONSAT	8000
RC	7000
Toolbox	3000

Table 3.1: Sizes of Application Programs and (sequential and parallel) Toolbox, in lines of *somewhat commented* Common Lisp

The sequential (toolbox) versions of these applications run under Allegro Common Lisp, and should also run under other Common Lisp systems. The parallel (toolbox) versions of the applications run under Allegro CLiP on Sequent Symmetry multiprocessors. They may be ported to another parallel Common Lisp-based system on shared-address space multiprocessors by re-implementing the toolbox under the new parallel Lisp system. The description of CONSAT in Chapter 4 is based on a paper "CONSAT: A Parallel Constraint Satisfaction System" by Ho, Guesgen and Hilfinger [28]. The description of RC in Chapter 6 is based on an unpublished paper "RC: A Recursive-Type Analysis Program" by Ho, Wang and Hilfinger [31].

Chapter 4

Application: CONSAT

4.1 CONSAT Overview

CONSAT is a system for the definition and satisfaction of constraints in arbitrary finite discrete domains. A constraint consists of a set of variables and a relation among the variables, and a constraint network is a set of constraints connected by common variables. For example, in a constraint network that represents an electrical circuit, the variables may represent the voltages at nodes connecting circuit devices, while the constraints may represent relations between nodes imposed by circuit devices such as resistors, transistors and voltage sources. A *globally consistent* solution of a constraint network is a tuple of values, one per variable of the network, that satisfies all the constraints of the network simultaneously. CONSAT is a global constraint-satisfaction problem (CSP) solver whose constraint satisfaction technique is based on *filtering* [68], i.e., on the successive deletion of inconsistent values from the set of potential values for the variables. Unlike traditional filtering algorithms, CONSAT uses values that are associated with some additional information (called *tags*) for maintaining interrelationships among values (of different variables). This section gives an informal explanation of how CONSAT modifies *local constraint propagation*, a technique commonly used to compute locally consistent solutions, with *tagging* to compute the globally consistent solutions of a constraint network. As global constraint satisfaction problems over finite domains are generally NP-complete, the algorithm used by CONSAT is exponential in the worst case. A more formal treatment of CONSAT is given elsewhere [22].

Constraint Network Example Figure 4.1 defines constraint network A, which will be used in the examples throughout this section. There are three variables (`Va, Vb, Vc`) and six constraints (`C1-C6`). For example, constraint `C1` restricts the possible values of `Va` to `R` or `Y`, and constraint `C4` restricts the values of (`Va`,`Vb`) to one of the combinations (`R,G`) or (`R,B`). `Va` and `Vb` are called the *adjacent* variables of constraint `C4`. (A constraint may be adjacent to more than two variables, i.e., CONSAT is not limited to binary constraints.) Figure 4.2 defines constraint network B, which will also be used as an example. The steps of CONSAT as it computes the global solutions of constraint networks A and B are shown in Figures 4.3 and 4.4 respectively.

Notation In Figures 4.3 and 4.4, each constraint being activated (leftmost column) is shown with the feasible sets of its adjacent variables *after* its activation. An "*" at the end of a variable means that its feasible set has *not* been changed in the current activation. The feasible set of any variable non-adjacent to the current constraint can be found by searching backwards from the current activation.

```
C1(Va)    = { R,Y }
C2(Vb)    = { G,Y,B }
C3(Vc)    = { B,Y,G }
C4(Va,Vb) = { (R,G),(R,B) }
C5(Vb,Vc) = { (G,B),(B,G) }
C6(Va,Vc) = { (R,B),(R,G) }
```

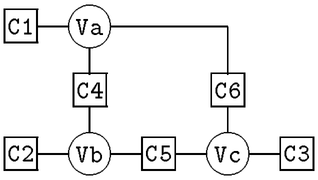


Figure 4.1: Constraint Network A: Variables are represented by circles and constraints by rectangles. An edge between a circle and a rectangle means that the corresponding variable belongs to the constraint represented by the rectangle.

```
C1(Va)    = { R,G }
C2(Vb)    = { R,G }
C3(Vc)    = { R,G }
C4(Va,Vb) = { (R,G),(G,R) }
C5(Vb,Vc) = { (R,G),(G,R) }
C6(Va,Vc) = { (R,G),(G,R) }
```

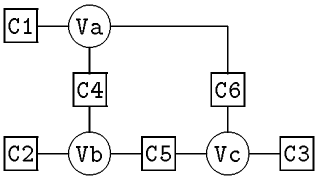


Figure 4.2: Constraint Network B

Tagged Values For any variable of a constraint network, the potential values are associated with additional information called *full tags*, and values of different variables with the same full tag form a global solution of the constraint network. For a constraint network of m constraints, each full tag is an m-tuple, where the rth component (the rth *subtag*) indicates the value tuple in the rth constraint that is part of the global solution. (For example, in constraint `C4` of Figure 4.1 the tuple `(R,B)` corresponds to the subtag 2.) The special symbol "-" is the wildcard character for subtags and indicates that a value tuple for the corresponding constraint has not been chosen, so a full tag containing "-" as subtags represents a set of full tags without "-". A subtag with a tuple number is said to be *determined*, while one with the special symbol "-" is said to be *undetermined*. As an example, consider the global solutions of constraint network A shown in Figure 4.3. The two solutions `(Va,Vb,Vc)` are `(R,G,B)` with full tag `(1,1,1,1,1,1)` and `(R,B,G)` with full tag `(1,3,3,2,2,2)`. An intuitive explanation of the use of tags for constraint network A is given in Figure 4.5.

Filtering of Tagged Values The *feasible set* of a variable contains its current set of potential (tagged) values, and is initialized to the special value `UnConstrained`, meaning the variable may take any value from the domain under consideration. The filtering function f of the constraint network uses local propagation of tagged values to eliminate inconsistent values from the feasible sets, and replaces undetermined subtags with determined subtags. Guesgen showed in his thesis that filtering of

```
Initial:                                                           Step
    Va = Vb = Vc = UnConstrained                                      0

C1: Va = { R(1,-,-,-,-,-), Y(2,-,-,-,-,-) }                           1

C2: Vb = { G(-,1,-,-,-,-), Y(-,2,-,-,-,-), B(-,3,-,-,-,-) }           2

C3: Vc = { B(-,-,1,-,-,-), Y(-,-,2,-,-,-), G(-,-,3,-,-,-) }           3

C4: Va = { R(1,1,-,1,-,-), R(1,3,-,2,-,-) }                           4
    Vb = { G(1,1,-,1,-,-), B(1,3,-,2,-,-) }

C5: Vb = { G(1,1,1,1,1,-), B(1,3,3,2,2,-) }                           5
    Vc = { B(1,1,1,1,1,-), G(1,3,3,2,2,-) }

C6: Va = { R(1,1,1,1,1,1), R(1,3,3,2,2,2) }                           6
    Vc = { B(1,1,1,1,1,1), G(1,3,3,2,2,2) }

C4: Va = { R(1,1,1,1,1,1), R(1,3,3,2,2,2) } *                         7
    Vb = { G(1,1,1,1,1,1), B(1,3,3,2,2,2) }

C5: Vb = { G(1,1,1,1,1,1), B(1,3,3,2,2,2) } *                         8
    Vc = { B(1,1,1,1,1,1), G(1,3,3,2,2,2) } *

C1: Va = { R(1,1,1,1,1,1), R(1,3,3,2,2,2) } *                         9

C2: Vb = { G(1,1,1,1,1,1), B(1,3,3,2,2,2) } *                        10

C3: Vc = { B(1,1,1,1,1,1), G(1,3,3,2,2,2) } *                        11

Solution:
{ Va = R(1,1,1,1,1,1); Vb = G(1,1,1,1,1,1); Vc = B(1,1,1,1,1,1) }
{ Va = R(1,3,3,2,2,2); Vb = B(1,3,3,2,2,2); Vc = G(1,3,3,2,2,2) }
```

Figure 4.3: Activations of constraint network A: (Component-wise unification of full tags is the key idea behind the algorithm.)

```
Initial:                                                      Step
    Va = Vb = Vc = UnConstrained                                 0

C1: Va = { R(1,-,-,-,-,-), G(2,-,-,-,-,-) }                      1

C2: Vb = { R(-,1,-,-,-,-), G(-,2,-,-,-,-) }                      2

C3: Vc = { R(-,-,1,-,-,-), G(-,-,2,-,-,-) }                      3

C4: Va = { R(1,2,-,1,-,-), G(2,1,-,2,-,-) }                      4
    Vb = { G(1,2,-,1,-,-), R(2,1,-,2,-,-) }

C5: Vb = { R(2,1,2,2,1,-), G(1,2,1,1,2,-) }                      5
    Vc = { G(2,1,2,2,1,-), R(1,2,1,1,2,-) }

C6: Va = { }                                                     6
    Vc = { }

C4: Va = { } *                                                   7
    Vb = { }

C5: Vb = { } *                                                   8
    Vc = { } *

No solution
```

Figure 4.4: Activations of constraint network B

```
  (Va,Vb,Vc) = (R,G,B) with full tag (1,1,1,1,1,1)

C1(Va)    = { R,Y }             1  Va      = R
C2(Vb)    = { G,Y,B }           1  Vb      = G
C3(Vc)    = { B,Y,G }           1  Vc      = B
C4(Va,Vb) = { (R,G), (R,B) }    1  (Va,Vb) = (R,G)
C5(Vb,Vc) = { (G,B), (B,G) }    1  (Vb,Vc) = (G,B)
C6(Va,Vc) = { (R,B), (R,G) }    1  (Va,Vc) = (R,B)


  (Va,Vb,Vc) = (R,B,G) with full tag (1,3,3,2,2,2)

C1(Va)    = { R,Y }             1  Va      = R
C2(Vb)    = { G,Y,B }           3  Vb      = B
C3(Vc)    = { B,Y,G }           3  Vc      = G
C4(Va,Vb) = { (R,G), (R,B) }    2  (Va,Vb) = (R,B)
C5(Vb,Vc) = { (G,B), (B,G) }    2  (Vb,Vc) = (B,G)
C6(Va,Vc) = { (R,B), (R,G) }    2  (Va,Vc) = (R,G)
```

Figure 4.5: Globally Consistent Solutions of Constraint Network A

tagged values is guaranteed to terminate with the globally consistent solutions of a constraint network in a finite number of steps provided that it is *fair* [22]. This means each constraint is evaluated (or *activated*) at least once, and if a constraint changes the feasible set of any adjacent variable during filtering, other constraints adjacent to the modified variable have to be re-activated. (This is not the usual definition of fairness in the parallel programming literature.) Upon termination of local propagation, each tuple of values—one value per variable—with identical full tags form a globally consistent solution of the constraint network (see Figure 4.3). If the feasible set of any variable becomes empty, there is no solution for the constraint network (see Figure 4.4). Local propagation computes the fixed point of the filtering function f of the constraint network.

Tagged and Ground Values A tagged value of the form `R(1,-,-,-,-,-)` may be thought of as a placeholder for a set of of *ground* values, say { `R(1,1,1,1,1,1)`, `R(1,3,3,2,2,2)` }. A ground value is a tagged value with no undetermined subtags. A solution of a constraint network consists of ground values only.

Consider a constraint network of m constraints. Let T_r be the set of subtags used by constraint C_r. A tagged value $v(i_1, \ldots, i_m)$ is *well-formed* if v is an element of the domain of simple values, and i_r is an element of T_r or "-". For a well-formed tagged value $v(i_1, \ldots, i_m)$, define

$$\begin{aligned} Ground(v(i_1, \ldots, i_m)) = \{v(i'_1, \ldots, i'_m) \mid \ & \text{if } i_r = \text{“-” then } {i'}_r \in T_r \\ & \text{else } {i'}_r = i_r \\ & \text{for } r \in \{\, 1, \ldots, m \,\}\ \} \end{aligned}$$

Ground computes the set of all ground values for a given tagged value.

Constraint Activation This section describes how subtags are computed by the filtering function f_c of a typical constraint c when c is activated. This description is based on a joint paper "*A Tagging Method for Parallel Constraint Satisfaction*" with Guesgen and Hilfinger [21].

When a constraint C_r is activated, the Cartesian product of the feasible sets of its adjacent variables is computed and then intersected with C_r. The *common* rth subtag of each tuple of values in the intersection is then computed. If all of the rth subtags of a value tuple are undetermined, the common rth subtag is the tuple number of the corresponding value tuple in the definition of C_r. In step 1 of Figure 4.3, the 1st subtag of `R` is `1` because `R` is the first tuple of constraint `C1`, while the 1st subtag of `Y` is `2` because `Y` is the second tuple of `C1`. If some of the rth subtags of a value tuple are determined, and if all of them are identical, then the common rth subtag is this subtag, and all the undetermined rth subtags of the value tuple are updated by this subtag. If some of the determined rth subtags of a value tuple do not match, the common subtag is undefined. A value tuple with an undefined common subtag represents an invalid value combination with respect to the subtags and is deleted from the set of permitted tuples.

For each remaining value tuple (with identical rth subtags) the full tags for all the values of the tuple are *unified*. For each constraint C_j: If all the jth subtags of a value tuple (one per value) are undetermined, the jth subtags are unchanged. If some of the jth subtags of a value tuple are determined, and if all of them are identical, then all the undetermined jth subtags of the value tuple are updated by this subtag. If the determined jth subtags of a value tuple are not identical, the values of the tuple are deleted from the feasible sets of the respective variables. In step 4 of Figure 4.3, the value tuples `(Va,Vb)` are `(R,G)` and `(R,B)`. For the value tuple `(R,B)`, the common 2nd subtag of `R(Va)` and `B(Vb)` is `3` because the 2nd subtag of `R` in step 1 is undetermined, while the 2nd subtag of `B` in step 2 is `3`. In step 6 of Figure 4.4, the value tuples `(Va,Vc)` are `(R,G)`, `(R,R)`, `(G,G)` and `(G,R)`. `(R,R)` and `(G,G)` are eliminated because their simple values are rejected by `C4`. `(R,G)` and `(G,R)` are eliminated because the common 1st (also 2nd and 4th) subtags of each value tuple are not identical (because $2 \neq 1$).

Filtering is Monotonic with respect to Partial Order $\leq$ of Tagged Values

For two tagged values v_a and v_b with identical domain values, $v_a \leq v_b$ iff every determined rth subtag in v_a has an identical rth subtag in v_b for any constraint C_r.

More formally, $v_a \leq v_b$ iff $Ground(v_b) \subseteq Ground(v_a)$. For example, `R(1,-,-,-,-,-)` $\leq$ `R(1,1,-,1,-,-)` $\leq$ `R(1,1,1,1,1,1)`.

For two sets of tagged values V and W, I extend the definition such that $V \leq W$ iff for every $w \in W$, there exists $v \in V$ such that $v \leq w$. More formally, for $V = \{v_1, \ldots, v_p\}$ and $W = \{w_1, \ldots, w_q\}$, $V \leq W$ iff $\cup_{i=1}^{q} Ground(w_i) \subseteq \cup_{j=1}^{p} Ground(v_j)$. In addition, $\leq$ is defined such that `UnConstrained` $\leq V$ for any set of tagged values V. This definition of $\leq$ is useful for comparing the feasible sets of a variable before and after a constraint activation. ($\leq$ is not defined for general sets of tagged values.) For the activation of constraint network A in Figure 4.3, `Va.Step1` = { `R(1,-,-,-,-,-)`, `Y(2,-,-,-,-,-)` }, and `Va.Step4` = { `R(1,1,-,1,-,-)`, `R(1,3,-,2,-,-)` }. Since Ground(`Va.Step4`) $\subseteq$ Ground(`Va.Step1`), `Va.Step1` $\leq$ `Va.Step4`.

Intuitively, filtering either eliminates a tagged value if it is inconsistent, or replaces one or more undetermined subtags of a tagged value by determined subtags. A determined subtag is *never* replaced by an undetermined subtag or another determined subtag. Informally, $\leq$ is the superset comparison ($\supseteq$) between successive states of a feasible set.

4.2 CONSAT as a Parallel Application

CONSAT is a challenging symbolic application for parallel programming systems because it is a medium-sized system (of about 8,000 lines of Common Lisp) that has been developed without consideration for parallel implementations. The system seems to involve many independent and therefore parallelizable computations, yet it is appears to be difficult to obtain good parallel performance. The computations are highly data-dependent, and concurrent accesses to shared data objects have to be performed in non-trivial ways to prevent them from becoming bottlenecks. A good understanding of the application is necessary for creating an efficient parallel implementation. An automatic parallelization tool is unlikely to have much success with CONSAT.

4.3 Parallelizing CONSAT

In the following sections I will describe the sources of parallelism in the CONSAT algorithm, and explain how the parallelism and data-sharing abstractions of the toolbox simplify the creation of a parallel version of CONSAT.

4.3.1 Sequential CONSAT

```
function CONSAT(All-Variables, All-Constraints)
    let WorkQueue = Make-PQ()
        for each v ∈ All-Variables
            Feasible-Set(v) := UnConstrained
        for each c ∈ All-Constraints
            PQ-Insert(c, WorkQueue)
        while not PQ-Empty-p(WorkQueue)
            Activate-Constraint(PQ-Delete(WorkQueue))
```

Figure 4.6: Sequential CONSAT: Activate-Constraint(c) may add other constraints to `WorkQueue`. `All-Constraints` and `All-Variables` represents all the constraints and all the variables of the constraint network respectively.

Figures 4.6 and 4.7 illustrate the main loop of sequential CONSAT. The globally consistent solutions of the constraint network are found in `Feasible-Set` when `WorkQueue` becomes empty.

4.3.2 Introducing Parallelism

Guesgen proved in his thesis that the correctness of CONSAT does not depend on the order the constraints are activated. (CONSAT is an example of a large class of algorithms with what Barth calls *allowable indeterminacy* [6].) If accesses to global data structures (`Feasible-Set` and `WorkQueue`) are synchronized properly, multiple instances of `Activate-Constraint` can operate on different constraints concurrently. This source of parallelism may be expressed using the `Fixed-Point` parallelism abstraction from the parallel programming toolbox:

```
function Activate-Constraint(c)
    ;; Activate-Constraint is a local function of CONSAT.
    ;; c and d are constraints.
    ;; FS-Copy is local to Activate-Constraint.
    ;; Variable WorkQueue is visible.
    for each variable i adjacent to c                        (R)
        FS-Copy(i) := Copy(Feasible-Set(i))
    for each variable i adjacent to c                        (M)
        eliminate values inconsistent with c from FS-Copy(i)
    for each variable i adjacent to c                        (W)
        if Feasible-Set(i) ≠ FS-Copy(i) then
            Feasible-Set(i) := FS-Copy(i)
            for each d (d ≠ c) adjacent to i
                PQ-Insert(d, WorkQueue)
                [FP-Insert-fn(d)]
```

Figure 4.7: Activate-Constraint: For any variable i its (shared) feasible set is denoted Feasible-Set(i). When using Fixed-Point (Section 4.3.2) the last line is replaced by FP-Insert-fn(d). Locking details have been omitted.

`Fixed-Point(Activate-Constraint, Sch-Strategy, All-Constraints)`

`Fixed-Point` invokes `Sch-Strategy` to create a work queue (`WorkQueue`, which is not shown) with an application-specific scheduling strategy, enqueues all the constraints (`All-Constraints`) into `WorkQueue`, and repeatedly invokes `Activate-Constraint` on each constraint removed from `WorkQueue` until `WorkQueue` becomes empty. The call `PQ-Insert(`d`, WorkQueue)` in Figure 4.7 is now replaced by the call `FP-Insert-fn(`d`)`. `FP-Insert-fn` is a function defined by `Fixed-Point`, and is used to add new elements to `WorkQueue`. By providing sequential and parallel implementations of `Fixed-Point`, the version of CONSAT using `Fixed-Point` can be executed on sequential and parallel machines without modification. All the low-level parallel programming details are handled by the implementation of `Fixed-Point`.

4.3.3 Application-Specific Scheduling

CONSAT uses an application-specific work queue for both correctness and performance. This work queue guarantees that multiple instances of any given constraint are not activated concurrently for correctness. If a constraint c is currently active, the work queue ensures that it is added to the work queue in the blocked state. (This happens when another constraint d modifies one of c's variables while c is being activated.)

4.3.4 Shared Data Accesses

In this section I explain the use of an optimistic protocol for accessing `Feasible-Set`, the set of current feasible values of the constraint variables, to obtain good parallel performance. The theory behind this optimistic protocol is described in Chapter 5.

Activate-Constraint as Read-Modify-Write Each call to `Activate-Constraint` destructively modifies `Feasible-Set` in a read-modify-write operation (Figure 4.7). In `Activate-Constraint` the feasible sets of the variables adjacent to the constraint being activated are copied from the shared `Feasible-Set` into `FS-Copy` (read); values inconsistent with the constraint are eliminated from `FS-Copy` (modify); and the new `FS-Copy` is written back to `Feasible-Set` destructively (write). These phases are denoted by (R), (M) and (W) in Figure 4.7 respectively.

A simple parallel implementation of this read-modify-write operation may be constructed by invoking `Activate-Constraint` in a critical section. As the shared `Feasible-Set` is heavily contended, this implementation effectively serializes the concurrent calls to `Activate-Constraint`, leading to poor parallel performance.

Monotonic Asynchronous Iteration To improve the parallel performance of concurrent calls to `Activate-Constraint`, I use an optimistic protocol, *monotonic asynchronous iteration*, to handle concurrent accesses to `Feasible-Set`. Monotonic asynchronous iteration does not lock `Feasible-Set` during the time-consuming modify phase of the computation. Instead, it uses the monotonicity predicate $\leq$ defined in Section 4.1 (Page 71) as an application-specific test to detect *incompatible* changes to `Feasible-Set` by concurrent calls to `Activate-Constraint` during the modify phase of the current call to `Activate-Constraint`. (Not all changes to `Feasible-Set` by concurrent `Activate-Constraint` calls are necessarily incompatible.) This test is shown as part of the (I) phase in Figure 4.8. If an incompatible change to `Feasible-Set` is detected, (the body of) the current call to `Activate-Constraint` is re-executed. Monotonic asynchronous iteration is difficult to use because it requires the programmer to use high-level understanding about the application to define incompatible race conditions, but has the potential of improving the performance of concurrent accesses to a heavily contended object such as `Feasible-Set`. Substantial changes to an existing program may be necessary to convert it to use monotonic asynchronous iteration.

Optimistic Read-Modify-Write Figure 4.9 shows an alternative formulation of the monotonic asynchronous iteration version of `Activate-Constraint`, written using a data-sharing abstraction from the parallel programming toolbox called *Optimistic Read-Modify-Write* (`ORMW`). The explicit locking of `Feasible-Set` and the retry loop in the case of incompatible races in Figure 4.8 has been replaced by `ORMW`, which has a sequential interface. `ORMW` takes five arguments: the object to be destructively modified (`Feasible-Set`) and four functions corresponding to the read, modify, write and incompatibility-testing phases of monotonic asynchronous iteration respectively. `ORMW` returns when the write phase succeeds, i.e., no incompatible race to `Feasible-Set` has been detected (in the current attempt). `ORMW` enables the programmer to express a high performance, optimistic update protocol for `Feasible-Set` without getting involved with the low-level parallel programming details, which are hidden by the implementation of `ORMW`.

4.4 Summary

In this chapter I described the parallelization of CONSAT, a constraint system that computes globally consistent solutions, by using the parallel programming toolbox. CONSAT is a challenging application for symbolic parallel programming systems because it is a large system that has been written without consideration for parallel implementations.

The initial toolbox version of CONSAT is developed by converting the explicit work queue of sequential CONSAT to use `Fixed-Point`. An application-specific work queue is developed to ensure that multiple instances of any given constraint are not activated concurrently in the parallel implementation. (This condition is necessary for correctness.) Finally, accesses to `Feasible-Set` are synchronized using an autolock. These relatively simple changes resulted in a toolbox version of CONSAT that runs (correctly) on uniprocessors and multiprocessors. However, the parallel performance of this version is relatively poor because of contention for `Feasible-Set`. By using the probe process provided by the toolbox (Section 8.1), it is determined that each read-modify-write access to `Feasible-Set` takes a relatively long time to complete. Since these accesses are synchronized using an autolock, the parallel performance is poor.

The next (current) toolbox version of CONSAT is developed by using optimistic concurrency control to handle concurrent accesses to `Feasible-Set`. A detailed analysis of the theory behind the CONSAT algorithm yielded the condition for legal state transitions for successive values of `Feasible-Set` (in a parallel implementation). As this analysis requires an understanding of the CONSAT algorithm, an automatic parallelization tool is extremely unlikely to have much success with CONSAT. The condition for legal state transitions was not derived in the original theoretical work on CONSAT because the condition is automatically satisfied in any sequential implementation of the CONSAT algorithm. The condition for incompatible races is then derived, and accesses to `Feasible-Set` are rewritten to use `ORMW`. A significant amount of code has to be rewritten. The `ORMW` version of CONSAT gives good sequential and parallel performance.

The theoretical work on parallelizing CONSAT is generalized to a scheme for performing optimistic parallel discrete relaxations. This will be discussed in detail in Chapter 5.

```
function Activate-Constraint(c)
;; c is a constraint.
;; FS-Copy is local to Activate-Constraint.

again:
    for each variable i adjacent to c                          (R)
        FS-Copy(i) := Copy(Feasible-Set(i))
        ;; fast (pointer) copy
    for each variable i adjacent to c                          (M)
        eliminate values inconsistent with c
        from FS-Copy(i)
    Lock(Feasible-Set)                                         (I)
    for each variable i adjacent to c
        if Feasible-Set(i) ⊈ FS-Copy(i) then
            Unlock(Feasible-Set)
            goto again
        for each variable i adjacent to c
            if Feasible-Set(i) ≠ FS-Copy(i) then
            ;; set (not pointer) comparison
                Feasible-Set(i) := FS-Copy(i)                  (W)
                ;; fast (pointer) assignment
                for each constraint d ≠ c adjacent to i
                    FP-Insert-fn(d)
    Unlock(Feasible-Set)
```

Figure 4.8: Activate-Constraint using Monotonic Asynchronous Iteration: The (R) and (M) phases are not locked, while the (I) and (W) phases are performed in a single critical section. There is no locking in the (R) phase because the representation of Feasible-Set guarantees that Copy(Feasible-Set(i)) will return either the current value of Feasible-Set(i) or an outdated (but formerly valid) value, i.e. the feasible sets are updated atomically. For a shared object that (unlike Feasible-Set) is not updated atomically, explicit locking has to be added to the (R) phase.

```
function Activate-Constraint(c)
;; c is a constraint.
;; FS-Copy is local to Activate-Constraint.
    function Read(Feasible-Set)                              (R)
        for any variable i adjacent to c
            FS-Copy(i) := Copy(Feasible-Set(i))
            ;; fast (pointer) copy
        return(FS-Copy)

    function Modify(FS-Copy)                                 (M)
        for any variable i adjacent to c
            eliminate values inconsistent with c
            from FS-Copy(i)
        return(FS-Copy)

    function Incompat-p(Feasible-Set, FS-Copy)               (I)
        for any variable i adjacent to c
            if Feasible-Set(i) ⊈ FS-Copy(i) then
            ;; set (not pointer) comparison
                return(t) ; incompatible race:  retry
        return(nil)

    function Write(Feasible-Set, FS-Copy)                    (W)
        for any variable i adjacent to c
            if Feasible-Set(i) ≠ FS-Copy(i) then
            ;; fast (pointer) assignment
                Feasible-Set(i) := FS-Copy(i)
                for each constraint d ≠ c adjacent to i
                    FP-Insert-fn(d)

    ORMW(Feasible-Set, Read, Modify, Write, Incompat-p)
```

Figure 4.9: Activate-Constraint using `ORMW`: The (R) and (M) phases are not locked, while the (I) and (W) phases are performed in a single critical section. The (R) phase is not locked because updates to Feasible-Set are atomic.

Chapter 5

Optimistic Parallel Discrete Relaxation

Discrete relaxation is frequently used to compute the fixed point of a discrete system $X = f(X)$, where f is monotonic with respect to some partial order $\leq$. Given an appropriate initial value for X, discrete relaxation repeats the assignment $X \leftarrow f(X)$ until a fixed point of f is found. Monotonicity of f with respect to $\leq$ is a sufficient (but in general not necessary) condition for iterative, hill-climbing techniques such as discrete relaxation to find the fixed point of f [53].

Discrete relaxation is widely used in the solution of constraint satisfaction problems (CSPs), and many parallel implementations of discrete relaxation for CSPs have been reported [43, 57]. These attempts have all focused on CSP solvers that compute locally consistent (arc consistent) solutions, which are relatively straightforward to parallelize as the computations are inherently monotonic. On the other hand, discrete relaxation algorithms used in CSP solvers that compute globally consistent solutions are very difficult to parallelize because for this class of problems, monotonicity is a necessary correctness condition that is not automatically satisfied. The need to maintain monotonicity (for correctness) often limits the amount of concurrency available in a parallel implementation, degrading the performance significantly.

[1] This chapter is based on a paper co-authored with Hilfinger and Guesgen, "Optimistic Parallel Discrete Relaxation" [30].

In this chapter I introduce *monotonic asynchronous iteration* as a novel way of implementing parallel discrete relaxation in problem domains for which monotonicity is a necessary condition. This is an optimistic technique that maintains monotonicity without limiting concurrency, resulting in good parallel performance. I will illustrate this technique with the parallel implementation of CONSAT [22], a constraint satisfaction system that computes globally consistent solutions. I believe that monotonic asynchronous iteration is applicable to parallel discrete relaxation in general.

5.1 Discrete Relaxation

Consider the problem of finding the fixed point of a discrete system $X = f(X)$ with the following properties:

- f: $D \times D$, for some domain D, is a relation. I shall notate it as a nondeterministic function, writing, for example, $Y \leftarrow f(X)$ to mean "set Y to some value standing in relation f with X." An equation $Y = f(X)$ means that Y is a value satisfying the relation.

- Values in D are structured, so that for any $X \in D$, $X = \langle X_1, \ldots, X_n \rangle$, where $X_i \in D_i$ and $D = D_1 \times \cdots \times D_n$.

- $I = \{1, \ldots, n\}$ is the *index set.*

- There exist functions f_c $(c \subseteq I)$ such that a solution of $X = f(X)$ is a solution of a system of equations $X = f_c(X)$ for all $c \subseteq I$, where each f_c: $D \rightarrow D$ is deterministic. Each relaxation step $X \leftarrow f(X)$ is equivalent to $X \leftarrow f_c(X)$ for some $c \subseteq I$. For $i \in I$ and $c \subseteq I$, $f_c(X) = \langle Y_1, \ldots, Y_n \rangle$, where

$$Y_i = \begin{cases} X_i & \text{if } i \notin c \\ {}_if_c(X) & \text{if } i \in c \end{cases}$$

  The new value of any such component i $(i \in c)$ is given by ${}_if_c(X)$ $({}_if_c : D \rightarrow D_i)$. $f_c(X)$ only differs from X at the indices in c.

- f is monotonic with respect to some partial order $\leq$, i.e., $X \leq f(X)$ for any $X \in D$. Consequently, $X \leq f_c(X)$ for any $c \subseteq I$, and $X_i \leq_i {}_if_c(X)$ for $i \in I$. For $X, Y \in D$, where $X = \langle X_1, \ldots, X_n \rangle$ and $Y = \langle Y_1, \ldots, Y_n \rangle$, $X \leq Y$ iff $X_i \leq_i Y_i$ for all $i \in I$.

Assumption I assume that domain-specific knowledge has been used to ensure that f has a fixed point that can be computed in a finite number of steps using discrete relaxation.

Notation Let X^j denote the value of X computed in the jth iteration of a discrete relaxation, and let X^0 be the initial value of X.

Sequential Iterations Given an appropriate X^0, discrete relaxation repeats the assignment $X \leftarrow f(X)$ until a fixed point for f is found. In a sequential implementation of discrete relaxation, $X \leftarrow f(X)$ becomes $X^j = f(X^{j-1})$, or $X^j = f_c(X^{j-1})$ for some $c \subseteq I$.

For $i \in I$ and $j = 1, 2, \ldots$, a *sequential iteration* has the following form:

$$X_i^j = \begin{cases} X_i^{j-1} & \text{if } i \notin c \\ {}_if_c(X^{j-1}) & \text{if } i \in c \end{cases}$$

where $c \subseteq I$ is not fixed for successive values of j. X^j only depends on X^{j-1}, and not on other (older) values of X.

Asynchronous Iterations To compute the fixed point of f in parallel, f_c functions for different values of c ($c \subseteq I$) may be computed concurrently. For example, f_c and f_d ($c, d \subseteq I$) should be allowed to proceed in parallel if c and d do not intersect. This is not possible under sequential iterations, because X^j is computed using the value of X from the most recent iteration (X^{j-1}) only, serializing the computations for f_c and f_d. Intuitively, f_c and f_d may be computed in parallel if the restriction that X^j is computed using the value from the most recent iteration (X^{j-1}) is relaxed so that values from some *relatively recent* iterations may be used instead.

Following Baudet [7], I define *asynchronous iterations* by removing the restriction imposed by sequential iterations. An asynchronous iteration $X^j = f'_c(X^{src(j,1\ldots n)}, X^{j-1})$ is defined as follows:

$$X_i^j = \begin{cases} X_i^{j-1} & \text{if } i \notin c \\ {}_if_c(X^{src(j,1\ldots n)}) & \text{if } i \in c \end{cases}$$

where $X^{src(j,1\ldots n)} = \langle X_1^{src(j,1)}, \ldots, X_n^{src(j,n)} \rangle$, and $src(j,k) \in \{0, \ldots, j-1\}$. The function $src(j,k)$ determines what past value of X_k is used in the computation. For any $k \in I$, $X_k^{src(j,k)}$ is the value of X_k used by ${}_if_c$ to compute X_i^j. There are two observations:

- $src(j,k)$ is not a function of i. For any value of $k \in I$, ${}_if_c$ $(i \in c)$ uses the value of X_k computed in iteration $src(j,k)$, $X_k^{src(j,k)}$.
- For a given value of j, $src(j,k)$ is a function of k. Consequently, the value of X_k $(k \in I)$ used to compute X_i^j may have been computed in different iterations. This freedom reduces the amount of synchronization required in a parallel implementation of asynchronous iterations significantly.

For example,

$$\begin{aligned} X_2^7 &= {}_2f_{\{2,3\}}(\langle X_1^{src(7,1)}, X_2^{src(7,2)}, X_3^{src(7,3)} \rangle) \\ &= {}_2f_{\{2,3\}}(\langle X_1^5, X_2^3, X_3^4 \rangle), \text{and} \\ X_3^7 &= {}_3f_{\{2,3\}}(\langle X_1^{src(7,1)}, X_2^{src(7,2)}, X_3^{src(7,3)} \rangle) \\ &= {}_3f_{\{2,3\}}(\langle X_1^5, X_2^3, X_3^4 \rangle) \end{aligned}$$

For sequential iterations $src(j,k) = j - 1$.

The value of X^j computed by an asynchronous iteration

$$X^j = f'_c(X^{src(j,1\ldots n)}, X^{j-1})$$

depends on multiple previous states. The components of X not modified by f'_c come from the most recent state, X^{j-1}, while the components of X modified by f'_c come from $X^{src(j,1\ldots n)}$. As X^j is not computed using a single (consistent) state of X, f'_c is *not* monotonic in general, and so $X^{j-1} \not\leq f'_c(X^{src(j,1\ldots n)}, X^{j-1})$. Consequently, asynchronous iterations may lead to incorrect results for parallel discrete relaxations.

Monotonic Asynchronous Iterations Given an arbitrary discrete, monotonic and non-deterministic function f with functions f_c ($c \subseteq I$), it is relatively difficult to derive a condition such that the corresponding f'_c functions defined by an asynchronous iteration are monotonic. To use asynchronous iteration for parallel discrete relaxation, I apply an application-specific test for monotonicity to the result computed by f'_c such that if $X^{j-1} \leq f'_c(X^{src(j,1\ldots n)}, X^{j-1})$, then X^j is updated using this value of f'_c. Otherwise, f'_c is re-computed (using more recent values of X). I call this optimistic scheme *monotonic asynchronous iteration*:

$$X_i^j = \begin{cases} {}_if_c(X^{src(j,1\ldots n)}) & \text{if } i \in c \text{ and } \forall \alpha \in c, \\ & X_\alpha^{j-1} \leq {}_\alpha f_c(X^{src(j,1\ldots n)}) \\ X_i^{j-1} & \text{otherwise} \end{cases}$$

If f'_c and f'_d ($c, d \subseteq I$) are computed concurrently, a *race* occurs if X is modified by f'_d after the values of $X^{src(j,1\ldots n)}$ used to compute ${}_if_c(X^{src(j,1\ldots n)})$ have been read, but before X is modified by f'_c. Monotonic asynchronous iteration *only* detects (and rejects) races that would violate monotonicity. Races that maintain monotonicity are allowed, so X^{j-1} and $X^{src(j,1\ldots n)}$ do not have to be equal. In fact, these *legal* races provide a source of concurrency not found in other parallel relaxation schemes, and lead to improved performance. Intuitively, f_c and f_d may be computed concurrently if they modify disjoint components of X, or if they modify common components of X in such a way that monotonicity between successive values of X is maintained.

I do not have a general way of deriving the application-specific test for monotonicity for a given system. Instead, I expect the programmer to provide this test by using high-level knowledge about the problem domain. In the context of CSP solvers that compute globally consistent solutions, this test reduces to a subset test between successive sets of value combinations that may be associated with the variables of a constraint network (see Section 4.1).

Parallel Implementation of Monotonic Asynchronous Iterations A step of a monotonic asynchronous iteration $X^j = f'_c(X^{src(j,1\ldots n)}, X^{j-1})$ with partial order $\leq$ may be implemented in parallel in the following way:

1. Read current value of X, $X^{src(j,1\ldots n)} = \langle X_1^{src(j,1)}, \ldots, X_n^{src(j,n)} \rangle$
   (X_ks may come from different iterations.)

2. Compute ${}_if_c(X^{src(j,1\ldots n)})$ for all $i \in c$

3. Lock X

4. if $\forall\ i \in c$: $X_i^{j-1} \leq {}_if_c(X^{src(j,1\ldots n)})$
   then for each $i \in c$, update X:$X_i^j = {}_if_c(X^{src(j,1\ldots n)})$
   else schedule f'_c for re-execution

5. Unlock X

X is not locked while the ${}_if_c(X^{src(j,1\ldots n)})$ values are computed in step 2. If these computations are time-consuming, this implementation can lead to much better parallel performance than a naive alternative that locks X for the entire step of the monotonic asynchronous iteration. Nevertheless, the maximum speedup is bounded by $1+(T_1+T_2)/T_4$, where T_i is the time for step i. For example, if T_4 is 10% of (T_1+T_2), the speedup limit is 11.

Monotonic Asynchronous Iteration using `ORMW` A monotonic asynchronous iteration may be implemented using the `ORMW` data-sharing abstraction. Step 1 above corresponds to `Read-fn` of `ORMW`, Step 2 corresponds to `Modify-fn`, the update of X in Step 4 corresponds to `Write-fn`, and the $\leq$ tests together correspond to `Incompatible-p-fn`. Low-level parallel programming details, including the locking of X and re-execution in the event of incompatible races, are hidden by the implementation of `ORMW`. The programmer simply specifies what has to be done using the `ORMW` interface, while the toolbox implementation decides how the actions are actually performed. The use of `ORMW` in CONSAT is described in Section 4.3.4.

Practical Considerations Monotonic asynchronous iteration does not specify how a function f'_c $(c \subseteq I)$ is chosen for execution in any iteration j, or whether f'_c and f'_d $(c, d \subseteq I)$ should be executed concurrently in a parallel implementation for optimal performance. There is considerable freedom in applying application-specific scheduling strategies for good performance.

5.2 Discrete Relaxation and Fixed-Point Computations

The formulation of discrete relaxation in Section 5.1 is very similar to the formulation of fixed-point computations in Section 2.1.1. There are two minor differences:

- For any $i \in I$, the component X_i in discrete relaxations is equivalent to the component $X_{\{i\}}$ in fixed-point computations.
- The notation X_c, where $c \subseteq I$, is not used in discrete relaxations.

5.3 CONSAT as Discrete Relaxation

In this section I show that CONSAT is a special case of discrete relaxation as defined in Section 5.1.

- CONSAT computes the fixed point of the discrete system $X = f(X)$ for the non-deterministic f that corresponds to the filtering function of the entire constraint network. This fixed point corresponds to the set of all the globally consistent solutions of the constraint network.
- $I = \{1, \ldots, n\}$ is the set of all the variables of the constraint network.
- f may be decomposed into component functions f_c for constraints $c \subseteq I$. f_c: $D \rightarrow D$ is the filtering function of constraint c, and eliminates value combinations inconsistent with c from the feasible sets of its adjacent variables. Without loss of generality, I assume that a constraint may be uniquely identified by its set of adjacent variables. For example, $f_{\{2,3,5\}}$ is the filtering function of the constraint that is adjacent to variables 2, 3 and 5. f_c only depends on the values of variables $i \in c$, and $f_c(X)$ only differs from X for the same set of

variables. The new value of any such variable i is given by ${}_i f_c(X)$. The details of the f_c functions for CONSAT are described on page 70.

- For $X \in D$, $X = \langle X_1, \ldots, X_n \rangle$. Each $X_i \in D_i$ is the feasible set of variable i.
- X^0 is the initial state of the feasible sets of all the variables. All its components have the special value `UnConstrained`.
- The partial order $\leq$ for $X \in D$ has been defined on page 71.

I have shown that CONSAT is a special case of discrete relaxation, and have defined an application-specific monotonicity test between successive states of the relaxation. Consequently, an efficient parallel implementation of CONSAT may be obtained by using monotonic asynchronous iteration.

5.4 Summary

In this chapter I proposed monotonic asynchronous iteration as a correct and efficient way of implementing parallel discrete relaxation for systems for which monotonicity is a necessary correctness condition. Monotonic asynchronous iteration uses an optimistic scheme to compute a possible next state of the system. This optimistic scheme is highly efficient but is not necessarily monotonic (i.e., correct). An application-specific test for monotonicity is then applied to the computed state, and the state transition is made (atomically) if the test succeeds. Otherwise, the computation is repeated. I have applied monotonic asynchronous iteration successfully to the parallel implementation of a constraint satisfaction system that computes globally consistent solutions, for which monotonicity is a necessary correctness condition that is not automatically satisfied. I believe monotonic asynchronous iteration is applicable to parallel discrete relaxation in general. It will be interesting to see if this application-specific monotonicity test is easy to derive for other discrete relaxation problems.

Chapter 6

Application: RC

6.1 Introduction

RC performs conservative type determination of Lisp programs using dataflow analysis that is commonly done by optimizing compilers. The results of its analysis may be used by a Lisp compiler to eliminate unnecessary runtime type checks.

RC is a challenging application for symbolic parallel programming systems because it is an example of a medium-sized system (of about 7,000 lines of Common Lisp) that was written without consideration for parallelism. In particular, many techniques used by the system appear to be inherently sequential. There is a large variety of shared data objects in the parallel implementation, and a good understanding of the application is necessary for the realization of an efficient parallel implementation. An automatic parallelization tool is unlikely to have much success. In this chapter I give a brief overview of RC, and describe the challenges that were overcome to produce an efficient parallel implementation.

6.2 RC Overview

RC is a prototype implementation of an experimental type analysis algorithm. It derives recursive type information from type checks in a program, and uses the information to remove other type checks in the same program. For example, information from the type checks in one loop over a list can be used to remove checks in subsequent loops over the same list.

Abstractly, the type description is a set of productions called a *p-set*. Each production is in the form $(u_0 \rightarrow t u_1 \ldots u_k)$, where t is a terminal (a type symbol), and the u_i's are nonterminals. A nonterminal is a set of small integer identifiers representing variables and expressions. For example, if $\bar{x}$ and $\bar{y}$ are variable identifiers (for variables x and y, respectively), and 10 is an expression identifier, and I assume there are only three types (`cons`, `nil`, and `other`), then this is a valid p-set that describes x and y:

$$
\begin{aligned}
&\{\bar{x}\} \rightarrow \texttt{cons}\ \{\bar{y}\}\ \{10\}\\
&\{\bar{y}\} \rightarrow \texttt{nil}\\
&\{10\} \rightarrow \texttt{cons}\ \emptyset\ \{10\}\\
&\{10\} \rightarrow \texttt{nil}\\
&\emptyset \rightarrow \texttt{cons}\ \emptyset\ \emptyset\\
&\emptyset \rightarrow \texttt{nil}\\
&\emptyset \rightarrow \texttt{other}
\end{aligned}
$$

Without being very precise, this p-set says that x must be a `cons`, the car of which is y (a `nil`), and the cdr is described by the nonterminal $\{10\}$, which is a list (a string of zero or more `cons` ending in `nil`). All other elements of the list are of an unknown type. They are described by $\emptyset$, which has productions that can expand into all possible structures within the constraint of the three types.

The algorithm is a forward flow analysis. A p-set is associated with each edge in the program flow graph. For each node in the flow graph, the algorithm defines the p-sets on the out-edges in terms of the union of the p-sets on the in-edges. Initially, all p-sets are empty except on specific entry edges. Then an iterative relaxation is used to repeatedly compute new p-sets until convergence.

In Lisp terms, a p-set is a list of productions. A production is a structure:[1]

```
(defstruct pd
  lhs
  rhs
```

[1] Some fields have been omitted.

```
  mark
  scratch)
```

The field `pd-lhs` is a nonterminal; `pd-rhs` is a list, the first element of which is a type (a symbol), and the rest are nonterminals. The other fields will be explained later. A nonterminal is also a structure:

```
(defstruct nt
  vars
  ids
  pds
  mark
  scratch)
```

The fields `nt-vars` and `nt-ids` are sets of variable and expression identifiers. Both are sorted lists of integers.

Both productions and nonterminals are *interned*, so that equal productions or nonterminals are also identical (`eq`). For this purpose, all nonterminals are stored in a global hash table (an `equal` hash table, keyed by `(cons nt-vars nt-ids)`). Productions are kept in tries in the `nt-pds` fields of their left-hand sides.[2] More precisely, a production p is in a trie stored in `(nt-pds (pd-lhs` p`))`, which is indexed by `(pd-rhs` p`)`.

This interning of objects makes possible several efficient programming techniques. For example, set operations on p-sets are common, and they can be done in linear time using the `pd-mark` field:

```
(defun ps-union (a b)
  (let ((mark (cons nil nil)))
    (dolist (x a)
      (setf (pd-mark x) mark))
    (dolist (x b)
      (unless (eq (pd-mark x) mark)
        (push x a))))
  a)
```

[2]A *trie* is a search structure indexed by a sequence of keys. A description of it can be found in any reasonable data structure book (such as Volume 3 of Knuth [44]).

First, a new unique mark is generated. Then a loop marks all elements of p-set `a`. Finally, all unmarked elements of `b` are added to `a`.

Set operations on nonterminals are less common. More often, `nt-mark` is used in conjunction with `nt-scratch` to create nonterminal-to-nonterminal functions. For example, for an assignment (`setq` x y), the algorithm defines the output p-set, Π_{out}, in terms of the input p-set, Π_{in}, as a renaming of nonterminals:

$$\Pi_{out} = \{(r(u_0) \rightarrow tr(u_1)\ldots r(u_k)) : (u_0 \rightarrow tu_1 \ldots u_k) \in \Pi_{in}\}$$

where r is a function on nonterminals. If $\bar{x}$ and $\bar{y}$ are the variable identifiers, and d is the identifier for the `setq` expression in question, then r is defined this way, over all nonterminals u:

$$r(u) = \begin{cases} u \cup \{\bar{x}, d\} & \text{if } \bar{y} \in u \\ u - \{\bar{x}\} & \text{otherwise.} \end{cases}$$

The corresponding Lisp implementation uses `nt-mark` and `nt-scratch` to represent r:

```
(let ((mark (cons nil nil)))
  (dolist (p p-in)
    (let ((nt (pd-lhs p)))
      (unless (eq (nt-mark nt) mark)
        (setf (nt-mark nt) mark)
        (setf (nt-scratch nt)
              (if (member y (nt-vars nt))
                (nt-add-var-and-id nt x d)
                (nt-delete-var nt x))))))
  (setq p-out (ps-subst-nt p-in mark)))
```

Like the formal definition, this piece of Lisp code computes `p-out` from `p-in`. It does this by first defining the equivalent of the function r in the `nt-scratch` field, using `nt-mark` to avoid recomputation, then applying it to all nonterminals in `p-in` to produce `p-out` (via `ps-subst-nt`).

```
function RC(S, succ)
    let WorkQueue = Make-PQ()
        for each n ∈ S
            if Π_in(n) ≠ ∅ then
                PQ-Insert(n, WorkQueue)
        while not PQ-Empty-p(WorkQueue)
            Activate-Node(PQ-Delete(WorkQueue))
```

Figure 6.1: Sequential RC: As mentioned in Section 6.2, p-sets are initially empty except on specific entry nodes. These make up the initial content of `WorkQueue`.

6.3 RC as a Fixed-Point Computation

6.3.1 Notation

The program flow graph, G, can be thought of as the pair $(S, succ)$, where S is the set of nodes (program statements, roughly equivalent to Lisp expressions), and *succ* is the function on S that maps a node to the set of its (immediate) successors. For the type analysis, p-sets are ideally associated with program points, which are edges in the flow graph. However, due to data structure constraints, RC in fact labels *nodes* with p-sets. For each $n \in S$, $\Pi_{in}(n)$ is the p-set going into node n. Also, if $pred(n)$ is the set of immediate predecessors of n, then

$$\Pi_{in}(n) = \bigcup_{p \in pred(n)} f_{(p,n)}(\Pi_{in}(p)),$$

where the f's are functions defined by the algorithm, one on each edge.

6.3.2 Fixed-Point Computation

A sequential implementation of RC is shown in Figures 6.1 and 6.2. The program flow graph G is specified by S and *succ*. RC may be considered a fixed-point computation $X = F(X)$, where X is the collection of all Π_{in}'s. This version of `RC` is a straightforward implementation of iterative relaxation. It begins with an initial work queue, then repeatedly invokes `Activate-Node` on elements of the queue, removing each as it does so, until the work queue is empty. An invocation of `Activate-Node`

```
function Activate-Node(n)
    ;; Activate-Node is a local function of RC.
    ;; Variables S, succ, and WorkQueue are visible.
    for each s ∈ succ(n)
        let x = f_(n,s)(Π_in(n))
            if x ⊈ Π_in(s) then
                Π_in(s) := Π_in(s) ∪ x
                PQ-Insert(s, WorkQueue)
```

Figure 6.2: `Activate-Node`: When using `Fixed-Point` (see Section 6.3.3), the last line `PQ-Insert(`s`, WorkQueue)` should be replaced by `FP-Insert-fn(`s`)`. Locking details have been omitted (see Section 6.4.1).

```
function RC(S, succ)
    Fixed-Point(Activate-Node, PQ, {n ∈ S : Π_in(n) ≠ ∅})
```

Figure 6.3: RC using `Fixed-Point`

on node n recomputes the operation associated with each out-going edge of n, and if necessary, propagates the result to the successor node and adds it to the work queue. A monotonicity condition ensures a stable solution is eventually reached. In the case of RC, the condition is that the Π_{in}'s can only grow as the computation progresses. In other words, for all n, $\Pi_{in}(n)$ from an earlier iteration is always a (not necessarily proper) subset of $\Pi_{in}(n)$ from a later iteration. This condition will also be useful in a parallel implementation of RC (Section 6.4.1).

6.3.3 Introducing Parallelism

In Figure 6.3, the explicit loop of sequential RC (Figure 6.1) has been replaced by a call to `Fixed-Point`. The first argument to `Fixed-Point` (`Activate-Node`) is the per-node activation function. The second argument (`PQ`) specifies the scheduling strategy of the work queue to be created by `Fixed-Point`. The third argument is the initial content of the queue. `Fixed-Point` creates the work queue, initializes it, and repeatedly removes nodes from the queue, calling the activation function with each node (which may insert more nodes), until the work queue is empty. (Thus, a parallel implementation of `Fixed-Point` returns when the queue becomes empty *and* there is no call to `Activate-Node` in progress.) The function `Activate-Node` is defined as before (Figure 6.2), except the call `PQ-Insert(`s`, WorkQueue)` must now be replaced by `FP-Insert-fn(`s`)`, which is a function defined by `Fixed-Point`, and is used by the application to add new elements to the (now hidden) work queue.

If accesses to global data structures (p-sets, etc.) are synchronized properly, more than one instance of `Activate-Node` can be active at the same time. For example, branches of a conditional node can be traversed in parallel. By providing sequential and parallel implementations of `Fixed-Point`, the version of RC in Figure 6.3 can be executed on sequential and parallel machines without change. All the low-level parallel programming details are handled by `Fixed-Point`.

For correctness or performance, the application may specify a work queue implementation (`PQ`) that enforces application-specific dependencies between the (potentially parallel) computations, without being involved in the details of the internals of `Fixed-Point`.

6.3.4 RC as a Dag Traversal

As most control-flow graphs can be partitioned into directed-acyclic graphs (dag) of strongly-connected components (SCCs, or simply *components*), RC can compute the fixed point more efficiently by working on one component at a time, in a topological order dictated by the dag.

In Figure 6.4, the dag D corresponding to G is computed by `Digraph-to-Dag`. The parallelism abstraction `Dag-Traversal` of the toolbox traverses D such that a component is not visited (by `Dag-Op`) unless all its predecessor components have been visited. For each component C visited by `Dag-Op`, `Fixed-Point` is used to compute the p-sets for the nodes of C. The work queue for the fixed-point computation is initialized to contain all the nodes of C with non-empty p-sets. In a parallel implementation of `Dag-Traversal`, more than one component may be visited concurrently.

```
function RC(S, succ)
    let (Comp-List, Succ-Comps-fn, Comp-to-Nodes-fn,
         Node-to-Comp-fn)
              =Digraph-to-Dag(S, succ)
         Dag-Traversal(Comp-List, Succ-Comps-fn, Dag-Op)

function Dag-Op(C)
    Fixed-Point(Activate-Node, PQ,
                {n ∈Comp-to-Nodes-fn(C): Π_in(n) ≠ ∅})
```

Figure 6.4: RC using `Dag-Traversal`: `PQ` specifies an application-specific work queue that does *not* enqueue a node if it does not belong to the current component, thus isolating the fixed-point computation of each component. The dag is specified by a list of the strongly-connected components (`Comp-List`), a function that maps components to their successors (`Succ-Comps-fn`), a function that maps components to their constituent nodes (`Comp-to-Nodes-fn`), and a function that maps nodes to the components they belong to (`Node-to-Comp-fn`).

In addition, multiple nodes of any given component may be visited concurrently by `Fixed-Point`. Thus, there are two levels of parallelism in this parallel implementation of RC.

6.4 Shared Data Accesses

In this section I describe how the data-sharing abstractions of the parallel programming toolbox simplify concurrent accesses to shared data objects in the (parallel) toolbox version of RC. I will describe how p-sets are modified, how productions and nonterminals are interned, and how false sharing for production and nonterminal fields are eliminated.

6.4.1 P-sets

In function `Activate-Node` of Figure 6.2, the new p-set, x, is propagated to $\Pi_{in}(s)$ using set union. In parallel RC, multiple predecessors of s may attempt to perform the union with $\Pi_{in}(s)$ concurrently, leading to incorrect results.

```
;; s is a successors of n
;; x is the computed p-set (f_(n,s)(Π_in(n)))
again:
let Old-Pin = Copy(Π_in(s))                                    (R)
     if x ⊈ Old-Pin then
          let Modified-Pin, New-Pin
               if One-predecessor-p(s) then
                    Modified-Pin := x                           (M)
               else
                    Modified-Pin := Old-Pin ∪ x                 (M)
               Lock(s)
               New-Pin := Π_in(s)
               if (Old-Pin ≠ New-Pin)                           (I)
                    Unlock(s)
                    goto again
               Π_in(s) := Modified-Pin                          (W)
               Unlock(s)
               FP-Insert-fn(s)
```

Figure 6.5: Locking details of `Activate-Node`: This code segment replaces the body of the `let` block in Figure 6.2.

Figure 6.5 shows the details of an explicitly parallel (i.e. not toolbox-style) algorithm to propagate a new value x to $\Pi_{in}(s)$. To ensure that a successor node with multiple predecessors does not become a bottleneck in parallel RC, the set union operation is *not* performed in a critical section. Instead, it is done between x and a *copy* of $\Pi_{in}(s)$, and an application-specific test is then performed in a critical section to determine whether $\Pi_{in}(s)$ has been modified by an *incompatible* race condition. (Not all concurrent changes to $\Pi_{in}(s)$ are incompatible.) If such a race is detected, the propagation is reexecuted. If not, then $\Pi_{in}(s)$ is updated by its new value while

```
function Read(m)                                              (R)
     return Copy(Π_in(m))

function Modify(Old-Pin)                                      (M)
     let Modifiedp = false, Modified-Pin = x
          if x ⊈ Old-Pin then
               Modifiedp := true
               if One-predecessor-p(s) then
                    Modified-Pin := x
               else
                    Modified-Pin := Old-Pin ∪ x
          return (Modified-Pin, Modifiedp)

function Write(m, Modified-Pin)                               (W)
     Π_in(m) := Modified-Pin

function Incompat-p(m, Old-Pin, Modified-Pin)                 (I)
     let New-Pin = Π_in(m)
          return (Old-Pin ≠ New-Pin)

let (New-Value, Changedp)
       = ORMW(s, Read, Modify, Write, Incompat-p)
     if Changedp then
          FP-Insert-fn(s)
```

Figure 6.6: Locking in `Activate-Node` using `ORMW`

still in the critical section.

Figure 6.6 is the toolbox version of the propagation algorithm, written using the `ORMW` data-sharing abstraction (Section 3.6). The explicit locking and the retry loop in Figure 6.5 have been replaced by `ORMW`, which has a sequential interface. The programmer specifies the conditions for incompatible races, while the `ORMW` implementation is responsible for the detection of these races and the re-executions. `ORMW` takes five arguments: the object to be destructively modified, and four functions corresponding to the read, modify, write and incompatibility-testing phases of the original code. It returns when the write phase succeeds. Like `Fixed-Point`, `ORMW` presents a high-level interface that permits good parallel performance while hiding the parallel programming details.

6.4.2 Interning Productions and Nonterminals

To ensure that productions are interned, a new production is not created if an equivalent one may be found in a dictionary for productions. (A similar dictionary is used for interning nonterminals.) This dictionary supports two operations: lookup and insert. In parallel RC there may be concurrent lookup and insert operations on such a dictionary. The lookup and insert pairs of interning operations are converted to use the conditional-insert operations of the parallel programming toolbox. Details of this operation are discussed in Section 3.4.

6.4.3 False Sharing in RC

The linear-time set union algorithm of Section 6.2 appears to be inherently sequential if an element may belong to multiple sets, as concurrent set operations using this algorithm may conflict at the `pd-mark` fields of elements belonging to multiple sets and lead to incorrect results. This phenomenon is called false sharing.

The toolbox extends the Common Lisp structure-defining facility `defstruct` to support per-thread field definitions. The resulting data-sharing abstraction is called `pdefstruct`. (Thread creation is *not* part of `pdefstruct`. Threads are usually created transparently by a parallelism abstraction, say `Fixed-Point`.) In the set union case the `pdefstruct` has the following form:

```
(pdefstruct pd
    lhs
    rhs
    (mark default :private)
    ...)
```

The form `(mark` *default* `:private)` defines a per-thread slot `mark`, where *default* is the usual default initial value allowed by `defstruct`. In the parallel implementation, all the per-thread locations corresponding to slot `mark` are initialized to the *same* initial value. A structure may have more than one per-thread slot.

Little change was necessary to convert RC to use `pdefstruct`. Structure declarations were changed to replace `defstruct` by `pdefstruct`, and per-thread fields were modified to include the `:private` option. More importantly, there was no change to code accessing or updating a per-thread field (`pd-mark`). Through the use of `pdefstruct`, the algorithms described in Section 6.2 were parallelized relatively easily.

6.5 Summary

In this chapter I described the parallelization RC, an experimental type analysis program, using abstractions from the parallel programming toolbox. RC is a challenging application for symbolic parallel programming system because it is a large system that was originally written without consideration for parallel implementations, using techniques that are difficult to parallelize.

The initial toolbox version of RC is developed by converting the explicit work queue of sequential RC to use `Fixed-Point`. An application-specific work queue is developed to ensure that multiple instances of a given node are not activated concurrently in the parallel implementation. (This condition is necessary for correctness.)

The current toolbox version of RC takes advantage of the fact that the fixed-point computed by RC may be found more efficiently by first decomposing the input program flow graph G into a dag of strongly-connected components, and computing the fixed point of each component separately. The parallelism abstraction `Dag-Traversal` is used to identify the potential parallelism in the dag traversal of G, while `Fixed-Point` is used to find the fixed point of each component. Thus, there are two levels of parallelism in RC.

When a node is activated, a new p-set is propagated to all its immediate successor nodes. The p-sets in the system are updated using the `ORMW` data-sharing abstraction to prevent contention between concurrent updates to the same p-set. (This happens when multiple predecessor nodes of a given node N attempt to propagate their new p-set values to N concurrently.) Productions and nonterminals are interned using toolbox dictionaries. These dictionaries support the conditional-insert operation, which provide a sequential interface, and help eliminate a lot of common race conditions. Finally, the false sharing of fields in productions and nonterminals is eliminated by using structures with per-thread locations to represent productions and nonterminals. Only the definitions of the production and nonterminal datatypes have to be changed—the algorithms that access and update these falsely shared fields are unchanged.

Part III

Immplementation and Evaluation

Chapter 7

Toolbox Implementation

The sequential version of the prototype toolbox is written in Common Lisp, and runs in Allegro Common Lisp (version 4.1). It may be ported to any other Common Lisp implementation relatively easily. The parallel version of the toolbox is written in CLiP [17], a version of Allegro Common Lisp with the multiprocessing features of SPUR Lisp [76]. CLiP currently runs on Sequent Symmetry shared-memory multiprocessors. There are approximately 3,000 lines of code in the (sequential and parallel) implementation of the toolbox.

In this chapter I give a brief overview of the multiprocessing features of SPUR Lisp and CLiP. This set of features may be used to determine the feasibility of porting the toolbox to another parallel Lisp system. I will then describe some performance bottlenecks in the CLiP implementation, and the changes that have been made to remove some of the bottlenecks. These bottlenecks represent the limitations of a real parallel Lisp implementation, and may be of interest to implementors of other parallel Lisp systems.

7.1 Multiprocessing SPUR Lisp

Multiprocessing SPUR Lisp is a superset of Common Lisp with added features for programming shared-address space multiprocessors. These features are flexible but low-level, and are designed to be used to implement various high-level abstractions.

The features include *processes*, which are multiple threads of control in a shared address space; *mailboxes*, which are unbounded FIFOs (of arbitrary Lisp objects) with synchronization for communication between threads; *futures*, which are placeholders for yet uncomputed values introduced in Multilisp [23]; and asynchronous signals (*signal, suspend/resume* and *kill*). SPUR Lisp *special variables* generalize the semantics of Common Lisp special variables to a multi-threaded environment. See Section 7.7 for a discussion of the semantics of SPUR Lisp special variables.

7.2 CLiP

CLiP is a version of Allegro Common Lisp that includes all the multiprocessing features of SPUR Lisp. In addition, *spin locks* and *sleepy locks* are provided. Spin locks are hardware supported busy-waiting locks, while sleepy locks are locks that spin for a fixed number of attempts before blocking. CLiP currently runs on Sequent Symmetry shared-memory multiprocessors. For comparison purposes, a 16MHz 80386 processor of the Sequent Symmetry is rated at 4.3 SPECmark89, while a 20MHz Sun SPARCstation I is rated at 10 SPECmark89.

7.3 Parallel Lisp Features Used

The following SPUR Lisp or CLiP features are used by the toolbox implementation. This list may be used as a starting point to evaluate the feasibility of porting the toolbox to another parallel Lisp system.

Process: A process is a dynamic creation of thread of control in a shared address space.

Sleepy lock: A process acquiring a sleepy lock spins for a given number of times and then blocks until the lock is released.

Mailbox: A mailbox is an unbounded FIFO with implicit synchronization. FIFO elements may be arbitrary Lisp objects. Mailboxes are used for communication and synchronization between threads.

Special variable: CLiP does not implement the full semantics of SPUR Lisp special variables. (See Section 7.7 for details.)

Backoff lock: These are Anderson-style spin locks with exponential backoff, and are currently used by the user-level multi-threaded allocator described in Section 7.6 and the toolbox implementation.

Asynchronous features: The asynchronous features of SPUR Lisp are not used because they are not fully supported by CLiP.

7.4 Changes made to CLiP

The following optimizations have been made to CLiP version 3.1.alpha.16:

No Futures Support for futures have been eliminated because they are not used by the toolbox implementation. This change eliminates the overhead of inline implicit *touch* operations that are scattered throughout compiled code in both the CLiP implementation and application code.

Cons allocation is Multi-threaded CLiP objects are divided into four classes internally for allocation purposes: cons, symbol, root (for generation-scavenging garbage collection) and other (vectors, arrays, structures etc). Concurrent allocation of these objects (by different threads) are synchronized using spin locks, with one lock controlling the allocation of each of these four classes of objects. This simple implementation results in severe performance bottlenecks for any program that allocates any significant number of objects.

The allocation of *newspace* cons cells in CLiP is modified so that cons cells are now allocated from per-processor pools of free cells. In the usual case when a per-processor pool is not empty, concurrent cons allocations on different processors are not synchronized. The effect of this improvement is quantified in Section 7.5.

Spin Lock with Exponential Backoff Spin locks with exponential backoff [49] have been added as an alternative to the spin and sleepy locks supported by CLiP. These locks are used by the user-level object allocation facility (Section 7.6) and the toolbox implementation, but are not currently used by the CLiP runtime system internally.

7.5 CLiP Allocation Bottlenecks

In this section I describe a simple benchmark to illustrate the effects of the allocation bottlenecks of CLiP on performance. The benchmark program consists of n threads (with n Dynix processes) on n processors, each allocating 1,000,000 cons cells, 100,000 simple vectors of size 2 or 100,000 structures of 2 fields. The benchmark code is compiled with the highest optimization setting for speed. The real time taken for all the threads to terminate is measured. The results are shown in Tables 7.1, 7.2 and 7.3 respectively. Note that these numbers are taken from a Sequent Symmetry that is *not* otherwise idle, and the Dynix processes are *not* bound to processors using the tmp_affinity system call. (Complete specifications for the Sequent Symmetry used is given in Section 8.4.)

Table 7.1 compares the time taken for each processor to allocate 1,000,000 cons cells before and after the (internal) change to the allocation of newspace cons cells. The original implementation performs better for one to three processors, but contention for the global free cons pool becomes a serious problem for four or more processors. The multi-threaded cons allocator performs much better for four or more processors.

7.6 User-Level Object Allocation

Because of the complexities involved with multi-threading the allocation of *other* objects (structures and vectors) in CLiP, I decided to use user-level per-processor allocation for vectors and structures (which are heavily used by the applications) for better performance. (User-level object allocation is not required by the toolbox ap-

Processors	Synchronized/s	Multi-threaded/s
1	12	38
2	21	39
3	37	39
4	52	38
5	79	39
6	113	39
7	155	40
8	207	39

Table 7.1: Cons Allocation: *Real* time (in seconds) for the specified number of processors to *each* allocate 1,000,000 cons cells. `Synchronized` refers to the original CLiP implementation, and `Multi-threaded` refers to CLiP with newspace cons allocation multi-threaded (internally).

proach for parallel programming, and is strictly speaking not a part of the toolbox implementation. It is provided to circumvent inherent limitations of the CLiP implementation.) An application program has to be modified to use user-level object allocation.

The types or sizes of the popular *other* objects (structures or vectors) for an application are identified by the programmer, and per-process pools of these objects are pre-allocated by explicit calls to the user-level object allocation facility. The application program is modified to use a different function call to create such an object (e.g. `my-make-array` instead of the Common Lisp `make-array`), and allocations out of a per-process pool are not synchronized in the common case when the pool is not empty. The performance of the new user-level multi-threaded array and structure allocators are shown in Tables 7.2 and 7.3.

Table 7.2 compares the time taken for each processor to allocate 100,000 simple arrays (vectors) of size 2 using the CLiP allocator (by calling `(make-array 2)`) and the multi-threaded user-level array allocator (by calling `(my-make-array 2)`). The original implementation performs better than the user-level allocator for one to six processors, but the absolute time taken by the original implementation rises dramat-

Processors	Synchronized/s	Multi-threaded/s
1	4	15
2	5	17
3	6	17
4	8	17
5	10	17
6	13	18
7	19	19
8	25	18

Table 7.2: Array Allocation: *Real* time (in seconds) for the specified number of processors to *each* allocate 100,000 one-dimensional simple arrays (vectors) of size 2. `Synchronized` refers to the original CLiP implementation, and `Multi-threaded` refers to the user-level multi-threaded array allocator described in Section 7.6.

ically as the number of processors increases. (The eight-processor case of the original synchronized implementation takes more than six times the time taken by the one-processor case.) On the other hand, the user-level multi-threaded array allocator does not show any significant performance degradation as the number of processors increases.

Table 7.3 compares the time taken for each processor to allocate 100,000 structures with two fields using the CLiP allocator and the multi-threaded user-level structure allocator. The CLiP implementation is faster for a small number of processors, but its performance decreases rapidly as the number of processors increases. The multi-threaded allocator scales much better as the number of processors increases.

7.7 Future Improvements to CLiP

In this section I describe some of the changes that are necessary to make CLiP a better parallel Lisp system. In some sense this is a list of challenges facing the implementor of a high-performance parallel Lisp system. Time did not permit me to pursue these performance optimizations. Details about selected parts of the CLiP implementation have been described elsewhere [35, 34].

Processors	Synchronized/s	Multi-threaded/s
1	5	15
2	6	17
3	6	17
4	8	17
5	9	17
6	12	17
7	17	18
8	24	18

Table 7.3: Structure Allocation: *Real* time (in seconds) for the specified number of processors to *each* allocate 100,000 structures with 2 fields. `Synchronized` refers to the original CLiP implementation, and `Multi-threaded` refers to the user-level multi-threaded structure allocator described in Section 7.6.

Multi-threaded Allocation The allocation of all objects (including vectors and structures) should be multi-threaded (in the implementation) so that no synchronization is necessary in the common case. This change is more complicated than the multi-threading of cons allocation described.

Garbage Collection CLiP uses a generation-scavenge garbage collection algorithm [77] that is sequential. When a processor attempts to allocate an object of type X (e.g. cons) and discovers that the global pool for X is empty, it informs other processors that a scavenge is to be performed. All the processors then synchronize (at *safe* points in the code) using a barrier-style synchronization, and then *one* processor performs the entire garbage collection sequentially. (The other processors busy-wait.) After the collection, the processors resume the computations they were performing before the collection.

The performance of this simple implementation may be improved by using a concurrent or multi-threaded garbage collector [16, 65, 66, 33]. For example, a collector thread may run concurrently with the mutator threads (the threads of the application). In addition, the collector may also be multi-threaded, such that different sections of the allocation heap are traversed concurrently to reduce the time required for a collection.

Special Variables Common Lisp provides dynamically-bound variables known as *special variables.* In SPUR Lisp the semantics of special variables is extended so that a child thread C inherits all the dynamically-created bindings of its parent P that are in effect *when* C *is created.* (C may later re-bind and hence override a special variable binding established in P.)

One major consequence of the semantics of SPUR Lisp special variables (which is generally what the programmer wants) is that the *shallow-binding* implementation of special variables commonly found in sequential Common Lisp systems has to be modified, since the binding of a special variable now depends on the thread accessing the variable. A naive implementation of SPUR Lisp special variables requires the use of the *deep-binding* technique, which is relatively inefficient. Using this technique, each special variable access requires a potentially unbounded search of the binding stack.

As a result of the performance problems with the deep binding technique, CLiP does not fully implement the SPUR Lisp semantics for special variables [34]. In particular, the dynamic bindings established by a parent thread are not visible to its children threads. To get around this limitation, dynamic bindings that have been created by a parent P and should be visible to a child C of P have to be passed explicitly from P to C when C is created (as arguments), and rebound in C explicitly. This is very inconvenient and error prone. This limitation of CLiP has a severe impact on the implementations of parallelism abstractions (e.g. `Fixed-Point` or `Dag-Traversal`). If an application program that uses special variables is converted to use a parallelism abstraction, the *implementation* of the parallelism abstraction has to be modified to handle the special variables of the *application* explicitly. (This modification would not be necessary in a parallel Lisp system that supports special variables with SPUR Lisp semantics.)

Various techniques for improving the performance of special variable accesses in parallel Lisp systems (with SPUR Lisp semantics) have been attempted [19, 64]. These techniques improve the access time for repeated accesses to any variable through the use of caching. Unfortunately, details of these attempts have not been published.

Locks CLiP uses numerous spin locks internally in the runtime system to implement short critical sections. These locks tend to create unnecessary bus traffic, leading to memory contention even when no particular memory location is a hot spot [49]. Spin locks with exponential backoff as described by Anderson [4] or software queuing as described by Mellor-Crummey [49] should be used in the CLiP runtime system instead for improved performance.

Hash Tables In the current CLiP implementation accesses to *all* hash tables are synchronized using a *single* global mutex lock (to protect against races related to rehashing). As a result, concurrent operations involving unrelated hash tables may be serialized unnecessarily. A hash table should be provided with its own private mutex lock. (The toolbox provides a per-thread dictionary as a temporary solution to this problem.)

Chapter 8

Performance Tuning

In this chapter I will explain the procedure for tuning the performance of a toolbox-based parallel program. I will then introduce a definition of speedup that is meaningful in the presence of *sequential* garbage collection (in CLiP). Finally, I will give hardware performance numbers for the multiprocessor used in the performance measurements that will be reported in Chapter 9.

8.1 Locating Performance Bottlenecks

In this section I describe the facilities provided by the toolbox for locating performance bottlenecks. The use of built-in performance hooks to improve the parallel performance of a toolbox-based application will be discussed in Section 8.2.

8.1.1 Why is a Parallel Program Slow?

A parallel program may not achieve good parallel performance for the following reasons:

- There is insufficient parallelism in some phase of the application, especially during the initial phase (when work is getting generated) or during the final phase (when work is running out). By Amdahl's Law, the speedup of an application is limited by the relative durations of its sequential phases.

- An application using `Fixed-Point` may be processing work queue elements in an order that is not optimal. (In RC, the fixed point of the digraph G is computed more efficiently by partitioning G into a dag of components, and then computing the fixed points of these components in some topological sort order.)

- The granularity of work for an application using `Fixed-Point` or `Dag-Traversal` is too small compared to the overhead of using the abstraction.

- The implementation of `Fixed-Point` or `Dag-Traversal` is not scalable to a large number of processors. (This is unlikely to be the case for the small number of processors used in CONSAT or RC.)

- The autolock abstraction uses a dictionary internally for maintaining object-lock associations. If the number of subspaces is too small or if the object-lock pairs are not evenly distributed among the subspaces, autolock operations for objects in the more crowded subspaces may take longer to complete. (The object-lock pairs in a subspace are organized as an association list.) In addition, the `Split-fn` for mapping an object to its subspace (i.e. lock) may also be slow.

- Dictionaries used by the application may also be plagued by the above inefficiencies.

- Some objects protected by autolocks may be highly contended.

- There may be excessive contention for backoff locks used in the toolbox implementation.

- The CLiP allocation bottlenecks for *other* objects (Section 7.5) may have an impact on the performance of an application that allocates a large number of structures or vectors.

8.1.2 Locating Performance Bottlenecks

The performance bottlenecks in an application may either be located by observing the state changes of relevant data structures in real time during an execution, or by examining various statistics after an execution.

Statistics After an Execution

- The work queue for `Fixed-Point` may maintain a histogram of the number of activations of each work queue element. This information may be useful in deriving a better scheduling heuristic.

- Spin locks with exponential backoff are used in the implementation of autolocks and other toolbox abstractions. Each backoff lock is instrumented to record the number of attempts to acquire the lock, the number of unsuccessful attempts to acquire the lock (because the lock is held by another thread), the average delay for the exponential backoff, and the maximum delay for the exponential backoff. By examining these statistics, locks that are bottlenecks may be identified easily.

- The number of elements in the subspaces of a dictionary (including the one maintained by autolock) may be too large or unbalanced. The dictionary abstraction provides facilities for examing the distribution of objects in the various subspaces of a dictionary.

- The CLiP runtime system is instrumented so that contention for the allocation of newspace *other* objects (vectors and structures) may be detected easily.

Transient Information The probe process provided by the toolbox may be used to display the states of interesting data structures periodically during the execution of an application. The periodicity may be varied by the programmer.

The information displayed may include the status of internal scheduler queues of CLiP, the work queue of `Fixed-Point` (including the histogram of activations of work queue elements), the status of a lock (locked or unlocked) and its contention statistics, the number of elements in each subspace of a dictionary, and other application-specific data structures. As an example, the probe for CONSAT displays the number of determined subtags for each variable in `Feasible-Set`, which measure the progress of the computation toward the final solution.

The support for performance tuning provided by the toolbox is relatively primitive. A complete programming environment for tuning the performance of toolbox-based applications would include a graphical user interface, and would display the state of each thread in the implementation as well as the status of locks and other relevant data structures. This ideal environment is not provided as it could easily become the focus of another dissertation.

8.2 Removing Performance Bottlenecks

Once the performance bottlenecks in an application have been located, the performance hooks provided by the toolbox abstractions may be used to remove them.

- If the application does not have enough parallelism, the units of work for `Fixed-Point` or `Dag-Traversal` may sometimes be partitioned into a finer granularity. The toolbox does not currently provide any support for automatic partitioning. On the other hand, it is also possible that the application simply does not have sufficient parallelism.

- `Fixed-Point` may be used with an application-specific work queue. The work queue may use runtime information—including the state of global data and the history of the computations—in its scheduling decisions, and allows arbitrary dependencies between elements of the work queue to be enforced.

- `Fixed-Point` may sometimes be used in conjunction with `Dag-Traversal` (cf RC).

- The degree of lock contention for concurrent insert operations on a shared dictionary may be reduced by decreasing the granularity of locking for the dictionary (by increasing the number of subspaces). Contention may also be reduced by using an application-specific `Split-fn` function that distributes keys to subspaces more evenly, such that concurrent dictionary operations are less likely to be synchronized by the same (internal) lock.

- Time consuming mutation operations on an object protected by an autolock may be rewritten as an optimistic read-modify-write operation if possible. This involves the derivation of an application-specific definition of incompatible race condition. A good understanding of the application is required, and substantial changes to the application may be necessary. Alternatively, a composite object may *sometimes* be decomposed into a number of sub-objects, each protected by its own autolock.

- *Performance*-related false sharing may be eliminated using per-thread locations. For example, a global counter used to generate globally unique identifiers (integers) may be replaced by a set of per-thread counters that generate unique identifiers from disjoint subspaces of the space of unique identifiers. (See Section 3.7.4.)

- Exponential backoff parameters for a heavily contented backoff spin lock may be adjusted to reduce the amount of contention and bus traffic. The parameters include the initial and maximum delays for the exponential backoff.

- In the CLiP implementation of the toolbox, explicit, programmer-controlled object pre-allocation is used to get around the CLiP allocation bottlenecks for vectors and structures. Frequently allocated types of vectors and structures may be pre-allocated, and the amount of each type of object pre-allocated may also be varied by the programmer.

- The internal implementation of `Fixed-Point` or `Dag-Traversal` may be distributed so that it is scalable to a large number of processors. (This is the responsibility of the expert parallel programmer who implemented the toolbox.)

8.3 Speedup and Garbage Collection

As explained in Section 7.7, all the mutator threads of an application under CLiP stops and synchronizes before a single collector thread performs a garbage collection. To estimate the potential performance of a parallel application in the absence of garbage collection, I measure the real time spent in an application, *excluding* time spent performing garbage collection. (The time spent in the barrier synchronization before each collection is also excluded.) This measurement gives an upper bound on the performance of a parallel application in a parallel Lisp system with a more sophisticated garbage collector.

Definition of Speedup I define the speedup of an application on n processors as the ratio between the time taken by a sequential implementation (*not* a one-processor parallel implementation) to the time taken by an implementation on n processors. Because of the limitations of the CLiP garbage collector, I will use real time *excluding* garbage collection for both quantities in speedup computations.

8.4 Multiprocessor Used

The Sequent Symmetry model S81 used in the measurements has 20 16MHz 80386 processors and 64MB of shared memory. It is a bus-based multiprocessor with hardware support for cache coherency. The operating system is Dynix v3.2.0 NFS. For comparison purposes, a 16MHz 80386 processor is rated at 4.3 SPECmark89, while a 20MHz Sun SPARCstation I is rated at 10 SPECmark89.

8.4.1 Measurement Conditions

The Sequent Symmetry was *not* run in single-user mode when the performance numbers were collected. On the other hand, very few CPU-bound processes belonging to other users were executed during the measurements. The Dynix processes used by CLiP were bound to specific processors using the Dynix tmp_affinity system call, and the setpriority system call was used to ensure that the scheduling priority of long running Dynix processes were not reduced (i.e., increased numerically) by the Dynix process scheduler.

Chapter 9

Evaluation

9.1 Introduction

In Chapter 1 I introduced the toolbox approach for symbolic parallel programming. I identified a number of parallelism and data-sharing abstractions, and implemented them as part of a prototype toolbox described in Chapters 2 and 3. I then converted two significant symbolic applications to use the toolbox. These applications include CONSAT, a constraint satisfaction system described in Chapter 4, and RC, a type analysis system described in Chapter 6. In this chapter I shall reflect on my experience of converting CONSAT and RC to use the toolbox, and extrapolate this experience to the feasibility of the toolbox approach for symbolic parallel programming by novice parallel programmers in general. Performance numbers for the sequential and parallel toolbox versions of the two applications will also be presented.

The following four criteria will be used to evaluate the toolbox approach for symbolic parallel programming:

- Can a small toolbox be used for a large class of applications?
- Is the source code for the toolbox version of an application truly portable between uniprocessors and multiprocessors?
- Can the toolbox abstractions be used by novice parallel programmers easily?

- Do toolbox versions of the applications perform reasonably efficiently, on uniprocessors and multiprocessors?

Small toolbox for a large class of applications The toolbox approach is not universal, i.e. a toolbox may be used for an application only if it contains all the necessary abstractions. This approach is only cost-effective if the effort required to developed an abstraction is amortized over the large number of applications that use the abstraction. In the worst case the approach reduces to the hiring of a parallel programming expert for each application parallelized.

As I have only parallelized two applications with the toolbox in this research, a definitive answer about the cost-effectiveness of the approach cannot be given. However, the results from CONSAT and RC are encouraging. CONSAT and RC share the following abstractions: `Fixed-Point`, counter, dictionary, autolock, `ORMW` and per-thread location. `Dag-Traversal` is only used by RC, but I expect it to be useful in many graph algorithms.

Portability of Source Code In general, the toolbox version of CONSAT (or RC) may be executed on uniprocessors and multiprocessors without modifications. Only the implementation of the toolbox has to be changed. The major exception to the portability of source code lies in the application-specific schedulers used with `Fixed-Point`. Based on my limited experience with CONSAT and RC, it appears that the scheduling algorithm has a more pronounced effect on the performance of the parallel implementation than the sequential implementation, and consequently the parallel implementation will probably use a more sophisticated scheduler than the sequential implementation. Because of the clean interface between `Fixed-Point` and the scheduler, all the potentially non-portable scheduler code will be cleanly isolated from the rest of the application code.

Ease of Use by Novice Parallel Programmers A programmer using the toolbox is expected to *identify* all sources of parallelism and instances of data sharing in an application, and select the appropriate parallelism abstractions and data-sharing abstractions respectively. As the toolbox abstractions have sequential interfaces, I argue that novice parallel programmers may be expected to perform this task with relative ease.

Reasonable Performance The toolbox provides some primitive support for locating the performance bottlenecks in an application. In addition, performance hooks are built into many of the toolbox abstractions. These hooks enable the programmer to tune the performance of an application using application-specific information without knowing the low-level parallel programming details in the toolbox implementation.

It is not clear if this primitive support for performance tuning is sufficient, or if performance tuning may be accomplished by inexperienced parallel programmers easily. As an example, it is unlikely that an application-specific scheduler for `Fixed-Point` that gives good parallel performance can be developed without a parallel model of the computation. More experience with the toolbox is necessary to determine if the performance of the toolbox version of a typical application is comparable to the version developed by a parallel programming expert without using the toolbox.

9.2 The Toolbox Experience

In this section I describe my subjective experience with re-implementing CONSAT and RC using the toolbox. I will indicate the relative ease with which individual abstractions are used. Details about CONSAT and RC may be found in Chapters 4 and 6 respectively.

9.2.1 CONSAT

The central loop of sequential CONSAT is converted into a work queue-style computation that uses `Fixed-Point`. An application-specific scheduler is used to ensure that multiple instances of any given constraint are not activated concurrently (for correctness). A moderate amount of code has to be rewritten.

Concurrent accesses to `Feasible-Set` are initially synchronized using an autolock.

This modification is trivial, but unfortunately results in a severe performance bottleneck. There is practically no concurrency in such a parallel implementation. The access and update operations on `Feasible-Set` are then rewritten into a read-modify-write model in which only the write phase is locked (using autolock). A substantial amount of code has to be rewritten. This modification greatly alleviates the contention for `Feasible-Set`, but unfortunately introduces a very subtle race condition that took a long time to detect and understand. The race condition results from a class of state transitions (for successive states of `Feasible-Set`) that does not occur in a sequential implementation. The elimination of this race condition results in the theoretical work on optimistic parallel discrete relaxation in Chapter 5. The key to eliminating the race condition without reducing concurrency excessively is the observation that not all race conditions are incompatible. Consequently, only computations that lead to incompatible race conditions are aborted and re-executed. This correct and efficient optimistic scheme is expressed using the `ORMW` data-sharing abstraction. While a substantial amount of code has to be reorganized to use `ORMW`, the effort required is trivial compared to the high-level understanding of CONSAT behind the use of `ORMW` for updating `Feasible-Set`.

The implementation of the unique-id generator (Section 3.7.4) for unique subtags is changed from an atomic counter to a `pdefstruct` with per-thread locations so that concurrent constraint activations are not serialized unnecessarily. No explicitly parallel code is used, and a small amount of code is modified.

Some (temporary) hash tables used in CONSAT are replaced by per-thread dictionaries from the toolbox. Each per-thread dictionary is designed to be accessed by a single thread, and operations on the dictionary are not synchronized. These dictionaries are used because *all* CLiP hash table operations on *all* hash tables are synchronized using a *single* global mutex lock.

The application-specific scheduler used with `Fixed-Point` is modified to collect statistics about the number of times a constraint is activated, and the number of constraint activations that are aborted because of incompatible races by concurrent `ORMW` operations. These statistics are used to help the programmer derive a good scheduling strategy that will reduce the number of constraint activations required to compute the solution of a constraint network. As there is usually no theoretically optimal scheduling strategy for the activation of a generalized constraint network, a scheduling heuristic that performs reasonably well and does not take a lot of time to compute is chosen.

For each variable V in the constraint network, the number of determined subtags in the feasible set of V increases monotonically from zero to the total number of constraints in the network. (The solution is found when the subtags for all the variables of a constraint network are determined.) These numbers are used as progress metrics for the computation, and are displayed periodically by the probe provided by the toolbox.

To summarize, the most difficult challenge in deriving a parallel toolbox version of CONSAT lies in the high-level understanding about incompatible race conditions for updating `Feasible-Set`. A non-trivial amount of code has to be modified to use the toolbox abstractions, but these modifications are relatively straightforward.

9.2.2 RC

The central loop of sequential RC is converted to use `Fixed-Point`. As in the case for CONSAT, there is no obvious way to derive an optimal scheduling strategy for use with `Fixed-Point`, which is probably input-dependent. A simple application-specific scheduler that prevents multiple instances of any given node from being visited concurrently is used.

Instead of computing the fixed point for all the nodes of the input program graph G directly, RC is modified to decompose G into strongly-connected components (using a library provided by the toolbox). The fixed-point computation on G is then replaced by fixed-point computations on the components of G. These fixed-point computations are coordinated using `Dag-Traversal`. A moderate amount of code is rewritten. This transformation improves the performance of the sequential and parallel versions of RC, and a sequential programmer can be expected to perform this transformation easily.

Nonterminals are interned using a global hash table in the original version of RC.

The toolbox version replaces the hash table with a dictionary. This change is trivial to implement.

Productions are interned using a trie that is stored in the relevant nonterminal in the original version of RC. The toolbox version initially uses an autolock to synchronize concurrent update operations on the trie. Eventually each trie is replaced by a dictionary, which provides better parallel performance by allowing multiple concurrent updates to a dictionary. The change is straightforward.

A unique-id field is added to each nonterminal and production so that a fast `Split-fn` that uses the unique-id field may be used with the dictionaries. The number of subspaces for the dictionaries are chosen so that the number of entries per subspace is not excessive. The changes are straightforward.

Certain temporary fields in nonterminals and productions are used in ways that will result in false sharing in a parallel implementation (Section 6.4.3). I attempted to rewrite the algorithms involved so that these temporary fields are unnecessary, but abandoned the effort because the changes would require a very good understanding of algorithms that use the fields, and would involve major changes to scattered sections of the code. Instead, nonterminals and productions are defined using `pdefstruct` with per-thread locations so that the existing algorithms that use the temporary fields may be used in a parallel implementation without modification. The changes required are trivial—only the definition of the objects have to be changed.

`Conditional-Insert-Dict` operations on dictionaries for interning nonterminals and productions are converted to `Conditional-Insert-Funcall-Dict` to reduce the amount of objects created unnecessarily. This change is trivial to implement, and is perfectly understandable for a sequential programmer.

`ORMW` is used to update p-sets to ensure that concurrent updates by multiple predecessor nodes of a given node N do not lead to excessive contention. If the concurrent updates are incompatible, they are re-executed by the `ORMW` implementation (transparently). The programmer simply specifies the condition for incompatible updates without getting involved with the low-level parallel programming details.

To summarize, most of the changes made to derive the toolbox version of RC are relatively straightforward, and inexperienced parallel programmers can be expected to convert the original sequential implementation of RC to use the toolbox.

9.3 CONSAT Performance

9.3.1 Sample Constraint Network

I measured the performance of CONSAT for a problem in machine vision that assigns three-dimensional edge labelings (convex, concave, or occluding) to line drawings in a polyhedral world of trihedral vertices [32]. The constraints restrict the labeling of edges meeting at a vertex to be the few combinations physically possible. (These constraints correspond to L, fork and arrow junctions.) In addition, each edge is constrained to have the same label at both ends (where it meets other edges).

For $n \geq 2$, Stair(n) is a constraint network that represents a synthetically generated three-dimensional scene of a simple staircase with n raised steps. There are $6n+3$ variables (edge labels) and $10n+6$ constraints ($6n+6$ 2-variable constraints and $4n$ 3-variable constraints). In the following I give performance figures for Stair(5), which has 33 variables and 56 constraints (36 2-variable and 20 3-variable constraints).

9.3.2 Uniprocessor Performance

I measured the performance of the sequential (*not* one-processor) toolbox version of CONSAT for the constraint network Stair(5) in Allegro CLiP on the Sequent Symmetry (Section 8.4). The sequential toolbox implementation only uses the features of Common Lisp. No multiprocessing features of CLiP are used. CONSAT was compiled with the highest optimization setting for speed (`(speed 3) (safety 1)`). The execution of Stair(5) took 78 seconds of *real* time (75s non-gc + 3s gc).

9.3.3 Multiprocessor Performance

I measured the performance of the parallel toolbox version of CONSAT for Stair(5) in Allegro CLiP on the Sequent Symmetry. As in the uniprocessor case, CONSAT was compiled with the highest optimization setting for speed (`(speed 3) (safety 1)`).

Processors	no gc	gc	Total Time	Speedup	Steps
Sequential	75	3	78	1.0	447
1	78	2	80	0.96	447
2	76	1	77	0.99	449
3	83	1	84	0.90	448
4	81	1	82	0.93	456
5	82	1	83	0.91	457
6	81	1	82	0.93	445
7	83	1	84	0.90	459
8	80	1	81	0.94	445

Table 9.1: Performance of CONSAT for Stair(5) using Autolocks on the Sequent Symmetry: All times are real time in seconds. `non-gc` is time without garbage collection, and `gc` is time for garbage collection. `Speedup` is based on times *excluding* garbage collection (Section 8.3). `Steps` is the number of constraint activation steps.

Processors	no gc	gc	Total Time	Speedup	Steps	Aborted
Sequential	75	3	78	1.0	447	0
1	78	2	80	0.96	447	0
2	43	1	44	1.7	454	9
3	29	1	30	2.6	455	11
4	23	1	24	3.3	468	22
5	19	1	20	3.9	460	15
6	16	1	17	4.7	465	17
7	14	1	15	5.4	464	16
8	14	1	15	5.4	468	17

Table 9.2: Performance of CONSAT for Stair(5) using `ORMW` if Incompat-p returns `t` whenever `Feasible-Set` is changed, i.e. the RMW is not optimistic. `Steps` is the number of constraint activation steps, *including* non-monotonic ones. `Aborted` is the number of non-monotonic steps (re-executed).

Processors	non-gc	gc	Total Time	Speedup	Steps	Aborted
Sequential	75	3	78	1.0	447	0
1	75	3	78	1.0	447	0
2	42	3	45	1.8	453	5
3	28	4	32	2.7	456	10
4	23	4	27	3.3	470	19
5	18	5	23	4.2	450	5
6	15	5	20	5.0	451	3
7	13	6	19	5.8	455	5
8	14	6	20	5.4	467	16

Table 9.3: Performance of CONSAT for Stair(5) using `ORMW`. Incompat-p is defined using the monotonicity condition defined in Chapter 4.

The results for Stair(5) are summarized in Tables 9.1, 9.2 and 9.3. In Table 9.1 updates to `Feasible-Set` are synchronized using an autolock. There is virtually no speedup because of contention for `Feasible-Set` by the concurrently activated constraints. In Table 9.2 updates to `Feasible-Set` are synchronized using `ORMW`. However, `Incompatible-p-fn` is defined so that *all* concurrent accesses to `Feasible-Set` are defined to be incompatible, i.e., the RMW is not optimistic. The speedup varies from 1.0 on one processor to 5.4 on eight processors. Table 9.3 updates to `Feasible-Set` are synchronized using `ORMW`. `Incompatible-p-fn` is defined so that only concurrent accesses to `Feasible-Set` that violate the monotonicity condition defined in Chapter 4 are defined to be incompatible. The speedup varies from 1.0 on one processor to 5.8 on seven processors.

In all cases the one processor time is within 5% of the sequential time, showing that the parallel implementation is reasonably efficient. The speedup is less than linear because a small number of constraint activations have to be re-executed because incompatible races were detected, i.e. the (aborted) changes to `Feasible-Set` violate monotonicity. The speedup for the two versions using `ORMW` (Tables 9.2 and 9.3) are quite similar. I speculate that the performance gained by the use of the optimistic version (Table 9.3) is somewhat offset by the longer execution time of the `Incompatible-p-fn` test. (`Incompatible-p-fn` is executed inside a critical section in the `ORMW` implementation.)

9.4 RC Performance

9.4.1 Sample Input Program

I have measured the performance of RC using a synthetic input `(Chainx 20 10)`. For integers A and B, `(Chainx` A B`)` generates a Lisp function with a number of strongly-connected components roughly proportional to A and with the size of each component roughly proportional to B.

The source code for RC comes with a preprocessing phase and an extra checking phase that are not included in the timings. The preprocessing phase builds the flow graph from the input Lisp program and prepares it for type analysis. The checking phase is a separate, sequential implementation of the RC algorithm, and is used to verify the correctness of the result of parallel RC.

9.4.2 Uniprocessor Performance

I measured the performance of the sequential (*not* one-processor) toolbox version of RC for `(Chainx 20 10)` in Allegro CLiP on the Sequent Symmetry (Section 8.4). The sequential toolbox implementation only uses the features of Common Lisp. No multiprocessing features of CLiP are used. RC was compiled with the highest optimization setting for speed (`(speed 3) (safety 1)`). The execution of `(ChainX 20 10)` took 16 seconds of *real* time. (There was negligible time spent in garbage collection.)

9.4.3 Multiprocessor Performance

I measured the performance of the parallel toolbox version of RC for `(Chainx 20 10)` in Allegro CLiP on the Sequent Symmetry. As in the uniprocessor case, RC was compiled with the highest optimization setting for speed. Both parallel and sequential RC ran on the same machine. (The sequential version is simply RC with the sequential toolbox.)

Processors	Time	Speedup
sequential	16	1.0
1	68	0.24
2	50	0.32
3	43	0.37
4	44	0.36
5	42	0.38
6	49	0.33
7	64	0.25
8	99	0.16

Table 9.4: Performance of RC for `(Chainx 20 10)` on the Sequent Symmetry: All times are real time in seconds. Negligible time is spent in garbage collection. `Speedup` is based on times *excluding* garbage collection (Section 8.3).

The results are summarized in Table 9.4. All times are measured in seconds of real time, and none of the runs requires garbage collection. The speedup is rather disappointing because of the high overhead of using the parallel features of CLiP (by the toolbox implementation) compared to the grainsize of work in RC. The performance of parallel RC does improve slightly as the number of processors increase, but the improvement is limited by the lack of parallelism in the structure of program flow graphs. In addition, contention for the allocation of newspace *other* objects in the CLiP implementation (Section 7.5) becomes significant as the number of processors increase.

Chapter 10

Related Work

This chapter covers related work in the following areas: parallel programming systems (with shared-address space models) suitable for symbolic applications, asynchronous iterations, and parallelized applications that are similar to CONSAT or RC.

Section 10.1 compares the toolbox with other symbolic parallel programming systems that support a shared-address space model. Most of the systems described support an explicitly parallel programming model, as techniques for the automatic detection of parallelism are not yet practical for symbolic applications, especially those written in imperative programming languages. Section 10.2 compares the parallelism abstractions of the toolbox with other approaches for introducing parallelism into an application, and Section 10.3 examines how side effects on shared data structures are handled by other parallel programming systems, as well as interesting architectural ideas for supporting the implementation of data-sharing abstractions. Section 10.4 compares asynchronous iterations with other parallel iteration schemes, Section 10.5 describes related work in the parallel constraint satisfaction area, and Section 10.6 describes other compiler optimization applications that perform graph traversals similar to `Dag-Traversal`.

10.1 Explicitly Parallel Programming Systems

Algorithmic Skeletons At an abstract level the parallelism abstractions of the parallel programming toolbox are very similar to the *algorithmic skeletons* proposed by Cole to simplify the programming of parallel machines [14]. Algorithmic skeletons are templates or high-order functions that can be specialized by problem-specific procedures, and represent common algorithmic techniques with efficient parallel implementations. Most of the details of parallel programming are hidden by the implementation of the skeleton. As the skeletons are completely independent of one another, new skeletons may be introduced without interference from existing skeletons. The skeletons proposed include *fixed degree divide and conquer*, *iterative combination*, *clustering* and *task queues*.

Algorithmic skeletons are similar to parallelism abstractions in that the programming model provided is not *universal*, because there are parallel programs that cannot be expressed by a given set of algorithmic skeletons or parallelism abstractions. On the other hand, all parallelism abstractions have sequential interfaces while some skeletons have sequential interfaces and others have parallel interfaces. The total independence between skeletons requires concurrent accesses to shared data structures to be handled on a per-skeleton basis. For example, the task queue skeleton has an explicitly parallel interface, and requires the programmer to handle concurrent data accesses in an ad hoc manner.

CHARM Charm is a parallel programming system for MIMD multiprocessors with or without shared memory [40, 39]. It provides an explicitly parallel programming model based on message passing. A *chare* (instance) is somewhat similar to a process (or *Qlambda* in Qlisp [18]), and contains *entry-points* that are associated with non-blocking sections of C code. Chares communicate by sending messages to the entry points of each other. A chare may only handle one message at a time, and the system supports a set of predefined queuing strategies for messages. There is no provision for application-specific queuing strategies for messages, and messages for all the entry points (of different chares) all share the same strategy. A chare has access to its local data, and multiple chares may only access explicitly declared shared data structures called *specifically shared variables*. The specifically shared variables of Charm will be discussed in more detail in Section 10.3.2.

The specifically shared variables of Charm are similar to the data-sharing abstractions of the toolbox. They differ in that the specifically shared variables of Charm have interfaces based on message passing, and do not have the performance hooks provided by the data-sharing abstractions. A detailed comparison between individual data-sharing abstractions and specifically shared variables is given in Section 10.3.2.

Chameleon Chameleon provides a set of abstractions for writing *effectively portable* parallel programs on shared-memory multiprocessors [2, 3]. An effectively portable parallel program has low software development and porting costs, and runs with good performance on different multiprocessors. Chameleon achieves this goal by selecting abstractions for activities that are crucial for good performance, and by providing optimized implementations of these abstractions on different multiprocessors. For example, an *interval* computation applies a function to disjoint intervals of a fixed continuous domain in parallel, while a *tree* computation is characterized by subcomputations that are solved recursively (e.g. branch and bound). Abstractions including data representation (distribution and layout) and partitioning-scheduling have been proposed for these computations. At a high level the Chameleon approach is very similar to that of the parallel programming toolbox. However, as Chameleon is targeted for experienced parallel programmers who wish to achieve (very) high performance, the abstractions have explicitly parallel interfaces with elaborate performance hooks, and Chameleon programs are very different from sequential programs. Finally, the data representation abstractions of Chameleon are provided for high performance by improving the locality of reference and reducing memory contention (especially on NUMA multiprocessors), while the data-sharing abstractions of the toolbox are designed to provide the programmer with a simple programming model.

p4 p4 is a library for writing portable and efficient parallel programs for shared-memory and message-passing multiprocessors [9], and is a successor of the m4-based

Argonne macros [48]. p4 supports the *cluster* model in which processes within the same cluster may use shared memory for communication, while message passing is used for communication between processes of different clusters. p4 supports primitives for process creation, message passing, broadcast and global reduction operations (for numerical datatypes). It also provides monitors, locks, barriers, shared counters, and *askfor* monitors for processes that cooperate using the shared-memory model. With the exception of askfor, which is described in more detail in Section 10.2.1, the shared-memory programming primitives provided are relatively low level.

Formal Derivation Techniques Yelick describes a *transition-based* approach for synthesizing fast and correct parallel symbolic applications [74]. In this approach the problem is specified as a set of legal state transitions, which are refined into an equivalent but more efficient set. Each state transition is then coded as a procedure in some programming language, and necessary synchronization is added. Finally performance is optimized by the use of an application-specific scheduler. A set of simple concurrent datatypes is an integral part of this approach. These datatypes are described in more detail in Section 10.3.2. Formal refinement approaches for developing parallel programs have also been proposed for UNITY [11], and by Back [5].

The transition-based approach for program development is substantially different from the usual methodology for developing sequential programs, and the resulting parallel program will probably be entirely different from the initial sequential version. On the other hand, the abstraction of scheduling policies from the main application, and the use of an application-specific scheduler for performance tuning is similar to that of the `Fixed-Point` parallelism abstraction of the toolbox.

Parallel Lisp Dialects Explicitly parallel Lisp dialects such as Multilisp [23], Qlisp [20], SPUR Lisp [76] and STING [37, 36] have been proposed for use in parallel symbolic computations.

Multilisp and Qlisp are examples of the dialects that provide high-level abstractions for the programmer. These high-level abstractions suffer from the all-or-nothing problem: if the parallel computation to be performed fits some abstraction provided by the language, the computation may be parallelized relatively easily. On the other hand, the programmer is on his own if the computation does not fit any of the abstractions provided. As an example, Multilisp supports *futures*, which are placeholders for values that have not been computed. Futures may be used to parallelize sections of side-effect-free code, but does not simplify the job of the parallel programmer for code with side effects.

SPUR Lisp and STING are examples of dialects that provide flexible but low-level primitives. These primitives are intended for use in implementing high-level abstractions (or languages), and are not designed for use by application programmers directly. The programmer has to manage low-level details such as thread creation, scheduling, termination detection or the effect of mutation operations on shared data structures.

10.2 Parallelism Abstractions

10.2.1 Fixed-Point

In this section I survey other work queue-style abstractions for parallel programs, and the interface design of these abstractions regarding scheduling and partitioning decisions.

Work Queue Abstractions `Fixed-Point` has an interface that is very similar to that of the *task queue* algorithmic skeleton proposed by Cole [14], which takes a parameterized work procedure (cf. `Op`), a queuing strategy selected from a given set, and a list of initial elements of the task queue. Unlike `Fixed-Point`, there is no provision for using application-specific schedulers with a task queue, and a task may only be executed after its parent has completed. Furthermore, Cole makes no attempt to hide the parallel nature of shared data accesses, and considers the task queue an explicitly parallel abstraction. The programmer is solely responsible for the correctness of concurrent accesses to shared data. `Fixed-Point` is also similar to the *askfor* monitor of p4, which is a generalized work queue interface that supports application-specific work queues provided by the programmer. The interface is fairly clean and high level, and does not require the programmer to perform any synchronization for operations on the work queue. One limitation of askfor is that the work queue interface does not support an operation analogous to `After-op` of `Fixed-Point`, and as a result some scheduling strategies cannot be implemented.

The *tree task* abstraction of Chameleon is an example of the work queue model specialized for tree-style computations (e.g. tree searches). The abstraction takes an initial tree node, a work procedure (cf. `Op`), a cutoff depth and a partitioning-scheduling strategy, and expands a subtree of computations starting at the given node up to the cutoff depth. Work queue abstractions for parallel programming are also provided by the *Uniform System* [67] and *Qlisp* [20], among others.

One major drawback of explicitly parallel programs written using these work queue models is that they do not provide a sequential interface, and consequently the resulting parallel program is significantly different from the original sequential version. `Fixed-Point` is unusual in that it provides a simple, *sequential* interface that may be used by novice parallel programmers easily, while allowing the expert parallel programmer complete freedom in implementing application-specific schedulers.

Scheduling and Partitioning Decisions `Fixed-Point` gives the programmer full control over the scheduling decisions of the work queue, and encourages the programmer to separate the basic algorithm from the performance-related scheduling decisions. As a result, different scheduling strategies may be experimented with by modifying a very small and restricted portion of the code. The application-specific scheduler may use runtime information (including the state of global data) to improve performance. In addition, the interface between `Fixed-Point` and the work queue allows the programmer to enforce arbitrary dependencies between elements of the work queue for correctness or performance. Parallel programs for the Chameleon system also separate the basic algorithm from the partitioning and scheduling decisions [2]. However, the Chameleon interface is significantly more complicated than `Fixed-Point`, and there is no provision for the partitioning-scheduler to enforce arbitrary dependencies between outstanding units of computations.

A partitioning strategy is frequently used with a work queue-style parallel program to control the grainsize of computations performed by the individual threads of control. Examples include *WorkCrews* [56], *dynamic partitioning* [71], *lazy task creation* [52] and *Chameleon*. `Fixed-Point` does not allow the programmer to supply a partitioning strategy, and all partitioning must be done by the programmer explicitly before the computations are added to the work queue maintained by `Fixed-Point`. It remains to be seen whether this simple interface is sufficient for providing good parallel performance for a wide range of applications.

10.2.2 Dag-Traversal

Sarkar experimented with the use of *macro dataflow* (compile-time partitioning and runtime scheduling) for the single assignment language SISAL on shared-memory multiprocessors [59]. The partitioning algorithm only generates tasks with acyclic intertask dependencies, and a task is scheduled for execution if all its predecessors have finished execution.

10.3 Data-Sharing Abstractions

Most parallel programming systems provide very limited support for handling concurrent accesses to shared data. The primitives provided usually consists of locks, semaphores or monitors, which are low level and hard to use. In the following I examine how other explicitly parallel languages or systems simplify concurrent accesses to shared data, and survey related work for each data-sharing abstraction of the parallel programming toolbox.

10.3.1 Other Approaches to Concurrent Accesses

Jade Jade is a language designed for coarse-grained parallelism on shared-memory and distributed-memory multiprocessors as well as heterogeneous systems [55]. A Jade program consists of a sequential imperative program (in C) annotated with *task decompositions* and *access specifications*. Task decompositions specify the sources of parallelism in a program, while access specifications identify the shared objects accessed by the tasks, as well as the modes of accesses (e.g., read-only, write-only, read-write). The access specifications are dynamically interpreted by the implementation at runtime, and may contain arbitrary code segments containing access declaration statements that make use of runtime information. For example, runtime information may be required to determine if two variables are aliased to the same object. If a *conflict* (e.g., two concurrent writes to the same object from different tasks) is detected, the execution order specified in the sequential version of the program is preserved. Access specifications may also be modified dynamically at runtime so that the granularity of synchronization is not limited to task boundaries.

The Jade approach is similar to the toolbox approach in that the programmer (and not some automatic analyzer) identifies the shared accesses in a program. The main distinction between these two approaches is that the Jade implementation handles these shared accesses automatically on the basis of access declarations, while the toolbox requires the programmer to select appropriate data-sharing abstractions to ensure that the accesses are correct. As the Jade implementation does not understand the application-specific correctness conditions of the program, it follows the sequential execution order whenever a conflict is detected. This execution order is sufficient but (in general) not necessary for the correctness of the program, and may lead to suboptimal parallel performance. To enhance the performance of a Jade program, the programmer may decrease the granularity of shared objects, or supply more accurate access specifications. In contrast, a programmer using the toolbox may instead use his high-level understanding of the application to select a different (and perhaps more complicated) data-sharing abstraction for the program for higher performance.

Concurrent Aggregates Concurrent Aggregates [13] is an object-oriented language for fine-grained message-passing machines, and is loosely based on the Actor model [1]. An *aggregate* is a multi-access data abstraction that is not serializing. It has a message interface, and multiple identical representative objects cooperate to handle multiple messages concurrently. A message sent to an aggregate is delivered to an arbitrary representative of the aggregate. *Aggregates* may be used to construct multi-level abstractions without limiting concurrency. The programmer has explicit control over data repetition, partitioning and consistency decisions among the representatives of an aggregate. The aggregate model is different from coherent shared memory in that only the level of consistency required for an application is provided, and the programmer is given more flexibility in the implementation. Aggregates seem more appropriate as an implementation platform for parallel programming experts to construct high performance abstractions than a user-friendly language for programmers not experienced in parallel programming.

Multilisp Multilisp *futures* are placeholders for yet uncomputed values [23]. They may be used for the concurrent computation of independent expressions, or to introduce parallelism between the production and use of certain values. They may be inserted into sequential programs with no side effects to introduce concurrency relatively easily, but do not simplify the construction of parallel programs with side effects.

M-structures Barth recently proposed the introduction of M-structures (mutable structures) to the non-strict and implicitly parallel functional programming language Id [6]. M-structures are data structures with atomic read-modify-write protocols, and support synchronization styles that are more general and flexible than regular critical sections. (See the description of autolocks in Section 10.3.2 for an example.) They are added to Id to reduce the amount of data copying and *threading*—the explicit passing of state up and down the call tree—commonly found in functional programs. An M-structure supports two implicitly synchronized operations, *take* and *put*. Take locks an M-structure and reads its value, while put updates an M-structure and unlocks it. Programs written using M-structures are more declarative (simpler) than their functional counterparts, and are more efficient because the need to copy data is reduced. On the other hand, the introduction of this imperative datatype destroys the referential transparency of functional Id, and consequently M-structure programs are indeterminate.

M-structures are designed for use in computations with *allowable indeterminacy.*

A computation with allowable indeterminacy either computes one of many legal values, or has more than one legal way to compute the same value. Because of this indeterminacy, M-structure programs are more difficult to reason about than functional Id programs. Parallel programming issues such as scheduling and deadlock have to be taken into consideration by the programmer. In addition, debugging is also more difficult because certain race bugs may not be easily reproducible. At the implementation level M-structures are a language-level feature that requires extensive compiler support to implement efficiently, while the data-sharing abstractions of the toolbox may be added to any language by the inclusion of the appropriate runtime library.

Two abstractions from the toolbox are similar to the work on M-structures at a high level, though this similarity was not discovered until the abstractions have been developed. Barth's notion of allowable indeterminacy in an algorithm captures the sources of parallelism identified by a parallelism abstraction. In addition, the take and put operations on M-structures bear some resemblance to the `ORMW` data-sharing abstraction.

Chameleon Chameleon supports abstractions for shared data that may be bound to different (optimized) representations on different NUMA multiprocessors. The distribution of shared data structures among memory modules (on a NUMA or a distributed-memory multiprocessor) is determined by the Chameleon system. A shared object may be stored at only one memory module, be spread across several modules, or be fully replicated in each module. The layout of shared data structures is determined by the programmer. The optimal layout is usually a function of the access pattern of the computation performed, but independent of the machine architecture.

Steele Steele proposed a programming model that combines the flexibility of MIMD programming with the simplicity of SIMD programming [63]. This is a shared-memory MIMD model with severe restrictions on side effects. Formally, two operations on shared memory that are not *casually related* must commute (for all possible memory states). Consequently, threads in such a system may not use shared memory for communication or synchronization. They may only synchronize using thread termination. The restriction ensures that the unpredictability of program execution order among threads is not externally observable, so that a given input will always produce the same output. A dynamic scheme to ensure that forbidden operations with side effects on shared memory do not occur is also sketched. This programming model requires extensive changes to existing sequential or parallel (shared-memory MIMD) programs. In addition, the dynamic scheme requires extensive runtime support, and will probably require hardware support to implement efficiently. On the other hand, debugging is very simple in this model because *one* correct execution of a program with a given input will ensure that *all* executions of the program with the same input are correct.

Multipol Multipol is a library of data structures for irregular computations on distributed-memory multiprocessors [10]. These data structures have clean interfaces, and different implementations of the library may be tuned to specific architectures to provide good portable performance. This goal is similar to Chameleon's notion of effectively portable programs. Multipol data structures, called *relaxed objects*, are shared mutable objects with relaxed, user-defined notions of consistency based on the semantics of the operations on the objects. By taking advantage of the relaxed consistency requirements, more efficient implementations of the objects may be created. As an example, a Multipol (distributed) task queue may treat task priorities as hints, such that task priorities are only observed within a per-processor local queue, and not across all the per-processor queues. As a result of the relaxed consistency requirements, such a task queue may be implemented much more efficiently than one in which task priorities have to be observed across all the per-processor queues.

10.3.2 Per-Component Comparisons

Simple Concurrent Datatype Gong and Wing describe a library of *concurrent objects* (concurrent datatypes) with provably linearizable implementations [73]. Their library is designed for use by application programmers, and includes datatypes such as FIFO, priority queue, semiqueue, stuttering queue, set, multiset and register. Yelick describes an alternative set of concurrent datatypes that includes lock, counter, queue, dynamic array, and mapping [74]. Charm provides a set of concurrent datatypes for sharing called *specifically shared variables*. These variables (datatypes) include *read-only* variables, *write-once* variables, *accumulators*, *monotonic* variables and *dynamic tables*. The parallel programming toolbox supports the more common simple concurrent datatypes. Other concurrent datatypes will be added to the toolbox as the need arises.

Conditional Update for Concurrent Datatypes Multilisp provides a family of low-level atomic primitives called *replace-if-eq* for the construction of higher-level synchronization operations [24]. For example, the atomic `(replace-car-eq C V V')` operation tests if `(car C)` is `eq` to `V'`, and if so changes `(car C)` to contain `V`. A boolean is returned from this operation to indicate if the replacement has occurred. Currently this family of operations is only supported for the `car` and `cdr` fields of cons cells, and is not supported for arbitrary datatypes.

Dictionary Dictionaries are frequently implemented as hash tables in a sequential implementation. If a single mutex lock is used to synchronize concurrent operations on a shared dictionary in a parallel implementation, the lock may become a bottleneck, even for accesses involving different keys.

Yelick describes an implementation of the *mapping* (dictionary) datatype based on *multiported objects* [74]. A multiported object is a shared object with per-thread data for each thread that accesses the object, and each thread's version of the state of the object (shared and per-thread data) is called a *port*. The multiported dictionary implementation does not use locks, and uses the full power of sequentially consistent shared memory. It provides much higher throughput for operations on a mapping object than an implementation based on a single mutex lock, at the cost of significant implementation complexity. It would be interesting to compare the performance of this mapping implementation with the dictionary of the toolbox.

Charm *dynamic tables* are dictionaries with integer keys. The dynamic table operations—insert, delete and find—are non-blocking, and the result of a dynamic table operation may be sent back to a specified *entry point* in a message from the implementation. (Threads of control in a Charm system communicate using messages.) A variant of the insert operation supports a conditional insert-style operation provided by dictionary datatype of the toolbox.

The use of an application-specific (hash) function that partitions the key space into a programmer-specified number of subspaces for improved performance is somewhat similar to state partitioning or domain decomposition techniques used for distributed memory computers such as the Cosmic Cube, where the state of a data structure is distributed over a number of processors, and each processor is responsible for handling requests for its part of the data structure [60].

Barth describes a parallel hash table in which each bucket is implemented using an M-structure to reduce contention [6]. This design allows concurrent insert operations to the same bucket to be pipelined, at the cost of a parallel (complicated) interface. On the other hand, the dictionary implementation of the parallel programming toolbox supports per-bucket locking through a sequential interface. Concurrent insert operations to the same bucket are synchronized, but the interface is much simpler.

Autolock The programming language Mesa provides *monitored objects* for associating monitors with individual objects [45]. The lock of a monitored object is passed as an argument to the entry procedure of the monitor, and is automatically acquired and released by the monitor implementation. However, the object-lock association is still left to the application programmer.

Id with M-structures provides *atomic objects*, which are mutable structures that are transparently associated with locks. An *atomic scope* is similar to a critical section provided by `With-Autolock`, except that *only* the take operations with side effects on the atomic object in the scope are synchronized. Other operations are allowed to proceed without delay to achieve more throughput. The compiler ensures that an atomic object is only accessed within an atomic scope.

Midway is a distributed shared memory programming system for multicomputers [8] that supports a novel weakly consistent memory model called *entry consistency*. In this model shared data is guaranteed to be consistent at a processor when the processor acquires a synchronization object (e.g. lock or barrier) known to guard the shared data. The association between a synchronization object and shared data is made explicit using annotation in Midway programs, either as a declaration (e.g. `shared`) or dynamically by a call to the runtime system. The association between shared data and synchronization objects in Midway programs can benefit from a facility such as autolocks, which maintains the association between shared data and synchronization objects transparently. With this facility accesses to shared data may be made without making explicit references to the synchronization object.

Optimistic Read-Modify-Write Charm provides *monotonic variables* whose values increase monotonically according to some programmer-defined metric. A monotonic variable supports an atomic update operation that is commutative, associative, monotonic and idempotent. This update operation modifies a monotonic variable only if the new value satisfies some partial order predicate with respect to the existing value of the variable. Monotonic variables are designed for use in optimization problems in which the value of a variable (say the current best bound in a branch-and-bound program) increases monotonically. The update operation corresponds to a monotonic *non-optimistic* read-modify-write operation in which the read, modify, write and monotonicity test operations are lumped into the monotonic update operation. The monotonicity test is somewhat analogous to the test for compatibility between concurrent updates of `ORMW`, but the monotonic variable interface does not allow the implementation to use optimistic protocols to reduce contention for concurrent update operations on a monotonic variable.

Herlihy's methodology for implementing *lock-free* and *wait-free* concurrent datatypes for large objects [25] is somewhat similar to the `ORMW` abstraction of the toolbox. In both cases only a part of the object is modified, and the programmer uses the semantics of the datatype to minimize the parts of the object that has to be copied. The major conceptual distinction between these two approaches lies in their treatment of concurrent updates to a shared object (race conditions). Under `ORMW` an *application-specific* test is used to determine if any of the concurrent updates to a shared object are compatible (e.g., linearizable), based on the *semantics* of the object.

If so, the concurrent updates will (all) complete successfully. Otherwise, the incompatible updates are aborted and re-executed. Herlihy's approach is a special case of `ORMW` in which all concurrent updates are considered incompatible. Consequently, concurrent updates are effectively serialized at the implementation level. Herlihy has also mentioned the use of the semantics of the object to increase concurrency as a possible refinement of his methodology. Another difference is that the current implementation of `ORMW` uses locking, so `ORMW` operations are neither lock-free nor wait-free.

Concurrent Aggregate, Chameleon, Multipol and Midway all support application-specific notions of data consistency for shared mutable objects in some way.

Per-Thread Locations Multiprocessing SPUR Lisp provides dynamically-bound *special* variables that may be bound to different values in different processes [76]. A special variable (name) has different bindings in different processes, and a process is unable to access or modify the binding of a special variable rebound by another process. Top-level special variables, however, may be accessed by all the processes in the system. These two cases are similar to the private and non-private (default) slots of structures defined by pdefstruct respectively.

Structures defined by pdefstruct of the parallel programming toolbox are very similar to the multiported objects defined by Yelick. For example, a multiported work queue may be implemented with one internal queue for each thread that accesses the queue. To reduce contention, each thread would examine its own queue for work before checking queues that belong to other threads. The ports of a multiported object are much more closely coordinated than the per-thread locations of a shared structure, which are used to eliminate false sharing.

In Concurrent Aggregates a shared data structure with per-thread data may be implemented as an aggregate, with one representative for each thread that accesses the data structure. The per-thread data is stored in the corresponding representative.

Barth describes the use of M-structures in Id to construct per-thread hash tables for storing per-thread data about nodes in shared graph structures. As an example, a *mark* field in a graph node may be used to denote that the node has been visited in the current graph traversal. To use these mark fields in an algorithm in which there may be multiple concurrent traversals, the mark fields of all the nodes may be replaced by a per-thread hash table that maps nodes to mark values. (Alternatively, a per-node hash table may also be used to map threads to mark values.) However, the use of hash tables for storing per-thread data changes the way the per-thread fields of a node are accessed, and requires substantial changes to the original program.

10.3.3 Architectural Support for Data-Sharing Abstractions

This section describes architectural features that may be used to implement data-sharing abstractions more efficiently.

Load-linked, Store-conditional Herlihy describes a methodology for deriving *lock-free* and *wait-free* implementations of a concurrent datatype from a stylized sequential implementation of the datatype [25]. An implementation is lock-free if *some* process accessing the object will complete an operation in a finite number of steps, and wait-free if *each* process will complete an operation in a finite number of steps. The methodology is based on a pair of atomic primitives on shared memory—*load-linked* and *store-conditional*—that is available on the MIPS [41] and DEC Alpha [61] architectures.

Transactional Memory Herlihy recently proposed the use of *transactional memory* to support the efficient implementation of *lock-free* data structures on shared-memory multiprocessors [26]. A lock-free data structure does not require the use of mutual exclusion in its implementation, and is attractive because locking frequently results in problems including priority inversion, convoying and deadlock. A transactional memory supports customized multi-word read-modify-write operations, and can be implemented by extensions to the multiprocessor cache-coherency protocol. It will be interesting to see whether a transactional memory-based toolbox implementation outperforms a lock-based implementation for real applications, especially in applications with high contention, and whether transactional memory will be adopted by future multiprocessor designers. In addition, it will be interesting to compare the performance and implementation complexity of a transactional memory-based toolbox with one based on (regular) shared memory.

Cedar Synchronization Instructions A set of synchronization instructions have been proposed for the Cedar multiprocessor [75]. These atomic instructions operate on Cedar synchronization variables, which have *key* and *data* fields. The key field contains an integer while the data field may contain an integer or a floating point number. A synchronization instruction performs a (numerical) comparison test between the key field and an immediate value. If the test succeeds, separate operations are performed on the key and data fields. Operations on the key field include increment, decrement, fetch, fetch & add, store etc. Operations on the data field include fetch and store. The result of the test is available to the processor. A variant of these synchronization instructions repeats the test until it succeeds. A Cedar synchronization instruction may be considered as a hardware implementation of a conditional update operation for numerical datatypes. The repeating variant of these instructions bears some resemblance to the retry feature of `ORMW`.

10.4 Asynchronous Iterations

The formulation of asynchronous iterations in Chapter 5 is similar to *generalized iterations* defined by Pohlmann in the context of parallel discrete event simulation [54]. Generalized iterations operate in domain of infinite streams of elements, each of which corresponds to an event over time in the system being simulated.

Parallel relaxation is also used by various schemes for solving systems of equations $X \leftarrow f(X)$ in the domain of real numbers in parallel, including *chaotic relaxations* defined by Chazan and Miranker [12], *asynchronous iterations* defined by Baudet [7] and *chaotic iterations with delay* defined by Miellou [50]. In these systems f is made up of component functions f_c, and the state X of the fixed-point computation may be decomposed into a set of (potentially disjoint) components that are computed by different f_c functions concurrently. Each f_c may be computed using *multiple* previous states, with a different state for each component of X, to minimize the amount of synchronization required. These schemes are optimized for the domain of real numbers by using the properties of real numbers, and their convergence criteria are somewhat analogous to the test for monotonicity used in monotonic asynchronous iterations. My formulation of asynchronous iterations for parallel discrete relaxation is a domain-independent generalization of these parallel iteration schemes. In addition, the use of an optimistic test for monotonicity for improved parallel performance is unique to my approach.

At an abstract level asynchronous iteration is also similar to implementations of certain data structures (e.g. B-trees) based on multi-version memory [72]. In this scheme the correctness of concurrent operations on a shared structure is not affected by the use of an outdated version of the structure, though a performance penalty may be incurred. Multi-version memory may be implemented more efficiently than coherent shared memory because multi-version memory allows outdated versions of an object to be cached, while coherent shared memory requires that all outdated cached copies be invalidated.

10.5 Parallel Constraint Satisfaction

Many algorithms for parallelizing constraint satisfaction have been published [57, 58]. These algorithms only solve for local consistency, not global consistency, and are used as a preprocessing step to reduce the size of the search space before another (more expensive) algorithm is used to find a globally consistent solution. CONSAT is unusual among constraint satisfaction systems in that it computes globally consistent solutions by applying a local technique.

Kasif analyzed the use of massive parallelism to achieve local consistency, and concluded that massive parallelism is unlikely to improve the complexity of known sequential solutions significantly [42]. On the other hand, he explicitly left open the possibility of using a constant (small) number of processors to speedup the computation, as I have described in this dissertation.

10.6 Dag-Traversal

10.6.1 Circular Grammar Evaluator

RC performs a fixed-point computation over a directed graph G by partitioning G into a dag D of strongly-connected components, and computing the fixed points of these components in an order that obeys the precedence constraints in D (Section 6.3.4). This algorithm is significantly more efficient than a naive algorithm that computes the fixed point of G directly, and is inspired by Jones' work on the evaluation of circular attribute grammars [38]. Parallel RC generalizes Jones' algorithm by computing the fixed points of multiple components concurrently (subject to the precedence constraints). In addition, the fixed point of any individual component may also be computed (internally) in parallel.

10.6.2 Hybrid Dataflow Analysis Algorithms

Lee et al. proposes the use of *hybrid* dataflow analysis algorithms on distributed memory multiprocessors [47]. A hybrid algorithm decomposes a program flow graph G into a dag of strongly-connected components, and uses a process called *factorization* to express the dataflow information at the exit nodes of an component in terms of the dataflow information at the entry node of the component. A (sequential) fixed-point algorithm is then used to express the dataflow information at each internal node of C in terms of the information at the entry of C. Then intercomponent dataflow information is propagated in some topological order, starting from the components without predecessors. Finally, information for each internal node is computed.

RC and hybrid algorithms both decompose the program flow graph into a dag of components, and uses dag traversal over the components. However, RC does not (and indeed cannot) factorize the flow problem, and thereby loses some degree of parallelism, some of which it regains by parallelizing within a component. Therefore, the RC approach is weaker, but works on unfactorizable problems and is thus more general.

Chapter 11

Conclusion

In this chapter I will review the contributions of the research described in this dissertation, and close with some directions for future work.

11.1 Contributions

In this dissertation I proposed the use of a toolbox to simplify symbolic parallel programming for novice parallel programmers. I have identified and implemented a number of parallelism abstractions and data-sharing abstractions, and used them in the parallelization of two medium-size symbolic applications. The toolbox-based versions of these applications may be ported between uniprocessors and shared-memory multiprocessors without modification, and perform reasonably efficiently on both platforms. As all the parallel programming details are hidden in the toolbox implementation, these abstractions may be used by programmers who are not experienced in parallel programming. I have also proposed monotonic asynchronous iteration as an efficient parallel implementation of discrete relaxation. A more detailed summary of the contributions follows.

Parallel Programming Toolbox Various parallel programming languages or systems differ in the amount of programmer effort required to create a parallel program. Automatic program analysis tools or parallelizers represent one end of the spectrum.

For example, Curare transforms a sequential Scheme program into an equivalent parallel program automatically [46]. Data dependencies between multiple threads of control in the parallel version are serialized using locks. No programmer intervention is required. As the parallelizer is only given a sequential *implementation* of the underlying algorithm, potential sources of concurrency in the *algorithm* may not be discovered by the parallelizer. For example, the sequential implementation may contain extraneous orderings that are not necessary for the correctness of the algorithm, but would prevent the *sequential implementation* from being parallelized successfully.

Languages or systems for explicitly parallel programs represent the other end of the spectrum of programmer effort required. The programmer may use high-level knowledge or understanding of the *algorithm*—perhaps extracted from a sequential implementation—to create a parallel implementation that has good parallel performance. However, these systems are designed for use by parallel programming experts, and a large amount of low-level parallel programming details—including the synchronization of concurrent accesses to shared data—has to be specified by the programmer. In general, the sequential and parallel versions of the same program are substantially different, and there are significant software engineering costs in maintaining two versions of a given program. In extreme cases this cost alone may preclude the parallelization effort of a large existing sequential program.

The toolbox approach to parallel programming represents an engineering compromise between these two extremes of programmer involvement. The programmer may use his high-level understanding of the underlying algorithm to identify the sources of parallelism, and is also responsible for identifying all instances of data sharing in the parallel implementation. The toolbox provides parallelism abstractions for introducing parallelism, and data-sharing abstractions for simplifying concurrent data accesses in the common cases.

A parallelism abstraction has a sequential interface, and allows the programmer to specify the sources of parallelism in an algorithm as a set of computations that may be executed concurrently. On a multiprocessor these computations may be executed concurrently by the implementation of the abstraction. All the low-level parallel programming details are hidden from the the programmer. For example, the

`Fixed-Point` abstraction may be used to specify the sources of parallelism in iterative algorithms for finding the fixed point of a system, and may also be used in general work queue-style computations. As another example, the `Dag-Traversal` abstraction visits the nodes of a dag such that a node is not visited until all its predecessor nodes have been visited.

The data-sharing abstractions cover a spectrum that trades ease of use with performance, and allow a programmer to use more complicated abstractions for heavily contended data structures to obtain better performance. The data-sharing abstractions include simple concurrent datatypes (including dictionaries), autolocks, optimistic read-modify-writes and per-thread locations. Simple concurrent datatypes and dictionaries are supported by the toolbox directly, and may be used as drop-in replacements for their sequential counterparts. The use of conditional update operations on these datatypes eliminates a major source of race conditions in parallel programs. Autolocks may be used to synchronize concurrent accesses to arbitrary (lightly contended) objects not supported as concurrent datatypes by the toolbox, and very little modification to the source code is required. Optimistic read-modify-write operations may be used to synchronize concurrent updates to heavily contended objects by making use of the semantics of the datatype to achieve more concurrency. They have the potential of providing significant performance improvements over implementations based on simple critical sections, but require some high-level understanding of the operations being performed. Per-thread locations in structures help eliminate false sharing of structure fields in parallel code, and simplify the parallelization of a class of sequential programs (with benevolent side effects) significantly.

The data-sharing abstractions are designed for use by novice parallel programmers, and may be retrofitted to existing (sequential) programs with minimal modifications. In addition, programs written using these abstractions may be executed on uniprocessors and shared-memory multiprocessors without modification. As most of the low-level parallel programming details are hidden by the toolbox implementation, programs that use these tools are less error-prone than explicitly parallel programs that do not use them.

To locate the performance bottlenecks in an application that uses the toolbox, a probe process (provided by the toolbox) may be used to periodically display the states of interesting (internal) data structures, such as the status of the work queues and contention statistics of various locks. The programmer may also customize this probe process to display the status of application-specific data structures. Once the bottlenecks are located, the performance hooks provided by the abstractions may be used to remove them. For example, `Fixed-Point` may be used in conjunction with an application-specific scheduler to reduce the number of activations needed to compute the fixed point of the system. The granularity of locking for concurrent operations on a dictionary may be adjusted to trade space for time. In addition, an application-specific `Split-fn` that is faster and distributes objects more uniformly than the toolbox default may be used for the dictionary. Other data-sharing abstractions are metered so that data contention can be identified easily. This contention may be reduced by locking program objects (using autolocks) at a finer granularity, by using optimistic read-modify-write operations instead of autolocks, and by overriding the default parameters for the backoff spin locks used to implement the abstractions. Through the use of these performance hooks, performance bottlenecks may be identified and removed relatively easily.

Monotonic Asynchronous Iteration I have introduced monotonic asynchronous iteration as a correct and efficient way of implementing parallel discrete relaxation for systems for which monotonicity is a necessary correctness condition. Monotonic asynchronous iteration uses an optimistic scheme to compute a possible next state of the system. This optimistic scheme is highly efficient but is not necessarily monotonic (i.e., correct). An application-specific test for monotonicity is then applied to the computed state, and if the test succeeds the state transition is made (atomically). Otherwise, the computation is repeated. I have applied monotonic asynchronous iteration successfully to the parallel implementation of a constraint satisfaction system that computes globally consistent solutions, for which monotonicity is a necessary correctness condition that is not automatically satisfied. I believe monotonic asynchronous iteration is applicable to parallel discrete relaxation in general. It will be interesting to see if this application-specific monotonicity test is easy to derive for other discrete relaxation problems.

11.2 Future Directions

The ultimate goal of this research is to make the power of multiprocessors accessible to sequential programmers. I have chosen to focus on the area of symbolic (irregular and dynamic) applications, and used shared-memory multiprocessors as the initial target.

The toolbox abstractions described in this dissertation have all been designed in the process of parallelizing real applications. An obvious extension of this research is to use the toolbox to parallelize larger symbolic applications. In particular, the toolbox may be specialized for a particular domain (e.g. VLSI CAD or heuristic search) by providing application-specific abstractions.

Another possible extension of this research is to implement the toolbox on the current generation of shared-address space multiprocessors, including those from Sun, SGI, KSR and Tera. Looking further, the feasibility of using the toolbox approach on distributed-memory multiprocessors or networks of workstations may be investigated. Issues such as data partitioning and placement will have to be considered.

On the performance side, the task of identifying and removing performance bottlenecks in a toolbox-based program should be simplified. The current toolbox implementation collects contention statistics that enable the programmer to identify and remove performance bottlenecks manually. A future implementation of the toolbox may use these statistics to improve the performance of subsequent runs of the same program by varying certain parameters automatically. The current implementation of the parallelism abstractions does not support any form of dynamic partitioning strategy to control the grain size of concurrent computations. A mechanism that will allow the programmer to use a dynamic partitioning strategy [71, 56, 51] to balance the overhead of a parallelism abstraction against grain size should be provided.

Bibliography

[1] Gul A. Agha. Actors: A model of concurrent computation in distributed systems. Technical Report 844, MIT Artificial Intelligence Laboratory, Cambridge, MA, June 1985.

[2] Gail A. Alverson and David Notkin. Abstracting data-representation and partitioning-scheduling in parallel programs. In *Proceedings of the International Symposium on Shared Memory Multiprocessing*, pages 138–151, Tokyo, Japan, April 1991.

[3] Gail A. Alverson and David Notkin. Program structuring for effective parallel portability. *IEEE Transactions on Parallel and Distributed Systems*, 4(9):1041–1059, September 1993.

[4] Thomas E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[5] R. J. R. Back. A method for refining atomicity in parallel algorithms. In Eddy Odijk, Martin Rem, and Jean-Claude Syre, editors, *PARLE 89 Parallel Architectures and Languages Europe*, volume II: Parallel Languages of *Lecture Notes in Computer Science 366*, pages 199–216, Eindhoven, The Netherlands, June 1989.

[6] Paul S. Barth. *Atomic Data Structures for Parallel Computing.* PhD thesis, MIT Laboratory for Computer Science, Cambridge, MA, March 1992.

[7] Gerard M. Baudet. Asynchronous iterative methods for multiprocessors. *Journal of the ACM*, 25(2):226–244, April 1978.

[8] Brian N. Bershad, Matthew J. Zekauskas, and Wayne A. Sawdon. The Midway distributed shared memory system. In *Digest of Papers: COMPCON SPRING '93*, pages 528–537, San Francisco, CA, February 1993.

[9] Ralph Butler and Ewing Lusk. User's guide to the p4 programming system. Technical Report ANL-92/17, Argonne National Laboratory, Mathematics and Computer Science Division, Argonne, IL, October 1992.

[10] Soumen Chakrabarti and Katherine Yelick. Distributed data structures and algorithms for grobner basis computation. *Lisp and Symbolic Computation*, 1994. To appear.

[11] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A foundation*. Addison–Wesley Publishing Company, Reading, MA, 1988.

[12] D. Chazan and W. Miranker. Chaotic relaxation. *Linear Algebra and its Applications*, 2:199–222, 1969.

[13] Andrew Andai Chien. *Concurrent Aggregates(CA): An Object-Oriented Language for Fine-Grained Message-Passing Machines*. PhD thesis, MIT Artificial Intelligence Laboratory, Cambridge, MA, July 1990.

[14] Murray Cole. *Algorithmic skeletons: Structured management of parallel computations*. Research monographs in parallel and distributed computing. MIT Press, Cambridge, MA, 1989.

[15] Kendall Square Research Corp. KSR1 Technical Summary. Waltham, MA, 1993.

[16] John R. Ellis, Kai Li, and Andrew Appel. Real-time concurrent collection on stock multiprocessors. Technical Report 25, DEC Systems Research Center, Palo Alto, CA, February 1988.

[17] Franz Inc. *Allegro CLiP Manual*, release 3.0.3 edition, March 1990.

[18] Richard P. Gabriel and John McCarthy. Queue-based multi-processing LISP. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, pages 25–44, Austin, TX, August 1984.

[19] Ron Goldman. Implementation of Qlisp special variables. Personal communication, July 1990.

[20] Ron Goldman and Richard P. Gabriel. Qlisp: Parallel processing in Lisp. *IEEE Software*, 6(4):51–59, July 1989.

[21] Hans W. Guesgen, Kinson Ho, and Paul N. Hilfinger. A tagging method for parallel constraint satisfaction. *Journal of Parallel and Distributed Computing*, 16(1):72–75, September 1992.

[22] Hans Werner Guesgen. *CONSAT: A System for Constraint Satisfaction*. Research Notes in Artificial Intelligence. Morgan Kaufmann, San Mateo, CA, 1989.

[23] Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[24] Robert H. Halstead, Jr. An assessment of Multilisp: Lessons from experience. *International Journal of Parallel Processing*, 15(6):459–501, 1986.

[25] Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, November 1993.

[26] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *The 20th International Symposium on Computer Architecture*, pages 289–300, San Diego, CA, May 1993.

[27] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[28] Kinson Ho, Hans W. Guesgen, and Paul N. Hilfinger. Consat: A parallel constraint satisfaction system. Submitted to *Lisp and Symbolic Computation*, 1994.

[29] Kinson Ho and Paul N. Hilfinger. Managing side effects on shared data. In Robert H. Halstead, Jr. and Takayasu Ito, editors, *Parallel Symbolic Computing: Languages, Systems, and Applications (US/Japan Workshop Proceedings)*, number 748 in Lecture Notes in Computer Science, pages 205–232, Cambridge, MA, November 1993. Springer-Verlag.

[30] Kinson Ho, Paul N. Hilfinger, and Hans W. Guesgen. Optimistic parallel discrete relaxation. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, volume 1, pages 268–273, Chambery, France, August 1993.

[31] Kinson Ho, Edward Wang, and Paul N. Hilfinger. RC: A recursive-type analysis program. Unpublished paper, 1994.

[32] Berthold Klaus Paul Horn. *Robot Vision*. MIT Electrical Engineering and Computer Science Series. MIT Press, Cambridge, MA, 1986.

[33] Akira Imai and Evan Tick. Evaluation of parallel copying garbage collection on a shared-memory multiprocessor. *IEEE Transactions on Parallel and Distributed Systems*, 4(9):1030–1040, September 1993.

[34] George K. Jacob. What price, parallel Lisp? In *High Performance and Parallel Computing in Lisp*, Twickenham, London, United Kingdom, November 1990.

[35] George K. Jacob and Robert F. Rorschach. Building a commercial-quality parallel Common-LISP system. In *The First European Conference on the Practical Application of Lisp*, pages 101–108, Cambridge, United Kingdom, March 1990.

[36] Suresh Jagannathan and Iim Philbin. A customizable substrate for concurrent languages. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 55–67, San Francisco, CA, June 1992.

[37] Suresh Jagannathan and Iim Philbin. A foundation for an efficient multi-threaded Scheme system. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, pages 345–357, San Francisco, CA, June 1992.

[38] Larry G. Jones. Efficient evaluation of circular attribute grammars. *ACM Transactions on Programming Languages and Systems*, 12(3):429–462, July 1990.

[39] L. V. Kale. The design philosophy of the Chare Kernel parallel programming system. Technical Report UIUCDCS-R-89-1555, University of Illinois, Urbana-Champaign, IL, November 1989.

[40] L. V. Kale. The Chare Kernel parallel programming language and system. In *The Proceedings of the International Conference on Parallel Processing*, volume II, pages 17–25, St. Charles, IL, August 1990.

[41] Gerry Kand and Joe Heinrich. *MIPS RISC Architecture*. Prentice–Hall, Inc., Englewood Cliffs, NJ, 1992.

[42] Simon Kasif. On the parallel complexity of discrete relaxation in constraint satisfaction networks. *Artificial Intelligence*, 45(3):275–286, October 1990.

[43] Simon Kasif and Azriel Rosenfeld. The fixed points of images and scenes. In *Proceedings CVPR '83: IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 454–456, Washington, DC, June 1983.

[44] Donald E. Knuth. *The Art of Computer Programming*, volume 3: Sorting and Searching. Addison-Wesley, Reading, MA, 1973.

[45] B. W. Lampson and D. D. Redell. Experience with processes and monitors in Mesa. *Communications of the ACM*, 23(2), February 1980.

[46] James R. Larus. *Restructuring Symbolic Programs for Concurrent Execution on Multiprocessors*. PhD thesis, Computer Science Division (EECS), University of California, Berkeley, CA, May 1989.

[47] Yong-fong Lee, Thomas J. Marlowe, and Barbara G. Ryder. Experiences with a parallel algorithm for data flow analysis. *The Journal of Supercomputing*, 5(2-3):163–188, October 1991.

[48] Ewing Lusk, James Boyle, Ralph Butler, Terrence Disz, Barnett Glickfeld, Ross Overbeek, James Patterson, and Rick Stevens. *Portable Programs for Parallel Processors*. Holt, Rinehart and Winston, Inc., New York, NY, 1987.

[49] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. Technical Report 342, Computer Science Department, University of Rochester, Rochester, NY, April 1990.

[50] Jean-Claude Miellou. Iterations chaotiques a retards; etudes de la convergence dans le cas d'espaes partiellement ordonnes (Chaotic iterations with delay; studies of convergence for the case of partially ordered spaces). *Comptes Rendus Hebdomadaires des Seances De L'Academie des Sciences*, 280, Series A(4):233–236, January 1975. In French.

[51] Eric Mohr. *Dynamic Partitioning of Parallel Lisp Programs*. PhD thesis, Department of Computer Science, Yale University, New Haven, CT, October 1991.

[52] Eric Mohr, David A. Kranz, and Robert H. Halstead, Jr. Lazy task creation: A technique for increasing the granularity of parallel programs. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, pages 185–197, Nice, France, June 1990.

[53] D. Stott Parker. Partial order programming. Technical Report CSD-870067, Computer Science Department, University of California, Los Angeles, CA, December 1987.

[54] Werner Pohlmann. A fixed point approach to parallel discrete event simulation. *Acta Informatica*, 28(7):611–629, October 1991.

[55] Martic C. Rinard, Daniel J. Scales, and Monica S. Lam. Jade: A high-level, machine-independent language for parallel programming. *IEEE Computer*, 26(6):28–38, June 1993.

[56] Eric S. Roberts and Mark T. Vandevoorde. WorkCrews: An abstraction for controlling parallelism. Technical Report TR 42, Digital Equipment Corporation System Research Center, Palo Alto, CA, April 1989.

[57] Azriel Rosenfeld, Robert A. Hummel, and Steven W. Zucker. Scene labeling by relaxation operations. *IEEE Transactions on Systems, Man and Cybernetics*, 6(6):420–433, June 1976.

[58] Ashok Samal and Tom Henderson. Parallel consistent labeling. *International Journal of Parallel Processing*, 16(5):341–364, October 1987.

[59] Vivek Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. Research monographs in parallel and distributed processing. MIT Press, Cambridge, MA, 1989.

[60] Charles L. Seitz. The Cosmic Cube. *Communications of the ACM*, 28(1):22–33, January 1985.

[61] Richard L. Sites. Alpha AXP Architecture. *Communications of the ACM*, 36(2):33–44, February 1993.

[62] Guy L. Steele Jr. *Common Lisp: The Language, Second Edition*. Digital Press, Bedford, MA, 1990.

[63] Guy L. Steele Jr. Making asynchronous parallelism safe for the world. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 218–231, San Francisco, CA, January 1990.

[64] Tomoyuki Tanaka. Implementation of special variables in TOP-1 Common Lisp. Personal communication, 1990.

[65] Tomoyuki Tanaka and Shigeru Uzuhara. Multiprocessor Common Lisp on TOP-1. In *Proceedings of the Second IEEE Symposium on Parallel and Distributed Processing*, pages 617–622, Dallas, TX, December 1990.

[66] Tomoyuki Tanaka and Shigeru Uzuhara. Futures and multiple values in parallel Common Lisp. In *Proceedings of Information Processing Society of Japan SYM Meeting 91-SYM-60*, Fujisawa, Japan, June 1991.

[67] Robert H. Thomas and Will Crowther. The Uniform System: An approach to runtime support for large scale shared memory parallel processors. In *Proceedings of the 1988 International Conference on Parallel Processing*, volume 2, pages 245–254, August 1988.

[68] D. L. Waltz. Generating semantic descriptions from drawings of scenes with shadows. Technical Report AI-TR-271, MIT Laboratory for Computer Science, Cambridge, MA, 1972.

[69] Edward Wang. *Analysis of Recursive Types in an Imperative Language*. PhD thesis, Computer Science Division (EECS), University of California, Berkeley, CA, 1994. To appear.

[70] Edward Wang and Paul N. Hilfinger. Analysis of recursive types in Lisp-like languages. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, pages 216–225, San Francisco, CA, June 1992.

[71] Joseph Simon Weening. *Parallel Execution of Lisp Programs*. PhD thesis, Department of Computer Science, Stanford University, Stanford, CA, June 1989.

[72] William E. Weihl and Paul Wang. Multi-version memory: Software cache management for concurrent B-Trees (extended abstract). In *Proceedings of the Second IEEE Symposium on Parallel and Distributed Processing*, pages 650–655, Dallas, TX, December 1990.

[73] Jeannette M. Wing and Chun Gong. A library of concurrent objects and their proofs of correctness. Technical Report CMU-CS-90-151, School of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, July 1990.

[74] Katherine Anne Yelick. *Using Abstraction in Explicitly Parallel Programs.* PhD thesis, MIT Laboratory for Computer Science, Cambridge, MA, July 1991.

[75] Chuan-Qi Zhu and Pen-Chung Yew. A scheme to enforce data dependences on large multiprocessor systems. *IEEE Transactions on Software Engineering*, SE-13(6):726–739, June 1987.

[76] Benjamin Zorn, Kinson Ho, James Larus, Luigi Semenzato, and Paul Hilfinger. Multiprocessing extensions in Spur Lisp. *IEEE Software*, 6(4):41–49, July 1989.

[77] Benjamin G. Zorn. *Comparative Performance Evaluation of Garbage Collection Algorithms.* PhD thesis, University of California, Berkeley, CA, November 1989.