THE COOL PARALLEL PROGRAMMING LANGUAGE: DESIGN, IMPLEMENTATION, AND PERFORMANCE

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Rohit Chandra
April 1995

© Copyright 1995
by
Rohit Chandra
All Rights Reserved

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

John L. Hennessy
(Principal Advisor)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Anoop Gupta
(Co-advisor)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Monica S. Lam

Approved for the University Committee on Graduate Studies:

Abstract

Effective utilization of multiprocessors requires that a program be partitioned for parallel execution, and that it execute with good data locality and load balance. Although automatic compiler-based techniques to address these concerns are attractive, they are often limited by insufficient information about the application. Explicit programmer participation is therefore necessary for programs that exploit unstructured task-level parallelism. However, support for such intervention must address the tradeoff between ease of use and providing a sufficient degree of control to the programmer.

In this thesis we present the programming language COOL, that extends C++ with simple and efficient constructs for writing parallel programs. COOL is targeted towards programming shared-memory multiprocessors. Our approach emphasizes the integration of concurrency and synchronization with data abstraction. Concurrent execution is expressed through parallel functions that execute asynchronously when invoked. Synchronization for shared objects is expressed through monitors, and event synchronization is expressed through condition variables. This approach provides several benefits. First, integrating concurrency with data abstraction allows construction of concurrent data structures that have most of the complex details suitably encapsulated. Second, monitors and condition variables integrated with objects offer a flexible set of building blocks that can be used to build more complex synchronization abstractions. Synchronization operations are clearly identified through attributes and can be optimized by the compiler to reduce synchronization overhead. Finally, the object framework supports abstractions to improve the load distribution and data locality of the program.

Besides these mechanisms for exploiting parallelism, COOL also provides support for the programmer to address the performance issues, in the form of abstractions that can be used to supply hints about the objects referenced by parallel tasks. These hints are used by the runtime system to schedule tasks close to the objects they reference, and thereby improve data locality. The hints are easily supplied by the programmer in terms of the objects in the program, while the details of task creation and scheduling are managed transparently within the runtime system. Furthermore, the hints do not affect the semantics of the program and allow the programmer to easily experiment with different optimizations.

COOL has been implemented on several shared-memory machines, including the Stanford DASH multiprocessor. We have programmed a variety of applications in COOL, including many from the SPLASH parallel benchmark suite. Our experience has been promising: the applications are easily expressed in COOL, and perform as well as hand-tuned codes using lower-level primitives. Furthermore, supplying hints has proven to be an easy and effective way of improving program performance. This thesis therefore demonstrates that (a) the simple but powerful constructs in COOL can effectively exploit task-level parallelism across a variety of application programs, (b) an object-based approach improves both the expressiveness and the performance of parallel programs, and (c) improving data locality can be simple through a combination of programmer abstractions and smart scheduling mechanisms.

Acknowledgements

It is a pleasure to acknowledge the people who have contributed in different ways to this dissertation. Foremost amongst them is my advisor, John Hennessy, who provided constant encouragement and enthusiasm, and never hesitated to remind me that this dissertation was about designing a programming language. I was very fortunate to have Anoop Gupta as my co-advisor, who was there to remind me that the language had to solve the problems of real parallel programmers. I am grateful to him for his criticisms and his total hands-on involvement (how many advisors will sit and hack with you for two days?). Monica Lam served on my orals committee and read this dissertation, and helped balance my perspective on the thesis. Mendel Rosenblum served on my orals committee, and Krishna Saraswat chaired the oral examination.

Many students worked with me on COOL during various stages as the language evolved. The primary co-conspirators were Arul Menezes and Shigeru Urushibara, who participated in the initial language design and implementation. Several students subsequently used COOL for various application projects, including Avneesh Agrawal, Maneesh Agrawala, Denis Bohm, Michael Lin, Greg Lindhorst, Steve Speer, Robert Wilson, and Kaoru Uchida. I am grateful to them for developing many of the COOL applications presented here, for their patience with the prototype implementation, and for providing valuable feedback on the language design.

Members of the DASH group provided the incredibly broad environment essential for systems research; through them I learnt about other aspects of parallel computer systems. J. P. Singh and Ed Rothberg developed many of the parallel applications used in this dissertation, and cheerfully helped in porting them to COOL. Jonathan Chew and Dave Nakahira kept DASH alive and well, and Charlie Orgish kept the maze of machines in CIS working.

Several friends offered relief when I was too deeply buried in the thesis. Kourosh Gharachorloo, Ravi Soundararajan, Ben Verghese, Scott Devine, and Mendel Rosenblum provided refreshing diversions from COOL into other research areas. Avneesh Agrawal rescued me from tennis deprivation at all hours, David Cyrluk introduced me to the intramural table-tennis competition, and Hanna Djajapranata humored my pizza and ice-cream excursions. Kourosh, my office-mate for all these years, inflicted me with reluctant

learning of memory consistency models. Ashok Subramanian, my house-mate for some years, coined the name of the language (COOL). JP, my house-mate for the other years, contributed to my stamina through house-hunting tests of endurance.

Jumping through the bureaucratic hoops involved would not be possible without the administrative support provided by Margaret Rowland, Darlene Hadding, and the amazing CSD staff. They have kept the bureaucracy off my back, and kept my student life running smoothly in spite of all my efforts to the contrary (how else could I file a study list two quarters late?). I am grateful to Silicon Graphics for graciously providing me with time off to finish this dissertation, and to DARPA for patiently providing several years of funding for this research.

My final thanks are to my parents, my family, and my wife, for their constant love and support.

This dissertation is dedicated

to the memory of my father.

Contents

Abstract **iv**

Acknowledgements **vi**

1 Introduction **1**

1.1 The Problem 2

1.2 Our Approach 4

1.2.1 Expressing Parallelism 5

1.2.2 Improving Data Locality and Load Balancing 6

1.2.3 Potential Drawbacks 6

1.3 Evaluation 7

1.4 Contributions of the Thesis 7

1.5 Organization of the Thesis 8

2 Language Design and Programming Experience **10**

2.1 Runtime Execution Model 10

2.2 Language Constructs 11

2.2.1 Expressing Concurrency 11

2.2.2 Mutual Exclusion 13

2.2.3 Event Synchronization 14

2.2.4 Task-Level Synchronization 16

2.2.5 Summary 17

2.3 Examples 17

2.3.1 Illustrating the Constructs . . . 18
2.3.2 Object-Level Synchronization . . . 20
2.3.3 Exclusive Access to Multiple Objects . . . 22
2.4 Experience with Applications . . . 23
2.4.1 Water . . . 24
2.4.1.1 Exploiting Concurrency . . . 26
2.4.1.2 Expressing Synchronization . . . 26
2.4.1.3 Performance and Summary . . . 27
2.4.2 Ocean . . . 28
2.4.2.1 Exploiting Concurrency . . . 28
2.4.2.2 Expressing Synchronization . . . 30
2.4.2.3 Performance and Summary . . . 30
2.4.3 LocusRoute . . . 31
2.4.3.1 Exploiting Concurrency . . . 32
2.4.3.2 Expressing Synchronization . . . 32
2.4.3.3 Obtaining Private Per-Task Storage . . . 32
2.4.3.4 Performance and Summary . . . 35
2.4.4 Panel Cholesky . . . 35
2.4.4.1 Exploiting Concurrency . . . 37
2.4.4.2 Expressing Synchronization . . . 37
2.4.4.3 Other Performance Issues . . . 38
2.4.4.4 Summary . . . 39
2.4.5 Block Cholesky . . . 39
2.4.5.1 Exploiting Concurrency . . . 40
2.4.5.2 Expressing Synchronization . . . 40
2.4.5.3 Performance and Summary . . . 42
2.4.6 Barnes-Hut . . . 43
2.4.6.1 Expressing Concurrency and Synchronization . . . 45
2.4.6.2 Summary . . . 46
2.4.7 Discussion . . . 47
2.4.7.1 Exploiting Concurrency . . . 47
2.4.7.2 Expressing Synchronization . . . 47

2.4.7.3 The Object-Based Approach . . . 48
2.5 Alternate Design Choices . . . 49
2.5.1 Return Value of Parallel Functions . . . 49
2.5.1.1 A Variation of the Current Design . . . 50
2.5.1.2 Future Types in oldCOOL . . . 52
2.5.2 Waitfor . . . 55
2.6 Summary . . . 55

3 Data Locality and Load Balancing **57**
3.1 Exploiting the Memory Hierarchy . . . 59
3.2 Our Approach . . . 60
3.3 The Abstractions . . . 60
3.3.1 Defaults . . . 62
3.3.2 Simple Affinity . . . 62
3.3.3 Task and Object Affinity . . . 63
3.3.4 Processor Affinity . . . 63
3.3.5 Multiple Affinity Hints . . . 64
3.3.6 Task Stealing . . . 64
3.3.7 Summary of Affinity Hints . . . 65
3.4 Object Distribution . . . 65
3.5 Summary . . . 69

4 Implementation and Optimizations **70**
4.1 Implementing the Constructs . . . 70
4.1.1 Concurrency: Parallel Functions . . . 71
4.1.2 Synchronization: Monitors . . . 75
4.1.2.1 Translation of a Monitor . . . 76
4.1.2.2 The Synchronization Algorithm . . . 77
4.1.3 Synchronization: Condition Variables . . . 82
4.1.4 Synchronization: waitfor . . . 83
4.2 Overheads and Optimizations . . . 85
4.2.1 Overheads . . . 85
4.2.2 Optimizations . . . 86

4.2.2.1 Monitors with only Mutex Functions . . . 88
4.2.2.2 Spin Monitors . . . 88
4.2.2.3 Private Condition Variables . . . 89
4.2.2.4 Directly Recursive Calls . . . 90
4.2.2.5 Tail Release Optimization . . . 90
4.2.2.6 Parallel Monitor Functions . . . 90
4.2.3 Evaluating the Optimizations in Applications . . . 91
4.3 Runtime Scheduling Support for Locality Optimizations . . . 94
4.4 Summary . . . 96

5 Case Studies **97**
5.1 Ocean . . . 98
5.2 Water . . . 101
5.3 LocusRoute . . . 105
5.4 Barnes-Hut . . . 109
5.5 Panel Cholesky . . . 112
5.6 Block Cholesky . . . 117
5.7 Summary . . . 122

6 Related Work **123**
6.1 Other Parallel Languages . . . 124
6.1.1 Early Monitor-based Languages . . . 124
6.1.2 Parallel Runtime Libraries . . . 127
6.1.3 Variations in Monitor Design . . . 129
6.1.4 Concurrent Object-Based Languages . . . 131
6.1.5 Languages Based on Serial Semantics . . . 133
6.2 Exploiting Locality . . . 134
6.2.1 Operating System Based Approaches . . . 134
6.2.2 Compiler Based Approaches . . . 135
6.2.3 Explicit Approaches . . . 135
6.2.4 Scheduling in Object-Based Systems . . . 136
6.3 Summary . . . 137

7 Conclusions **139**

7.1 Future Directions 141

Bibliography **143**

List of Tables

2.1 Summary of COOL constructs. 17
2.2 Summary of experience with SPLASH applications. 24

3.1 Summary of affinity specifications. 65
3.2 Summary of data distribution constructs. 66

4.1 Instruction overhead of various COOL operations. 87
4.2 Time to execute a barrier for different numbers of processors (in microsecs). 92

List of Figures

1.1 Exploiting parallelism at different levels in a program. 2
1.2 Multiprocessor memory hierarchy. 3

2.1 Task-level synchronization using the waitfor construct. 16
2.2 A wind tunnel in PSIM4, with particles contained in space cells. 18
2.3 Expressing concurrency and synchronization in PSIM4. 19
2.4 Object-level synchronization in the Water code. 21
2.5 Acquiring access to multiple objects in a transaction based system. . . . 22
2.6 Water code expressed in COOL. 25
2.7 Grid structure in Ocean. 28
2.8 Ocean code expressed in COOL. 29
2.9 LocusRoute expressed in COOL. 31
2.10 Panel reduction structure in Cholesky. 35
2.11 Panel Cholesky expressed in COOL. 36
2.12 A matrix partitioned into blocks. 40
2.13 Block Cholesky expressed in COOL. 41
2.14 Physical space with bodies, and the corresponding tree representation (Figure courtesy of J. P. Singh). 43
2.15 Barnes-Hut expressed in COOL. 44
2.16 Illustrating futures in Multilisp. 49
2.17 Synchronization for the return value of a parallel function. 51
2.18 Potential problems with future data types. 53

3.1 Multiprocessor memory hierarchy. 59

3.2 Illustrating the affinity hints. 61
3.3 COOL code illustrating TASK and OBJECT affinity in Gaussian elimination. 64
3.4 Blocked distribution of a two-dimensional matrix through the *distribute* statement. 67
3.5 Implementation of *distribute* using *migrate*. 68

4.1 Compilation of a COOL Program. 71
4.2 Recognizing the invocation of a parallel function. 72
4.3 Translating a parallel function invocation. 73
4.4 Translating a monitor function. 76
4.5 Monitor data structures. 78
4.6 Entry code for monitor operations. 80
4.7 Exit code for monitor operations. 81
4.8 Release code for monitor operations. 83
4.9 Implementing the waitfor construct. 84
4.10 Implementation of a spin monitor. 89
4.11 The effect of synchronization optimizations in the Water code. 91
4.12 A barrier abstraction in COOL and its implementation. 93
4.13 The impact of optimizing the barrier abstraction in the Ocean code. . . . 94
4.14 Task queue structure to support task and object affinity. 95

5.1 Concurrency and affinity in the Ocean code. 99
5.2 Performance improvement with affinity hints and noStealing for Ocean. . 100
5.3 Cache miss statistics for Ocean, before and after affinity optimizations. . 100
5.4 Water code expressed in COOL. 102
5.5 Performance improvement with affinity hints and noStealing for Water. . 103
5.6 Cache miss statistics for Water, before and after affinity optimizations. . 104
5.7 The CostArray composed of regions. 106
5.8 Specifying affinity hints in LocusRoute. 107
5.9 Performance improvements in LocusRoute with affinity hints. 108
5.10 Cache miss statistics for LocusRoute, and the affect of the affinity optimizations. 108
5.11 Expressing concurrency and affinity in Barnes-Hut. 110
5.12 Barnes-Hut: The performance of different scheduling schemes. 111

5.13 Expressing concurrency and affinity in Panel Cholesky. 113
5.14 Performance improvement with affinity hints for Panel Cholesky (matrix BCSSTK33). 114
5.15 Cache miss statistics for Panel Cholesky, and the affect of the affinity optimizations. 116
5.16 The Block Code. 118
5.17 A 2-D round-robin distribution of blocks used in the ANL code (shown for four processors). 119
5.18 Performance improvement with affinity hints for Block (DENSE1000). . 119
5.19 Performance improvement with affinity hints for Block (BCSSTK15). . . 120
5.20 Cache miss statistics for Block Cholesky (Dense matrix DENSE1000). . 121
5.21 Cache miss statistics for Block Cholesky (Sparse matrix BCSSTK15). . . 121

6.1 Example code illustrating Capsules. 130

Chapter 1

Introduction

Application programs exhibit a seemingly unquenchable thirst for ever higher compute power. As computers become faster, more applications become computable within a reasonable amount of time, further driving the need for high performance computing. With the increasing availability of fast and cheap microprocessors, parallel processing has emerged as a way of obtaining high performance in a cost-effective manner. Several computer vendors, such as Intel [43, 56], Encore [34], Silicon Graphics [12, 39], Sun [22, 37], IBM [2], and Convex [27] now offer a variety of commercial multiprocessors that include both shared-memory and message-passing systems, and range in size from a few to thousands of processors. However, the increasing availability of such multiprocessors has not been matched by an equally rapid acceptability by the user community, primarily due to the difficulty of programming these computers. Harnessing the performance potential of these systems requires that programs be partitioned to efficiently utilize the parallel resources of the machine, which can be a difficult problem.

Parallelism can be exploited at several levels in an application program, including instruction-level, loop-level, and task-level parallelism (see Figure 1.1). Both instruction and loop-level parallelism can be successfully exploited through automatic techniques: compiler/hardware mechanisms can identify independent instructions and exploit instruction-level parallelism [36, 68, 79, 86, 97, 98, 102] while sophisticated compiler analysis techniques have been developed to identify independent loop iterations and thereby exploit loop-level parallelism [108, 112]. However, many codes, including many scientific and numerical applications, require the exploitation of unstructured task-level parallelism that is not amenable to such compiler-based parallelization, and must be indicated by the programmer. In this thesis we address the problem of exploiting coarse-grained parallelism at the task-level that is explicitly specified by the programmer.

We focus in particular on programming shared-memory multiprocessors. Compared

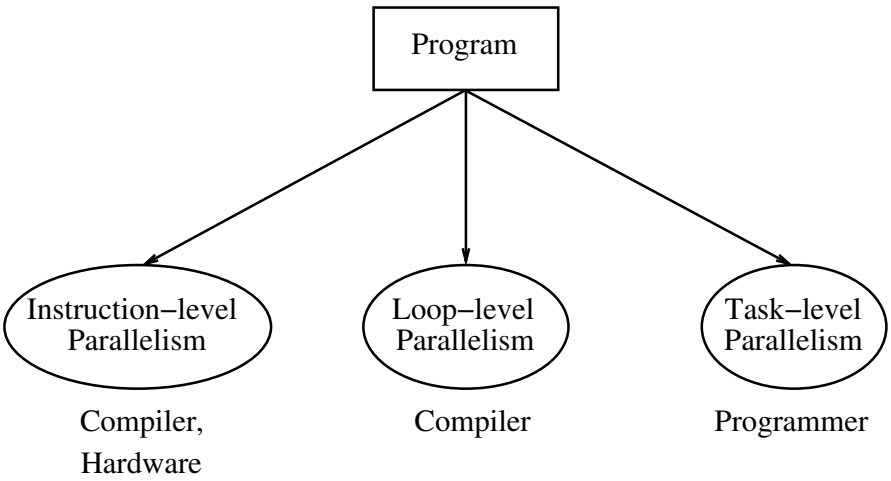

Figure 1.1: Exploiting parallelism at different levels in a program.

to message-passing systems where communication between processors must be specified through explicit messages between processors, shared-memory architectures are easier to program since inter-processor communication can be specified implicitly through load/store operations to shared memory locations. In addition, concerns regarding the scalability potential of shared-memory architectures have been successfully addressed by recent research prototypes such as the Stanford DASH multiprocessor [74], the MIT Alewife machine [1], the Kendall Square Research KSR multiprocessor [64], and the Convex Exemplar [27].

While there has been much previous research on providing explicit support for parallel programming (e.g., the ANL macros [18], CST [59], Concurrent C/C++ [40, 41], HPF [78], Linda [5], Multilisp [46], and PRESTO [14], to name just a few), the increasing availability of commercial multiprocessors in recent years has generated substantial new interest in parallel processing. Furthermore, it has enabled us to evaluate our ideas in the context of a fully implemented programming system on the Stanford DASH multiprocessor [74]. This experience has been instrumental in the development of our ideas and in focusing our research on the important problems in exploiting parallelism.

1.1 The Problem

Exploiting task-level parallelism requires that the program be partitioned for parallel execution on a target multiprocessor. The programmer first develops a parallel algorithm, either by designing an entirely new algorithm, or adapting an existing sequential algorithm for parallel execution. The parallel algorithm specifies the design of the data structures, the partitioning of the algorithm into the concurrent activities (tasks), and the coordination necessary between the parallel tasks. The algorithm is then expressed in the target parallel language.

To express parallelism as described above, we need language mechanisms to specify

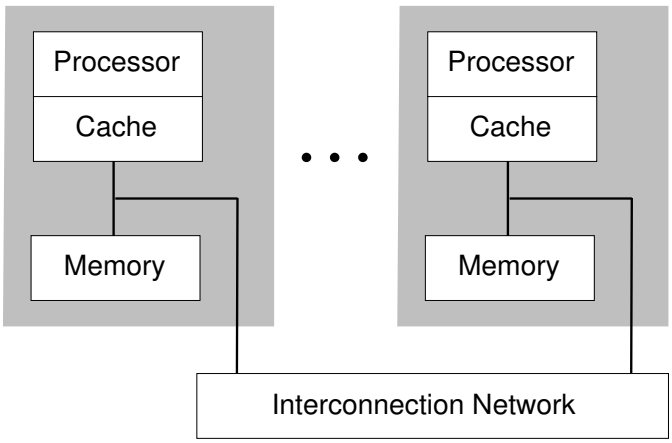

Figure 1.2: Multiprocessor memory hierarchy.

both concurrent execution as well as the communication and synchronization between the parallel tasks. The design of these language features is driven by multiple goals, primary amongst which are efficiency and expressiveness. *Efficiency* is important in that the language constructs should be efficiently implementable, and a parallel program should not incur the overheads of features that it does not use. Furthermore, the language must be *expressive* to flexibly support different concurrency patterns, thereby allowing various decompositions of a problem and supporting a wide variety of applications. However, these goals often conflict with each other. For instance, low-level primitives (such as the test-and-set instruction for synchronization) provide very efficient execution but can be cumbersome to use. On the other hand, high-level abstractions (such as the guarded command [30] for synchronization) are very expressive but can incur high implementation overheads. The design of a language, therefore, is usually a balancing act between multiple desirable goals.

Expressing the concurrency in the program is enough to get parallel execution, but is often not sufficient for good performance. Obtaining high speedups requires good data locality and load balance in the execution of the program. This is particularly important in modern multiprocessors where the performance of an application is severely affected by the high latency of memory references. Modern multiprocessors, such as the Stanford DASH [74] and the MIT Alewife [1], typically have deep memory hierarchies consisting of per-processor caches and distributed global memory (see Figure 1.2), and are termed *NUMA* architectures (for Non Uniform Memory Access). Achieving high performance on such machines requires that the computation (tasks) and the data structures (objects) in the program be appropriately assigned to the underlying processors and memory modules respectively, so that (a) the work is distributed uniformly across processors for good load balancing, and (b) tasks execute close to the data they reference in the underlying memory hierarchy for good data locality. Therefore, a parallel programming system must not only provide support for expressing parallel execution but must also address these performance issues. Furthermore, a solution should cover the range of optimizations necessary for good performance, and should be portable and robust across architectures

with different memory hierarchies.

1.2 Our Approach

We have designed the COOL programming language (Concurrent Object Oriented Language) to exploit task-level parallelism on shared-memory multiprocessors. Furthermore, COOL provides support to address the performance issues of data locality and load balancing. We elaborate on each of these components in this section.

The design of COOL has two main themes. First, as mentioned above, COOL must balance the goals of expressiveness and efficiency in the abstractions that it presents to the programmer. Since COOL is meant for high-performance parallel programming, we have chosen to emphasize efficient execution, sometimes at the cost of ease of use. COOL therefore provides simple and efficient constructs that allow straightforward applications to be expressed easily, yet enable the programmer to get full control over the parallel execution of applications that have more complex requirements. This applies to the constructs for expressing parallelism, as well as to the mechanisms for improving data locality.

Second, COOL is designed to explore the benefits of an object-based approach to writing parallel programs. Object-oriented programming has been shown to offer many benefits—it enhances the expressiveness of programs and makes them easier to understand and modify. However, organizing concurrency and synchronization around the objects in a program also has advantages that are particular to parallel programming. For instance, it enables us to build abstractions that encapsulate concurrency and synchronization within the object while providing a simple interface for its use, thereby enhancing the expressiveness of the language. Second, the synchronization operations (and the data protected by the synchronization) can be clearly identified by the compiler, enabling optimizations for efficient synchronization support. Finally, by associating tasks with the objects they reference (either automatically or through programmer supplied information), the runtime system can move tasks and/or data close to each other in the memory hierarchy, and thereby improve both data locality and load balance. An object-based approach, therefore, has the potential to improve both the expressiveness and the efficiency of a parallel program.

Overall, this research is not tied to any particular ideological approach. Rather, as we shall see throughout the thesis, our approach is pragmatic in nature. Our emphasis is on identifying the key problems in writing high-performance parallel programs, and addressing them in an effective programming system. As the language has evolved, we believe that the final design of COOL adequately addresses the important requirements for high-performance parallel programming.

Finally, COOL is designed for exploiting unstructured task-level parallelism. Regular programs that exhibit loop-level parallelism are usually more easily handled through automatic compiler-based parallelization, explicit programmer-supplied directives, or the use of data-parallel languages such as HPF [78]. COOL is meant for irregular programs

that are not amenable to such loop-level parallelization.

1.2.1 Expressing Parallelism

Rather than designing an entirely new language, we chose to extend an existing language with mechanisms for expressing parallel execution. This has the advantage of offering concurrency features in a familiar programming environment. We chose C++ [99] as our base language because of its support for defining abstract data types, its widespread use, and the availability of good compilers. However, COOL is not tied to C++, and its ideas could easily be applied to any programming language with support for abstract data types. Furthermore, our emphasis is on designing the individual constructs in COOL to exploit the data abstraction mechanisms in C++; their interaction with inheritance, while important, is not our primary concern.

The constructs in COOL are provided within the class mechanism of C++, and thereby exploit the object structure of a C++ program. Concurrent execution in COOL is expressed through the invocation of *parallel* functions that execute asynchronously when invoked. Parallel functions execute in the same address space and therefore communicate through load/store references to shared data. Synchronization between parallel functions is through monitors and condition variables [55] for mutual exclusion and event synchronization respectively.

The majority of the constructs in COOL have individually been proposed before in other languages, such as ConcurrentPascal [47], Mesa [70], and Modula-3 [21]. Our contribution has been to bring together a small and simple set of features that mesh well together, and afford the programmer flexibility and control. COOL offers a combination of different features that set it apart from previous languages. For instance, the original monitor-based languages were designed for operating systems programming on uniprocessor systems (e.g., ConcurrentPascal [47, 48]), where the primary requirement was coarse-grained sharing of resources between heavy-weight operating system processes that execute in their own private address space. In contrast, we have integrated these constructs into a language designed specifically for writing parallel programs that require fine-grained sharing of data between tasks executing in the same shared address space. We have also studied the issues that arise in implementing monitors and condition variables on a shared-memory multiprocessor, and developed compiler techniques to optimize their overheads. Together, the language design and implementation has enabled us to evaluate the usefulness of these constructs for writing high-performance parallel programs. In addition, the COOL constructs are designed to exploit the object structure of the underlying program for expressing parallelism. Finally, COOL offers language-level support for improving data locality, which is described next.

1.2.2 Improving Data Locality and Load Balancing

Good data locality and load balancing are essential for high performance in a parallel program, and require a careful distribution of tasks and data across the machine. Automatic compiler or operating system techniques to perform these optimizations have had only limited success so far, since the optimizations usually require knowledge about the program that is beyond the scope of automatic analysis. To avoid being limited by the capabilities of current techniques, it becomes important to consider programmer help in improving performance. Most current systems, however, provide minimal support for programmer participation, typically consisting of low level machine-specific primitives to schedule a process onto a particular physical processor, or to allocate data from a particular processor's memory. As a result, achieving high performance usually requires that these optimizations be explicitly hand-coded directly in terms of the underlying physical processor/memory modules.

The support provided in COOL for performing these optimizations is both easy-to-use and portable across a range of memory hierarchies, including those found in bus-based machines [39], COMA architectures [64], and NUMA multiprocessors [74]. COOL provides abstractions for the programmer to supply hints about the data objects referenced by parallel tasks. These hints are used by the runtime system to appropriately schedule tasks and migrate data, and thereby exploit locality in the memory hierarchy without affecting the semantics of the program. With this partitioning of functionality the programmer can focus on exploiting parallelism and supplying hints about the object reference patterns, leaving the machine-specific details of task creation and scheduling to the implementation and the runtime system. The hints in COOL provide a hierarchy of control, so that most common optimizations are easily obtained, yet at the same time providing full control over a range of optimizations if necessary.

1.2.3 Potential Drawbacks

Our approach does have some potential drawbacks. First, since it favors giving control to the programmer, COOL is sometimes not as expressive as other languages that provide more elaborate constructs, and some applications that can be directly expressed in other languages require the COOL programmer to explicitly build the desired synchronization. In this regard, therefore, COOL represents a particular design point in the tradeoff between expressiveness and efficiency. Other, perfectly reasonable languages, may well choose to make a different tradeoff, and emphasize ease-of-use at the cost of efficient execution.

Second, although the object-based approach is usually beneficial, there are applications where there is a mismatch between the object granularity and the desired concurrency, synchronization, and/or communication granularity. In such situations the object structure is useless (at best) and may even be cumbersome. In COOL, therefore, we provide the programmer the flexibility to circumvent the object structure when necessary.

Finally, the mechanisms in COOL to improve locality and load-balance only make it

easier to actually *perform* the desired task and data distribution optimizations. Determining the useful optimizations requires an understanding of both the application structure as well as the memory hierarchy, and continues to be incumbent upon the programmer.

1.3 Evaluation

To evaluate the ideas presented in the thesis we have implemented COOL on several shared-memory multiprocessors and programmed a variety of applications in the language. The machines we have ported to include the SGI workstations [12] (8 processors), the Encore Multimax [34] (32 processors), and the Stanford DASH multiprocessor [74] (64 processors). Our applications have been chosen from the SPLASH [96] suite of parallel benchmarks. These are important classes of applications from the scientific and engineering domain that are likely to benefit from the performance potential of parallel processing. Most of these applications are irregular in structure and not amenable to automatic compiler-based parallelization. The programs are large, ranging from 3,000 to 9,000 lines of code, and are coded using the ANL macros [18] to express parallelism.

Having a set of realistic applications implemented on a modern multiprocessor has enabled us to do a thorough evaluation of the design, implementation, and performance of the language. We evaluate the effectiveness of the language constructs in expressing different forms of concurrency and synchronization. We evaluate the efficiency of the language by analyzing the performance of the COOL programs on the DASH multiprocessor. We evaluate the effectiveness of our approach to improving data locality by applying it to the COOL version of the applications, monitoring the programmer effort involved, and measuring the performance gains. Along with measuring overall improvements in application performance, we use the hardware performance monitor on DASH [75] to monitor the effects on the cache and memory system behavior in detail. We also compare the performance of the COOL applications to that of equivalent codes hand-tuned using the ANL macros [18] in the SPLASH suite.

COOL has been operational for over two years and has been used in research projects by several undergraduate and graduate students. In addition, it has been used in a graduate class that attracts students from many engineering departments and includes a substantial parallel programming project. COOL has also been distributed to other university and industrial sites. Feedback from the diverse set of users has been integral to the evolution of the language.

1.4 Contributions of the Thesis

In this thesis we describe the design and implementation of COOL, as well as our programming experience and performance evaluation using a variety of parallel applications. Based on this experience, this thesis makes the following contributions.

The most important lesson we have learned is the importance of simplicity and efficiency in designing constructs for parallel programming. We show that simple constructs that provide the control and flexibility to build application-specific abstractions can offer programming ease as well as enable efficient programs.

Previously it was shown that monitors and condition variables were good for operating systems programming. In this thesis we show that these constructs are also effective for writing high-performance parallel programs. They are appealing because they can directly express the requirements of most parallel applications, as well as serve as flexible building blocks for expressing more complex applications. Furthermore, they can be implemented very efficiently.

We show that an object-based approach that integrates concurrency and synchronization with data abstraction improves both the expressiveness and the efficiency of the language. The programmer can build abstractions that encapsulate the details of concurrency and synchronization within the object. A compiler can analyze the operations on monitor objects, and optimize their implementation to substantially reduce the synchronization overheads in the program.

We develop a new approach to improving data locality that combines programmer supplied pragmas with scheduling mechanisms in the runtime system. We show that this approach is very effective—it is easy to use and portable across different architectures. Yet, as demonstrated by our results, COOL programs optimized in this fashion perform as well as hand-tuned codes.

1.5 Organization of the Thesis

In Chapter 2 we present the design of COOL. We outline the basic design philosophy behind the language, describe each of the constructs, and illustrate them through example programs that include small kernels as well as large applications.

In Chapter 3 we describe the support provided in COOL for improving data locality and load balance in parallel programs. We present the abstractions provided to the programmer, and the scheduling heuristics used by the runtime system.

In Chapter 4 we describe the COOL implementation, indicating the techniques employed to support each construct efficiently. We then describe several compiler techniques that we have developed to optimize the overheads associated with the synchronization constructs. Finally, we describe the runtime task-queue structures to support the scheduling optimizations.

In Chapter 5 we present case-studies of several applications written in COOL. We present performance results on the Stanford DASH multiprocessor [74], and evaluate the effectiveness of our approach to improving data locality and load balance in these applications.

In Chapter 6 we present related work. We compare COOL with other parallel programming languages, and compare our approach to exploiting locality with other automatic and manual techniques.

Finally, in Chapter 7 we present concluding remarks and discuss directions for future research.

Chapter 2

Language Design and Programming Experience

In this chapter we describe the COOL parallel programming language. Our presentation is organized as follows. We first describe the runtime execution model for a COOL program. We then describe the constructs in COOL and show how they are integrated within the class mechanism of C++. We illustrate the constructs through small examples and then describe how we programmed several large parallel applications in COOL. In light of this programming experience, we discuss the strengths and limitations of the COOL constructs and evaluate the benefits of our object-based approach. Finally we present some alternate choices that we considered (and discarded) during the design of the language.

2.1 Runtime Execution Model

An executing COOL program creates units of work called *tasks* (commonly referred to as a *thread* of execution), with multiple tasks executing within the same shared address space. The execution of these tasks is implemented in the COOL runtime system as *user-level* or *light-weight threads* [28, 31]. The runtime system manages the execution of tasks in a manner analogous to the management of processes by the operating system, except that the creation and scheduling of tasks is managed entirely within user space. As a result tasks are implemented very efficiently—the overhead of creating and then executing a task in a COOL program is about 400 machine instructions on a MIPS R3000 processor (see Chapter 4). This is far more efficient than either Unix processes or kernel-level threads.

Programming with light-weight threads is significantly easier than programming in

process-oriented models such as the ANL macros [18]. The abstraction presented to the programmer in these process-oriented models is that of multiple heavy-weight operating system processes executing concurrently, with operations such as creating a process (fork) or switching contexts from one process to another consuming several thousand instructions. In this programming model, therefore, the programmer creates processes only at the start of the program and thereupon explicitly manages the allocation of work across these processes, in effect building an explicit task management system. Even though this allows the programmer to hand-tune the allocation of work to be very efficient, explicitly managing the creation and scheduling of tasks can be very cumbersome in practice. In contrast, light-weight threads handle the details of task management automatically within their runtime system.

An important consideration in expressing concurrent execution is the granularity of a parallel task. There is an inherent tradeoff between task granularity and load balance: large numbers of small tasks allow for good load balancing of the work across processors, but if the tasks are too small then the overheads of concurrency may be excessive. On the other hand, large tasks (perhaps to the extreme of a single task per processor) incur minimal overhead, but variations in the amount of work within each task can result in poor load balancing. The granularity of tasks must therefore be chosen carefully and may require the programmer to divide large tasks into smaller ones, or aggregate several small tasks into a larger task. We shall see instances of these different scenarios later in this chapter when we discuss our experience writing parallel applications in COOL.

2.2 Language Constructs

Writing a parallel program has three basic components: specifying the concurrency, the communication, and the synchronization between the concurrent activities. In COOL *concurrency* is expressed through parallel functions; *communication* between parallel functions is through shared variables; and the two basic elements of *synchronization*—mutual exclusion and event synchronization—are expressed through monitors and condition variables respectively. In addition, we also provide a construct to express fork-join style synchronization at the task level. We discuss each of these components in detail below.

2.2.1 Expressing Concurrency

The only way to specify concurrent execution in a COOL program is through the invocation of *parallel* functions. Both C++ class member functions and C-style functions can be declared parallel by prefixing the keyword `parallel` to the function header. An invocation of a parallel function creates a task to execute that function; this task executes asynchronously with other tasks, including the caller. In addition, the task executes in the same shared address space as all other tasks and can access the program variables like an ordinary sequential invocation. The COOL runtime system automatically manages the

creation and scheduling of these tasks. Since tasks are light-weight threads of execution, they can be dynamically created to express medium to fine-grained concurrency, making it easier to specify parallelism at a level that is natural to the problem.

It is useful to have certain invocations of a parallel function execute serially within the caller when the default of parallel execution results in unnecessary parallelism. For instance, in an implementation of mergesort, one may wish to invoke the sort on the left half of the array in parallel, while the sort on the right half is invoked serially. This is because the caller must wait for both halves to be sorted anyway before merging them. We therefore allow a parallel function to be invoked serially by specifying the keyword *serial* at the invocation. However, the serial specification must be used carefully: depending on the synchronization requirements in the program, executing a parallel function serially can sometimes lead to deadlock.

Since a parallel function executes asynchronously with the caller, what should be the semantics of returning a value from a parallel function? The solution adopted in Multilisp [46] is for a parallel function to return a *placeholder* for the return value. This placeholder remains unresolved until the parallel function completes, at which point it becomes resolved with the value returned by the parallel function. Non-strict operations on the placeholder (those that only need a *reference* to the value, rather than the value itself) execute without delay, while strict operations (those that need the actual value) automatically block until the parallel function has completed. We describe our experiments with language constructs to support returning general values from parallel functions later in Section 2.5.1. For now, we observe that fully-general futures incur high implementation overheads, whereas future-like solutions can result in complicated semantics, such as returning a value that may be modified by the calling task even while a parallel function is still executing to produce the return value.

In COOL, therefore, *all* parallel functions implicitly return a pointer to an event of type condH (described later in Section 2.2.3). When the parallel function is invoked an event is *automatically* allocated and a pointer to it returned *immediately* to the caller. To synchronize for the completion of the parallel function, the caller can store this pointer and continue execution, then later wait on this event for the parallel function to complete. The called parallel function *automatically* broadcasts a signal on this event upon completion, thereby resuming any waiting tasks.

Returning a pointer to an event is a simple mechanism to synchronize for the completion of parallel functions. Synchronization is automatically provided by the implementation within the caller, without requiring the programmer to modify the called function. This features does not allow parallel functions to return a value other than a pointer to an event. The results produced by a parallel function must be communicated through global variables and pointer arguments to the function. We illustrate this through an example in Section 2.3.2.

It is the programmer's responsibility to free the storage associated with the event returned by a parallel function once it is no longer required. The compiler performs simple optimizations such as (a) not allocating the event at all for invocations of parallel functions where the return value is ignored, and (b) deallocating the event once it is no

longer accessible in the program (e.g., exiting the scope of an automatic variable that stores the only pointer to the event). The compiler performs the optimizations safely, i.e., it ensures that an event is not deallocated by both the programmer and the compiler. Although the compiler can optimize simple situations, the ultimate responsibility for deallocating the event rests with the programmer.

2.2.2 Mutual Exclusion

The next key functionality needed is for parallel functions to communicate and to have controlled access to shared data. Just like ordinary functions in a C++ program, parallel functions in COOL communicate through variables in the shared address space. Controlled access to shared data is provided through monitors for mutual exclusion, and condition variables for event synchronization. We describe the mutual exclusion support in this section, and event synchronization in the following section.

With traditional monitors [55] synchronization for a shared object (an instance of a class) is expressed by specifying the class to be a monitor. All the operations on that class are assumed to modify the object and must acquire exclusive access to the object before executing. While COOL monitors provide the same basic synchronization as traditional monitors, they differ in the following three respects. First, monitor synchronization in COOL is specified at the granularity of individual operations rather than the entire class. Second, COOL supports recursive calls on the same monitor object. Finally, COOL allows the programmer to access a monitor object without synchronization. We elaborate on each of these differences below.

First, instead of specifying a *class* to be a monitor, synchronization for a shared object in COOL is expressed by specifying an attribute, either mutex or nonmutex, for the individual *member functions* of that class. A function so attributed must acquire appropriate access to the object while executing, thereby synchronizing multiple accesses to the object. A *mutex* function requires exclusive access to the object instance that it is invoked on; it is therefore assured that no other function (mutex or nonmutex) executes concurrently on the object. A *nonmutex* function does not require exclusive access to the object; it can therefore execute concurrently with other nonmutex functions on the same object but not with another mutex function. Typically, functions that modify the object are declared mutex while those that reference the object without modifying it are declared nonmutex, automatically providing the commonly used multiple reader (nonmutex) single writer (mutex) style synchronization for the shared object. Functions with neither attribute execute without synchronization like ordinary C++ functions; their behavior and usefulness is discussed later in this section.

The mutex and nonmutex attributes provide synchronization between functions on the *same* object, unaffected by functions on other instances of the class. Furthermore, since this synchronization is based on objects, C style functions cannot be declared mutex or nonmutex. A C++ class member function can have only one of the mutex and nonmutex attributes. However, these two attributes are orthogonal to the parallel attribute. A function that is both parallel and mutex/nonmutex executes asynchronously

when invoked *and* appropriately locks the object before executing. Finally, invocations of monitor operations can rely on not being *starved*. The implementation guarantees that a blocked monitor operation is not indefinitely overtaken by subsequent monitor operations, and is ultimately serviced.

The second difference between traditional and COOL monitors is that in a traditional monitor recursive monitor calls to the same object by a task result in deadlock, since the task is trying to *reacquire* an object. However, in COOL such recursive monitor calls proceed without attempting to reacquire the object, since the task has already satisfied the necessary synchronization constraints. Along with immediate recursion, this includes calls to other methods on the same object, as well as indirectly recursive calls that may go through methods on other objects before invoking a method on the initial object. Furthermore, if a task invokes a mutex function on an object on which it is already executing a nonmutex function, then it (a) waits for other executing nonmutex functions to complete, (b) acquires a mutex lock on the object and executes the mutex function, and (c) reverts back to the nonmutex lock upon completion. These semantics for monitor functions provide reasonable behavior while still maintaining the necessary synchronization requirements, and avoid obscure deadlock situations in perfectly reasonable programs. However, deadlock is still possible in a COOL program when two tasks attempt to acquire *multiple* objects in a different order. This is a common problem when acquiring multiple resources; we discuss solutions to it in Section 2.3.3.

Finally, COOL allows the programmer to bypass the synchronization on a monitor object. Often the programmer knows that certain references to a shared object can be performed safely without synchronization. For instance, a task may wish to simply examine the length of a queue object to decide whether to dequeue an element from this queue or to move on to another queue. Although unsynchronized accesses violate the synchronization abstraction offered by the mutex and nonmutex attributes, they avoid the synchronization overhead and serialization when they can be performed safely. We permit unsynchronized accesses in two ways. First, we do not *require* functions to have a mutex/nonmutex attribute. Those without an attribute can execute on the object without regard for other concurrently executing functions—mutex or nonmutex. Second, we maintain the C++ property that the public fields of an object may be accessed directly rather than through member functions alone, again independently of executing functions. When performed safely, these unsynchronized references avoid the synchronization overhead and enable additional concurrency. Although the burden for determining when these unsynchronized references can be performed safely is left entirely to the programmer, we believe that such compromises are frequently necessary for efficient execution. This is illustrated by several applications (e.g., Water, Barnes-Hut, Panel, and Block Cholesky) later in this chapter.

2.2.3 Event Synchronization

Waiting for an event is a common requirement of parallel programs. For instance, a consumer may need to wait for a value to be made available by a producer. In COOL

this synchronization is expressed through operations on condition variables. A condition variable is an instance of a predefined class *cond* in the language. The main operations on condition variables are wait, signal, and broadcast. A *wait* operation always blocks the current task until a subsequent signal. A *signal* wakes up a single task waiting on the event, if any. It has no effect otherwise. A *broadcast* wakes up all tasks waiting for that event, and is useful for synchronizations like barriers that resume all waiting tasks.

Within a mutex/nonmutex function, a task may need to wait for an event that is signaled by another function on the same object instance. This is expressed through the *release* operation on a condition variable. The release operation atomically blocks the function on an event and releases the object for other functions. Invoking a wait operation may lead to deadlock since the function has locked the object, preventing the signaling function from acquiring access to the object. When the event is signaled, the function resumes execution at the point that it left off after (a) waiting for the signaling function to complete, and (b) reacquiring the appropriate lock on the object. Thus the synchronization requirements of mutex and nonmutex functions are maintained.

The condition variables just described do not have history, i.e., a wait operation is resumed by subsequent signals only. In some situations, such as a wait operation on the event returned by a parallel function, the wait should obviously continue without blocking if the event had been signaled before the wait operation. Although this functionality can be implemented with the existing mechanisms in the language, for convenience we provide the class *condH* (cond+History) in which a signal operation is never lost; if a task is waiting then it is resumed, otherwise the signal is stored in the condition variable. A wait operation blocks only if there is no stored signal, otherwise it consumes a signal and continues without blocking. The broadcast operation is equivalent to infinitely many signals; all subsequent wait operations continue without blocking. We provide an *uncast* operation to reset the stored signals to zero, and a nonblocking *count* operation that returns the number of stored signals. The latter operation is useful while writing nondeterministic programs, that test an event and take different actions depending on whether the event has been signaled or not.

In traditional monitors condition variables are allowed only as private variables within a monitor. Their wait operation is like our release—it releases the monitor and blocks until a subsequent signal is performed. This behavior is desirable in traditional monitors since another thread can signal the waiting thread only by acquiring exclusive access to the monitor. In contrast, an event can also be accessed directly in COOL (i.e., outside of a monitor object), hence both the wait and the release operations are useful. These differences in condition variables between traditional monitors and COOL are not fundamental, however. For instance, an abstraction that behaves like an event and is globally accessible can easily be built using traditional monitors and condition variables. We simply provide global events as a built-in language feature in COOL for convenience, since condition variables are often useful outside a monitor for general event synchronization between tasks.

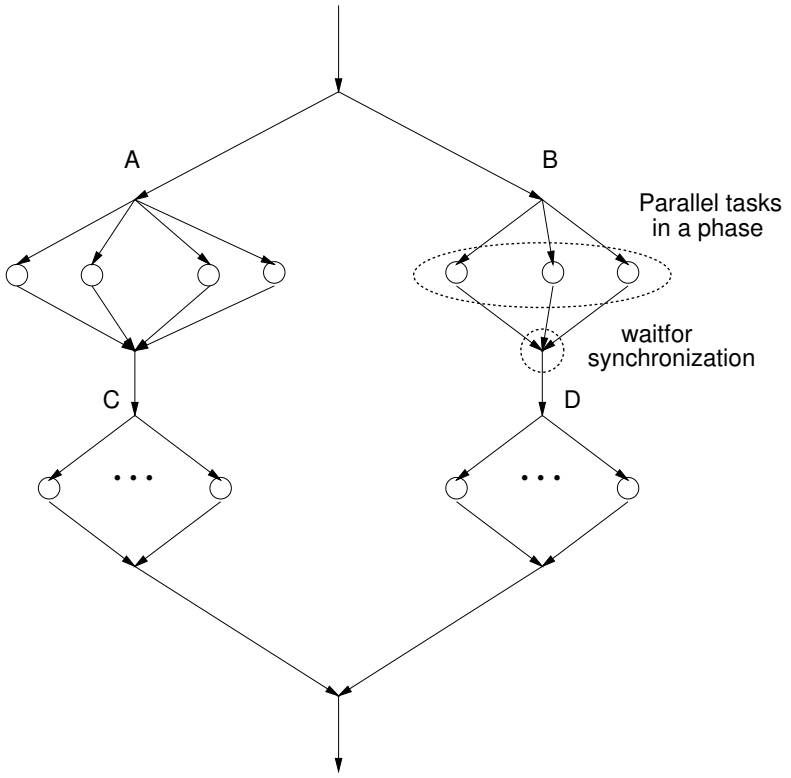

Figure 2.1: Task-level synchronization using the waitfor construct.

2.2.4 Task-Level Synchronization

Application programs that exhibit phase-structured computation—such as a simulation performed over several time steps—often exploit parallelism within each phase. Rather than synchronizing for each individual object to be produced by the different parallel tasks in the phase, it is often simpler and more efficient to wait for all the tasks in that phase to complete. In COOL, this task-level synchronization can be expressed using the *waitfor* construct, by attaching the keyword waitfor to a scope wrapped around the block of statements of that phase. The end of the scope causes the thread executing the statements in the scope to block until all tasks created within the scope of the waitfor have completed. This is precisely the dynamic call graph of all functions invoked within the scope of the waitfor, and it includes all parallel functions either invoked directly by the task, or invoked indirectly on its behalf by other functions.

In a process-oriented programming model such as the ANL macros [18], synchronization for the completion of such phase-structured computation is usually expressed using barriers. The waitfor construct in COOL offers greater flexibility compared to a barrier. For instance, consider the example in Figure 2.1 that shows two independent parallel phases A and B, followed by the phases C and D respectively. While a waitfor can express synchronization independently for each individual phase, the same is quite difficult using barriers. We can partition the available processors across the two phases and have two independent barriers at the end of each phase, but this does not allow a

Table 2.1: Summary of COOL constructs.

Purpose	*Construct*	*Syntax*
Concurrency	Parallel functions	**parallel condH*** egClass::foo ()
Mutual exclusion	Mutex and nonmutex functions	**mutex int** egClass::bar ()
Mutex with concurrency	Parallel mutex functions	**parallel mutex condH*** egClass::baz ()
Event synchronization	Condition variables	**condH** x; **cond** y;
	Wait for an event	x.**wait** ()
	Signal an event	x.**signal** ()
	Wait and release monitor	x.**release** ()
	Broadcast an event (**condH**)	x.**broadcast** ()
	Reset an event (**condH**)	x.**uncast** ()
	Number of stored signals (**condH**)	x.**count** ()
Task-level synchronization	Waitfor	**waitfor** { ... }

processor to service tasks from another phase. In contrast, the waitfor construct does not have any of these problems and offers greater flexibility.

2.2.5 Summary

Table 2.1 summarizes the COOL constructs. As we can see, COOL offers a small and simple set of constructs additional to C++: parallel functions to express concurrency, monitors and condition variables for mutual exclusion and event synchronization respectively, and the waitfor construct for expressing task-level synchronization. The constructs are designed to exploit the data abstraction features of the underlying language (in this case C++). Furthermore, the language makes deliberate allowances for programmers to leverage their knowledge of the code for additional expressiveness and/or performance benefits. For instance, we allow the programmer to bypass the abstraction provided by monitor operations and reference the public fields of a monitor object directly without synchronization (see Section 2.2.2). Such allowances are not desirable from safety concerns, but offer sufficient performance benefits to justify offering them in a controlled fashion.

2.3 Examples

We now illustrate the language through some example programs. We first present the basic concurrency and synchronization features of COOL with a particle-in-cell code (PSIM4 [81]). We then demonstrate the ability to build synchronization abstractions in COOL with some examples that perform object-level synchronization. The first example implements synchronization for multiple values produced by a parallel function (taken from the Water [96] application). The second example shows how to acquire exclusive access to multiple objects without causing deadlock (common in database applications). Besides these code fragments, we describe our experience with implementing several applications from the SPLASH [96] benchmark suite in COOL in the following section.

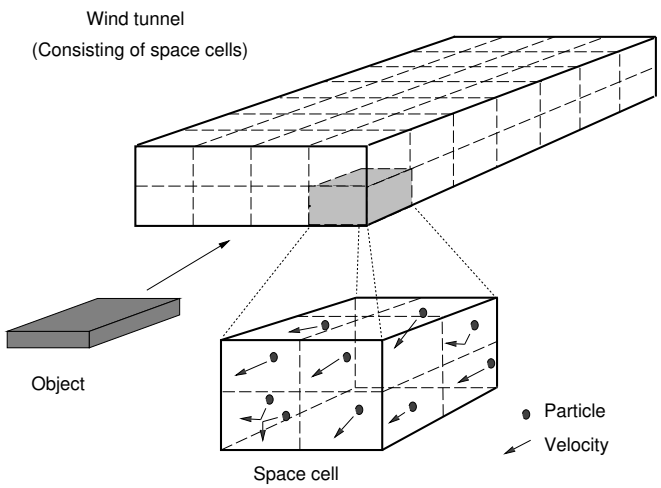

Figure 2.2: A wind tunnel in PSIM4, with particles contained in space cells.

2.3.1 Illustrating the Constructs

PSIM4[1] is a three dimensional particle-in-cell code that is used to study the pressure and temperature profiles created by an object flying through the upper atmosphere (see Figure 2.2). The program contains *particle* objects that represent air molecules and *space cell* objects that represent the wind tunnel, the flying object, and the boundary conditions. It evaluates the position, velocity, and other parameters of every particle over a sequence of time steps. The main computation in each simulated time step consists of a *move* phase and a *collide* phase. The move phase calculates a new position of the particle using the current position and velocity vectors, and moves the particles to their destination cell based on their new position. The collide phase models collisions among particles within the *same* space cell.

Concurrency is organized around space-cell objects in the parallel version of the algorithm, as shown in Figure 2.3. Within the move phase particles in different space cells can be moved concurrently, as expressed through the parallel *move* function on a space cell. In the collide phase, based on the semantics of the application, only particles in the same space cell may collide with each other, so collisions in different space cells can be modeled concurrently through the parallel *collide* function on a space cell.

During the course of the simulation, particles that move from one space cell to another are removed from their present space cell and added to the destination space cell, by invoking the *addParticle* function on the destination cell. Each cell maintains a list of incoming particles (*incomingPartList*) that provides the mutex function *add*; the *addParticle* function calls the *add* function to enqueue particles on this list. The incoming particles are incorporated with the other particles in the space cell in the beginning of the collide phase. While transferring the new particles, the incomingParticleList can be manipulated directly without synchronization since all cells have already completed their

[1] An improved version of the more widely known MP3D program from the SPLASH [96] benchmarks suite.

```
class ParticleList_c {
  . . .
public:
  mutex void add (Particle*);
};


class Cell_c {
  . . .
  ParticleList_c incomingParticleList;
public:
  . . .
  // Perform the move phase for particles in this cell.
  parallel void move () {
    . . .
    // If the particle 'p' has moved beyond a cell
    // boundary then add it to the cell 'dest'
    Cell[dest].addParticle (p);
  }

  // Model collisions for particles in this cell.
  parallel void collide () {
    // Transfer particles from the incomingParticleList
    // and add them to the list of particles in this cell.
    // incomingParticleList can be accessed without
    // synchronization.
    . . .
  }

  void addParticle (Particle* p) {
    incomingParticleList.add (p);
  }
} Cell[N];
```

```
main () {
  . . .
  while (timeStep > 0) {
    // For each time step do
    . . .
    waitfor {
      // Move particles in each cell concurrently.
      for (i=0; i<N; i++)
        Cell[i].move ();
    } // Wait until move phase completes.

    waitfor {
      // Model collisions in each cell concurrently.
      for (i=0; i<N; i++)
        Cell[i].collide ();
    } // Wait until collide phase completes.

    timeStep−−;
  }
}
```

Figure 2.3: Expressing concurrency and synchronization in PSIM4.

move phase. Finally, since each phase must complete before going on to the next phase, the synchronization is naturally expressed by wrapping the waitfor construct around each phase.

This example shows the basic COOL constructs in action within the context of a real application, a particle-in-cell simulator. This is all the COOL code required to express the concurrency and synchronization in this application, and includes parallel functions, monitor functions, and the waitfor construct. This example also illustrates the usefulness of 'attribute-less' functions on a monitor object, since accesses to the incomingParticleList must be serialized in the move phase, but can proceed without synchronization during the collide phase.

2.3.2 Object-Level Synchronization

The previous example showed how a straightforward application of the COOL constructs could be used to express an interesting application in parallel. We now use the Water code [96] to illustrate how the constructs can be used to build more complex synchronizations. This example also illustrates the benefits of C++: we use the data abstraction mechanisms in C++ to build an abstraction where the synchronization details are encapsulated within the object, providing a clean interface for its use.

The water code is an N-body molecular dynamics application that evaluates forces and potentials in a system of water molecules in the liquid state. Several molecules are processed together as a group (a task on an individual molecule is too fine-grained; this is discussed later in Section 2.4.1), with a parallel function *stats* on a molecule group to compute various internal statistics like the position, velocity, and potential energy of the molecules in that group. In this example we show how to synchronize for individual values produced by a parallel function as they become available, rather than waiting for the entire function to complete, thus enabling the caller to exploit additional concurrency.

In Figure 2.4 we show the synchronization for the variable *PotEnergy*, representing the potential energy of a molecule group. This example exploits the ability in C++ to define type conversion operators that are automatically invoked to construct values of the desired type. We define a class *double_s* that adds a condition variable for synchronization to a *double*. This class has two operators. The first is an assignment operator invoked when a value of type *double* is assigned to a variable of type *double_s*. The operator stores the value in the variable and broadcasts a signal on the condition variable. The second is a cast operator invoked when a value of type *double_s* is used where a *double* is expected. The operator waits for the event to be signaled and then returns the value. With these operators a *double* can be safely used in place of a *double_s* and vice versa.

The caller passes the address of *PotEnergy* to the *stats* function and continues execution. When it later references the value of *PotEnergy*, it invokes the cast operator which automatically blocks until the value is available. The pointer to *PotEnergy* can be passed to other functions without blocking, thus exploiting additional concurrency. The called function computes the potential energy and stores it in the supplied variable—this

```
class double_s {
  double value;
  condH x;
public:
  // Initialize the value to be unavailable.
  double_s () {
    x.uncast ();
  }

  // Operator to cast a double to a double_s.
  // Called when a double is assigned
  // to a variable of this type.
  void operator= (double v) {
    value = v;
    x.broadcast ();
  }

  // Operator to cast a double_s to a double.
  // Called when the value is referenced.
  operator double () {
    x.wait ();
    return value;
  }
};


// Function to compute internal statistics of the molecules
// in the group, like position, velocity, and potential energy.
parallel void moleculeGroup::stats (double_s* poteng, ...) {
  double subtotal;
  . . . Compute local potential energy into subtotal . . .
  *poteng = subtotal;
  . . . Continue - compute position, velocity etc . . .
}
```

```
main () {
  moleculeGroup mg;
  double totalEnergy = 0;
  double_s PotEnergy;
  // The value is initially unavailable.
  . . .
  // Parallel invocation.
  mg.stats (&PotEnergy, ... <other arguments> ...);

  . . . perform other useful computation . . .

  // Now the value of PotEnergy is required.
  // The reference to PotEnergy blocks if the
  // corresponding function has not completed.
  // e.g. Compute the total energy.
  totalEnergy += PotEnergy;
  // Continue after the value becomes available.
  . . .
}
```

Figure 2.4: Object-level synchronization in the Water code.

```
class sh_obj {
  objtype value;
  enum { FREE, BUSY } status;
  cond sync;
  . . .
public:
  mutex void claim () {
    if (status != FREE)
      // Release mutex and wait until the object
      // becomes available.
      sync.release ();
    // The object is now available; continue.
    status = BUSY;
  }

  mutex void surrender () {
    status = FREE;
    // Wake up a task waiting for this object, if any.
    sync.signal ();
  }
};
```

```
main () {
  sh_obj x, y;

  // Begin Transaction:
  // Acquire exclusive access to both objects.
  // To avoid deadlock, objects must be claimed
  // in the same order by all transactions.
  x.claim (); y.claim ();

  . . . Reference and modify the two objects . . .

  // End Transaction: Release the objects.
  x.surrender (); y.surrender ();
}
```

Figure 2.5: Acquiring access to multiple objects in a transaction based system.

assignment invokes the assignment operator which stores the value and signals all tasks waiting on the condition variable. The function *stats* continues execution to compute the other parameters.

This example builds an abstraction of a shared object with synchronization. This abstraction allows the caller (consumer) to execute asynchronously until it needs the value, and allows the called function (producer) to execute independently, making the various parameters available as they get computed. In addition, we use the ability to define operators in C++ to build a *double_s* abstraction; the synchronization details are encapsulated within the object, with the object being used like an ordinary double. This enables us to integrate synchronization as a property of the shared object, similar to 'future' variables in MultiLisp [46] (discussed later in Section 2.5.1).

2.3.3 Exclusive Access to Multiple Objects

Mutex and nonmutex functions protect access to a single object instance. However, there are situations where a program updates two (or more) objects and requires exclusive access to all of them at the same time. For instance, in a hash-table moving an object from one bucket to another requires exclusive access to both buckets (otherwise another

task may find none/duplicate copies of that object). Another example is a transaction based system in which updates to several objects must be performed atomically. This synchronization can be easily built by the programmer using mutex functions and condition variables, by explicitly maintaining the status of the objects and providing operations to acquire and release them (see Figure 2.5). A *claim* operation acquires the object if it is free and blocks otherwise. The *surrender* operation grants access to a waiting task, if any. Exclusive access to all the objects is obtained by claiming each individual object. However, all tasks must claim the objects in the same order to avoid deadlock—this can be ensured through macros or functions that claim the objects in order of increasing virtual addresses, for example. This simple protocol can be easily extended to support specific application requirements like operations to test if an object is available and either blocking or retrying immediately/later.

2.4 Experience with Applications

The previous sections have described how various common synchronizations can be expressed in COOL. For a more thorough evaluation of the language, we have rewritten several of the SPLASH [96] applications in COOL. The SPLASH suite is a collection of large scientific and engineering applications that are quite representative of important parallel applications. These applications have been explicitly hand-parallelized, and are therefore good candidates to evaluate both the programmability and the performance of the language.

All of the SPLASH applications were originally written in C (with the exception of Ocean, which is written in Fortran), and use the ANL macros [18] for concurrency. LocusRoute [89] is an application to route wires in a VLSI circuit. Panel Cholesky [91] performs Cholesky factorization of a sparse matrix using a Panel decomposition of a matrix. Block Cholesky [92] also performs Cholesky factorization of a sparse matrix, but uses a block decomposition of the matrix and therefore requires a different algorithm. Ocean [95] is an application that studies the influence of eddy and boundary currents on large-scale ocean currents. Water [96] and Barnes-Hut [93] are N-body applications—Water simulates various physical parameters in a system of water molecules, while Barnes-Hut uses a hierarchical algorithm to simulate the evolution of galactic systems.

We now provide a detailed description of our experience programming each of these applications in COOL. Our starting point for each application was the parallel version developed by the original authors using ANL macros [18], which we then coded in COOL. For each application we describe the basic structure of the parallel algorithm, and then describe how the desired concurrency and synchronization was expressed using the COOL constructs. Table 2.2 summarizes the object decomposition, and the concurrency and synchronization features of COOL employed in each application. Performance results of these applications running on the Stanford DASH multiprocessor [74] are presented later in Chapter 5.

Table 2.2: Summary of experience with SPLASH applications.

Program	*Object Structure*	*Concurrency (Parallel Function)*	*Synchronization*
Water	Molecules, Molecule Groups	Operate on a Molecule group	• waitfor tasks on molecule groups • synch for values from functions • serialize updates to shared values
Ocean	Grids, Regions	Process a Region	• waitfor region operations to complete
LocusRoute	Wires	Route a Wire	• waitfor all wires to be routed
Panel Cholesky	Panels	Update a Panel	• serialize updates to a panel • synch for source panel to be ready • waitfor updates to complete
Block Cholesky	Blocks	Update a Block	• serialize updates to a block • synch for source block to be ready • waitfor updates to complete
Barnes-Hut	Bodies, Space Cells	Process a Body	• serialize updates to space cells • waitfor tasks on all bodies

2.4.1 Water

Water [94, 96] is an N-body molecular dynamics application that models forces and potentials in a collection of water molecules over a period of time-steps. The application consists of about 2000 lines of code. The main data structures are various physical properties such as position, velocity, and potential/kinetic energy for each molecule in the system. These properties are organized into a molecule object, and there is an array of such molecule objects in the program. (The original SPLASH version of this application had a different organization of the data structures, consisting of a separate array for each different physical property.) The computation in each time step consists of several phases (see Figure 2.6), with each phase computing various properties of the molecules in the system.

```
class atom_c;
class mol_c {
  atom_c atom[3];
  . . . physical properties of the molecule . . .
public:
  . . . functions to process a molecule . . .
  void predic ();
  // Update molecule during interf phase
  mutex void interf_update ();
};

class molGroup_c {
  mol_c* mymols; // molecules within this group.
  . . .
  // Predict values of displacement and its derivatives.
  void predic ();
  // Calculate intra-molecular force/mass on each atom.
  void intraf (double_s*);
  // Divide the final forces by the mass.
  void divForce ();
  // Correct the predicted values based on the forces.
  void correc ();
  // Check boundary conditions on the position.
  void bndry ();
  // Compute kinetic energy in each dimension.
  void kineti ();
public:
  // Parallel functions to process molecules within a group.
  parallel condH* predic_intraf (double_s* poteng) {
    predic ();
    intraf (poteng);
  }
  parallel condH* interf ();
  parallel condH* divForce_correc_bndry_kineti () {
    divForce ();
    correc ();
    bndry ();
    kineti ();
  }
  . . .
} *molGroup;

main () {
  . . .
  for (all time steps do) {
    // Perform the different phases.
    waitfor {
      // process all the molecule groups in parallel.
      for (all molecule groups 'i')
        molGroup[i].predic_intraf ();
    }

    waitfor {
      for (all molecule groups 'i')
        molGroup[i].interf ();
    }

    waitfor {
      for (all molecule groups 'i')
        molGroup[i].divForce_correc_bndry_kineti ()
    }
  }
  . . .
}
```

Figure 2.6: Water code expressed in COOL.

2.4.1.1 Exploiting Concurrency

Since the values computed in one phase are typically used in the following phase, we cannot exploit concurrency across phases. Instead we exploit concurrency within each phase by processing the various molecules concurrently. However, the computation associated with processing a single molecule is too small to be usefully executed as a concurrent task. We therefore introduce the notion of a *moleculeGroup* or a collection of molecules. We exploit concurrency at the granularity of a group, expressed through parallel functions on a moleculeGroup object.

The size of a moleculeGroup (i.e., the number of molecules) must be chosen carefully to balance both granularity and load-balancing concerns. Since the computation is uniform and the same work is performed for each molecule, processing an equal number of molecules on each processor results in a good load distribution. We therefore have as many moleculeGroups as processors, each containing an equal portion of the molecules. This partitioning is done based on the runtime value of a variable that maintains the total number of processors employed in that particular execution of the program.

2.4.1.2 Expressing Synchronization

The first synchronization requirement is that the phases must be performed in sequence on each moleculeGroup. The processing of different moleculeGroups can in general proceed independently, with the exception of those phases that require communication between moleculeGroups (e.g., divForce). These phases can proceed only after the previous phase has finished on *all* moleculeGroups. These requirements are expressed by (a) calling a parallel function to perform each set of independent phases (e.g., predic followed by intraf, and divForce followed by correc then bndry then kineti), and (b) synchronizing for a set of phases (such as predic-intraf) to finish using the waitfor construct (see Figure 2.6).

The second requirement is the accumulation of some global parameters such as the potential energy of the entire system. Each moleculeGroup accumulates the potential energy of the molecules within that group into a local variable and then assigns it to the parameter poteng of type double_s (described earlier in Section 2.3.2). The main thread automatically synchronizes for these values to be computed as it accumulates them into the global potential energy. An initial version of the program in which the potential energy of each molecule was accumulated directly into the global value performed poorly due to severe contention; the local accumulation within each group does not require any synchronization, and performs much better.

The final requirement arises while modeling pairwise interaction between each pair of molecules in the system within the *interf* function. This function models the interaction of each molecule m within its group with exactly half of the other molecules in the system, starting from the molecule to the immediate right of m in a linear ordering of the molecules and wrapping around if necessary; this keeps the same interaction from being modeled twice. The interf function clearly references molecules in other moleculeGroups and must therefore acquire exclusive access to each molecule before updating it.

Since a mutex function operates on only one molecule at a time, this would ordinarily require the protocol for acquiring exclusive access to multiple objects simultaneously, described earlier in Section 2.3.3. However, in this particular instance it turns out that the values examined during a pairwise interactions (position and velocity) are entirely disjoint from the values that are updated based on the interaction (force and acceleration). Furthermore, the update is actually an increment, so that multiples updates to a molecule are commutative. The desired synchronization can therefore be implemented quite easily. Rather than acquiring exclusive access to both molecules together, the interf function acquires exclusive access to one molecule at a time, implemented by labeling the interf_update method on a molecule to be mutex. In addition, since the values being examined are not modified during this phase, we allow direct unsynchronized access to the internals of a molecule so that both interacting molecules can be examined together. Finally, although a molecule is now a monitor object, only the interf phase actually performs a synchronized update to the molecule. The other phases require no synchronization and reference/update the molecule through ordinary (i.e., non-monitor) functions.

2.4.1.3 Performance and Summary

Coding the application as described above was sufficient to exploit the desired parallelism in the program. In contrast to the COOL code described above, the original code written using ANL macros statically partitions the molecules equally across processors. Each processor operates on its assigned set of molecules, with barrier synchronization between phases and lock/unlock operations to enforce the mutual exclusion requirements.

In addition to exploiting parallelism, achieving high performance required some simple scheduling optimizations to improve data locality; these are described in detail in Chapter 5.2. We ran the program (with these optimizations) on an input of 512 molecules on a 32-node Stanford DASH multiprocessor [74]. The application performed nearly as well as the original code written using ANL macros, achieving a speedup of over a factor of 20 on 32 processors.

To summarize, the objects (molecules) in this application were too fine-grained to be processed concurrently, so we aggregated several molecules into a moleculeGroup object. Although this involved extra programmer effort, it was quite straightforward to express. Regarding the synchronization requirements, a waitfor was used to wait for a phase to complete; the data abstraction mechanisms were used in building the synchronized double object; and the flexibility of COOL monitors was helpful both in expressing the synchronization during the interf phase, and in bypassing it during the other phases to avoid unnecessary overhead. Finally, the object structure proved useful in several ways—the data structures were organized hierarchically and included atom, molecule, and moleculeGroup objects; concurrency was expressed by labeling methods on a moleculeGroup as parallel; synchronization was encapsulated within the double_s object; and the updates to a molecule were synchronized by labeling the update_interf method to be mutex.

2.4.2 Ocean

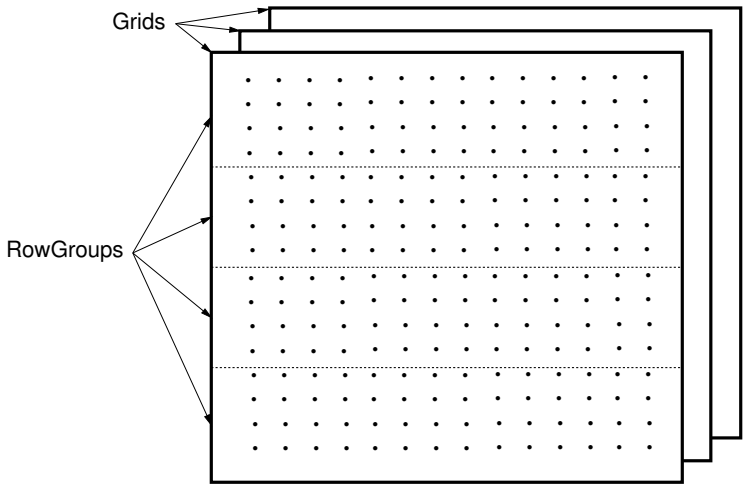

Figure 2.7: Grid structure in Ocean.

Ocean [95] is a program to study the influence of eddy and boundary currents on large-scale ocean movements. It simulates flows in a cuboidal basin over a series of time steps, solving a set of spatial partial differential equations within each step. The main data structures in the application are twenty-five double-precision floating-point grids. Each grid is a two-dimensional array, representing the value of a state variable at different locations in the ocean basin. The computation involves various regular grid operations in each time step (see Figure 2.8), including intra-grid operations such as nearest neighbor computation and inter-grid operations such as adding the corresponding elements of two grids. The application consists of about 3300 lines of code.

2.4.2.1 Exploiting Concurrency

We can exploit concurrency either *across* different operations that work on distinct grids or *within* an operation by processing different portions of a grid in parallel. The latter turns out to be much simpler for this application. We therefore exploit concurrency within each grid operation and wait for the entire grid to be updated before proceeding to the next operation. However, the work associated with an individual grid element is typically just a few arithmetic operations and is too small to perform concurrently. Therefore, similar to the Water code in the previous section, we aggregate several grid elements into a larger group that is processed concurrently with other groups. Each aggregate is simply a collection of rows or a *rowGroup*[2] (see Figure 2.7). Furthermore, since the computation involved is uniform across the various grid elements, each rowGroup is a simple partitioning of the rows across the processors, with as many rowGroups as

[2]For larger numbers of processors other partitionings such as a blocked decomposition of the grid may be more appropriate. However, each grid decomposition may well require a completely different organization of the data structures.

```
class row_c {
  double element[size];
};

class rowGroup_c {
  row_c myrow[numrows];
  . . . other data, such as row indices . . .
public:
  parallel condH* laplace (rowGroup_c*);
  parallel condH* sub (rowGroup_c*, rowGroup_c*);
  parallel condH* mult_add (int, rowGroup_c*, int, rowGroup_c*);
  parallel condH* jacobi (rowGroup_c*);
  . . .
};

class grid_c {
  rowGroup_c rowGroup[numgroups]; // grid composed of rowGroups
  . . .
public:
  void laplace (grid_c* p) {
    waitfor {
      for (all rowGroups 'i' do)
        // Process all the rowGroups in parallel.
        rowGroup[i].laplace (&(p->rowGroup[i]));
    } // wait for the operations to complete over all rowGroups.
  }
  . . . other grid operations . . .
};

main () {
  grid_c A, B, C, D;
  . . .
  for (all time steps) {
    A.laplace (B);
    C.jacobi (D);
    . . . other grid operations . . .
  }
}
```

Figure 2.8: Ocean code expressed in COOL.

processors. A grid, therefore, is a collection of *rowGroup* objects, a rowGroup is a collection of *row* objects, and a row contains the actual elements of one row of the grid in a one-dimensional array of elements. Parallel execution is expressed by invoking a parallel function on each rowGroup within the grid being updated (e.g., the function laplace in Figure 2.8).

As mentioned above, the grid operations include both inter-grid and intra-grid operations. An intra-grid operation only references the one grid that it is invoked upon, while inter-grid operations—such as adding two grids and storing the result in a third—must reference one or more additional grids along with modifying the grid that they are invoked upon. We allow the functions to directly access the internals of other grids in an inter-grid operation. While direct access to the elements within a grid object does compromise the modularity of objects in the program, it is useful when objects are shared at a fine-granularity, as in this application.

2.4.2.2 Expressing Synchronization

Synchronization for a grid operation to complete is expressed by wrapping the parallel rowGroup operations within a waitfor. This is the only synchronization requirement—the operations on individual grid elements are fully parallel and can proceed without synchronization. One grid operation that performs relaxed SOR (successive over relaxation) on a grid does contain nearest-neighbor communication between adjacent grid elements, but the algorithm is robust and converges even when these accesses are performed without synchronization.

2.4.2.3 Performance and Summary

We exploited concurrency in this fashion within each of the grid operations in the program in a fairly mechanical manner. The original code written using ANL macros statically partitioned the rows across processors in a similar fashion. Each processor operated on its assigned set of rows and synchronized using a barrier at the end of each phase.

Besides expressing the concurrency and synchronization, the program also required task and data distribution optimizations to achieve high performance. These optimizations are described in detail in Chapter 5.1. We ran the program on a 196x196 grid. The application performed well, achieving a speedup of over a factor of 22 on 32 processors. The ANL code was written in Fortran, and since the ANL Fortran macros had not been ported to DASH we could not compare the performance of the COOL program with the original code. However, we expect their performance to be similar.

To summarize, this application afforded concurrency down to the finest-granularity of an individual grid element. We aggregated several elements into a larger collection, a group of rows, that defined the grain of concurrent execution. The synchronization requirements were very simple and only required a waitfor around each grid operation. Finally, the data abstraction mechanisms of C++ were used to build a hierarchy of objects

```
struct {
  int xcost;
  int ycost;
} CostArray [XMAX][YMAX];

class Wire {
  . . .
public:
  // Functions to manipulate a wire.
  // Determine the cost of a route.
  int findCost ();
  // Parallel function to route a wire.
  parallel condH* Route ();
  . . .
};

main () {
  . . .
  while (not converged) {
    waitfor {
      for (all wires 'w' do) {
        // route the wires in parallel.
        w->Route ();
      }
    } // Wait for all wires to be routed.
  }
  . . .
}
```

Figure 2.9: LocusRoute expressed in COOL.

including elements, rows, rowGroups, and grids. We also exploited the flexible object structure in COOL to directly reference multiple objects during the inter-grid operations.

2.4.3 LocusRoute

LocusRoute [89] is a parallel algorithm for standard-cell placement and routing in integrated circuits. Given a circuit consisting of modules and connections, and a placement of the circuit modules, the program does *routing* to determine the paths of the connecting wires. The objective is to find a route that minimizes the area of the circuit. The program consists of about 9000 lines of code.

The two main data structures are *Wires* and the *CostArray*. A wire object contains the list of pin locations to be joined, where a pin location corresponds to a routing cell

in the circuit. The CostArray keeps track of the number of wires running through each routing cell and contains two values: the number of wires that pass horizontally and vertically through the routing cell. These values are updated as wires are routed. The cost of a route is given by the sum of the CostArray values for all the routing cells that it traverses. The program iteratively converges to a route for each wire. Within each iteration, LocusRoute invokes the function *Route* on each wire object (see Figure 2.9). This function rips out the previous route of a wire, generates the possible alternate routes, and chooses the lowest-cost route. After all wires have been routed in an iteration, this process is repeated in the next iteration.

2.4.3.1 Exploiting Concurrency

The primary source of parallelism in the program is to route different wires concurrently within an iteration.[3] In contrast to the previous two applications, exploiting wire parallelism is appropriate from both granularity and load-balancing viewpoints—routing a wire is a coarse-grained operation, and typical input circuits contain thousands of wires thereby generating sufficient parallelism. We therefore exploit wire parallelism in the COOL program, easily expressed by annotating the Route function to be parallel.

2.4.3.2 Expressing Synchronization

The only synchronization required in the application is that all wires be routed in an iteration before proceeding to the next iteration; this is expressed by wrapping a waitfor around an iteration. Although routing multiple wires concurrently may generate simultaneous updates to the CostArray, the algorithm is robust enough that it converges in spite of some inconsistencies in the CostArray. These inconsistencies are also present in the original SPLASH code.

2.4.3.3 Obtaining Private Per-Task Storage

In coding LocusRoute we encountered a problem that related to private per-task storage, and affected both programmability and performance. In the COOL programming model all the data is allocated from within the same shared-address space. In contrast, in programming models such as the ANL macros that use a 'fork' model of parallelism, all data is private by default unless explicitly allocated using a shared-malloc routine. In these latter programming models, if the task executes to completion on the same process without migrating to another process (as is the case in LocusRoute), then per-task storage can be allocated from within the private address space of each process. LocusRoute uses two different kinds of private scratch data structures while routing a wire—those that are dynamically allocated based on the wire parameters and others that are of fixed size and can be allocated statically. Both of these data structures can be allocated from within the

[3]There are secondary sources of parallelism as well [89], such as segment parallelism, route parallelism, and iteration parallelism. However, we focus on the primary source of parallelism, i.e., wire parallelism.

private space of a process when using the ANL macros; we now discuss how they are each handled in a COOL program.

The dynamically allocated data can continue to be implemented as before—'malloc'ed when a wire is selected, and 'free'd once the wire has been routed. Although allocating such data from the shared space is unnecessary, it is not erroneous. However, the initial COOL program with this scheme performed terribly! It turned out that memory allocation was a severe bottleneck and all the tasks were lined up waiting to execute malloc. The original ANL program did not encounter this problem because calls to malloc/free allocate data from the private address space of each process and can therefore execute concurrently. In the COOL program, however, these calls to malloc allocate data from the single shared address space and must execute serially.

A general solution to this problem is to write a distributed memory allocator, where calls to malloc/free from a processor continue to allocate data from within the shared address space, but from within some preallocated free space belonging to that particular processor, thereby avoiding contention. Occasional global communication may sometimes be necessary if either a processor runs out of free storage and has to go to the global allocator to allocate some more memory, or if a processor has too much free memory and wants to share it with other processors by (say) adding it to a global free pool. However, such global communication is likely to be infrequent. We did not implement this general solution, but instead implemented a distributed (i.e., per processor) free list for each data structure that was being heap-allocated during the routing of a wire. This scheme successfully avoided the contention encountered in the original version of the program. We have since implemented a distributed memory allocator in COOL.

The statically allocated data structures present a harder problem. In the original ANL program these data structures were declared as static variables, so that each process had its own private copy. However, since all the data in a COOL program—including global data—resides in the same address space, references by different processes to this scratch storage now resolve to the same location, which is an error.

There are two possible solutions to this problem. The first is an extension of the scheme used in the ANL macros of declaring the variables to be global, except that we now declare an *array* of such variables, one per processor. This array may be sized statically if the number of processes is known statically, otherwise it can be heap-allocated upon program startup. This solution, however, does require each process to have a unique identifier that can be used to index into this array for the private copy of that particular process. The problem is not over, though: how do we get a private copy of the identifier for each process—that was the problem we were originally trying to solve. The solution we use exploits an operating system feature of the SGI IRIX operating system (running on the Stanford DASH) that provides precisely one page in the shared virtual space that is mapped to a distinct physical page for each process. We use a location in this page to store a global variable *myid* that has the same virtual address for each process but resolves to a different physical location when referenced. References to this variable return the unique identifier of the process. Having this one private location per process is sufficient for our scheme—we can now use the value of myid to index into a global

array and thereby obtain per-process private data.

This solution does have some limitations, however. First, it provides private space on a *per-process* basis, not per-task. Therefore, it works fine if tasks execute to completion without blocking (as in LocusRoute), but if a task blocks leading to a context switch to (say) another wire-task then references to this scratch storage may conflict with the incomplete processing of the just suspended task, potentially leading to obscure programming errors. The second limitation is that references to static variables have now been replaced by references through a pointer to the array of such private storage, incurring the cost of an extra indirection at runtime. Depending on the usage of this scratch storage, this effect can potentially be a serious performance problem. However, the biggest drawback of this solution is that it violates the task-queue abstraction in COOL by exposing the underlying process-based implementation to the programmer.

An alternative to using arrays indexed by a process-id is to allocate the necessary storage elsewhere, such as on the stack of the Route function or within the Wire object. (In the latter case we would probably declare a pointer within the Wire object and allocate the storage dynamically when the wire was actually fetched for execution, rather than incur the storage overhead of the scratch data structures for all wires in the program.) This scheme does not have a problem with blocked tasks since the data is associated with the wire being routed, but may still incur the overhead of an extra indirection. However, the bigger problem with this scheme has to do with the accessibility of these data structures. In the previous schemes the scratch storage is global and can be accessed directly by all functions in the program. However, if the storage is allocated on the stack of the Route function then it must be explicitly passed to any function that needs access to these data structures. This is actually a severe problem in LocusRoute since this storage is used by tens of procedures and allocating it on the stack would require an extra parameter to all of these procedures, sometimes for the sole purpose of passing the pointer down to another procedure. Allocating the scratch storage within each Wire object (when it is fetched for routing) is another possibility, but this has the same problem that a Wire object is not globally accessible across the various procedures that need the storage, although to a much lesser extent than before. This solution would likely have been reasonable, although we ultimately used an array of global variables since that minimized the changes required to the original code.

This problem is not unique to COOL; obtaining private storage is an issue in any 'fully shared' programming model. Such programming environments should provide a private identifier such as the *myid* variable, since it cannot be improvised by the programmer in a fully shared environment in general. Having such a variable would allow the programmer to make the appropriate choice between the two solutions outlined above based on programmability. However, our initial experience suggests that although allocating per-task storage on the stack may sometimes be a little cumbersome, it is preferable to the process-based solution since the latter violates the task-based abstraction, makes programs difficult to understand, and can lead to obscure bugs. Regarding the performance overhead of the indirection, the reverse problem exists in 'fork' based programming models such as the ANL macros, but for *shared data*. Since all shared data is heap-allocated

through the shared-malloc call, it must be accessed indirectly through a pointer. For instance, the CostArray is a static data structure in COOL, but is heap-allocated through shared-malloc in the ANL macros.

2.4.3.4 Performance and Summary

The original ANL code exploits the same wire-parallelism by building explicit task queues and performing the task scheduling and execution functions within the user code. In contrast, this functionality is provided automatically by the COOL runtime system, and the programmer can simply label the Route method to be parallel.

After addressing the problems relating to per-task storage, we ran the application coded as described above, on an input circuit consisting of wires. Detailed performance results, as well as some task-scheduling optimizations that were performed to improve data locality, are presented in Chapter 5.3. The COOL version of LocusRoute performs better than the original ANL code, although the overall speedups (a factor of 12.5 on 28 processors) are low due to the high degree of communication.

To summarize, in contrast to the previous applications, parallelism at the granularity of wire objects was suitable from both granularity and load-balancing concerns. The additional COOL code required to express this application in parallel was very little: the Route function on a wire was annotated to be parallel, and each iteration was wrapped within a waitfor for synchronization. The object structure was useful with the primary computation organized around wire objects in the program. This application also illustrated the synchronization bottleneck of centralized memory allocation and the problem with obtaining private data structures in a shared programming environment.

2.4.4 Panel Cholesky

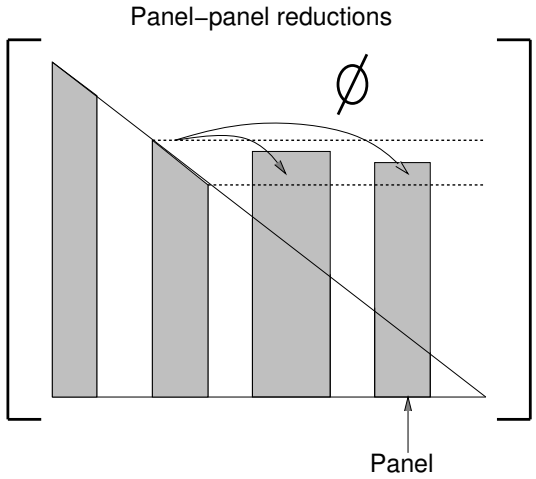

Figure 2.10: Panel reduction structure in Cholesky.

```
class panel_c {
  int remainingUpdates;
  . . .
public:
  // Update this panel by the given source panel.
  parallel mutex condH* updatePanel (panel_c* src) {
    . . .
    remainingUpdates−−;
    if (−−remainingUpdates == 0) {
      // This panel is now ready.
      completePanel ();
    }
  }

  // Perform internal completion of the panel.
  parallel condH* completePanel () {
    . . . perform internal completion . . .
    // Produce updates that use this panel.
    for (all panels 'p' modified by this panel)
      panel[p].updatePanel (this);
  }
} *panel;

main () {
  . . .
  waitfor {
    // Start with the panels that are initially ready.
    for (all panels 'p' that are initially ready)
      panel[p].completePanel ();
  } // Wait for all updates to complete.
}
```

Figure 2.11: Panel Cholesky expressed in COOL.

The next application, Panel Cholesky, performs parallel Cholesky factorization of a sparse positive definite matrix [91]. It consists of about 2500 lines of code. Given a sparse matrix A, the program finds a lower-triangular matrix L, such that $A = LL^T$. The primary operation in a column-oriented sparse Cholesky factorization is the addition of a multiple of one column of A into another column to its right in order to cancel a non-zero in the upper triangle. This operation is typically referred to as a column modification. Rothberg and Gupta [91] have suggested a matrix representation in which columns with identical non-zero structure are organized into *panels*, as shown in Figure 2.10. Operations are performed between panels—each panel has updates performed to it by relevant panels to its left, and once *all* the updates to a panel have been performed, the panel becomes 'ready' and can be used to update other panels to its right.

2.4.4.1 Exploiting Concurrency

Parallelism in this algorithm consists of performing the panel-panel updates concurrently. The partitioning of the matrix into panels tries to balance these granularity and load-balancing concerns by splitting large panels into smaller ones and aggregating small panels into larger ones. As a result the concurrency in this algorithm—that of panel-panel updates—is suitable from both a granularity and a load-balancing perspective. Typical inputs have tens of thousands of coarse-grained tasks.

The COOL code expressing this computation is shown in Figure 2.11. The main data structures in the program are panel objects, representing the partitioning of the matrix into panels. A panel object offers two methods of interest, updatePanel and completePanel. The *updatePanel* method is invoked on a destination panel—it updates the destination panel using the supplied source panel, and invokes completePanel if all updates have completed. The *completePanel* method performs internal completion on the panel and generates updates that use this panel as a source. Concurrent execution is expressed by labeling both these functions to be parallel. The computation is initiated in the *main* procedure by calling completePanel on those panels of the matrix that are initially ready.

2.4.4.2 Expressing Synchronization

The concurrent execution of multiple updates is subject to the synchronization constraints that (a) since an update modifies the destination panel, only one update can proceed on a destination panel at any time, and (b) all updates that are due to be performed on a panel must complete before that panel can itself be used to perform other updates. The overall computation finishes when all the updates have completed.

These synchronization requirements are expressed as follows. Multiple updates to a destination panel are serialized by simply annotating the updatePanel function to be mutex as well; it therefore has exclusive access to the panel being modified. (Updates to a destination panel are commutative and can therefore be performed in any order.) The second synchronization requirement, that all updates to a panel must complete before it can be used to perform other updates, is a little more complex. During the initial setup

phase, the algorithm counts the number of updates that must be performed on each panel and stores that number within each panel object. This number is decremented by each update operation. Once it reaches zero then all updates have been performed and the panel is ready. Although this is a specialized synchronization scheme that is explicitly constructed by the programmer, it is very efficient and easily coded in COOL. Finally, synchronization for the overall computation is expressed using the waitfor construct as shown.

Synchronization for a panel is necessary only when it is being updated; after all the updates have been performed, the panel can be read without any synchronization. For instance, the updatePanel method reads the values from the supplied source panel and updates the destination panel (the one it is invoked on). Since all updates to the source have already been performed, we allow the updatePanel method to directly reference the values within the source panel, thereby bypassing any synchronization. Similarly, the completePanel method is invoked after all updates have been performed and can execute without synchronization as well. These optimizations exploit the flexibility in COOL by enforcing synchronization for those panels being updated and bypassing the synchronization for the ready panels.

2.4.4.3 Other Performance Issues

This application exposed a potential problem in our implementation when we first ran it on large inputs that have a large number of updates performed to each panel. Recall that the updatePanel method is both parallel and mutex. As a result invocations of the function create a task, and the first thing this task does upon being dequeued is try to acquire exclusive access to the destination panel. If multiple tasks try to acquire access to the same destination panel then one of them will succeed while the others will block waiting to enter the monitor object, the destination panel. The underlying processes will then fetch other tasks, perhaps on the same panel, in which case these tasks will block as well, and the cycle continues. Apart from the overhead of suspending these tasks, the real problem is that suspended tasks consume precious stack resources (see Chapter 4). In this application the updatePanel function declares a large amount of scratch storage on its stack, and a stack size of 500KB is common for medium to large input matrices. Because of the large stack size, excessive numbers of suspended tasks result in several tens of mega-bytes of storage consumed just in stack resources, leading to severe thrashing and sometimes simply running out of swap space on the machine.

The problem in this application is the large numbers of suspended tasks consuming expensive stack resources. This problem cannot be completely solved in general, since an application could always be constructed to precisely contrive this situation. However, the problem can be alleviated to a large extent. The COOL implementation optimizes the support for parallel monitor functions by delaying the allocation of the stack until *after* the task has acquired access to the monitor object that it is invoked on (see Chapter 4.2.2.6 for details). This 'fix' solved the problem for Panel Cholesky, and the problem has not arisen in any other application. However, this solution will not suffice for applications

that invoke a synchronization operation later within a task. For those applications the programmer may just have to restructure the application to avoid this problem. We may also consider providing additional support in the language, such as allowing the programmer to identify the synchronization operations that occur within a task and can be acquired before the task begins execution.

2.4.4.4 Summary

Having optimized the implementation of parallel mutex functions, we coded the program as described above. Compared to the COOL code, the original program written using ANL macros built an explicit task-queue structure within the application, where each task corresponded to a panel-panel update. The synchronization requirements were built using locks/unlocks to serialize multiple updates, by maintaining explicit counts of the remaining updates for each panel, and by using a barrier to wait for the computation to finish. In contrast, the concurrency and synchronization were relatively easier to express in COOL by annotating methods on panels to be parallel or mutex.

To achieve good performance the application required both task and data distribution optimizations that are described later in Chapter 5.5. We used the matrix BCSSTK33 from the Boeing-Harwell set of sparse matrix benchmarks [32] as an input. This matrix has 8738 columns organized into 1201 panels with approximately 7.28 columns per panel. With the locality optimizations the application performed nearly as well as the original ANL code (that also performed similar optimizations), achieving a speedup of a factor of 14 on 24 processors.

To summarize, compared to the previous applications, Panel Cholesky had relatively complex concurrency and synchronization requirements. All the important computation was organized around panel objects—concurrent execution was expressed by labeling methods on panels to be parallel, while mutual exclusion for a panel was expressed by labeling the update method to be mutex. Our implementation of waiting for a panel to become ready illustrated how the programmer can build application-specific synchronizations in COOL. Finally, we exploited the flexibility of COOL monitors to synchronize accesses to a panel during the update phase and bypass the synchronization once a panel became ready.

2.4.5 Block Cholesky

The Block Cholesky code [92] performs the same computation as the Panel code, namely Cholesky factorization of sparse matrices. However, the Block code attempts to address some performance limitations of the Panel code, such as high communication and limited scalability, through a different matrix decomposition strategy (the details of these limitations are discussed in [92]). Rather than representing the matrix as a set of columns (which are then organized into panels), the matrix is represented as a set of rectangular *blocks* (see Figure 2.12). The basic operation is to update a destination block by two source blocks, rather than updating an entire panel by another panel, and affords

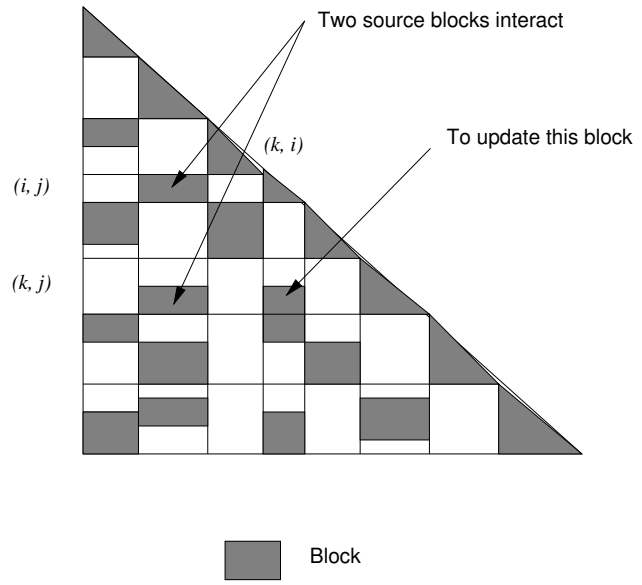

Figure 2.12: A matrix partitioned into blocks.

greater concurrency. As shown in the figure, blocks (i, j) and (k, j) (in the same column j, $i < k$) interact to reduce the destination block in the position (k, i). Similar to the Panel code, a block is updated by blocks to its left until all modifications to it have been performed. Thereafter the block become ready and is used to update other blocks to its right. The application consists of about 5500 lines of code.

2.4.5.1 Exploiting Concurrency

The COOL version of Block Cholesky is shown in Figure 2.13. Similar to the Panel code, multiple updates can execute concurrently in Block Cholesky as well. The main data structures in the program are block objects that make up the sparse matrix. The block class offers two methods of interest—the *completeBlock* method first performs internal completion of that block and then updates each block that is modified by this block by invoking the function updateBlock. The *updateBlock* method updates the destination block that it is invoked on, using the two source blocks supplied as parameters to the method. The main program sets the computation rolling by invoking completeBlock on all blocks that are initially ready.

2.4.5.2 Expressing Synchronization

The synchronization requirements in the application are (a) only one update on a destination block can execute at any time, (b) all updates to the source blocks must complete before they can be used to update other blocks, and (c) the overall computation completes only when all updates have been performed.

Multiple updates to a destination block are serialized by declaring the updateBlock

```
class block_c {
  . . .
  int remainingUpdates;
  condH ready;
public:
  // Update this block by the given source blocks
  parallel mutex condH* updateBlock (block_c* src1, block_c* src2) {
    . . . update the block . . .
    if (--remainingUpdates == 0)
      completeBlock ();
  }

  // Perform internal completion of a block
  parallel condH* completeBlock () {
    . . . perform internal completion . . .
    // I am now ready. Signal others that may be waiting.
    ready.broadcast ();
    // Generate updates that use this block.
    for (all blocks 'b' above 'this' in this column) {
      // Wait for that block to become ready.
      block[b].ready.wait ();
      dest = block updated by 'this' and 'b';
      block[dest].updateBlock (this, block+b);
    }
  }
} *block;

main () {
  . . .
  waitfor {
    for (all blocks 'b' that are initially ready)
      block[b].completeBlock ();
  }
  . . .
}
```

Figure 2.13: Block Cholesky expressed in COOL.

function to be mutex. The update operation is commutative, hence simple mutual exclusion is sufficient. The completeBlock method is invoked only when all other updates to the block have been performed and can therefore execute without synchronization. Furthermore, synchronization for a block is necessary only when the block is being updated, and references to the source blocks can proceed without synchronization as well.

Synchronization for a block to become ready is implemented in the following fashion. Each block maintains a count of the number of updates remaining to be performed (*remainingUpdates*) before the block becomes ready. Once a block becomes ready, it generates updates only with those blocks (i,j) that are *above* it within the same column j of the matrix; this avoids generating duplicate updates, one from each of the two source blocks required by the update. However, with this scheme an update may be generated before the other source block is ready. Waiting for the other source block to become ready is implemented using a condition variable within each block. This condition variable is broadcast once all the updates have been performed to that block. A source block (k,j) performs a wait operation for the other source block (i,j) to become ready before generating the update, thereby ensuring that both source blocks can now be used to perform the update. Finally, synchronization for the overall computation is expressed using the waitfor construct.

Waiting for the other source block requires a wait operation in the middle of the completeBlock function. As we saw in the previous application, synchronizing in the middle of a task can lead to large numbers of blocked tasks during the execution of the program and consume precious storage resources. In Panel Cholesky we had addressed this problem by optimizing the implementation of parallel mutex functions, but no such optimizations are possible here since the wait operation occurs in the middle of a parallel function. However, although we do get several blocked tasks in this application, each task requires very little stack storage to execute (less than 1 KB); therefore the application executes with hundreds (or even thousands) of blocked tasks without any problems.

2.4.5.3 Performance and Summary

The COOL program exploits the same concurrency as the original code using ANL macros. However, the ANL code builds explicit task-queues within the application, whereas in COOL we simply label methods on a block to be parallel. Furthermore, synchronization for the two source blocks is much more easily expressed in COOL—the ANL code maintains a separate data structure to identify the blocks that have become ready so far and continually examines this data structure to determine those updates whose source blocks are ready and can therefore proceed. In contrast, in COOL this synchronization is expressed by performing a wait on the condition variable within each block.

As the results in Chapter 5.6 show, the application as coded above performs better than the original ANL code (after both have been optimized for good locality and load-balance), achieving a speedup of a factor of 18 on 28 processors with dense matrices (input DENSE1000 of size 1000x1000), and 12 on 24 processors with sparse matrices

(input BCSSTK15 with 3948 columns organized into 3661 blocks).

To summarize, the basic structure of Block Cholesky application is similar to the Panel code discussed previously, although the synchronization requirements are a little more complex. Concurrency was expressed by annotating block methods to be parallel, and mutual exclusion was expressed by annotating the update method to be mutex. Synchronization for a block to become ready required a condition variable within each block. Furthermore, accesses to a block were synchronized while it was being updated but could proceed directly once the block became ready. These various synchronization requirements were significantly more complex in the original ANL code. Overall, the object structure proved useful, with both concurrency and synchronization organized around operations on block objects.

2.4.6 Barnes-Hut

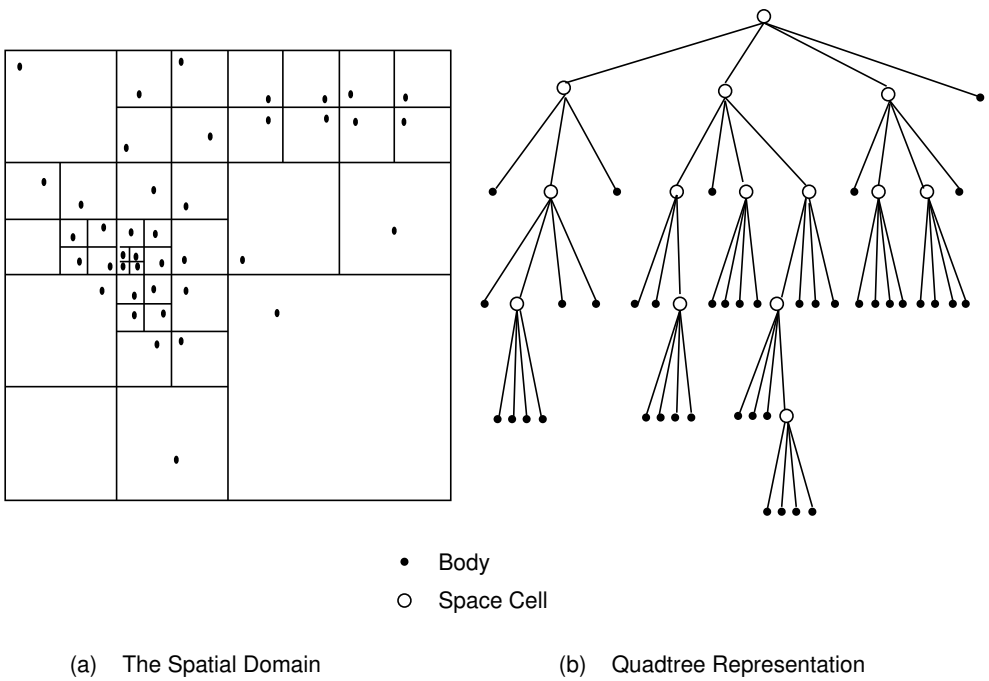

Figure 2.14: Physical space with bodies, and the corresponding tree representation (Figure courtesy of J. P. Singh).

Barnes-Hut [93] is a 3000 line N-body program that simulates the evolution of a system of bodies under the influence of gravitational forces. All bodies in the system are modeled as point masses and exert force on the other bodies. Rather than modeling the interaction between all possible pairs of bodies ($O(n^2)$), the application uses a more efficient hierarchical algorithm ($O(nlogn)$) due to Barnes-Hut, in which groups of distant bodies are instead approximated by their center of mass. Consequently, the interactions of a body with nearby bodies are modeled individually, while interactions with distant groups of bodies are modeled as an interaction with the center of mass of that group.

```
// Space cell object.
class cell_c {
   . . . pointers to child nodes in the tree . . .
public:
   mutex void addBody (body_c*);
   parallel condH* cofm ();
   void completecofm ();
   . . .
} *cell;

// Body object.
class body_c {
   . . .
public:
   parallel condH* addtoTree ();
   parallel condH* computeInteraction ();
   parallel condH* update ();
} *body;

main () {
   . . .
   for (all time steps) {
      // Build the tree of space cells.
      waitfor {
         // Process each body in parallel
         for (all bodies 'i')
            body[i].addtoTree ();
      }
      // Compute the center of mass of each space cell
      waitfor {
         // Determine subtrees of space-tree to process in parallel
         for (subtrees of the space cells 'i')
            cell[i].cofm ();
      }
      // Compute the center of mass of the remaining cells serially.
      for (remaining cells)
         cell[i].completecofm ();
      // Compute the N-body interactions.
      waitfor {
         for (all bodies 'i')
            body[i].computeInteraction ();
      }
      // Update the values in each body.
      waitfor {
         for (all bodies 'i')
            body[i].update ();
      }
   }
}
```


Figure 2.15: Barnes-Hut expressed in COOL.

The program is based on a hierarchical tree representation of space (see Figure 2.14). Physical space is recursively subdivided into smaller subcells until there is exactly one body per cell. A space cell corresponds to a node in the tree, and its subcells correspond to the child nodes in the tree. The root node of the tree corresponds to the entire space, while the leaves are the bodies themselves. The simulation is carried out over a period of time-steps, with several phases in each time step. The tree representation is rebuilt in the first phase, the center of mass computations are performed for each space cell in the second phase, the force computation is done in the third phase, and it is used to update the position and velocity of each body in the fourth phase.

2.4.6.1 Expressing Concurrency and Synchronization

In the parallel version of this application we exploit concurrency within each phase and wait for that phase to complete before moving on to the next phase (see Figure 2.15). We describe the concurrency and synchronization within each phase in turn.

Building the Tree: In the tree-building phase we start with a single root node representing the entire (empty) space and add the bodies by calling the function *addToTree* on each body. This function traverses the tree to locate the space cell node containing that body and calls addBody to add the current body to that cell. If the cell is empty, then the body is directly added to that cell, otherwise addBody creates four children in the space cell (representing a partitioning of physical space in two-dimensions) and adds the body to the appropriate child cell.

Parallelism in this phase consists of processing multiple bodies concurrently, expressed by labeling the *addtoTree* function to be parallel. Since adding a body modifies the space cell, we serialize multiple updates by declaring the addBody method (that actually inserts the body into a space cell) to be a mutex function. The tree traversals within addtoTree can proceed without synchronization since they only read the space cells; even nonmutex synchronization is unnecessary since space cells once divided do not change any further.

The ANL version of this code partitions the set of bodies across processors, with each processor working on its assigned set of bodies. Since the work involved in adding each body is sufficiently coarse, the COOL version of the code exploits concurrency at the granularity of an individual body through the parallel addtoTree function.

Computing Center of Mass: The next phase computes the center of mass of each space cell in the tree by starting from the bodies at the leaves and computing the center of mass of each space cell in a bottom-up fashion. The order of computation is important since the center of mass of a space cell is computed using the center of mass of each of its child cells.

In the ANL version of the code, each processor is said to *own* the space cells that it created in the previous tree-building phase, and parallelism consists of each processor

computing the center of mass of the cells that it owns. The COOL program, on the other hand, does not have the notion of owned cells and instead exploits concurrency by conceptually partitioning the tree into several subtrees that are processed concurrently. This is expressed by labeling the *cofm* (center of mass computation) method to be parallel. We use a waitfor to wait for all subtrees to be processed, and then (serially) compute the center of mass for the remaining space cells at the higher levels of the tree. Overall, however, this phase takes very little execution time, and different parallel schemes offer only marginal gains compared to just processing this phase serially.

Force Computation: The third phase performs the main processing, the force computation based on the interaction between bodies. The ANL version of the program exploits parallelism similar to the first phase in which the bodies are partitioned across processors. The COOL program instead exploits parallelism at the granularity of individual bodies by labeling the function *computeInteraction* on a body to be parallel. This function traverses the cells in the space tree to compute the interaction of this body with other bodies in the system. These traversals can proceed without synchronization since the tree is no longer modified during this phase. Furthermore, computeInteraction can update the body object without synchronization as well since each body is processed just once in each phase, and no other processor could be modifying the body concurrently.

Update Values: Finally, the fourth phase updates values within each body based on the force computation in the previous phase. It is similar to the third phase and parallelism in this phase is again exploited across bodies. No synchronization is required during this phase as well.

2.4.6.2 Summary

We coded the application as described above and ran it on an input of 32K bodies over two time-steps. The application performed nearly as well as the original ANL code and achieved a speedup of a factor of 28 on 32 processors.

Overall, while the Barnes-Hut application had complex data structures compared to the previous applications (bodies and space cells organized into a tree), the concurrency and synchronization requirements were quite simple and expressed by annotating methods on bodies and cells to be parallel (for concurrency) or mutex (for mutual exclusion). We also exploited the flexibility in COOL to bypass the synchronization on space cell monitor objects in the later phases of the program. Finally, the object structure helped in organizing all the important computation around cell and body objects, so that parallelism could be easily exploited based on those objects as described above.

2.4.7 Discussion

As this detailed look at several applications written in COOL has shown, the language has worked well in exploiting parallelism within these applications. We now review the effectiveness of the language constructs in expressing concurrency and synchronization and evaluate the benefits of exploiting the object structure.

2.4.7.1 Exploiting Concurrency

The task-based model of concurrency in COOL, provided through parallel functions, mapped directly onto the structure of several applications. This included routing wires concurrently in LocusRoute, updating a panel or a block in Panel/Block Cholesky, or processing a body in Barnes-Hut. Expressing concurrent execution in these applications was a simple matter of annotating a method to be parallel. The original versions of these applications using ANL macros built explicit task-queues within the program to exploit the same concurrency. In the Water and Ocean applications, granularity and load-balancing concerns required that several elements be aggregated together to form a larger unit of concurrency. While explicit programmer control was necessary, the flexibility of the language constructs enabled us to easily build these collections of elements. Furthermore, using the data abstraction mechanisms in C++, we expressed these collections as concurrent abstractions that encapsulated the parallelism within the object.

2.4.7.2 Expressing Synchronization

Amongst the synchronization constructs, the waitfor proved to be a simple way of waiting for a parallel phase to complete and was heavily used in all the applications. The monitor mechanism was most often used to serialize multiple updates to shared objects in the program, such as molecules in the Water code, panels/blocks in Panel/Block Cholesky, and space cells in Barnes-Hut. In each of these applications this synchronization was easily expressed by annotating the appropriate method on the shared object to be mutex. Furthermore, the flexibility of the monitor mechanism in COOL that allowed us to bypass the synchronization on a monitor object (either through direct access or through ordinary, non-monitor methods) proved extremely useful. Several applications, including Water, Panel, Block, and Barnes-Hut used this facility to synchronize accesses to the shared objects only during the phases in which the objects were being modified and bypassed the synchronization entirely in other phases where either the objects were read without being modified (e.g., Panel, Block, Barnes-Hut) or if some other synchronization ensured that there are no conflicting accesses (e.g., Water). None of the applications used nonmutex functions, instead preferring to bypass the synchronization entirely. Condition variables were used in building synchronization abstractions, such as the synchronizing double in the Water code, or the barrier abstraction in a version of the Ocean code. Finally, the data abstraction features in C++ helped us in building shared object abstractions that encapsulate the details of synchronization within the object.

One drawback of providing synchronization based on monitor objects is that object-based systems provide exclusive access to one object at a time (see Section 2.3.3). Therefore, task that naively try to acquire exclusive access to two different monitor objects may deadlock. We encountered this problem in the Water code while processing the pairwise interaction between each pair of molecules. Although a simpler solution sufficed in the Water application, other applications may well need the more general solution outlined in Section 2.3.3.

2.4.7.3 The Object-Based Approach

In all of these applications the computation has been organized around the main objects in the program. This has significantly eased the job of exploiting parallelism—for instance, expressing concurrency has often been as easy as annotating an object method to be parallel (e.g., routing a wire in LocusRoute, or updating a panel/block in Panel/Block Cholesky), while synchronization was usually expressed simply by declaring a method that modified a shared object to be mutex (e.g., updating a panel/block, modifying a molecule in Water). Furthermore, the data abstraction facilities of C++ have enabled us to hide the details of concurrency/synchronization within an object. We can thereby build concurrent abstractions such as rowGroups in Ocean or moleculeGroups in Water, or synchronization abstractions such as the synchronized double in Water.

On the other hand, there were situations when the 'natural' objects in the application did not represent the appropriate concurrency/synchronization grain in the program. For instance, the objects in the Water code (a molecule) and Ocean (a single grid element) were simply too fine-grained (alternatively, an entire grid object in Ocean was too coarse-grained) to be processed concurrently; we therefore used a collection of these elements as the unit of concurrency.

Another limitation of object-based systems arose while expressing fine-grained communication between objects. In object-based systems methods operate on a single object at a time. Furthermore, it is desirable (and required in many languages) that an object be referenced through access methods only. This model works fine if the computation can be expressed through coarse-grained operations on individual objects, but it is not suitable when objects are shared at fine granularities. In all the applications that we considered, there were situations when a particular piece of computation required intimate access to multiple objects. For instance, pairwise interaction between molecules in the Water code required access to two molecule objects, inter-grid operations in the Ocean code require access to several grid objects, routing a wire in LocusRoute required access to both the wire object and the CostArray, updating a panel or a block in Cholesky required access to both the destination and the source panel/block, and finally, processing the interaction of a body with other bodies in Barnes-Hut required access to both the space cells in the tree as well as other bodies in the system. If all accesses are required to be through interface operations, each access to a value within a second object must instead be replaced by a method invocation on that object, which can be very cumbersome.

```
(defun func (a, b)
  (. . .))

(let (a ...) (b ...)
  (setq x (future (func a b)))      ; parallel invocation of func
  (setq y x)                        ; simple assignment, non-strict access. do not block
  (foo x)                           ; parameter passing, non-strict access. do not block
  (+ (car x) 2))                    ; strict-access — block until the value is available
```

Figure 2.16: Illustrating futures in Multilisp.

Several aspects of the object structure in COOL have greatly simplified the programming task in the above situations. Similar to C++, COOL allows us to use objects where they help in structuring the program and to revert to ordinary C style data structures when there is no particular benefit from using objects. Furthermore, C++ is not as strict as some of the other object-oriented languages and provides mechanisms for the programmer to directly access an object from within a method on a different object. Although this violates the modularity of objects and places the burden for correct usage on the programmer, we have found that such compromises are often necessary from both programming ease and performance considerations.

2.5 Alternate Design Choices

In our design of the language, there were primarily two places where we struggled with alternative semantics. The first concerned returning values from parallel functions and synchronization for their completion. The second issue was the design of the waitfor construct. We discuss each of these below.

2.5.1 Return Value of Parallel Functions

Parallel functions in COOL are restricted to return a pointer to an event, and any values must be communicated through parameters or global variables. While designing the language, we considered two alternate schemes to support future-like [46] synchronization for the return value of a parallel function. However, we discarded these schemes because they complicated the semantics of the language in several situations. In this section we first review the future mechanism and then describe our two schemes.

The future construct in Multilisp [46] can be attached to a function invocation signifying that the function should be executed concurrently (see Figure 2.16). The parallel

invocation returns a *future object* to the caller as a placeholder for the return value. Strict operations on a future object—those that need the value—automatically block until the parallel function has completed, while non-strict operations—those that only need a reference to the object, such as passing a pointer—execute without delay. Futures are particularly attractive because synchronization for the return value is *entirely transparent* to the programmer—an invocation of a parallel function and access to its return value look syntactically the same as for serial functions.

This transparency comes at a cost, however, and requires a runtime check of the status of an object at each reference. These checks are much easier to implement in Multilisp because of the extreme simplicity of data types—everything is either a primitive element or a list of elements, and all functions return a pointer to a list. Providing the same level of transparency in a language like C++ with its rich variety of primitive and user-defined types is both complex and incurs high overheads. In the designs that we consider below, therefore, we expose the synchronization to the programmer in controlled ways so that synchronization can be provided only when necessary.

2.5.1.1 A Variation of the Current Design

We considered a variation of the current COOL design that allows parallel functions to return a value of any arbitrary type, and handles the invocation of a parallel function as follows. If the caller invokes the parallel function parFn through a simple assignment such as $x = parFn()$ (or if the return value is ignored), then parFn executes asynchronously with the caller. The caller continues execution after creating a task to evaluate parFn. Furthermore, the assignment statement itself, $x = parFn()$, is now executed by the *parallel task* executing parFn upon its completion, rather than being executed by the caller. If, on the other hand, the caller invokes the parallel function as part of an expression that requires the return value before it can proceed—i.e., a strict access such as $x = parFn() + 31$—then parFn is executed *serially* within the caller's context. This behavior is appropriate since the return value of parFn is required before the caller can proceed.

With this behavior, although the parallel function parFn stores the return value in the appropriate variable upon completion, the caller cannot determine when the parFn has completed. We now describe how the programmer can return a value with synchronization using this base semantics (see Figure 2.17). The programmer first defines a new *synchronizing type* (similar to the synchronizing double in Section 2.3.2) that extends the return value with a condition variable and provides operators to cast both to and from the base type along with appropriate synchronization. Second, the caller invokes parFn with the return value being assigned to a variable *rv* of the synchronizing type (*int_s* in the figure) rather than a variable of the base return type. As a result the caller continues execution after creating a task to execute parFn. Upon a subsequent strict access to *rv*, the caller automatically invokes the cast operator and blocks until the parallel function completes. The called function *parFn*, meanwhile, executes to completion and then executes the assignment *rv=return value*. This assignment invokes the assignment operator that stores the return value and signals a waiting thread (e.g., the caller), if any.

```
parallel int parFn () {
   int retval;
   return retval;
}

class int_s {
   int val;
   condH x;
public:
   // Operator that casts an int to an int_s.
   // Automatically called when the function completes,
   // and the assignment is executed.
   void operator= (int v) {
      val = v;
      x.signal ();
   }
   // Operator that casts an int_s to an int.
   // Called to return the value once it becomes available.
   operator int () {
      x.wait ();
      return val;
   }
};

main () {
   int_s rv;
   int x;
   rv = parFn ();
   . . . continue without waiting for function to complete . . .
   x = rv + 31; // Block until the value becomes available, then continue.
}
```

Figure 2.17: Synchronization for the return value of a parallel function.

This scheme allows the programmer to implement synchronization for the return value of a parallel function on top of the simple base semantics provided in the language. A parallel invocation now looks syntactically similar to a serial invocation, and the caller can refer to the return value as a value of the base type, with the synchronization being encapsulated within the object. Another advantage of this scheme is that synchronization for the return value can be implemented entirely within the caller's context without requiring any modification to parallel function, which continues to return a value of the base type.

Although this feature is useful when used in the stylized fashion described above, it presents problems in other situations. A potential problem occurs when the parallel function executes the assignment after the caller has completed and deallocated its stack, while the variable on the left-hand side of the assignment had been allocated within the caller's stack and is no longer accessible. However, a more serious problem is that while this design provides the illusion of a serial function call, the variable (on the left-hand side of the assignment) remains accessible and can be manipulated by the caller (and perhaps other threads) even while the parallel function is executing. This can lead to unexpected/unnatural behavior when two threads try to assign a value to the same variable. For instance, the caller could invoke a parallel function (storing its return value in rv) and subsequently decide to store a different value in rv through direct assignment. If the assignment is executed before the parallel function completes, then the parallel function will, upon completion, assign its return value to rv. This would unexpectedly destroy the value that had been stored in rv *subsequent* to the invocation of the parallel function. Several scenarios that exhibit similar non-intuitive behavior can be easily constructed. Since returning values from a parallel function can be easily implemented (with synchronization) through function parameters (see Section 2.3.2), we decided that the additional complexity of this feature was not worthwhile.

2.5.1.2 Future Types in old COOL

An earlier version of the language (we shall call it old COOL) described in [24] offered explicit *future* data types in the language. A future in old COOL was a type attribute that added synchronization when attached to a base type. An instance of a future type (a future variable) behaved like a value of the base type except with additional synchronization. The difference was that a future variable could be *unresolved*, signifying that its value was currently not available, but would be available later. An access to an unresolved future variable automatically blocked until the value was *determined*. A common way of using future variables was as placeholders for the return value of a parallel function. These placeholders could be passed around to other functions, synchronizing only when the return value was actually required, i.e., upon a *strict access*.

Futures in old COOL supported two kinds of synchronization. First, they provided synchronization for the return value of parallel functions in the caller—a future variable became unresolved when assigned to the return value of a parallel function, and was resolved with the return value when the function completes. However, in addition,

Task T1	**Task T2**
qx = parFn ();	
// parFn is a parallel function	
// Therefore qx becomes reserved	
. . .	
z = qx;	
// Strict access: block until parFn completes.	
	qx = 2;
	// What should this assignment do
	// if parFn has not completed yet?

Figure 2.18: Potential problems with future data types.

futures in oldCOOL provided operations to explicitly resolve and unresolve the future object. These operations enabled futures to be used for general event synchronization, and helped separate synchronization granularity from task granularity.

A future type could be in one of three possible states—*set* or resolved with a value, *reserved* or unresolved waiting for the return value from a parallel function, and *clear* or unresolved by an explicit clear operation and therefore not associated with the return value of any parallel function. The semantics of some common operations on future types were as follows. (We use names starting with a 'q' for future variables.) An assignment such as $qx = 3$ automatically resolved qx with the given value and resumed any blocked threads waiting for qx to be resolved. An invocation of a parallel function, $qx = parFn()$, made qx reserved awaiting the return value of the parallel function. A *strict* access such as $z = qx$ blocked the thread until qx was resolved and continued with the value thereupon. An assignment of one future variable to another, $qx = qy$, had the following behavior: if qy was resolved then qx became resolved with the value of qy; if qy was reserved then qx became reserved awaiting the return value of the same function as qy; and if qy was clear then qx became clear as well. Finally, operations to explicitly resolve/unresolve a variable (i.e., *set, clear, waitset*, and *waitclear*) were useful when implementing producer-consumer style synchronization between two (or more) concurrent tasks. The producer resolved a future variable with the produced value, while the consumer cleared the future variable upon consuming a value.

The various operations on future variables described above enabled the programmer to express different kinds of synchronizations in a program, such as returning values from a parallel function as well as more general producer-consumer style synchronization between tasks. However, we ran into several problems with future variables. The main problem was providing reasonable behavior in situations when a future that was unresolved awaiting the return value of a parallel function was subsequently modified before the parallel function could complete. For instance, consider the code fragment shown in Figure 2.18. With the assignment $qx = 2$ in task T2, qx itself will be resolved with the

value 2, irrespective of whether parFn has completed or not. However, what should be the effect of this assignment on the strict access, $z = qx$, that is suspended waiting for parFn to complete? We could wake up task T1, in which case execution could proceed past the assignment even though parFn was still executing. Or, we could let task T1 wait for the return value from parFn even though qx gets resolved. The problem then is that the behavior of $z = qx$ is different depending on whether qx is reserved or clear. In the former case it is unaffected by subsequent assignments to qx, while in the latter case it is made resolved with subsequent assignments to qx.

Now consider that task T2 had executed the statement $qx = qy$ rather than $qx = 2$. If qy is resolved then the situation is similar to the previous case. But if qy is clear, then should qx become clear, or should it continue to be reserved awaiting the value from parFn (note that both states are unresolved). And, if qy is reserved waiting for a parallel function parFn2, then qx should clearly await the return value of parFn2 as well. But, should the suspended strict-access $z = qx$ continue to await the value of parFn, or should it switch to waiting for the value from parFn2?

In this fashion we can construct several situations where the desired behavior is ambiguous and non-intuitive. Although these are corner cases where the programmer is using future variables in a decidedly muddled fashion—such as performing multiple unsynchronized assignments to a future variable—we felt that it was important to have sound and simple principles that could be used to determine the behavior of future variables in all situations. We found that providing these various kinds of functionality within future variables opens up loopholes in the language that make it all too easy to write bad programs that are difficult to reason about.

Besides these situations where the semantics of future variables gets complex, the other drawback of future variables is that supporting all of the necessary functionality can become quite expensive, both in terms of the storage required as well as the runtime overhead of the various operations. These overheads become particularly intolerable in programs that use futures in a simple fashion, yet have to pay for the more general (unused) functionality. For instance, a future variable that was only used for return value synchronization had to also support the possibility of being explicitly unresolved through the clear operation. Or, a future variable that was only being used for event synchronization continued to incur the overhead associated with supporting the unused facility for return value synchronization.

Based on the experiences outlined above, we came to the conclusion that we were overloading futures and trying to provide too much functionality all in one construct. We used futures for pure event synchronization as well as for integrating synchronization with shared data. In addition, we used futures for return value synchronization from parallel functions as well as for general event synchronization between tasks. In trying to simplify the constructs, we converged on the simple events or condition variables provided in COOL and described earlier in this chapter. The new design has several advantages. The most important benefit is that condition variables are simple constructs with all the operations having very straight-forward semantics. Furthermore, although condition variables themselves provide pure event synchronization only, they are easily

coupled with data abstraction mechanisms in C++ to build shared object abstractions that provide future-like synchronization, with the synchronization details encapsulated within the implementation of the object (e.g., the synchronizing double abstraction described in Section 2.3.2). Finally, because of their simple semantics, condition variables can be implemented very efficiently, and their implementation can be further optimized by the compiler in several situations that are outlined in Chapter 4. Although COOL no longer supports future style synchronization for the return value of parallel functions, values can be easily obtained through an extra argument to the function as shown in Section 2.3.2.

2.5.2 Waitfor

The waitfor construct is useful to wait for a phase of a program to complete and must be wrapped around the block of code representing the program phase; it is therefore syntactically restricted to be within the lexical scope of a function. However, if the code for the phase spans parts of two functions (e.g., the last half of one function and the first half of another) then a waitfor cannot be used, since the waitfor must obviously be entirely within a function.

We considered an alternate *fence* construct that can be invoked as a function call anywhere within the program, with the semantics that the task T invoking the fence blocks until the completion of all previous tasks created by T, either directly or indirectly. Therefore a fence is not identified with a static region of code, but instead waits for all the code dynamically executed within the context of that task to complete. This addresses the syntactic restriction in the waitfor construct.

However, we chose not to include this construct, primarily because supporting a fence operation adds a constant overhead to all task creation and completion operations. Since the invocation of a parallel function may at any time be followed by a fence, the runtime system must keep track of all tasks that have been created by a task at all points. In contrast, the waitfor construct explicitly identifies the code that must be synchronized for, allowing the runtime to only keep count of the tasks created within that scope. In addition, a fence can sometimes be tricky to use, leading to obscure bugs. For instance, a function that calls the fence operation may be invoked from multiple sites, most of which use the fence synchronization, but other invocations may end up executing the fence inadvertently, perhaps with disastrous results. The problem arises because the fence operation is not tied to the scope of a function, compared to the waitfor operation where the phase of the program being synchronized for is clearly identified. Finally, in our experience, synchronization in most programs could be easily expressed with the waitfor construct and did not require a fence.

2.6 Summary

The emphasis in the design of COOL has been to choose a simple and compact set of constructs that can be implemented efficiently. As we have shown, the simple constructs

chosen in COOL work very well in exploiting task-level parallelism over a variety of parallel applications. The requirements of most applications map directly onto the constructs in COOL. For other applications, although the programmer has to explicitly build additional concurrency and synchronization abstractions, these are easily expressed using the basic constructs.

The features in COOL are designed to exploit the object structure of the underlying C++ program. This object-based approach of COOL works particularly well when the computation is organized around the primary data structures in the program; in such applications both concurrency and synchronization are orchestrated around the main objects in the program. However, we have also found that being bound to a strict object structure can sometimes be quite restrictive. In such situations the flexibility to bypass the modularity of the object structure is often necessary to improve both programming ease as well as efficiency.

Finally, in this chapter we have explored the expressiveness of the COOL constructs in exploiting parallel execution. COOL also provides support in the language to optimize the placement of tasks and data and thereby improve data locality in the program's execution. This support is described in the following chapter.

Chapter 3

Data Locality and Load Balancing

Exposing the concurrency and the accompanying synchronization doesn't automatically ensure great speedups; obtaining good performance on a multiprocessor requires (a) good load balance so that processors do not sit idle for lack of available work and (b) good data locality so that individual processors do not waste time waiting for memory references to complete. The latter problem is particularly severe on modern multiprocessors where, with rapid improvements in processor speeds, the memory system is increasingly the limiting factor in the performance of computer systems. The latency of memory accesses can be especially high in large-scale shared memory multiprocessors which typically have deep memory hierarchies. For example, in the Stanford DASH multiprocessor [74], while the local cache access takes only a single clock cycle, a miss serviced by the local portion of shared memory takes about thirty clock cycles, and a remote miss takes over a hundred clock cycles. Improving data locality in parallel programs can reduce the time spent by a processor waiting for data and is critical for achieving good performance. Therefore, any parallel programming system must address these performance issues.

Data locality in parallel programs can be improved by scheduling computation and distributing data structures with an awareness of the underlying memory hierarchy, so that tasks execute close to the objects they reference. Several automatic techniques have been explored in the literature—for instance, the operating system can perform optimizations such as affinity scheduling for better cache reuse [23, 44, 76, 80, 105] or automatic page migration for better memory locality [16, 23, 71, 72]. A compiler can perform optimizations such as scheduling the iterations of a loop for cache locality or distributing individual elements of an array for memory locality [7, 45, 65, 77, 107, 108, 112]. These automatic techniques are usually limited in their effectiveness since determining an appropriate distribution of tasks and objects requires knowledge about the program that is usually not available to the operating system or the compiler. This application specific information is often readily known to the programmer. However, the programming support for the programmer to explicitly perform these optimizations

is very primitive. In most parallel programming systems today, these optimizations must be performed in a highly machine specific fashion—they are hard-wired to the particular memory hierarchy on a multiprocessor and use the runtime libraries provided on that machine. This effectively limits the portability of the parallel applications.

COOL provides explicit support in the language to address these performance issues [25]. In addition to support for expressing parallelism, COOL provides abstractions for the programmer to supply hints about the data objects referenced by parallel tasks. These hints are used only by the runtime system to appropriately schedule tasks and migrate data, and thereby exploit locality in the memory hierarchy. Furthermore, as hints these abstractions do not affect the semantics of the program. This approach provides a clear separation of functionality: the programmer can focus on exploiting parallelism and supplying hints about the object reference patterns, leaving the details of task creation and management to the implementation. Furthermore, this approach enhances the portability of COOL programs since the machine-specific portion is entirely encapsulated within the implementation.

In this chapter we first review the characteristics of the memory hierarchies found in modern multiprocessor systems. We outline our approach towards improving data locality, and present the abstractions provided in COOL to support programmer intervention, including the constructs for specifying the distribution of objects across processors' memories. We describe the scheduling optimizations performed by the runtime system, but defer a detailed discussion of the implementation to Chapter 4. We also postpone evaluation of the effectiveness of our approach through example programs to Chapter 5.

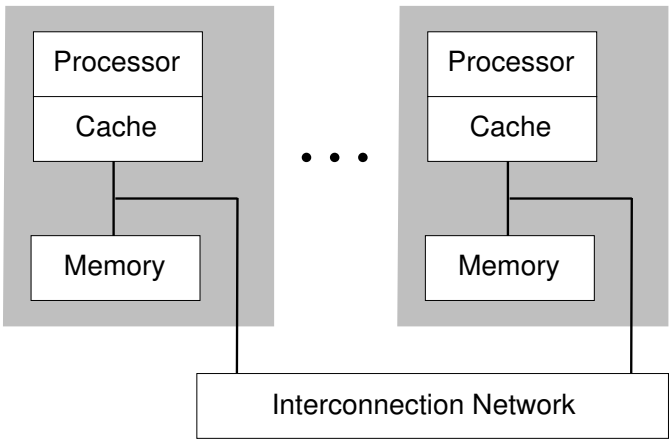

Figure 3.1: Multiprocessor memory hierarchy.

3.1 Exploiting the Memory Hierarchy

In this thesis we assume a three-level memory hierarchy (see Figure 3.1) consisting of (a) per processor caches, (b) local portion of shared memory, and (c) remote shared memory. We believe that this is a reasonable abstraction for many large scale multiprocessors, such as the Stanford DASH [74], MIT Alewife [1], and the KSR-1 [64]. In these multiprocessors the ratio of the latencies of local to remote references is usually much more significant than variations in the latencies to different remote processing elements. The memory hierarchy shown in Figure 3.1, therefore, captures the essential components of the memory hierarchy for such multiprocessors, while still allowing for a high-degree of machine independence.

For these deep memory hierarchies, the primary mechanisms to improve data locality in a parallel application are *task scheduling* and *object distribution*. The latency of references to an object can be reduced by scheduling computation and distributing objects so that a task executes on a processor that is close (in the memory hierarchy) to the objects referenced by the task. However, scheduling tasks to improve data locality often conflicts with the simultaneous goal of scheduling to achieve good load balance. Therefore, it is important to weigh the tradeoff between locality and load balance when determining a task schedule.

Given a specific task decomposition for an application, we can exploit two forms of locality—cache locality and memory locality. We can exploit *cache locality* (i.e., reuse in the cache) by scheduling tasks that reference the same objects on the same processor. Cache locality can be further enhanced by scheduling these tasks *back to back* to reduce possible cache interference caused by the intervening execution of other unrelated tasks. When cache locality alone is insufficient for good performance, we can exploit *memory locality* by identifying the primary object(s) referenced by a task and executing the task on the processor that contains the object(s) in its local memory. Thus, references to the object that miss in the cache will be serviced in local rather than remote memory, resulting in lower latency. However, ensuring a good load balance while exploiting

memory locality requires both task scheduling and an appropriate distribution of objects across the local memories of processors. Finally, the impact of long-latency memory references can be further reduced by dynamically migrating the object or by *prefetching* the object closer to the processor.

3.2 Our Approach

Determining an efficient task and object distribution requires knowledge about the program that is often beyond the scope of a compiler but may be readily known to the programmer. In COOL we therefore provide abstractions for the programmer to supply hints about the data objects referenced by parallel tasks. These hints are used by the runtime system to appropriately schedule tasks and thereby exploit locality in the memory hierarchy; they do not affect the semantics of the program. Since our approach involves programmer participation, it is important that the abstractions be intuitive and easy to use. At the same time the abstractions should be powerful and general enough to exploit locality at each level in the memory hierarchy, without compromising performance. We address these goals through the following key elements of our design. The abstractions are integrated with the task and object structure of the underlying COOL program so that the hints are easily supplied. Next, the abstractions are structured as a hierarchy of optional hints so that simple optimizations are easily obtained as defaults while more complex ones require incremental amounts of programmer effort. Finally, the hints for a task are dynamically evaluated each time a task is created, enabling the programmer to easily experiment with different optimizations.

The *hierarchy* of control provided in COOL ranges from smart defaults to simple hints to very specific hints. Smart default scheduling policies provide the basic optimizations without any programmer effort. When programmer participation becomes necessary, we have abstractions for providing simple hints about object usage. These hints are sufficient for most programs. For programs that require more complex optimizations, we have abstractions that provide greater control over task scheduling and data placement. These latter abstractions require a greater degree of understanding of the application and the memory hierarchy, but are simple extensions of the simpler abstractions and therefore require incremental amounts of additional programmer effort.

The programmer-supplied hints only influence the scheduling of tasks. In addition, we also provide support in the language for explicit data distribution. COOL provides constructs for allocating an object from a particular processor's memory and for dynamically migrating an object from one processor's memory to another. Ongoing compiler [7, 45, 65, 77, 107] and operating systems [17, 73] research has enjoyed some success in automatically distributing objects and could reduce this burden on the programmer.

3.3 The Abstractions

```
class column_c {
  . . .
public:
  . . .
  // Parallel function to update 'this' column
  // by the given 'src' column.
  parallel mutex void update (column_c* src)
    // Code specifying affinity hints. The runtime executes
    // this code whenever the function is invoked, to determine
    // how to schedule the corresponding task.
    [
      // By default the task has affinity for the column
      // being reduced ('this').
      affinity (this);

      // To instead express affinity for the column 'src'
      // being used to reduce the column.
      affinity (src);
    ];
} *column;

main () {
  int i, j;
  . . .
  for (i=0; i<N; i++) {
    for (j=0; j<i; j++) {
      // Invocations of the parallel function.
      // The corresponding tasks are scheduled
      // based on the specified affinity hints (or default).
      column[i].update (column+j);
    }
  }
}
```

Figure 3.2: Illustrating the affinity hints.

In COOL, information about the program is provided by identifying the objects that are important for locality for a task. To do so, along with a parallel function the programmer can specify a block of code that contains *affinity hints*. Figure 3.2 illustrates how this block of code may be supplied for a parallel function; the individual affinity hints are discussed later in this section. This block of code is executed when the parallel function is invoked and a corresponding task is created. The affinity hints themselves are simply evaluated by the runtime system to determine their effect on the scheduling of the task; they do not affect the semantics of the program (this can be automatically enforced by the compiler). We present the hierarchy of the affinity abstractions in this section.

3.3.1 Defaults

Parallel functions in COOL have a natural association with the object that they are invoked on. So by default, tasks created by the invocations of a parallel function are scheduled on the processor that contains the corresponding object in its local memory. (Tasks created by invocations of parallel C style functions—as opposed to C++ member functions—are scheduled on the server that created them). The task is therefore likely to reference the object in local rather than remote memory. In addition, if there are several tasks that operate on that object, then only the first task will need to fetch the object from memory; the rest are likely to find the object in the cache. The runtime system executes such tasks *back-to-back* on that processor to reduce the cache interference caused by the execution of unrelated intervening tasks. This further improves cache locality.

The following examples illustrate this default heuristic. Figure 3.2 presents the code to perform column-oriented Gaussian elimination of a matrix, in which the parallel update method is invoked on a destination column to reduce it using a given source column. Based on the heuristic described above, the update task is scheduled to exploit both cache and memory locality on the destination column. Considering the Panel and Block Cholesky applications described earlier in Chapter 2, the default heuristic schedules the updatePanel/updateBlock tasks to exploit locality on the destination panel/block respectively (see Figures 2.11 and 2.13). As we see later in Chapter 5, scheduling based on the default heuristic alone often leads to substantial performance improvements.

3.3.2 Simple Affinity

When a parallel function would benefit from locality on an object other than the base object, the programmer can override the default by explicitly identifying that object through an *affinity* specification for the function. When affinity for an object is explicitly identified, the runtime system schedules the task in a manner similar to the default described above, except that the scheduling is based on the specified object rather than the default object. As a result, the object is likely to be referenced in the cache, and references to the object that miss in the cache are serviced in local rather than remote memory. For instance, as shown in the example in Figure 3.2, the programmer can

supply an affinity hint to override the default affinity for the destination column and exploit locality on the supplied source column instead.

3.3.3 Task and Object Affinity

We have shown how to exploit each of cache and memory locality by simply specifying the objects referenced by the parallel function. It is often useful to *simultaneously* exploit memory locality on one object and cache locality on a different object.

We illustrate this need using an algorithm that performs column-oriented Gaussian elimination on a matrix. In the algorithm, each column of the matrix is updated by the columns to its left to zero out the entries above the diagonal element. Once the column has received all such updates (i.e., all entries above the diagonal element are zero) it is used to update other columns to its right in the matrix. A task in this algorithm is an invocation of the parallel function *update* (see Figure 3.3) that updates a destination column by a given source column. A desirable execution schedule and object distribution for this column-oriented algorithm are as follows (see Rothberg [91]). The algorithm executes an update on the processor where the destination column is allocated, thereby exploiting memory locality on the destination column (the number of columns per processor is so large that they are not expected to fit in the cache). Since tasks need locality on the destination column, distributing the columns across the local memories of processors in a round-robin fashion results in good load distribution. In addition, each processor executes multiple updates that involve the same source column; by executing these updates in a back-to-back manner the algorithm can exploit cache locality on the source column as well, by avoiding the execution of other tasks that might flush the source column from the cache.

The above example clearly shows that we wish to exploit cache locality on the source column and memory locality on the destination column. We therefore allow the keywords *OBJECT* and *TASK* to be specified with an affinity statement. The *TASK* affinity statement identifies tasks that reference a common object as a *task-affinity* set; these tasks are executed back-to-back to increase cache reuse. The *OBJECT* affinity statement identifies the object for memory locality; the task is collocated with the object. As shown in Figure 3.3, we can simultaneously exploit cache locality through task affinity on the source column, as well as memory locality through object affinity on the destination column. This exactly captures the way the algorithm was hand-coded using ANL macros [18] to run on the Stanford DASH multiprocessor; the same scheduling is very simply expressed in COOL.

3.3.4 Processor Affinity

Finally, for load balancing reasons it sometimes becomes necessary to *directly* schedule a task on a particular processor (in practice the corresponding server process), rather than indirectly through the objects it references. We therefore provide *processor affinity*

```
class column_c {
  . . .
public:
  . . .
  // Parallel function to update 'this' column by the given source column.
  parallel mutex void update (column_c* src)
    [
      // Object affinity for 'this' exploits
      // memory locality on the destination column.
      affinity (this, OBJECT);

      // Task affinity for 'src' exploits
      // cache locality on the source column.
      affinity (src, TASK);
    ];
};
```

Figure 3.3: COOL code illustrating TASK and OBJECT affinity in Gaussian elimination.

through the *PROCESSOR* keyword that can be specified for an affinity declaration. An integer argument is supplied instead of an object address, and its value (modulo the number of server processes) is used as the server number on which the task is scheduled.

3.3.5 Multiple Affinity Hints

If affinity is specified for multiple objects then the runtime system simply schedules the task based on the first object. It is the programmer's responsibility to supply appropriate affinity hints for tasks that would benefit from locality on multiple objects. For instance, to exploit memory locality on multiple objects, the programmer can allocate all the objects from the memory of one processor and supply an object affinity hint for any of those objects. To exploit cache locality on multiple objects, the programmer can pick one of the several objects and supply a task affinity hint for that particular object consistently across the various tasks. Greater experience with using these affinity hints may suggest heuristics to automatically make an intelligent scheduling decision when multiple affinity hints are supplied, such as determining the relative importance of objects based on their size and scheduling the task on the processor that has the *most* objects in its local memory.

3.3.6 Task Stealing

While the runtime system schedules tasks based on the affinity hints, tasks may be stolen by an idle processor for better load-balance. While this is desirable in most applications,

Table 3.1: Summary of affinity specifications.

Affinity Construct	*Description*	*Effect*
`affinity` (*obj-addr*)	Simple affinity	Collocate with object, schedule back-to-back
`affinity` (*obj-addr*, `OBJECT`)	Object affinity	Collocate with object
`affinity` (*obj-addr*, `TASK`)	Task affinity	Schedule tasks in task-affinity set back-to-back
`affinity` (*num*, `PROCESSOR`)	Processor Affinity	Schedule on specified processor

there are codes where a good load-distribution can be determined statically. In such applications it is often useful for the programmer to control load-balancing explicitly by scheduling tasks directly onto processors (using processor affinity, for example) and disabling automatic task stealing. We therefore allow task stealing to be explicitly disabled by exposing a COOL runtime flag (called *noStealing*) to the programmer. This flag can be toggled freely—a non-zero value disables task stealing in the entire COOL runtime, while a zero value (default) keeps task-stealing enabled. We see instances where this support is useful in the following chapter on case studies.

3.3.7 Summary of Affinity Hints

The various affinity hints are summarized in Table 3.1. They are used by the runtime scheduler hints to schedule tasks as described above. In addition to scheduling tasks for good locality, the scheduler employs several heuristics to maintain good load balancing at runtime. For instance, an idle processor steals tasks from other processors. An idle processor will also try to steal an *entire* task-affinity set, since the set can execute on any processor and benefit from cache locality. Several processors can execute tasks from the same task-affinity set if the common object is not being modified; modifications to the object will invalidate it in other processors' caches.

3.4 Object Distribution

In addition to affinity hints for a task, we also allow the programmer to distribute objects across memory modules, both when they are allocated, as well as by dynamically migrating them to another processor's local memory. Since tasks with object affinity are executed where the object is allocated, distributing objects across memory modules can improve the load balance of the program.

By default, memory is allocated from the local memory of the requesting processor. To allocate memory from within the local memory of a particular processor, a processor number can be supplied as an additional argument to the *new* operator (see Table 3.2). To dynamically move an allocated object from one processor's memory to another, we provide the *migrate* function. The migrate function takes a pointer to an object and a processor number, and it migrates the object to the local memory of the specified processor (modulo the number of server processes). An optional third argument that

Table 3.2: Summary of data distribution constructs.

Affinity Construct	*Description*
`x = new (P)` *type*	Allocate the object from the memory of processor P.
`migrate` (*obj-addr*, `P`)	Migrate the object to the memory of processor P.
`migrate` (*obj-addr*, `P`, `num`)	Migrate an array of 'num' objects to the memory of processor P.
`home` (*obj-addr*)	Return the processor P that contains the object in its local memory.
`distribute (x,M,1)`	Distribute elements of x (1-D) round-robin.
`distribute (x,M,b)`	Distribute elements of x (1-D) blocked round-robin.
`distribute (x,M,1,N,N)`	Distribute rows of x (2-D) round-robin.
`distribute (x,M,b,N,N)`	Distribute rows of x (2-D) blocked round-robin.
`distribute (x,M,M,N,1)`	Distribute columns of x (2-D) round-robin.
`distribute (x,M,M,N,b)`	Distribute columns of x (2-D) blocked round-robin.
`distribute (x,M,b1,N,b2)`	Distribute blocks of x (2-D) round-robin.

specifies the number of objects to be migrated is useful to migrate an array of objects. Finally, the programmer can determine where an object is allocated through the *home* function that returns the number of the processor that contains the given object allocated in its local memory.

These constructs allow the programmer to describe the desired distribution (if any) of objects in the program. The implementation cannot, however, guarantee this precise distribution since the actual distribution is limited by the data allocation granularity in the underlying architecture. For instance, the operating system on the Stanford DASH supports data allocation only at the granularity of a 4-KB page. Object distribution on these machines is therefore implemented through the migration of entire pages spanned by the object, rather than by the object alone. Hence, in addition to specifying precise object distributions, the programmer must also consider the interaction of the layout of different objects in memory and the underlying granularity of data distribution.

The new and migrate operations express general object distributions. Many parallel programs, especially scientific and engineering and numerical applications, have very regular data structures such as matrices and arrays. For these programs we provide the *distribute* construct that allows the programmer to easily specify the distribution of these regular data structures. Besides the array object, the distribute function is supplied a pair of values (N, b) for each dimension of the supplied array. In this pair N is the declared size of that dimension, and b is the desired block size in that dimension. The distribute function determines the blocking of the array and distributes the blocks in a simple round-robin manner across the local memories of the processors.

Figure 3.4 presents a blocked distribution of a two-dimensional array x, of size 15x18, during a 16 processor execution. As shown in the figure, the statement $distribute(x, 15, 3, 18, 3)$ chooses blocks of size 3x3 and distributes them in a round-robin fashion across processors. The number within each block denotes the processor number to which the particular block is assigned. Table 3.2 presents other examples of

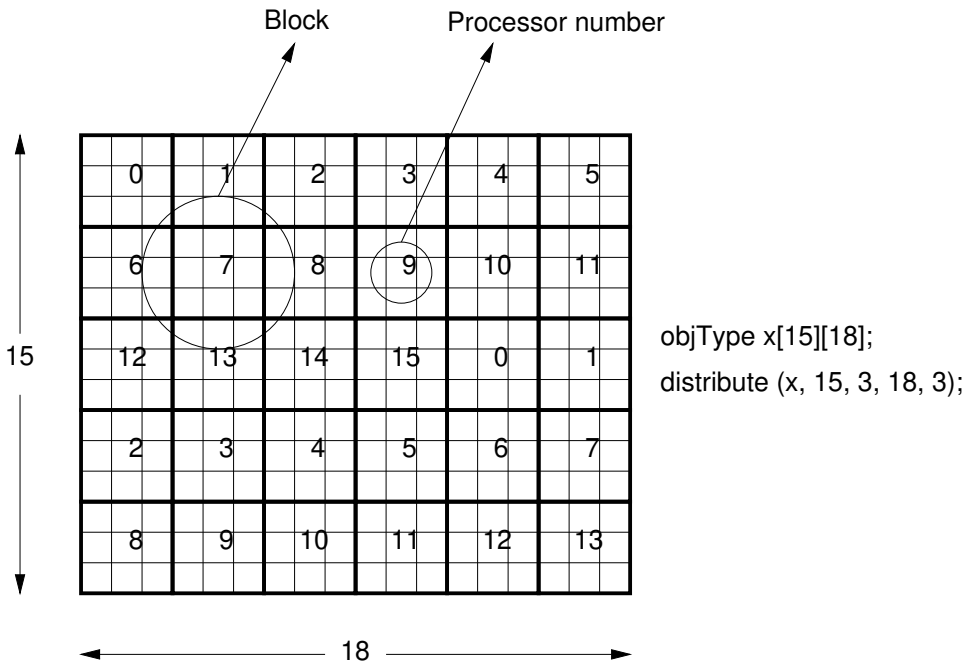

Figure 3.4: Blocked distribution of a two-dimensional matrix through the *distribute* statement.

several common distribution patterns specified using the distribute statement.

The *distribute* function can be used to dynamically redistribute arrays as well, which is useful when different phases of the program require a different object distribution. The blocks identified to the *distribute* function are distributed in a round-robin fashion across a *linear array* of processors (with wrap-around), as shown in Figure 3.4. We currently do not support more complex processor configurations such as a two-dimensional matrix of processors, although these basic constructs can be easily extended to support them. Furthermore, as before, the actual distribution may not precisely match the specified distribution due to the allocation granularity on the machine. We also do not consider transformations such as converting a blocked two-dimensional array into a four-dimensional array so that the data within each block can be laid out contiguously in memory. These transformations are usually performed explicitly by the programmer since the analysis required for automating them is often beyond the scope of a compiler. Finally, the distribute function does not provide any new functionality. Rather, it can be easily implemented using the migrate operation as shown in Figure 3.5. This figure presents an implementation of the distribute function for one and two-dimensional arrays using the migrate call. The distribute function is therefore provided only to make it easier to specify regular distributions for data structures such as arrays and matrices.

The distribute statement in COOL specifies data distribution similar to the primitives in HPF [78], but differs in the following respects. Most importantly, whereas the distribute statement in COOL is just a hint, the data distribution primitives in HPF are used by the compiler to (a) extract parallelism from the serial HPF program, (b) schedule the

```
// distribution of a 1-D array.
distribute (void* x, int M, int b) {
   int i;
   for (i=0; i<M; i+=b)
      migrate (x+i, i/b, b);
}

// distribution of a 2-D array.
distribute (void* x, int M, int b1, int N, int b2) {
   int i, j, k;
   for (i=0; i<M; i+= b1)
      for (j=0; j<b1; j++)
         for (k=0; k<N; k+=b2)
            migrate (&(x[i+j][k]), (i/b1)*b2 + k, b2);
}
```

Figure 3.5: Implementation of *distribute* using *migrate*.

parallel computation, and (c) manage the communication between the parallel tasks. Furthermore, COOL is designed exclusively for shared-memory architectures, whereas HPF can execute on both shared-memory and message-passing machines. As a result the distribute statement in COOL can be treated as a hint, with the communication of poorly placed data being automatically handled by hardware (at worst resulting in a performance penalty). In contrast, while HPF implementations on shared-memory machines can make such assumptions, implementations on message-passing machines must distribute the data precisely as specified, with the communication of data being handled by the software runtime system. Finally, compared to the simple blocked distributions across an array of processors expressed by the COOL distribute statement, the HPF primitives can express more general distributions across complex processor configurations. Such more general distributions must be explicitly specified using the migrate statement in COOL.

To summarize, the migrate and distribute constructs allow the programmer to distribute objects across the local memories of processors. The migrate call allows the programmer to specify arbitrary and irregular object distributions, while the distribute call allows regular round-robin distributions to be specified in a straight-forward fashion. These constructs only affect the physical allocation of data and do not affect the semantics of the program. Furthermore, the constructs serve only as hints since the actual object distribution is inherently limited by the granularity of memory allocation in the underlying architecture. Finally, the design of the data structures in the program is crucial in performing these optimizations successfully.

3.5 Summary

Exploiting data locality is crucial for high performance in modern multiprocessors. However, optimizations to improve data locality are difficult because they require both an intimate knowledge of the application structure and an understanding of the underlying memory hierarchy. As a result, in most systems today these optimizations are performed explicitly by the programmer in a very machine-dependent fashion.

In this chapter we have described the support provided in COOL to address these performance issues. Our approach is attractive because it allows the programmer to focus on supplying information about the application, leaving the low-level details of task scheduling to the implementation. The abstractions are easy to use and supplied in terms of the objects in the program. Furthermore, the abstractions are powerful enough to exploit locality at each level in the memory hierarchy and port transparently across a variety of shared-memory multiprocessor architectures. In Chapter 5 we apply our approach to several parallel applications and evaluate its effectiveness in practice.

Chapter 4

Implementation and Optimizations

We have implemented COOL on the Stanford DASH multiprocessor [74] (32-64 processors), the SGI 4D-340 multiprocessor workstations [12] (8 processors), and the Encore MultiMax [34] (32 processors). In this chapter we describe the COOL implementation in detail. We first give a high-level overview of the implementation, including the front-end translation process and the runtime execution model. Next we describe the implementation of the individual COOL constructs. We then present the techniques employed to optimize the overhead of each COOL construct. We also describe some compiler techniques that can very efficiently implement synchronization abstractions built using monitors and condition variables. Finally, we outline the task queue structure and scheduling support in the runtime system, and we present some base performance numbers of several applications written in COOL and running on the Stanford DASH multiprocessor [74].

4.1 Implementing the Constructs

Figure 4.1 shows the overall implementation and the compilation process for a COOL program. A COOL program is first translated to an equivalent C++ program by a yacc-based [61] source to source translator[1] that replaces the individual COOL constructs by calls to library support routines. The generated C++ program is then compiled for the target machine by a standard C++ compiler and linked with the COOL runtime libraries to produce an executable.

The COOL runtime system provides a task-queue model of parallelism [8, 26, 31, 28, 14] in which user-level tasks are scheduled onto a number of kernel-level processes that execute in the same shared address space. When a COOL program begins execution

[1] We used the yacc description of C++ provided with the GNU C++ compiler as our starting base, for which we are grateful to the Free Software Foundation.

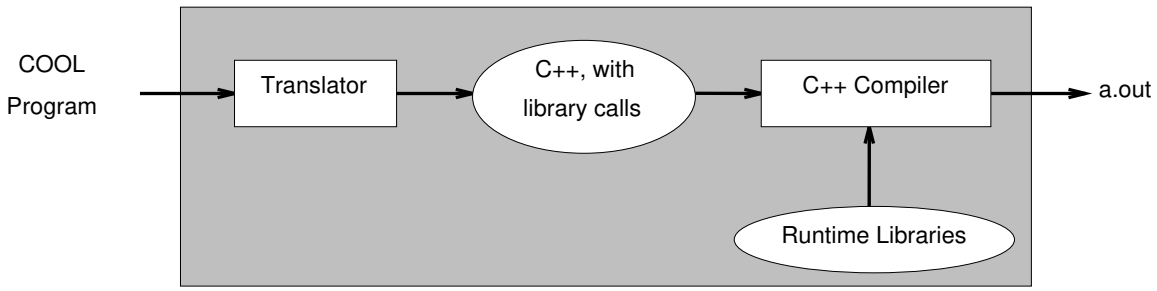

Figure 4.1: Compilation of a COOL Program.

several server processes are created, usually one per available processor. These server processes execute for the entire duration of the application program. They correspond to traditional heavy-weight UNIX processes and execute within the same shared address space. Each server process is assigned to a processor where it executes for its entire lifetime without migrating to another processor.[2] An invocation of a parallel function creates a *task*, a light-weight unit of execution. Tasks are implemented entirely within the COOL runtime system. They have their own stack and execute in the shared address space. Each server continually fetches a task and executes it without preemption until the task completes or blocks (perhaps to acquire a mutex object or wait on an event), whereupon the server process goes back to fetching another task. The program finishes execution when all tasks have completed.

In the following sections we describe the implementation of each COOL construct. We describe the *translation* process that generates the necessary calls to the runtime library routines and the *runtime* support for each construct.

4.1.1 Concurrency: Parallel Functions

A parallel function in COOL is implemented as follows. During the compilation process, the translator recognizes each invocation of a parallel function and replaces it by a call to a runtime routine to create a task. During execution this routine creates a task template specifying the parallel function and the arguments, and it enqueues the template onto a task queue. Finally, a server process dequeues the task template and executes the corresponding parallel function. We describe each of these in turn below.

To recognize the invocation of parallel functions, the translator maintains two tables—a table of all the parallel functions that have been declared so far and a table of all the variables (including both globals and locals) that have been declared so far. As the translator scans the input COOL program, it adds an entry to the table of parallel functions whenever it encounters a declaration for a new parallel function. C++ allows multiple functions to have the same name which are then disambiguated by the types of their arguments; therefore the table also stores the type of each argument to each parallel

[2]This approach may not be desirable in a multiprogrammed environment where the machine is simultaneously shared between several applications.

```
// C-style parallel functions.
// serial function, integer argument.
condH* foo (int);

// parallel function of the same name
// but different argument.
parallel condH* foo (double);

main () {
   . . .
   foo (3); // invoke (serial) foo (int).
   . . .
   foo (1.2); // invoke parallel foo (double).

}
```

```
// C++ style parallel functions.
// serial function (global).
condH* bar (int);

class myclass {
public:
   // parallel method, with the
   // same name and argument types.
   parallel condH* bar (int);
   void baz () {
      // parallel invocation of the method.
      bar (2);
      // again, parallel invocation of the method.
      this.bar (4);
      . . .
      // serial invocation of the C function.
      ::bar (5);
   }
}
```

Figure 4.2: Recognizing the invocation of a parallel function.

function. Since a parallel function may be a method belonging to a class, the table of parallel functions also maintains the class name (if any) for each function. The table of variable declarations maintains the type of each variable visible in the current scope, and is updated at variable declarations as well as when a new scope is created or destroyed in the program.

When the translator encounters a function invocation it performs a lookup in the table of parallel functions to determine if that function has been declared to be parallel.[3] A function invocation in a COOL program may either be a method invocation on an object or a regular C-style function invocation (see Figure 4.2). For method invocations the translator first performs a lookup in the table of variables to determine the class of the variable and then performs a lookup in the table of parallel functions to determine if the invoked function of the class was declared as a parallel function. A regular C-style function invocation, on the other hand, may refer to either a C function (i.e., not a method) or, if the function was invoked from within a method, it may refer to a method of the *enclosing class*. For such function invocations the translator checks for a match based on C++ scope-rules of the closest enclosing scope first. Therefore, it first checks for a

[3]This approach is facilitated by the ANSI C standard, which requires that a function must have been declared (through a function header or prototype) before it can be invoked. In our current implementation, if a parallel function is invoked before the function has been declared, then that invocation will result in sequential execution.

```
// Original COOL program.
// Parallel function header.
parallel condH* foo (double);

main () {
  condH* synch;
  . . .
  // Invocation of a parallel function.
  synch = foo (2.3);

  . . .
}
```

```
// Generated C++ program.

condH* foo (double);
// struct to hold arguments and return value
struct foo_double_s {
  double a0; // Argument
  condH* rv; // Return value
};
// Wrapper for parallel function
void foo_double_w (foo_double_s* sptr) {
  // unmarshall arguments and invoke foo.
  foo (sptr->a0);
  // signal the completion of the function.
  sptr->rv.signal ();
}

main () {
  condH* synch;
  . . .
  // Invocation of a parallel function.
  // Create task for the parallel function
  {
    // allocate template for arguments/return value.
    foo_double_s *sptr = new foo_double_s;
    // marshall the arguments.
    sptr->a0 = 2.3;
    // store the pointer to the return event.
    synch = sptr->rv = new condH;
    taskCreate (foo_double_w, sptr);
  }
  . . .
}
```

Figure 4.3: Translating a parallel function invocation.

matching method in the current class and then checks for a matching C-style function (see the invocation of the function bar on the right side of Figure 4.2).

Upon recognizing a parallel function, the translator removes the keyword parallel from the function header and leaves the body of the parallel function unchanged. Furthermore, it generates the following code for each parallel function (see Figure 4.3). For a parallel function header (the first header for each parallel function), the translator generates a struct to hold the arguments and the return value of the function. It also generates a

wrapper function that takes a pointer to the above struct, unmarshalls the arguments, and invokes the parallel function with the appropriate arguments. For each invocation of a parallel function, the translator replaces the actual invocation by code to (a) allocate an instance of the above struct, (b) marshall the actual arguments into the struct, and (c) call a runtime routine to create and enqueue a task template. The task template consists of a pointer to the wrapper function and a pointer to the struct containing the actual arguments.

This general scheme supports parallel functions with parameters of any arbitrary type, but incurs the overhead of invoking the intermediate wrapper function. Our implementation optimizes away this overhead in situations where the parameters of the parallel function are of simple primitive types such as integers or pointers. For such functions we do not use a wrapper function and instead store the function and the actual arguments directly in the task template; the runtime system subsequently invokes the function directly on the supplied arguments rather than going through the wrapper function.[4]

To support the return value of a parallel function (a pointer to an event), the translator generates code to allocate an event at the point of function invocation (see Figure 4.3). A pointer to this event is stored in the variable that is assigned the return value of the parallel function (the variable *synch* in the figure). This pointer is also stored in the struct containing the marshalled arguments. The former is used by the caller to synchronize for the completion of the parallel function, while the latter is used by the wrapper function to signal the event upon completion of the parallel function. A simple optimization performed is that if the caller ignores the return value of the parallel function then the event need not be allocated at all. Furthermore, as mentioned earlier in Chapter 2, the compiler can often deallocate these events automatically once they are no longer accessible, such as upon exiting the scope of a stack-allocated variable. However, our current implementation does not perform these optimizations. In any case, since the compiler is forced to be conservative, the ultimate responsibility for deallocating events continues to rest upon the programmer.

The implementation employs several tricks to keep the overheads down. For instance, the runtime system allocates a stack within which to run a task only after the task template is actually dequeued for execution. Allocating a stack as late as possible as in this scheme is important for applications that use large amounts of memory, since it avoids wasting memory in stacks that are allocated but unused. In addition, we maintain our own free-lists for all dynamically allocated data structures in the runtime system, such as task templates, stacks, and structs for marshalling the arguments of a parallel function. Allocation requests are therefore mostly satisfied from within the free-lists with little overhead and only occasionally need to go to the operating system for additional memory. Furthermore, an instance of every free-list is maintained for each processor, thereby avoiding contention while allocating/freeing these data structures. To keep the

[4]This optimization is performed only when the argument types are simple, and the runtime is able to pass the arguments to the parallel function in the processor registers. We exclude functions with arguments that are floats/doubles, since those arguments must be passed in the floating point registers and follow a more complex calling convention.

free-lists from getting imbalanced, we also maintain a global instance of each free-list. If the number of elements on a processor's free-list exceeds a certain threshold (which is different for different free-lists), then some excess elements are transferred to the global instance of this free-list. Conversely, when a processor finds a free-list empty, it tries to get elements from the global free-list before actually allocating more memory from the system. Although accesses to the global free-lists must be synchronized, these accesses are sufficiently infrequent that contention is not a problem.

Finally we address the issue of termination of a COOL program. As mentioned earlier, an invocation of a parallel function creates a task that is enqueued onto a task queue, while server processes continually fetch tasks from the task queue and execute them. The task queue structure is described in detail later in this chapter; we now describe the algorithm used to detect termination of the program within the runtime system. A straightforward strategy is to keep a global count of the number of tasks still executing, increment the counter when a task is created, and decrement the counter when a task completes execution. The program can terminate when the counter reaches zero, signifying that all tasks have completed. However, this single global counter needs to be referenced (incremented/decremented) within a lock at each task creation and completion event, and can become a bottleneck when using more than a few processors.

To reduce contention we implement a distributed termination algorithm built upon the waitfor construct. In this scheme we simply wrap the entire body of *main* within a waitfor; the semantics of waitfor ensures that execution resumes beyond the waitfor only after all tasks have completed and the program can terminate. At this point all the processors are directly notified that the program has terminated. This approach builds upon the implementation of the waitfor construct described later in this chapter and is in effect a distributed termination algorithm. Briefly, instead of the global counter, the runtime system maintains a separate counter for each task to track the number of tasks directly created by the task that are still executing. The counter for a task is incremented when it invokes a parallel function and creates a task, and is decremented by each child task upon completion. The program terminates once the counter for the main task reaches zero. These distributed counters therefore provide us with an efficient termination algorithm.

4.1.2 Synchronization: Monitors

Monitors are implemented by wrapping each monitor operation within some *entry* and *exit* code to perform the necessary synchronization for the object. When the monitor operation is invoked, the entry code is executed to acquire access to the object before continuing, waiting for the object to become available if necessary. Upon completion of the monitor operation the exit code is executed, which surrenders access to the object and wakes up a thread waiting to enter the monitor, if any. In this section we first describe the front-end translation of monitor operations, and then we describe the runtime synchronization algorithm.

```
// Original COOL Program
class monClass {
  . . . data declarations . . .

public:
  mutex void foo ();
  . . .

};
```

```
// Translated C++ program
class monClass {
    . . . data declarations . . .
  _mutexClass _mutexObj;
public:
  void foo ();
    void _wfoo () {
    _mutexObj.mutexEnter ();
    foo ();
    _mutexObj.mutexExit ();
  }
};
```

Figure 4.4: Translating a monitor function.

4.1.2.1 Translation of a Monitor

The translator modifies each monitor class (a class containing a mutex or a nonmutex method) as shown in Figure 4.4. It declares an additional object $_mutexObj$ of the predefined type $_mutexClass$ within the monitor class. This object contains the runtime data structures that maintain the status of the monitor and is described later in this section. Next, for each monitor operation foo in the class, the translator declares an additional wrapper function $_wfoo$ that invokes the actual function foo wrapped within enter and exit operations on the $_mutexObject$ to perform the necessary synchronization. Finally, each invocation of a monitor function foo is instead replaced by a call to the wrapper function, $_wfoo$. To perform these transformations on the original COOL program, the translator must recognize all invocations of monitor functions similar to the implementation of parallel functions described earlier in Section 4.1.1. The translator therefore maintains tables of the variables and mutex operations declared so far.

In contrast to this implementation, it would have been simpler to leave the invocations unchanged and instead modify the *body* of each monitor operation to call the enter and exit operations at the beginning and end of the function. However, our scheme allows the compiler to optimize individual invocations of monitor functions, by statically identifying those invocations that don't need to synchronize, and calling the method directly rather than the synchronizing wrapper function. These optimizations are described later in the chapter.

One minor implication of this implementation is that declaring the additional object $_mutexObject$ within a monitor changes the size of the object transparently to the programmer. Therefore, while the sizeof function works correctly, programs that do direct address arithmetic to move from one object to another without using the sizeof functions can get incorrect results. The compiler can address this problem by instead maintaining each $_mutexObject$ as a separate *shadow object*. All monitor operations perform the

necessary synchronization as before, but on the shadow object. This scheme can handle pointers as well by maintaining a shadow pointer for each pointer to a monitor object. All pointer manipulations would then be accompanied by corresponding manipulations of the shadow pointers to point to the appropriate $_mutexObject$. However, changing the size of a monitor object is not a serious problem in practice, especially since the problem already exists in C++ when the size of an object is modified transparent to the programmer to include space for a virtual function table pointer.

4.1.2.2 The Synchronization Algorithm

We first describe the basic synchronization algorithm for monitors, i.e., the entry and exit code for mutex and nonmutex functions and then describe the implementation of the more complex features.

The main data structures and the synchronization operations for a monitor are provided by the class $_mutexClass$ (see Figure 4.5). An instance of $_mutexClass$ is declared within each class in the user program with monitor functions. A monitor object can be in one of three states—*free* (no monitor operation), *shared* (one or more nonmutex operations executing), or *locked* (a mutex operation is executing). The entry code for both mutex and nonmutex operations is shown in Figure 4.6 and includes the more complex features that are described later in this section. Since a mutex operation must have exclusive access to the object, it can proceed only if the object is free and blocks otherwise. A nonmutex operation can execute if the object is in the free or shared state, but it must wait if the object is locked. The exit code for a monitor operation is shown in Figure 4.7. Upon completion each monitor operation checks whether there is an operation waiting to enter the monitor. If so, then the waiting operation is granted access as the completing operation exits the monitor. If there was no waiting operation, then the monitor is reset to be in the free state. To ensure fairness in execution, incoming monitor operations are serviced in first-come-first-served order (nonmutex operations execute concurrently, of course). In addition, to avoid the starvation of mutex operations, our implementation ensures that if a mutex operation is waiting to execute on a object, then newly arrived nonmutex requests do not jump over the waiting mutex thread.

If a monitor operation finds that the object is unavailable then the task executing the entry code must wait until the object becomes available. This waiting *must* be implemented by blocking the task so as to free the underlying processor to fetch and execute other tasks. (All synchronization operations in the task-based execution model of COOL must, in general, be implemented by blocking the task.) Since a task in COOL executes without preemption until it completes or blocks on synchronization, implementing the waiting with busy-waiting can lead to deadlock in some programs. For instance, consider the scenario where access to a monitor is implemented by pure busy-waiting, and all processes (except the one inside the monitor) are spinning to acquire the monitor. The process actually inside the monitor executes a wait operation, thereby suspending the task still holding the monitor. The process then goes and executes another task which (a) performs a signal on that condition variable and (b) invokes an operation

```
class _mutexClass {
    lock_t lk;                      // lock to protect access to this structure
    int status;                     // Monitor status—one of FREE, SHARED, and LOCKED
    queue_t Q;                      // Q for new monitor requests waiting to enter
    queue_t reenterQ;               // Q for operations resumed after a release waiting to reenter
    queue_t upgradeQ;               // Q for upgrade requests waiting to enter
    int upgrade;                    // upgrade flag: set if the current function is an upgrade
    int threadID;                   // threadID of currently executing mutex function
    int threadIDList[MAX];          // threadIDs of currently executing nonmutex functions
    int tidptr;                     // pointer into above array

public:
    int mutexEnter ();              // enter request for a mutex function
    void mutexExit (int);           // exit for a mutex function
    int nonmutexEnter ();           // enter request for a nonmutex function
    void nonmutexExit (int);        // exit for a nonmutex function

    int upgradeEnter ();            // call from nonmutex to mutex function to upgrade
    void upgradeExit (int);         // completion of an upgrade mutex function

    void releaseExit ();            // called to exit a monitor function with a release
    void releaseReenter ();         // call to reenter monitor after a release and subsequent signal

    _mutexClass ();                 // constructor
    ~_mutexClass ();                // destructor
};
```

Figure 4.5: Monitor data structures.

on the very same monitor and starts busy-waiting. We now have the situation where all the processes are busy-waiting to enter the monitor. However, the task actually inside the monitor, although enabled, will never get scheduled since all the processes are busy-waiting for access to that monitor. Therefore, a monitor operation (and any other synchronization) must in general block and surrender the processor if a monitor is not available.

However, blocking a task and switching to another context has high overhead. While blocking immediately on an unavailable monitor is efficient in situations where the operations execute for long durations, it has excessive overhead for objects with small operations. We therefore implement the entry to a monitor operation with a *two-phase* algorithm, in which the entry code busy-waits for a 'while' (heuristically chosen by the runtime system) and then blocks if the monitor object is still unavailable. This context switch is between user-level tasks and is analogous but orthogonal to the implementation

of locks in the context of operating system scheduling, where locks can be implemented with pure spin or with blocking. This two-phase scheme is successful in reducing unnecessary context switches for shared objects with small critical sections.

In addition to this basic synchronization, monitors in COOL provide the following additional functionality. First, access to an object must be on a thread basis, so that both direct and indirect recursive calls on an object by the same thread execute without deadlock. Second, if a nonmutex function invokes a mutex function, then it should upgrade from shared to exclusive access for the duration of the mutex function and revert back to shared state upon completion of the mutex function. Invocations of a nonmutex function from within a mutex function should execute without any additional synchronization. Finally, when a monitor function invokes a release operation on a condition variable, the task should surrender the object and block. When the condition variable is signaled, the task must reacquire the object before resuming execution inside the monitor. We discuss each of these in turn below.

Recursive Calls: Recursive monitors calls to the same monitor object in a COOL program should execute without requiring any synchronization since the thread has already acquired access to the monitor. For example, if a function foo invokes another monitor operation bar (or even foo) on the *same* object, either directly or indirectly through another function, then the second operation bar should not block since the thread has already acquired access to the monitor object through foo.

To detect recursive monitor calls, our implementation identifies each task (or execution thread) by a unique integer value. Each monitor object stores this identifier in the variable *threadID* for the task that is currently executing within the monitor. This variable is initialized to the thread identifier each time an operation first acquires an available monitor. A subsequent monitor operation that finds the object unavailable compares its identifier with the threadID stored within the monitor to determine if it already has access to the object. If so, then it proceeds without further synchronization. Thus we can dynamically handle both direct and indirect (daisy-chain) recursive calls on a monitor object. Indirect or daisy-chain recursion occurs when a function f does not contain the actual call to f, but invokes another function g which either invokes f or invokes a function h that invokes f, and so on. Upon exiting a monitor function, the object is surrendered only if the object was actually acquired for that invocation of the monitor function. This determination is made based on the value returned by the monitor enter operation: an enter operation returns 1 if the object was actually acquired and 0 if it was a recursive call (see Figure 4.6).

We detect recursive nonmutex calls in a similar fashion except that we now require a *list* of threadIDs within each object, one for each nonmutex operation that may be executing within the monitor. Thus a nonmutex operation checks whether the thread already has access to the object, in which case it continues execution. As before, the return value of the monitor operation is used while exiting the monitor to determine if the object should be actually surrendered to waiting threads. Furthermore, while in general we do not allow a nonmutex operation to jump ahead of a waiting mutex operation in

```
// Entry code for a mutex operation.
// Return 0 if this thread already had the object, 1 if it needed to acquire access.
int _mutexClass::mutexEnter () {
   if (recursive mutex) {
      return 0;
   }
   if (object is free) {
      . . . acquire object: initialize status, threadID . . .
      return 1;
   }
   if (I already have nonmutex access—upgrade) {
      if (there are other readers in the monitor) {
         // wait for other readers to complete
         upgradeQ.sleep ();
      }
      . . . acquire object: initialize status, threadID . . .
      return 1;
   }
   Q.sleep ();
   acquire object: initialize status, threadID
   return 1;
}

// Entry code for a nonmutex operation.
// Return 0 if this thread already had the object, 1 if it needed to acquire access.
int _mutexClass::nonmutexEnter () {
   if (recursive nonmutex) {
      return 0;
   }
   if (status != LOCKED) {
      // object is free or shared, so acquire object
      . . . acquire object: initialize status, threadIDList . . .
      ++Nreaders;
      return 1;
   }
   Q.sleep ();
   . . . acquire object: initialize status, threadIDList . . .
   return 1;
}
```

Figure 4.6: Entry code for monitor operations.

```
void _mutexClass::mutexExit (int t) {
  if (t == 0) {
    // recursive mutex
    return;
  }
  if (I am an upgrade) {
    . . . restore status to shared . . .
    . . . wake up readers on Q . . .
  }
  if notempty (reenterQ) renterQ.wakeup ();
    else if notempty (Q) Q.wakeup ();
         else status = FREE;
}

void _mutexClass::nonmutexExit (int t) {
  if (t == 0) {
    // recursive nonmutex
    return;
  }
  . . . delete from threadIDList . . .
  Nreaders -= 1;
  if (I'm the last nonmutex leaving) {
    if notempty (upgradeQueue) upgradeQueue.wakeup ();
    else if notempty (OldQueue) oldQueue.wakeup ();
      else if notempty (NewQueue) NewQueue.wakeup ();
        else status = FREE;
  }
}
```

Figure 4.7: Exit code for monitor operations.

the interests of fairness, a recursive nonmutex operation executes before waiting mutex operations to avoid deadlock.

Upgrade: When a nonmutex function invokes a mutex operation on the *same* object, either directly or indirectly, ordinarily it would find the object shared and block, leading to deadlock. However, a more natural behavior would be for the mutex operation to acquire exclusive access to the object and execute, reverting to the shared state upon completion (we call this an *upgrade*). Therefore, when a mutex operation is invoked, it checks whether the current thread already has access to the object through a nonmutex function, in which case it must upgrade. If so, then since a mutex operation requires exclusive access to the object, it must also wait until any executing nonmutex operations have exited the object before changing the object state to locked and executing. To support upgrades each monitor object has a queue on which upgrade requests block and wait. While the nonmutex operation is waiting to upgrade, new nonmutex requests are barred from entering the monitor even though the object is available for nonmutex functions; therefore new nonmutex requests cannot proceed if a thread is waiting on the upgrade queue. When the last executing nonmutex operations exits the monitor, it checks the upgrade queue, and resumes the waiting thread, if any. After the upgrade mutex operation has completed, it restores the object state to shared so that the enclosing nonmutex operation can continue. Furthermore, any nonmutex requests that arrived while the upgrade mutex operation was executing are woken up so that they can execute concurrently on the shared monitor object (however, nonmutex requests cannot jump over previous mutex requests as before).

Release: When a release operation is invoked on an event within a monitor, it surrenders the monitor if the event is yet to be signaled, resuming execution where it left off once the event is signaled. The implementation of the release operation is shown in Figure 4.8 and is straightforward, except that upon being signaled, the thread must wait for all executing monitor operations to complete before resuming execution inside the monitor (see the $releaseRenter$ operation). In particular, if the event was signaled from within a monitor operation foo, then foo must complete execution before the signaled thread can reenter the monitor, thereby ensuring that there is one mutex operation executes inside the monitor at any time. However, in our implementation the signaled thread is given priority over new monitor operations and therefore waits only for already executing operations to complete. This is implemented by having another, high-priority queue (called *reenterQ*) for resumed threads that are waiting to reenter the monitor.

4.1.3 Synchronization: Condition Variables

A condition variable provides an event queue on which threads can block and signal. The type cond therefore consists of a queue and a lock to serialize multiple operations on the condition variable. A wait operation blocks on the queue, a signal wakes up the first waiting thread, and a broadcast wakes up all waiting threads. A condition variable

```
void condH::release () {
  if (event not signaled yet) {
    _mutexObject.releaseExit ();
    . . . block on event . . .
    _mutexObject.releaseReenter ();
  }
}

void _mutexClass::releaseExit () {
  savedID = mythreadID; // to be restored upon reentry
  mutexExit (); // give up the monitor
}

void _mutexClass::releaseReenter () {
  if (monitor is FREE) {
    // acquire monitor, continue
    . . . Initialize monitor data structures . . .
    threadID = savedID; // restore threadID
    return;
  }
  // Otherwise wait to reacquire access
  reenterQ.sleep ();
  . . . Initialize monitor data structures, continue . . .
}
```

Figure 4.8: Release code for monitor operations.

with history, condH, contains an additional counter that counts the number of signals stored in the condition variable. If the number of stored signals is non-zero, a wait operation decrements the counter and continues without blocking. A signal operation either resumes a waiting thread (if any) or increments the counter of stored signals. A broadcast wakes up all waiting threads as well as sets the counter to be a very large value, while the uncast operation resets the counter to zero.

4.1.4 Synchronization: waitfor

The waitfor construct must keep track of all tasks created within the scope of the waitfor, either directly or indirectly, and wait for them to complete before continuing executing past the end of the scope of the waitfor. We first outline the implementation of a single waitfor construct and then describe how nested waitfors are implemented.

The basic data structure to implement a waitfor consists of a counter, a queue, and

```
struct waitfor_t {
  int count;
  LockType lock;
  QueueType queue;
}

beginWaitfor () {
  // allocate a waitfor struct
  struct waitfor_t* wf = new waitfor_t;
  wf->count = 0;
  . . . store wf in template of current task . . .
}

endWaitfor () {
  struct waitfor_t* wf = current waitfor;
  AcquireLock (wf->lock);
  if (wf->count) {
    // all tasks haven't completed yet, so block
    ReleaseLock (wf->lock);
    wf->queue.sleep ();
  }
  else
    ReleaseLock (wf->lock);
}
```

```
// waitfor code, executed when a task is created
. . .
if (current task is in a waitfor) {
  struct waitfor_t* wf = current waitfor;
  AcquireLock (wf->lock);
  wf->count++;
  ReleaseLock (wf->lock);
  . . . store wf in template of created task . . .
}
. . .

// waitfor code, executed when a task completes
. . .
if (current task is in a waitfor) {
  struct waitfor_t* wf = current waitfor;
  AcquireLock (wf->lock);
  if (--wf->count == 0) {
    // I am the last task
    if (someone is waiting on the queue)
      wf->queue.wakeup ();
  }
}
. . .
```

Figure 4.9: Implementing the waitfor construct.

a lock. The counter keeps track of the number of outstanding tasks that were created within the scope of the waitfor. The queue is used by the thread to block if it reaches the end of the waitfor and there are still some outstanding tasks. Finally, the lock is used to serialize multiple accesses to this data structure. Upon entering the scope of a waitfor, the runtime system allocates an instance of this waitfor structure and initializes the counter to zero. When a task is created within the waitfor, the counter is incremented and a pointer to the waitfor struct is passed to the created task. Upon reaching the closing scope of a waitfor, the thread blocks on the queue if some tasks are still outstanding. The created task upon completion decrements the counter in the waitfor struct; if the counter reaches zero then it wakes up the blocked thread on the queue, if any. Furthermore, if the task invokes a parallel function then a pointer to the struct is passed along to the child task as well.

If a child task executes a waitfor as well then it repeats the above algorithm—it allocates a new waitfor struct and passes a pointer to this new struct to any parallel tasks that it might create. Since this task waits for its children to complete, it maintains the

semantics of the original waitfor in its parent task as well. Similarly, multiple nested waitfors within the same task are implemented as a simple extension of this scheme. Each additional waitfor allocates a new struct, and subsequent child tasks are passed a pointer to the closest enclosing waitfor. Again, this is sufficient to maintain the semantics of each waitfor, since a waitfor cannot complete until all the waitfor constructs nested within that waitfor have completed.

4.2 Overheads and Optimizations

We now examine the overheads in the implementation of the COOL constructs described above, and describe several optimizations that help implement the language efficiently.

4.2.1 Overheads

Since a COOL construct typically executes within a few microseconds, it is difficult to measure its execution time accurately. While some constructs can be measured by executing them several hundreds (or thousands) of times, constructs that block the executing thread cannot be measured in this way. We therefore use the *pixie* [82] profiling tool that is available on machines based on the R3000 processors. Pixie provides an instruction tracing option that allows us to count the number of machine instructions executed from one point in the code to another in an actual execution of a program.[5] Therefore, rather than measuring the execution time, we measure the number of machine instructions (of a R3000 processor) executed in the implementation of each construct.

The instruction overheads for each COOL construct are shown in Table 4.1. It takes about 140 instructions in our implementation to create a task with one argument. These instructions are broken evenly between allocating/initializing the task template and locating the appropriate queue and enqueueing the task. Since this operation does not block we could time it by executing it thousands of times; we found that it took 16 microseconds to create a task. On a 33MHz R3000 processor, assuming one instruction per cycle, 140 instructions would ordinarily take 4.2 microseconds; the additional time reflects the penalty due to cache misses. To get an idea of the total overheads involved in performing a piece of work in parallel, we also measured the overhead of creating, fetching, and executing an empty task with one argument. This took a total of 400 instructions as shown in Table 4.1 and was measured to take nearly 30 microseconds. With these overheads, therefore, individual tasks should roughly be over a thousand instructions to get gains from parallel execution.

Table 4.1 also shows the overheads of the synchronization operations in COOL, such as entering and leaving a monitor object, wait and signal on a condition variable, and the waitfor operation. As we can see, the synchronization operations have small overheads if the operation can continue without blocking. Most of the overheads arise when a task

[5] We are grateful to Mike Smith for the utilities used in gathering the data.

needs to wait for the desired event: this includes blocking the task (78-107 instructions), waking up a task and placing it in the queue of ready tasks when the signaling event occurs (90-158 instructions), and dequeueing the task and resuming execution (140-155 instructions). For operations on monitors and condition variables the cost of waking up and enqueueing the waiting task on a ready queue is part of the monitor exit/signal operation respectively. For the waitfor construct, the waiting task is woken up and enqueued upon completion of the last task within the waitfor. This overhead is therefore part of the task completion activities. In the table, however, we show this cost explicitly for the waitfor construct.

Besides the overheads of the individual constructs, the bottom-line is the impact on the performance of actual applications; as the results presented later in this and the next chapter show, the overall COOL overheads are low, usually around 2-3% of the total execution time.

4.2.2 Optimizations

In the monitor implementation described above each monitor object requires 64 bytes of additional storage, and the enter and exit operations can take over 100 instructions if a thread needs to block. These overheads can be prohibitive in programs with fine-grained synchronization. We now present some optimizations that help eliminate most of these overheads. The optimizations also illustrate the benefits of integrating monitors with the object structure. This integration enables the compiler to (a) identify the data associated with the monitor synchronization and (b) identify the synchronization operations on an object. The compiler can therefore analyze the synchronization operations and optimize their implementation in various ways that we outline below.

Our optimization strategy is based on the observation that while the basic implementation is necessary in general, most applications built using monitors do not use all the functionality offered by a general monitor. A compiler can therefore analyze each individual monitor and tailor its implementation based on the features that it uses, thereby avoiding overheads for the functionality unused by that monitor. Examples of this strategy include a monitor that has only mutex functions and can therefore be implemented without the support for multiple readers. Or, a monitor that has operations with small critical regions is better implemented with pure busy-wait synchronization rather than the more general two-phase algorithm described earlier. We elaborate on these and other optimizations in this section. We have implemented several of the optimizations, while those that are beyond the scope of our current implementation are identified as we go along. With these optimizations, we show that synchronizations expressed using monitors in COOL can often be implemented as efficiently as hand-coded versions using low-level primitives.

Table 4.1: Instruction overhead of various COOL operations.

Construct	*Function*	*Instr Count*
Task creation	Save registers	7
	Allocate task template	16
	Store values	6
	waitfor	19
	Store scheduling hints	15
	Enqueue:	
	locate queue	20
	locate bucket, lock/unlock	25
	actual enqueue	15
	link queues	15
	Total	138
Create and execute a null task	Create a task	140
	Locate a non-empty queue	40
	Actual dequeue	80
	Allocate stack, load arguments and dispatch	70
	Cleanup: free storage and check waitfor	70
	Total	400
Monitor	Enter an available monitor	43
	Enter an unavailable monitor:	
	execute until block	107
	wakeup and enter	140
	Exit a monitor, no one waiting	45
	Exit a monitor, wake up a waiting thread	158
Condition variable	Wait operation, no blocking	16
	Wait operation, block:	
	execute until block	78
	wakeup and resume	150
	Signal operation, no one waiting	14
	Signal operation, wake up a waiting thread	90
Waitfor	Enter a waitfor	50
	Exit a waitfor, no waiting	35
	Exit a waitfor, must wait:	
	execute until block	87
	get woken up and enqueued	115
	wakeup and resume	155

4.2.2.1 Monitors with only Mutex Functions

It is common to find monitors in which the operations both read and modify the shared object. These objects have all mutex functions and no nonmutex functions. For such monitors the storage requirements can be considerably reduced. We can discard the storage required to support multiple readers (such as *threadIDList* and *tidptr* described in Figure 4.5 in Section 4.1.2.2) as well as the data structures for upgrade (the fields *upgradeQ* and *upgrade* in Figure 4.5). This reduces the space requirements by over 50%, down from the original 64 bytes to 28 bytes. Furthermore, the runtime checks upon monitor entry/exit for other readers and for an upgrade can also be discarded, reducing the execution time a little as well.

4.2.2.2 Spin Monitors

Parallel programs frequently have shared objects with very small critical sections. A typical example is a global counter on which the main operations are the increment and decrement operators. The overheads of even a two-phase synchronization algorithm are excessive for these shared counters: such monitors are best implemented with pure busy-wait semantics that we call *spin monitors*. Spin monitors (see Figure 4.10) are protected by a simple spin-lock which must be held for the entire duration of a monitor function. They are therefore highly attractive for small shared objects since they have minimal runtime and storage overhead—e.g., they require just a single word of extra storage for the lock rather than the original 64 bytes, and the enter and exit operations take 5 and 3 instructions respectively in contrast to the nearly 50 instructions required by the regular monitor. Therefore the potential savings with spin monitors are very large.

We have developed compiler heuristics to automatically identify spin monitors, i.e., those monitors that are suitable to implement with pure busy-wait semantics. A monitor can *safely* be implemented with busy-wait alone if none of the monitor functions call an operation that could block, i.e., a monitor operation on another object or a wait on an event. In addition, a monitor is *worthwhile* to implement as a spin monitor if all the monitor operations have a 'small' body (determined heuristically). We use heuristics to determine if a monitor operation is *small*, such as the functions should not call a function, and it should not contain any loops. One consequence of this optimization is that *all* monitor operations (of that particular monitor class), whether mutex or nonmutex, must be implemented with the same busy-wait protocol; multiple readers can therefore no longer execute concurrently as before. However, this should not be a problem in practice since the critical sections are small.

To implement this optimization the compiler must examine all the monitor operations within a class (and ensure that all are "small") before deciding the synchronization algorithm. This would have required two passes in our compiler and is therefore not implemented in our current system. Instead, the implementation supports an additional function attribute in the language, *cmutex*, that can be supplied for the member function of a class similar to the mutex and nonmutex attributes. This attribute is used by the

```
// Original COOL Program
class monClass {
  . . . data . . .

public:
  mutex void foo ();
  . . .




};
```

```
// Translated C++ program
class monClass {
  . . . data . . .
  lock_t _lk;
public:
  void foo ();
  void _foo () {
    AcquireLock (_lk);
    foo ();
    ReleaseLock (_lk);
  }
};
```

Figure 4.10: Implementation of a spin monitor.

programmer to identify monitor operations as before and additionally specify that synchronization for the monitor operations of that class be implemented with pure busy-wait synchronization.

4.2.2.3 Private Condition Variables

Condition variables are implemented with a lock that protects access to their internal data structures. However, this lock and the associated synchronization operations can be optimized away in some situations. For instance, monitors and condition variables are often used to build synchronization abstractions that are specific to an application; many of these were presented as examples in Chapter 2, such as the synchronizing double and the barrier abstraction. These abstractions typically consist of a monitor class with some private status fields and a condition variable. Furthermore, these private fields are referenced only within synchronizing mutex operations. The compiler can, in particular, recognize the private event variables that are accessed only within mutex functions. Operations on such condition variables, such as wait/signal, are therefore automatically serialized through the enclosing mutex function, and the lock that is normally associated with a condition variable to serialize multiple operations can be entirely discarded. This reduces the synchronization overhead associated with the operations on these abstractions by avoiding the redundant lock/unlock pair, resulting in several instructions worth of savings.

This optimization can be safely performed with spin monitors as well, even when they contain a release operation. Since accesses to a spin monitor are protected by a lock, the release operation unlocks the lock (to surrender the monitor) and blocks itself on the event. Upon being signaled on the event, the thread reacquires the lock (note that a resuming thread does not have priority over new requests in this scheme). In fact,

optimizing a private condition variable within a spin-monitor with a release constitutes a common and useful scenario that is typical of many synchronization abstractions. This optimization does require two-pass compilation and is not implemented in our current system.

4.2.2.4 Directly Recursive Calls

As mentioned before, monitor operations must check at runtime to determine whether they must acquire the object, since recursive operations execute within the access held by the thread without reacquiring the object. However, a compiler can easily identify a *directly* recursive call where one monitor operation invokes another on the same object directly. Such a recursive call is implemented like an ordinary function call with no synchronization at all, optimizing the nearly 25 instructions incurred in the runtime check, i.e., invoking the entry code and checking to determine if the thread already has the object.

This optimization is implemented in our compiler and is useful for concurrent data structures with operations that call each other with indiscriminate abandon. Upgrades and indirectly recursive monitor calls must still dynamically check whether they already have access to the object, since that cannot be determined statically.

4.2.2.5 Tail Release Optimization

A common scenario in synchronization abstractions such as barriers is a monitor operation that performs a release on an event as the *last* statement in the function. If the monitor operation blocks on the release, then upon being signaled it would ordinarily reacquire the monitor object, only to exit it immediately. By recognizing the occurrences of such release operations in the compiler, we can optimize the release to directly resume execution after the monitor operation.

We do not currently implement this optimization in our compiler, since we do not maintain sufficient information to perform this analysis of the control-flow graph. However, this optimization is important for many common synchronization abstractions, since it saves the additional overhead of reentering and then exiting the monitor. This overhead of reacquiring the monitor is about 58 instructions when the monitor is available and is even larger otherwise.

4.2.2.6 Parallel Monitor Functions

The final optimization is targeted towards making efficient use of the stacks required for parallel tasks. As described before, the underlying server process fetches and executes tasks, allocating a stack each time a new task is fetched for execution. When the task being fetched corresponds to a parallel monitor function, then the task will try to acquire the monitor object as soon as it begins execution and perhaps block in the attempt. Therefore, a program that invokes lots of parallel monitor functions contending for a few

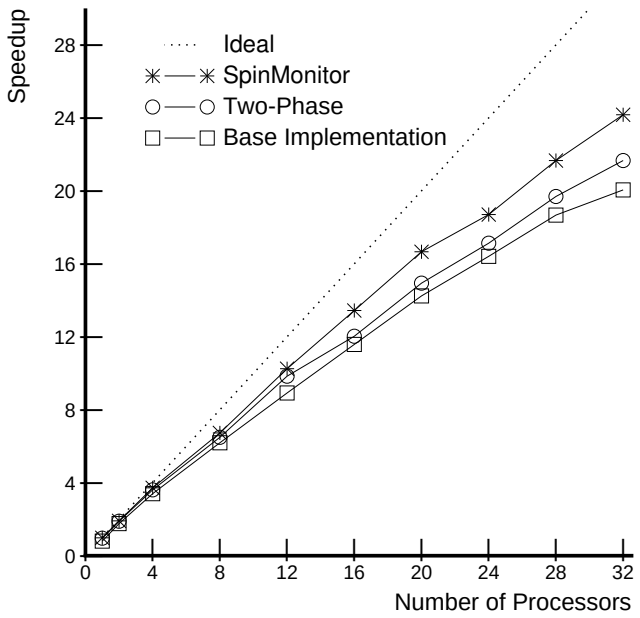

Figure 4.11: The effect of synchronization optimizations in the Water code.

monitor objects can easily find itself with hundreds or thousands of tasks suspended on a monitor, with each task consuming (and wasting) a stack that may easily be a megabyte or larger. We ran into this problem in the Panel and Block Cholesky applications and were unable to execute the applications since they ran out of memory.

We address this problem in the following fashion. The runtime system, instead of allocating a stack and finding that the task immediately attempts to acquire the monitor object, first acquires the monitor object on behalf of the task *before* allocating the stack. This ensures that the task doesn't grab a stack and then immediately block while acquiring the monitor. If the monitor object is unavailable, then the runtime system suspends the task waiting for the monitor and continues executing other tasks. When the task finally acquires the monitor and is woken up, then the runtime system allocates a stack for the task and resumes execution *inside* the monitor.

We have implemented this optimization in our runtime system and have found it useful for efficient storage management in programs such as Panel/Block Cholesky that would otherwise consume huge amounts of memory.

4.2.3 Evaluating the Optimizations in Applications

We now examine the applicability of the optimizations proposed above to real application programs and the corresponding performance benefits.

As discussed earlier in Chapter 2, monitors are most often used in two ways. First, they are used to express synchronization for shared data by serializing multiple accesses

Table 4.2: Time to execute a barrier for different numbers of processors (in microsecs).

Num Processors	1	4	8	12	16
Base Implementation	7.0 us	275 us	1280 us	2502 us	3780 us
After Optimization	2.6 us	80 us	385 us	690 us	1030 us

to a shared object. Second, monitors are used to build synchronization abstractions such as a barrier, exclusive access to multiple objects (Chapter 2.3.3), and object-level synchronization between tasks (Chapter 2.3.2). Monitors in this category typically consist of small operations that check some status conditions and perhaps wait on an event. The optimization techniques described above apply to each of these situations. For instance, the Water, Panel, and Block Cholesky applications use monitors to protect shared data, such as a molecule in Water and a panel/block of a matrix in Cholesky; these monitors are optimized to be simple spin-monitors. The Ocean application, on the other hand, uses monitors to build a barrier abstraction; such synchronization abstractions benefit from several optimizations—they can often be implemented as spin monitors, synchronization for private condition variables can be discarded, and the operations usually do not need to reenter the monitor after a release operation.

We present performance results for two applications, Water and Ocean. In the Water code each water molecule along with its many physical properties is declared to be a monitor. Figure 4.11 shows the performance of the Water code executing on a thirty-two processor DASH multiprocessor. We present three different versions of the code. In the base implementation we use blocking monitors (i.e., a monitor operation blocks immediately if a monitor is unavailable). In the two-phase implementation a monitor operation spins for a while for an unavailable monitor and then blocks. Finally, in a pure-spin monitor each operation keeps busy-waiting until it acquires the monitor. As we can see, the two-phase implementation performs better than a blocking monitor. In addition, since the critical sections of the monitor operations on a molecule are small, implementing the monitor as a spin monitor further improves performance by over 20% as compared to a blocking monitor.

The Ocean code uses a monitor to build a barrier abstraction as shown in Figure 4.12. The monitor has a single mutex barrier function which counts the number of threads that have arrived at the barrier. The first $N-1$ threads to arrive release the monitor and wait for the other threads to arrive, while the last (N^{th}) arriving thread wakes up all the earlier threads and continues execution after reinitializing the barrier. As shown in the right half of the figure, this abstraction can be implemented very efficiently. The compiler determines that the mutex function is tiny and implements it as a spin monitor with a single lock/unlock for the entry and exit operations. The condition variable $synch$ is private to the class and is implemented as just a queue of tasks with no additional synchronization. Finally, since the release statement is the last statement in the barrier operation, threads resuming after the release operation do not need to reenter the monitor. Incoming threads block on the condition variable until all the threads have arrived, at which point they resume execution past the barrier.

```
// Barrier abstraction in COOL.
class barrier_t {
  int total, count;
  cond synch;

public:
  void barrier_t (int P) {
    total = P;
    count = 0;
  }
  mutex void barrier () {


    if (++count == total) {
      // I am the last to arrive.
      // reinitialize monitor.
      count = 0;
      // wake everyone up

      synch.broadcast ();


    }
    // Wait for others to arrive
    else synch.release ();
  }
};
```

```
// Optimized implementation of the barrier.
class barrier_t {
  int total, count;
  Lock_type lk;
  Queue_type Q;
public:
  void barrier_t (int P) {
    total = P;
    count = 0;
  }
  void barrier () {
    // Entry code is just a lock.
    AcquireLock (lk);
    if (++count == total) {
      // I am the last to arrive.
      // reinitialize monitor.
      count = 0;
      // wake everyone up
      // No synchronization needed for queue
      Q.broadcast ();
      // Exit code is just an unlock
      ReleaseLock (lk);
    }
    // no need to reenter the monitor.
    else Q.sleep (lk);
  }
};
```

Figure 4.12: A barrier abstraction in COOL and its implementation.

To evaluate the effect of these optimizations on the overheads of the barrier, we measure the time taken for a set of processors to participate in a barrier. (For better accuracy we measure the time taken by several thousand instances of encountering the barrier: we create one COOL task per processor, where the task continually loops around a barrier and does no other work, thereby avoiding any load imbalance.) Table 4.2 shows the average time taken to complete a single instance of the barrier before and after optimization for different number of processors. As we can see, the optimized barrier is more than three times faster than the general implementation.

The performance of the Ocean application with and without these optimizations of the barrier abstraction is shown in Figure 4.13. The optimizations make a dramatic improvement in performance. Apart from the streamlined overhead, the gains are primarily due to large reductions in the number of times that a task needs to block. From the

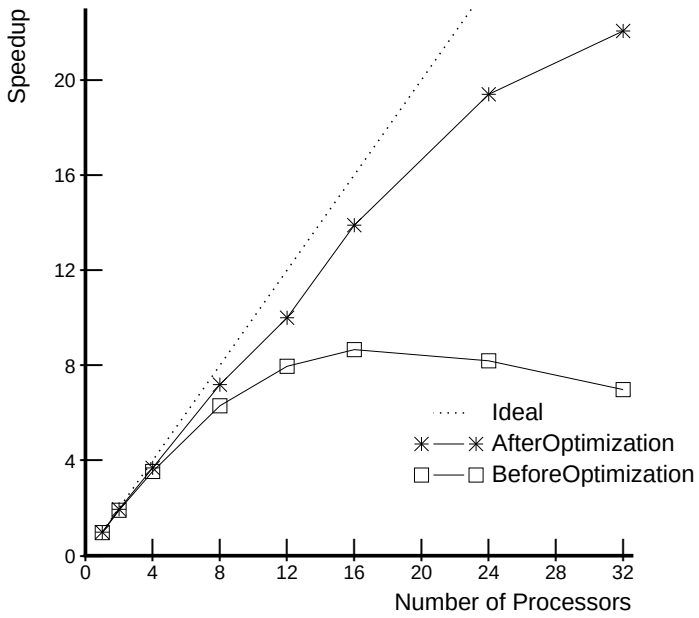

Figure 4.13: The impact of optimizing the barrier abstraction in the Ocean code.

barrier abstraction in Figure 4.12, $N - 1$ tasks need to block at each barrier waiting for the last task to arrive. In the naive implementation of the barrier, however, a task may also block within (a) the entry code for acquiring access to the monitor, as well as (b) the code to reenter the monitor when the task is resumed after the release operation. In the optimized monitor, the entry code for the monitor is implemented with busy-waiting, while the reentry code is discarded entirely and the task resumes execution directly beyond the barrier. The efficient synchronization, therefore, leads to large improvements in performance in the Ocean application.

Finally, the performance results for the Water and Ocean applications present the speedup of the parallel application relative to a serial version of the program running on a single processor. Looking at the speedup on one processor, i.e., the performance of a parallel program on one processor, gives us an indication of the COOL overheads incurred in the application. As we can see the COOL overheads are quite low; these COOL applications run about 3% slower than the corresponding serial version of the application.

4.3 Runtime Scheduling Support for Locality Optimizations

In this section we give an overview of the task queue structure and the scheduling mechanisms used in the COOL runtime system to support the scheduling optimizations

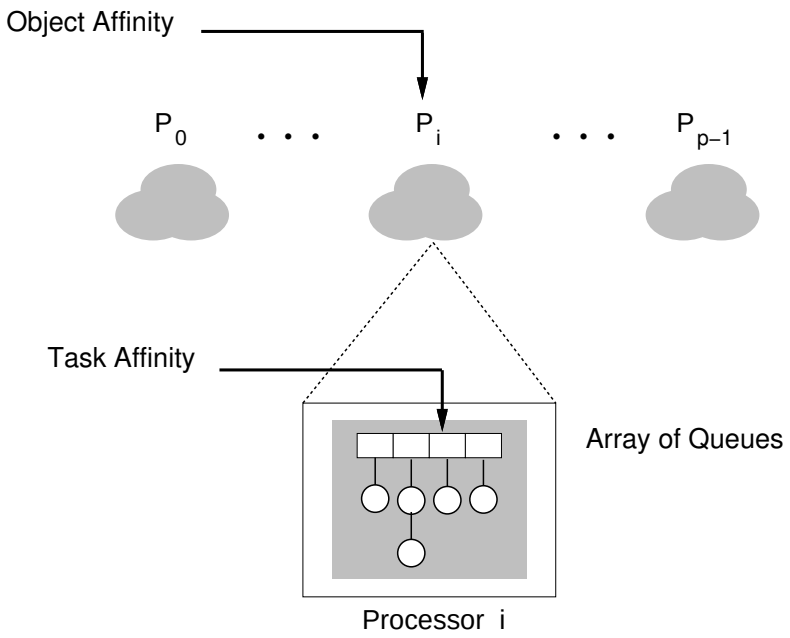

Figure 4.14: Task queue structure to support task and object affinity.

described in Chapter 3. The optimizations required are scheduling for each of object and task affinity. When a task has *object affinity* for an object, the runtime system should try to execute it on the processor that contains that object allocated in its local memory. To support object affinity we have a task queue per processor and a task with object affinity is collocated with the object (see Figure 4.14). When several tasks have *task affinity* for an object, the runtime system should try to execute these tasks in a back-to-back fashion on a processor. To support task-affinity, each processor must distinguish tasks that have affinity for the same object. Therefore each server actually has an *array* of queues, with each element of the array corresponding to a set of tasks.

Given the above task queue structure, when a task is created with affinity specified for some object, we schedule it on the processor that has the object allocated in its local memory. In addition, within the array of queues for that processor, the task is enqueued on the queue determined by hashing the object's virtual address onto an element of the array. The hash function is a simple modulo operation of the object address with the size of the array. If both task and object affinity are specified for a task, then the task is scheduled on the processor determined by the object affinity hint, i.e., on the processor that contains the specified object in its local memory. However, within that processor's array of task queues, it is scheduled within a bucket based on the object specified in the task-affinity hint, thereby identifying tasks in that task affinity set. This allows us to simultaneously exploit both cache and memory locality on different objects, as illustrated by the example in Chapter 3.3. Tasks in the same task-affinity set therefore get mapped onto the same queue, and can be serviced back-to-back. Collisions of different task-affinity sets on the same queue can be minimized by choosing a suitably large array size.

To determine the processor where an object is allocated for object affinity, the runtime system makes a call to the operating system. On DASH this system call takes a virtual address and returns the number of the processor that contains the physical page corresponding to the supplied virtual address. To avoid the cost of the system call, we can implement a *user-level cache* of the locations of the pages in the program. This cache could be kept updated by invalidating the corresponding entry each time a page was explicitly migrated from one processor to another, or swapped in or out by the operating system. We do not currently implement this optimization, but this is a simple way to avoid the overhead of expensive system calls.

This task queue structure is implemented efficiently. Determining where to schedule a task simply requires two modulo operations. Within the task queue structure of each processor, the non-empty queues in the array are linked together to form a doubly-linked list and provide fast enqueue and dequeue operations.

4.4 Summary

In this chapter we have described the COOL implementation and its two primary components: the front-end translator that replaces each COOL construct with equivalent C++ along with calls to library routines, and the runtime system that provides the execution environment and implements the concurrency and synchronization for each COOL construct. We have also described the task-queue structures and scheduling mechanisms that support the scheduling optimizations outlined in the previous chapter. Finally, we have described the techniques employed in the implementation to support each construct efficiently. We have outlined compiler techniques that analyze individual monitors in a COOL application program and tailor the implementation of each monitor based upon the functionality that it uses. As a result of these optimizations, the overheads of the COOL constructs are quite low; a parallel COOL program on a single processor runs with only 2-3% slowdown.

Chapter 5

Case Studies

In Chapter 2 we evaluated the expressiveness of COOL by studying several benchmark applications coded in the language. We now evaluate the efficiency of the COOL implementation by analyzing the performance of these applications. We also evaluate the effectiveness of the COOL approach to improving data locality. For each of the applications described in Chapter 2, we analyze the locality and load balancing issues in the application, and we show how the affinity hints can be used to optimize the distribution of tasks and obtain good performance. We analyze the performance impact of supplying these hints and contrast the programming effort required in our approach with performing these optimizations explicitly, as is necessary in other systems such as the ANL macros.

Through this analysis we evaluate the following aspects of our approach. First, do the hints provide the programmer with an adequate level of control over the optimization process? Are they powerful enough to exploit locality at each level in the memory hierarchy? Second, are the hints flexible—i.e., do they allow the programmer to easily experiment with different optimizations? Finally, does the underlying task and object structure of the program make it easier to supply these hints?

We present results from the six SPLASH [96] applications described earlier in Chapter 2. The first application, Ocean, has a simple structure, and default affinity between tasks and objects is sufficient to improve both cache and memory performance. In the next three applications—Water, LocusRoute, and Barnes-Hut—memory locality is less important and scheduling related tasks on the same processor is sufficient to get good cache locality. However, these applications have very different structures—in Water the computation is very regular and there is a direct mapping of tasks to processors; in contrast, in LocusRoute identifying related tasks to schedule for cache reuse is non-trivial and requires a deep understanding of the application semantics; and in Barnes-Hut the scheduling of related tasks is complicated by load-balancing concerns. The final two applications, Panel and Block Cholesky, are more complex and each requires careful

placement of objects and tasks to improve both cache and memory locality.

We present performance results running on a prototype of the Stanford DASH multiprocessor [74], consisting of thirty-two 33 MHz R3000 processors. The processors are organized into (eight) clusters, with each cluster containing four processors and some physical memory (28 MB each). Each processor has a 64 KB first level cache and a 256 KB second-level cache. References that are satisfied in the first-level cache take a single processor cycle, while hits in the second-level cache take about 14 cycles. The cache line size is 16 bytes. Memory references to data in the local cluster memory take nearly 30 cycles, while references to the remote memory of another cluster take about 100-150 cycles. We measure the time spent in the parallel portion of the code and plot its speedup with respect to the time taken by a serial version of the code running on one processor. We also use the hardware performance monitor on DASH [75], that enables us to monitor the bus and network activity in a non-intrusive manner. In particular, the performance monitor allows us to track the number of local and remote cache misses incurred during the execution of an application.

5.1 Ocean

We first study the Ocean [95] application, described earlier in Chapter 2.4.2. This case study illustrates how a simple object distribution together with the default affinity works well in improving data locality.

Figure 5.1 presents the COOL code for Ocean along with the necessary locality optimizations described below. As part of initialization, the COOL programmer explicitly distributes the rowGroups of the grids across the memories of the processors, so that corresponding rowGroups of different grids are allocated within the same local memory (see the function *distribute*). The runtime automatically schedules tasks for locality on the rowGroup objects, based on their default affinity. As a result both intra-grid and inter-grid tasks find the data they reference in local memory.

Upon running the application as described above, we found that although the tasks were enqueued appropriately, they almost never executed on the desired processor. (The COOL runtime system monitors the scheduling and stealing of tasks during the execution of the program and prints out these aggregate statistics when the program completes execution.) It turned out that most of the tasks were stolen within the COOL runtime for load-balancing. While the main thread was creating tasks on the queue of each processor, an idle processor would steal another processor's task even though its own task is either on the way or perhaps had already arrived. Although heuristics such as not stealing from an idle processor or spinning for a while after first noticing a remote task mitigate the problem somewhat, they do not solve the problem entirely. For instance, not stealing from an idle processor has the problem that the processor executing the main thread is not idle and a task scheduled on its queue may get stolen before the processor finishes executing the main thread. Given the limitations of such heuristics, we used the *noStealing* runtime flag (described earlier in Chapter 3.3.6) to disable task-stealing entirely.

```
class row_c {
  double element[numElements];
};

class rowGroup_c {
  row_c myrow[numRows];
  . . . other data, such as row indices . . .
public:
  parallel condH* laplace (rowGroup_c*);
  parallel condH* sub (rowGroup_c*, rowGroup_c*);
  parallel condH* mult_add (int, rowGroup_c*, int, rowGroup_c*);
  parallel condH* jacobi (rowGroup_c*);
  . . .
};

class grid_c {
  rowGroup_c rowGroup[numGroups]; // grid composed of rowGroups
  . . .
public:
  void distribute () {
    for (i=0; i<numGroups; i++)
      migrate (rowGroup+i, i);
  }
  void laplace (grid_c* p) {
    waitfor {
      for (all rowGroups 'i' do)
        // Process all the rowGroups in parallel.
        rowGroup[i].laplace (&(p->rowGroup[i]));
    } // wait for the operations to complete over all rowGroups.
  }
  . . . other grid operations . . .
};

main () {
  extern int noStealing;
  grid_c A, B, C, D;
  A.distribute ();
  B.distribute ();
  C.distribute ();
  D.distribute ();

  // Disable task stealing
  noStealing = 1;
  . . .
  for (all time steps) {
    A.laplace (B);
    C.jacobi (D);
    . . . other grid operations . . .
  }
}
```

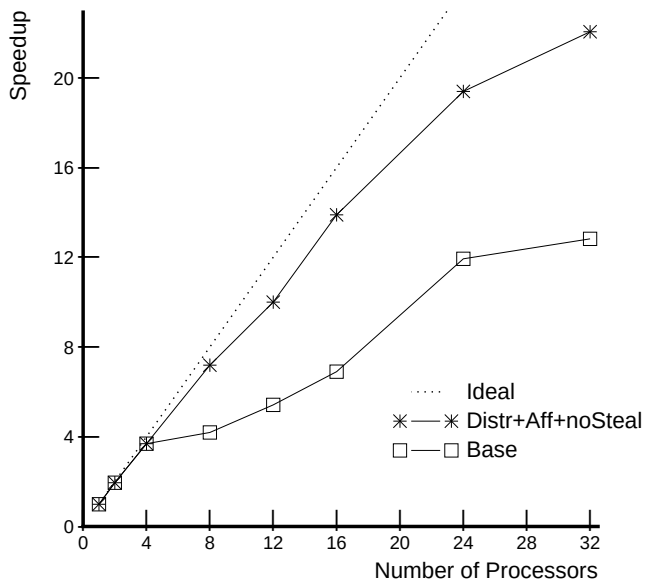

Figure 5.2: Performance improvement with affinity hints and noStealing for Ocean.

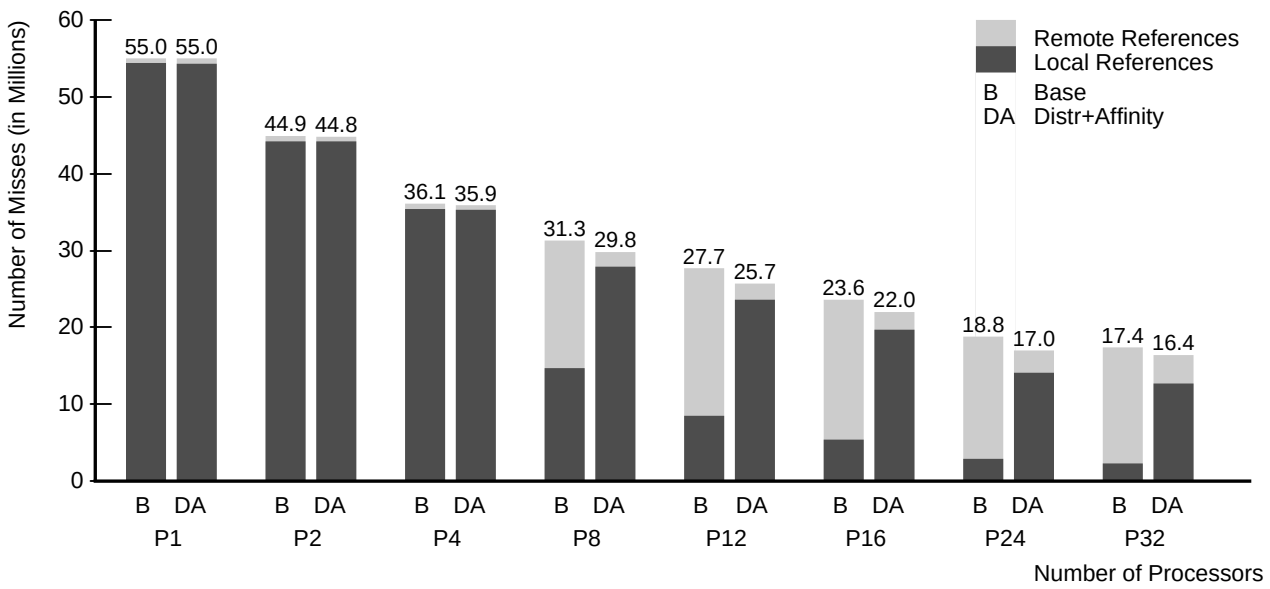

Figure 5.3: Cache miss statistics for Ocean, before and after affinity optimizations.

Figure 5.2 shows the performance results for the Ocean code on a square grid of 194x194 points, which takes about 105 seconds on a single processor (the input problem is kept fixed throughout).[1] Performance improves dramatically (75-100%) with the hints described above, compared to the base version in which tasks are scheduled on a single task queue. The statistics gathered with the hardware performance monitor on the 32-processor DASH are shown in the bar charts in Figure 5.3. The figure plots the total number of cache misses before (labeled B) and after optimization (labeled DA) during the execution of the parallel portion of the program, for different number of processors. Each bar corresponds to the total number of caches misses and is partitioned into two regions. The dark region represents the cache misses that were serviced in local memory while the light region corresponds to misses that were to data in remote memory. As shown in the figure most misses are to remote memory before optimization, but with the optimizations nearly all (80-90%) of the cache misses are satisfied in local memory. Thus the primary performance gains are due to the objects being referenced in local rather than in remote memory.

Recall that we chose to partition a grid into a single *array* of rowGroups. However, for some grid operations such as relaxed SOR that require inter-processor communication across rowGroups, the best communication to computation ratio is obtained if each grid is partitioned into square blocks rather than groups of entire rows. In a blocked decomposition, however, the elements of a block are no longer contiguous in memory making it difficult to distribute them across processors' memories. A common way of addressing this problem is to organize a two-dimensional grid as a four-dimensional array, with the first two dimensions referencing a particular block and the last two dimensions referencing the particular element within the block. However, a blocked decomposition is more complicated to express, regardless of the underlying language. We therefore used a simpler decomposition in our experiments.

To summarize, even though the problem can be statically decomposed, this example demonstrates the ease with which the program was optimized. The COOL program is decomposed into tasks on rowGroup objects, and the default affinity successfully improves both cache and memory locality. Explicit distribution of objects is required of the programmer, but is easily expressed through the migrate statement. Finally, experience with this example suggests an automatic object distribution strategy in which the objects associated with a parallel function are distributed in a round-robin manner across the local memories of processors. This appears to be a reasonable default and would be sufficient for programs with a regular structure. For more complex programs the programmer could always override this default with explicit object distribution.

5.2 Water

[1] The ANL version of the code has not been ported to DASH, hence we cannot compare its performance to the COOL program. However, the structure of the ANL code is similar: each processor operates on a set of rows, with a global barrier at the end of each grid operation to synchronize the processors. We therefore expect their performance to be similar as well.

```
class mol_c {
  atom_c atom[3];
  . . . physical properties of the molecule . . .
public:
  . . . functions to process the molecule . . .
};

class molGroup_c {
  mol_c* mymols; // molecules within this group.
  . . .
  void predic ();
  void intraf (double_s*);
  void divForce ();
  void correc ();
  void bndry ();
  void kineti ();
public:
  // Parallel functions to process molecules within the group.
  parallel condH* predic_intraf (double_s* poteng) {
    predic (); intraf (poteng);
  }
  parallel condH* interf ();
  parallel condH* divForce_correc_bndry_kineti () {
    divForce (); correc (); bndry (); kineti ();
  }
  . . .
} *molGroup;

main () {
  extern int noStealing; noStealing = 1; // Disable task stealing
  . . .
  for (all time steps do) {
    // Perform the different phases.
    waitfor {
      // process all the molecule groups in parallel.
      for (all molecule groups 'i')
        molGroup[i].predic_intraf () [ affinity (PROCESSOR, i);];
    }

    waitfor {
      for (all molecule groups 'i')
        molGroup[i].interf () [ affinity (PROCESSOR, i);];
    }

    waitfor {
      for (all molecule groups 'i')
        molGroup[i].divForce_correc_bndry_kineti () [ affinity (PROCESSOR, i);];
    }
  }
  . . .
}
```

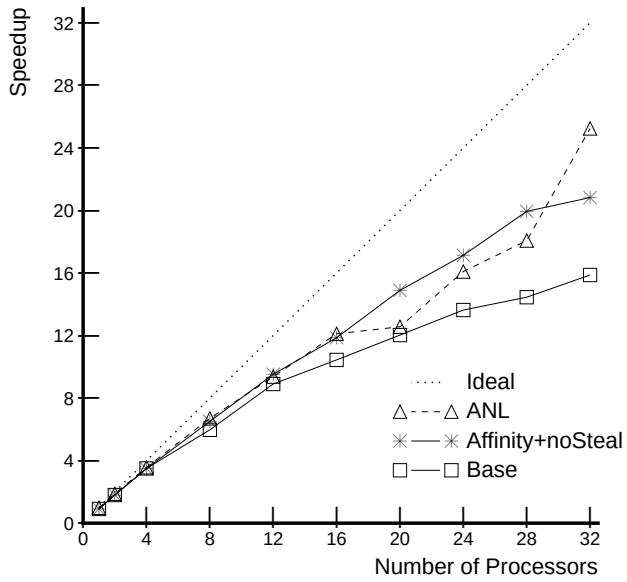

Figure 5.5: Performance improvement with affinity hints and noStealing for Water.

We next study the N-body Water application [94, 96], described earlier in Chapter 2.4.1. This application does not require data distribution and illustrates how scheduling related tasks for cache reuse is often sufficient to obtain good performance.

Figure 5.4 presents the COOL code for Water along with the affinity hints as described below. Since there is a uniform amount of work associated with processing each molecule, the load-balancing concerns are addressed by creating as many molecule groups as processors, each containing the same number of molecules. Regarding data locality, in most of the phases the data references are to the values stored within the individual molecule object being processed. In those phases each task primarily references the molecules within its group. One exception is the pairwise interaction phase that models the interaction between each pair of molecules in the system, and therefore requires all-to-all communication. We would like to exploit cache locality by processing tasks on the same molecule group on the same processor across different phases of the program, thereby reusing the molecule objects in the cache. To determine whether we need to exploit memory locality as well we do the following simple calculation. Each molecule consists of about 700 bytes of data, and typical inputs consisting of hundreds of molecules consume less than a mega-byte of data. Therefore the data set for each processor (a molecule group) is expected to fit within the processor's cache (64KB first-level cache, 256KB second-level cache); locality within the cache should be sufficient, and no data distribution appears to be necessary.

In the previous application (Ocean), row groups were distributed across processors and tasks were automatically scheduled based on default affinity. No special data distribution is performed in Water, however, and the desired scheduling must be explicitly specified through affinity hints for the tasks. Since we create one task per processor in each phase,

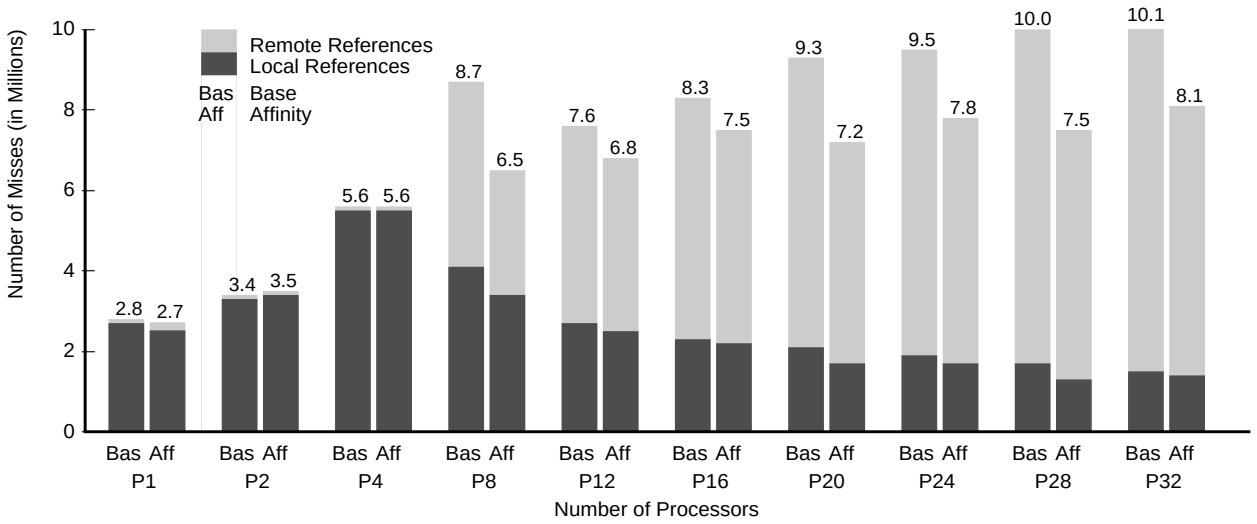

Figure 5.6: Cache miss statistics for Water, before and after affinity optimizations.

it is simplest to schedule the tasks directly onto processors using processor affinity. A task on the i^{th} molecule group is therefore scheduled on the i^{th} processor using the affinity hint shown in Figure 5.4. Multiple tasks on the same molecule group across different phases therefore execute on the same processor and benefit from cache reuse. The affinity hint is more easily supplied at the invocation of the parallel function rather than its declaration since the index of the molecule group (i) is trivially accessible at the function invocation. Finally, similar to the Ocean code, since the load-distribution is determined statically with one molecule group per processor, any task stealing can wreck the careful distribution of tasks described above. We therefore use the noStealing flag to disable task stealing during the execution of the program. As shown in Figure 5.4, the affinity hints and toggling the noStealing flag are all that is required to obtain the above scheduling optimizations.

The ANL version of the code differs from the COOL program in two aspects. First, since the program can be statically partitioned, in the ANL code a molecule group is assigned to a process for the entire computation. Synchronization for a phase to complete is expressed using a barrier between all processes in the system. In contrast, the COOL program creates tasks in each phase and waits for them to complete using a waitfor. A second difference is the organization of the molecule data structures. Compared to the array of molecule objects in COOL, the ANL code instead has multiple arrays, one for each physical property of the molecules. The latter organization exhibits better spatial locality within a cache-line, since the phases typically reference a few properties for all the molecules rather than referencing several properties for each individual molecule.

We evaluate the performance of the Water code on an input of 512 molecules over a period of 4 time steps. This problem takes about 110 seconds on a single processor. Figure 5.5 plots the speedup of the program over the serial version of the code and presents three curves: the Base version is the COOL program without any affinity hints in which tasks are scheduled based on default affinity. The molecule groups are allocated

from the memory of the processor that first initializes them during the serial startup portion of the program, hence all tasks subsequently get scheduled in the queue of that particular processor. The Affinity curve presents the performance of the program with the affinity hints described above, and the last curve presents the performance of the ANL version of the code.

As shown by the figure, the application performs well and exhibits good speedups (over a factor of 20 on 32 processors). Performance improves significantly (over 30%) with affinity scheduling compared to the Base case, reflecting the improved cache reuse. Finally, the performance of the ANL and COOL codes is comparable, with the ANL code starting to perform better for larger numbers of processors. This reflects the two reasons outlined earlier—the task creation overhead in COOL and the improved spatial locality in the ANL code.

Figure 5.6 presents the cache miss statistics for the two COOL versions of the code, with and without affinity optimizations. The total number of cache misses decreases by 20-30% with affinity scheduling, reflecting the improved cache locality on the molecule groups. The total number of cache misses is quite low and consume less than 10 seconds[2] in a program that executes for over a 100 seconds on a single (33MHz R3000) processor. Many of these remaining misses are likely to be true communication misses in the pairwise interaction phase of the computation, that cannot be optimized by either cache or memory locality. Therefore, data distribution is not likely to help in this application.

In summary, the Water code is an example of an application that can work well in its cache. For such applications simply scheduling related tasks on the same processor is sufficient for good cache reuse, and no data distribution is necessary. The desired scheduling was easily expressed using processor affinity in COOL. The water code had a one-to-one mapping of tasks to processors, and the ability to disable task stealing was necessary for affinity scheduling to be effective; this level of control appears to be essential for applications where a good load-distribution can be determined statically, and the programmer is willing to control load-balancing directly, as in Ocean and Water.

5.3 LocusRoute

LocusRoute [89] is a parallel routing algorithm for standard cell circuits and was described earlier in Chapter 2.4.3. Similar to the Water code, scheduling of related tasks is important for cache locality. However, this application is one where tasks are related indirectly through a *portion* of a shared object (the CostArray), and identifying the related tasks requires a semantic understanding of the application.

Figure 5.8 reproduces the code to express LocusRoute in COOL from Chapter 2.4.3, augmented with the locality optimizations described below. The algorithm spends most of its time in evaluating the cost of different routes for a wire, so locality on the CostArray is important. We can express this by viewing the CostArray as partitioned into geographical

[2]Less than 10 million misses, with each miss taking approximately 1 microsecond to service.

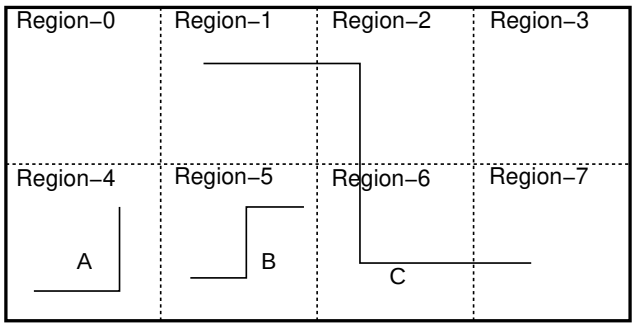

Figure 5.7: The CostArray composed of regions.

regions that correspond to a spatial partitioning of the circuit as shown in Figure 5.7 and have each processor route wires that lie within its region of the CostArray. Executing tasks within a region on the same processor will hopefully reuse that region of the CostArray in the cache. To simultaneously distribute the load across processors as well, we actually exploit this locality using processor affinity: each region of the CostArray is (conceptually) assigned to a processor, and tasks within that region are directly scheduled on the corresponding processor. Besides reusing the same region in the cache, each region is likely to be referenced by only one processor, thereby reducing the invalidations of the CostArray in caches of other processors. Finally, depending on the degree of reuse, physically distributing the regions of the CostArray across the memories of different processors may further improve performance, since the misses to the CostArray will be to local rather than remote memory.

The code to express these hints is shown in Figure 5.8. The function *Region(CurrentWire)* computes the midpoint of the wire and returns the region number that the midpoint lies within, which is then used as the server number in the processor-affinity hint. Partitioning the CostArray into a few large regions (say one per processor) will have better locality but perhaps poorer load balance, while larger numbers of smaller regions will have better load balance at the expense of data locality. These tradeoffs can be easily explored in the COOL program by varying the Region function. Finally, task stealing remains enabled to correct any load imbalance across regions.

Since we had only small input circuits available to us, we demonstrate our technique using a synthetically constructed input consisting of a dense network of wires within the regions of the circuit. This circuit consists of 8160 wires uniformly distributed in a circuit that is 1760 cells wide and 20 channels deep. The circuit can be thought of as composed of 32 regions of size 110 cells by 10 channels, with 255 wires distributed within each region. This problem takes about 100 seconds on a single processor. The performance results in Figure 5.9 show (a) the *Base* version of the program in which the wire tasks are scheduled across processors in a round-robin fashion without regard for locality, (b) the *Affinity* version in which processor affinity hint is supplied so that wires in a geographical region are likely to be routed by the associated processor, and

```
// Function to determine the CostArray region that the
// wire lies in, based on the mid-point of the wire.
int Region (Wire*);

class Wire {
  . . .
public:
  // Parallel function to route a wire.
  parallel condH* Route ()
    [ affinity (Region (this), PROCESSOR); ];
};

main () {
  . . .
  while (not converged {
    waitfor {
      for (all wires 'w' do) {
        // route the wires in parallel.
        w->Route ();
      }
    } // Wait for all wires to be routed.
  }
}
```

Figure 5.8: Specifying affinity hints in LocusRoute.

(c) the *Affinity+ObjectDistr* version in which the regions of the CostArray are physically distributed across processors' memories as well. Overall speedups are small due to the high degree of communication of shared data. However, with the affinity hints most of the wire tasks (over 80%) in a region are routed on the corresponding processor, resulting in significant performance improvements. As shown by the cache miss statistics in Figure 5.10, affinity scheduling nearly halves the number of cache misses. Distributing the CostArray improves performance further, although the gains are smaller. The number of cache misses remain unchanged but more of them are serviced in local rather than remote memory.

In contrast to the COOL program where the programmer only had to supply the affinity hints, the ANL version requires the programmer to explicitly maintain task queues per processor and manage task creation and scheduling based on the geographic locality described above. The COOL program outperforms the ANL code due to some additional optimizations for dynamic memory allocation—as discussed earlier in Chapter 2.4.3, several items of scratch storage are dynamically allocated using malloc/free in the ANL version of the code, while in the COOL code we maintain a free-list of such scratch storage. The COOL code is therefore more efficient in this regard. To factor out the effect

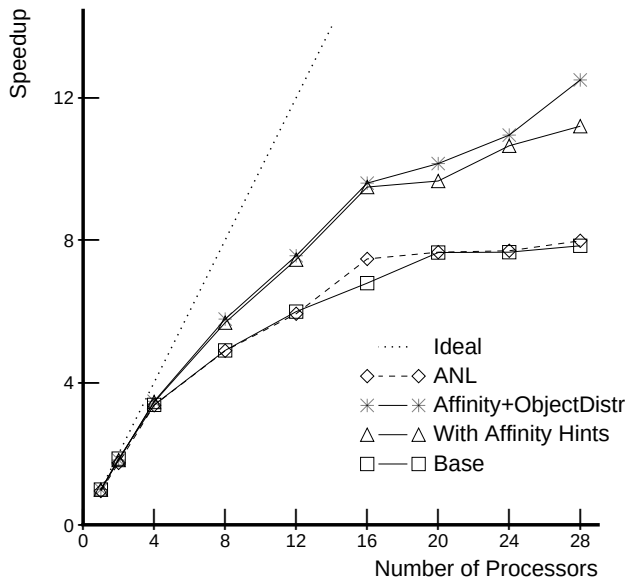

Figure 5.9: Performance improvements in LocusRoute with affinity hints.

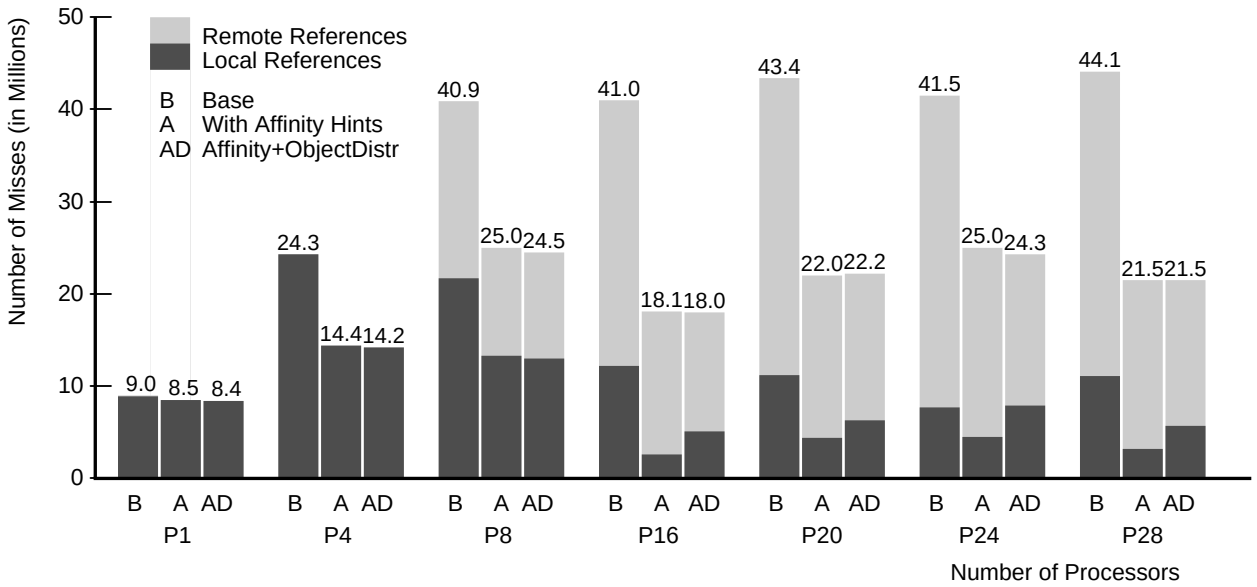

Figure 5.10: Cache miss statistics for LocusRoute, and the affect of the affinity optimizations.

of these optimizations, we present self-relative speedup for the ANL code in Figure 5.9. Even so, the COOL program exhibits better speedup, especially for larger number of processors.

While the actual affinity hints supplied are simple, it is important to realize that the hints are based on insights about the semantics of the application and would be impossible for a compiler to deduce. For instance, a compiler simply cannot infer that wire objects have end-points corresponding to a region in the CostArray and have affinity for that region. Programmer intervention is therefore necessary for such programs.

5.4 Barnes-Hut

We next discuss the N-body Barnes-Hut [93] application, described earlier in Chapter 2.4.6. This application performs well with good cache locality, and data distribution—which is difficult in this application—is not necessary. However, Barnes-Hut has complex load-balancing requirements, and it was necessary to explore several scheduling alternatives to study the tradeoff between locality and load-balance in this application; we show how the affinity hints in COOL enabled the programmer to easily experiment with different strategies.

Figure 5.11 presents the COOL code to express Barnes-Hut along with affinity hints as described below. We focus on the force computation phase since that is where the program spends most of its time. The two main data structures in the program are body objects that represent the bodies in the system and the tree of space cells representing a hierarchical partitioning of physical space. Concurrency during the force computation phase is exploited by processing different bodies in parallel. Computing the force on a body mainly requires references to (a) the bodies that interact with the current body, and (b) the space-cells in the tree as the tree is traversed to locate the nearby and distant bodies. Cache locality on these data structures can be improved by processing bodies that are close together in space on the same processor—nearby bodies are likely to interact in the same manner with other bodies (i.e., either directly or with the center of mass), so that the force computation for nearby bodies is likely to reference the same bodies as well as the same portions of the tree. However, the bodies move in space as the system evolves over a period of time-steps, therefore the "nearby" relationship changes over time. Furthermore, while assigning nearby bodies to the same processor, we must be careful to simultaneously distribute the load uniformly across processors.

The ANL code determines a partitioning of bodies across processors at the beginning of each time-step. The code experiments with a variety of partitioning schemes based on the criteria outlined above. However, the bodies remain assigned to the same processor for the entire duration of the time-step. In contrast, the COOL code exploits concurrency at the granularity of an individual body, and tasks may be stolen dynamically for load-balancing reasons. We now describe each of those schemes, show how they were expressed in COOL (the ANL versions are mimicked by disabling task-stealing), and compare their performance based on the speedups presented in Figure 5.12. This figure

```
class body_c {
  . . .
public:
  parallel condH* computeInteraction ();
} *body;

main () {
  . . .
  for (all time steps) {
    . . .
    // Compute the N-body interactions.
    waitfor {
      for (all bodies 'i')
        body[i].computeInteraction ()
          // Static/Random: noStealing and the following affinity hint
          [ affinity (i/numProcs, PROCESSOR);];

          // Static/CountZones: noStealing and the following affinity hint
          [ affinity (i/(numBodies/numProcs), PROCESSOR);];

          // Static/CostZones: noStealing and the following affinity hint
          [ affinity (proc[i], PROCESSOR);];

          // Dynamic/CountZones: the following affinity hint
          [ affinity (i/(numBodies/numProcs), PROCESSOR);];
    }
    . . .
  }
}
```

Figure 5.11: Expressing concurrency and affinity in Barnes-Hut.

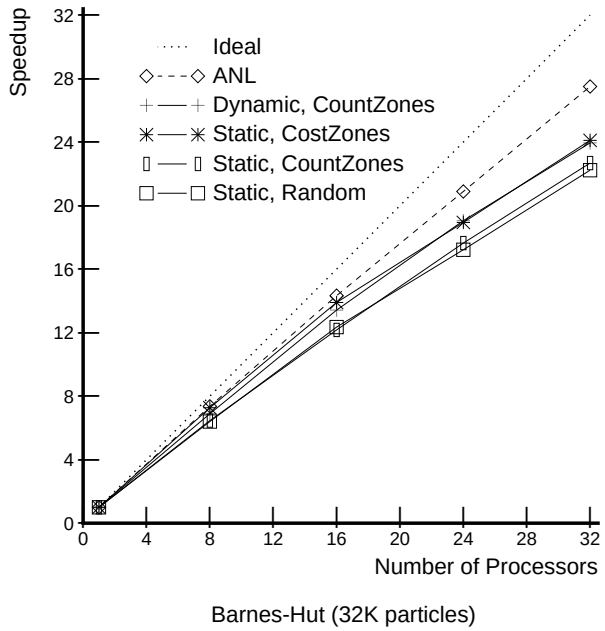

Figure 5.12: Barnes-Hut: The performance of different scheduling schemes.

presents results on an input of 32K bodies over two time-steps, that takes about 250 seconds to run on a single processor. (We did not collect cache miss statistics for this application.)

The first scheme *Static/Random* partitions the bodies equally across processors in a random fashion. As shown in Figure 5.11, this is expressed by the processor affinity hint with the tasks on each body being scheduled across processors in a round-robin fashion. Task stealing is disabled to mimic the assignment for an entire time-step. This scheme achieves reasonable load-balance but does not address any locality concerns.

The next scheme *Static/CountZones* addresses data locality by assigning nearby bodies to the same processor and divides the bodies equally among processors for load-balancing. This is expressed by the second processor affinity hint shown in Figure 5.11. The body array is organized based on the position of the body in space, so that bodies that are nearby in space are also close to each other within the body array. Therefore assigning adjacent elements of the body array to the same processor achieves the desired effect. However, although this scheme has good data locality, the amount of work associated with each body is not uniform, and partitioning the bodies equally across processors results in poor load-balance. The performance of this scheme is therefore only marginally better than the previous scheme.

The *Static/CostZones* was proposed to simultaneously address both locality and load-balancing concerns. This scheme assigns nearby bodies to the same processor similar to CountZones and at the same time tries to achieve good load-balance based on a fair partitioning of work rather than a simple body-count. It maintains a profile of the amount of work (specifically, a count of the number of body-body and body-cell interactions) performed for each body in the previous time-step and uses this work profile as a predictor

for the next time-step. Based on this work profile, nearby bodies are assigned to the same processor for good locality, and the load is balanced by choosing partitions with roughly the same amount of work rather than a body-count. This partitioning is maintained in the *proc* array in Figure 5.11, and expressed using processor affinity. Because it attempts to balance locality and load distribution in the program, the Static/CostZones scheme performs better than the other schemes described above.

The relatively complex CostZones scheme becomes necessary in the ANL code since bodies are assigned to processors for the duration of the entire time-step. In contrast, the COOL program exploits concurrency at a finer granularity (i.e., an individual body) and can benefit from dynamic load-balancing through task stealing. We therefore implemented the *Dynamic/CountZones* scheme, which is similar to Static/CountZones in that nearby bodies are processed on the same processor, but at the same time task-stealing is enabled to automatically correct any load-imbalance. This scheme is expressed by the last affinity hint in Figure 5.11 and is the same as the Static/CountZones hint but without disabling task-stealing. As shown in Figure 5.12, Dynamic/CountZones performs as well as the Static/CostZones scheme. Furthermore, compared to the CostZones scheme where the programmer explicitly maintains work profiles for each body, distributing bodies through task scheduling offers greater flexibility in experimenting with different partitionings as well as provides automatic load-balancing through task-stealing.

Finally, we also present results from the manually hand-tuned CostZones scheme in the ANL code in Figure 5.12. The performance of the COOL programs is slightly lower than that of the ANL code because the COOL program experiences poorer cache behavior in the center-of-mass computation phase due to differences in the organization of some data structures. However, within the different schemes, the automatic load-balancing in COOL is an attractive alternative to manual load distribution through explicit work profiles.

Overall, the example illustrates how the affinity hints in COOL allow easy experimentation with the various scheduling optimizations. It also illustrates how exploiting concurrency at the granularity of individual bodies enables better dynamic load-balancing.

5.5 Panel Cholesky

Our next case study is the Panel Cholesky application [91], which was described earlier in Chapter 2.4.4. This application requires careful task and data distribution to benefit from cache and memory locality; we illustrate how these requirements are met using the COOL constructs.

The COOL code expressing Panel Cholesky, along with locality hints, is shown in Figure 5.13. Most of the work is done in the updatePanel method, which reads the source panel and modifies the destination panel. By default, tasks corresponding to updatePanel have affinity for the panel that they are invoked on (the destination panel) and are automatically scheduled to exploit both cache reuse and memory locality on the destination panel. In addition, by distributing the panels across processors' memories we can both

```
class panel_c {
  int remainingUpdates;
  . . .
public:
  // Update this panel by the given source panel.
  parallel mutex void updatePanel (panel_c* src)
    [ affinity (src, TASK); affinity (this, OBJECT);] {
    . . . Update this panel by the given src panel . . .
    if (--remainingUpdates == 0) {
      // This panel is now ready.
      completePanel ();
  }

  // Perform internal completion of the panel.
  parallel condH* completePanel () {
    . . . perform internal completion . . .
    // Produce updates that need this panel.
    for (all panels 'p' modified by this panel)
      panel[p].updatePanel (this);
  }
} *panel;

main () {
  . . .
  // Distribute panels across processors memories,
  // in a round robin fashion.
  for (i=0; i<MaxPanels; i++) migrate (panel+i, i);
  waitfor {
    // Start with the initially ready panels.
    for (all panels 'p' that are initially ready)
      panel[p].completePanel ();
  } // Wait for all updates to complete.
}
```

Figure 5.13: Expressing concurrency and affinity in Panel Cholesky.

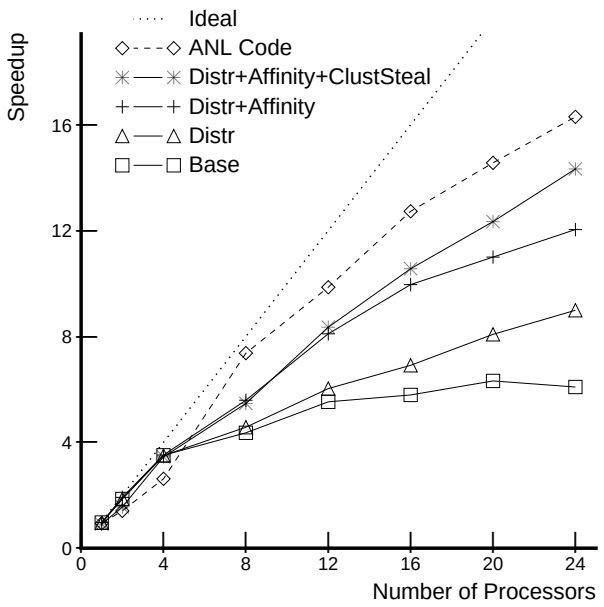

Figure 5.14: Performance improvement with affinity hints for Panel Cholesky (matrix BCSSTK33).

distribute the associated work and distribute the memory bandwidth requirements.

This scheme should perform well. Since the destination panel is being modified, it is preferable that modifications to a panel be performed on the processor that contains the destination panel in its local memory; otherwise the destination panel will continually be invalidated in different processors' caches. However, similar to the column-oriented Gaussian elimination example discussed in Chapter 3.3.3, we can also simultaneously exploit *cache reuse* on each source panel by identifying tasks with the same source panel as a task-affinity set. Each processor usually has multiple panels assigned to it that need to be modified by any given source panel. Executing these updates back-to-back on the processor will reuse the source panel in the cache.

These scheduling optimizations are easily obtained with the affinity hints shown in Figure 5.13. The panels are distributed across processors' memories in a round robin fashion. The default affinity for the base object automatically collocates tasks with the destination panel, while a task affinity hint supplied with updatePanel exploits reuse on the source panel.

The performance results are shown in Figure 5.14.[3] We use the matrix BCSSTK33 from the Boeing-Harwell set of sparse matrix benchmarks [32]. This matrix has 8738 columns organized into 1201 panels with approximately 7.28 columns per panel. This input runs for nearly 250 seconds on a single processor. The *Base* version is executed with no optimizations. *Distr* includes round-robin distribution of panels across processors, but

[3]We present results on up to 24 processors due to limitations in the amount of physical memory on the machine.

the tasks are scheduled randomly across processors. The improvement in performance is due to a better distribution of the memory bandwidth requirements. Performance further improves with scheduling tasks with affinity for the source and destination panels as described above, as shown by the plot *Distr+Aff*.

We now discuss an experiment we did with cluster-based scheduling. This experiment demonstrates the benefits of exploiting specific architectural features in improving data locality; the necessary optimizations, however, may require additional programmer abstractions and/or runtime support. Recall that although tasks are collocated with the destination panel on a particular processor, all processors within that cluster (in DASH) share the local memory containing the destination panel. We therefore ran the program with an idle processor allowed to steal tasks only from other processors within the same cluster; the stolen tasks would therefore continue to reference the destination panel in local rather than remote memory. This effect is controlled through a runtime flag that can be dynamically manipulated by the programmer. As shown by the plot *Distr+Aff+ClusterStealing*, stealing only within a cluster further improves performance.

Although in our experiments cluster scheduling was explicitly manipulated by the programmer, perhaps this effect can be automated within the runtime system. For instance, a runtime scheduler could try to always steal tasks from processors within the same cluster before trying to steal tasks from remote processors. Or, the scheduler may decide to steal tasks from remote processors based on the relative latencies of local and remote memory references. However, more experience is required before good defaults or automatic strategies can be developed; this example simply demonstrates the usefulness of such mechanisms.

As shown in the figure, the performance of the final COOL code is less than 10% slower than a hand-coded version of the code written using the ANL macros. Figure 5.15 displays the effect of the optimizations on the cache miss behavior of the program. Simply distributing the panels improves performance due to better utilization of the available memory bandwidth without affecting the cache performance. However, the proportion of misses that are serviced in local memory decreases somewhat from base to distr. Since data is allocated on a first-touch basis in the base case, most of it gets allocated from within one cluster. All tasks that execute within that cluster will therefore find their data in local memory. In the distr case, however, although the data is distributed across processors the tasks are distributed randomly. Hence the chances of a task finding its data in local memory are decreased. However, affinity scheduling and cluster scheduling significantly reduce the number of cache misses; in addition, since the tasks are collocated with the panel, more of the misses are serviced in local rather than remote memory.

In the second experiment we monitored the effect of task-affinity hints. We noticed that the program performed equally well with or without task affinity on the source panel (recall that task-affinity can be turned on with a simple affinity hint). So we took a closer look at the order in which tasks are executed on each processor. In the particular input matrix that we used, there just were not that many modifications per source panel (about 40). Furthermore, these modifications get split across different processors, based on the destination panel. Finally, even all these tasks don't necessarily execute back-to-back

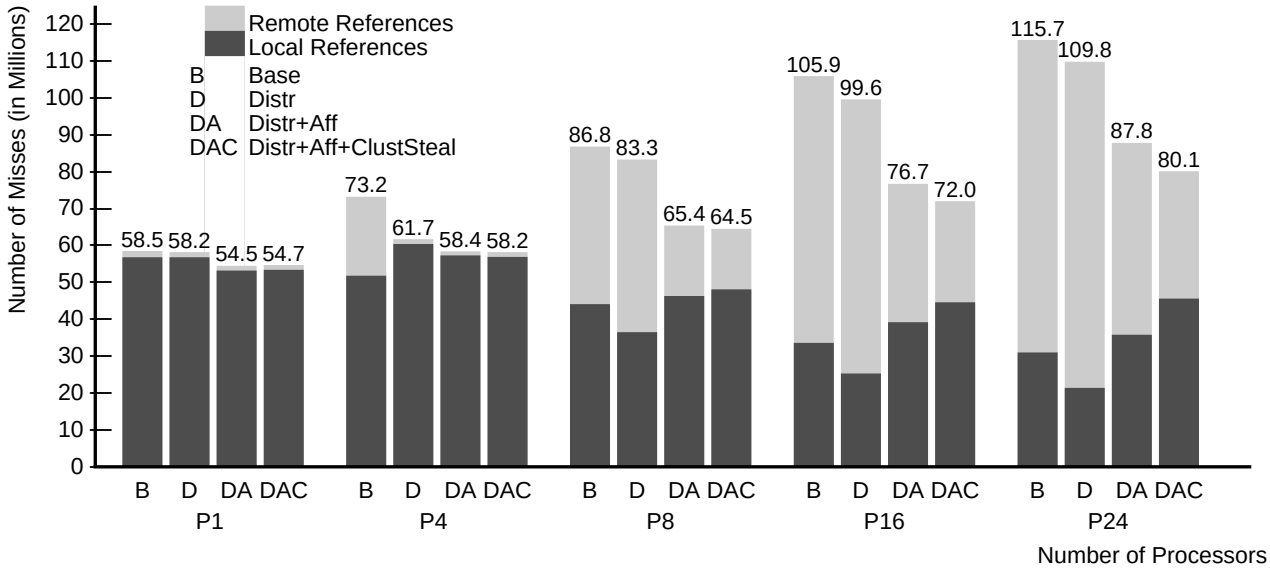

Figure 5.15: Cache miss statistics for Panel Cholesky, and the affect of the affinity optimizations.

since there are synchronization requirements. We instrumented the application to record a trace (over time) of the source panels used by tasks on each individual processor. This trace was post-processed to measure the number of consecutive tasks that used the same source panel, called the *run-length*. The run-lengths across all the processors were then averaged into a single number. We found that although the average run-lengths improved with task-affinity, the improvements were quite small. For instance, on 12 processors the run-length improved from 2.36 per source panel to 3.06 with task-affinity. The corresponding numbers for larger number of processors were—16 processors (up from 2.01 to 2.52), 20 processors (up from 1.77 to 2.17), and 24 processors (up from 1.60 to 1.96). With these small run-lengths—that get smaller as the number of processors increases—it is not surprising that we did not see significant performance gains. However, the scheduling hints were easily supplied and did achieve the desired effect; this effect may well be important for larger inputs to this application.

To summarize, the program was naturally expressed in terms of panel objects and panel-panel operations. While panels were explicitly distributed across processors, the scheduling defaults were sufficient to improve memory performance. Additional task affinity optimizations were easily expressed to simultaneously exploit cache reuse on source panels. Finally, the automatic object distribution strategy that we suggested for the Ocean code in Section 5.1 could adequately distribute panels across processors in this program as well.

5.6 Block Cholesky

Our final case study is the Block Cholesky code [92], discussed earlier in Chapter 2.4.5. Similar to Panel Cholesky, this application also requires a careful distribution of tasks and data to achieve good performance. In addition, there are several different scheduling options worth exploring in this code; we show how these requirements are met using the affinity hints in COOL.

Figure 5.16 presents the Block code expressed in COOL along with the locality optimizations described below. Based on default affinity tasks are collocated with the base object. Therefore invocations of the parallel completeBlock method are collocated with the block that becomes ready. The completeBlock method then invokes the updateBlock method to reduce each destination block that must be updated using the current source block. These updates are likely to reference this source block in local memory, but may reference the other source block and the destination block in local or remote memory. This is called the *source-based* scheme, in which we exploit locality on one of the source blocks. In contrast, another appealing strategy is the *destination-based* scheme, where we exploit locality on the destination block by performing the update on the processor that contains the destination block in its local memory.

We show how each strategy can be obtained within the same basic computational structure shown in Figure 5.16. Blocks are distributed across processors' memories in a round-robin fashion. The default affinity of a task for the base object provides the source-based scheme, as mentioned earlier. To express the destination-based scheme, we declare updateBlock to be a parallel function invoked on the destination block. As a result, invocations of updateBlock are automatically collocated with the corresponding destination blocks, thereby implementing the destination-based scheme. Although the destination scheme has some additional overhead, the extra concurrency may help load balance. Furthermore, the cache performance may improve since modifications to a block are likely to execute on the same processor.

The COOL version of Block Cholesky is similar to the original ANL version of the code [90], and exploits the same concurrency. However, rather than the round-robin distribution of blocks as in COOL, the ANL code treats the P processors as organized into a two-dimensional $\sqrt{P}$ by $\sqrt{P}$ grid. This grid is then used in a 'cookie-cutter' fashion to map sections of blocks to processors, as shown by the example in Figure 5.17 for four processors. Besides achieving a reasonable distribution of work, this scheme has the advantage that a row of blocks is mapped onto a row of processors, and a column of blocks is mapped onto a column of processors. This helps to reduce communication of a ready block because, since a ready block is used to reduce other blocks only in the same row and in the same column, it is now referenced only by the processors in the same row/column rather than *all* the processors. However, recall from Chapter 2.4.5 that the execution order of tasks in the ANL code must be carefully managed to avoid deadlock. Therefore the task distribution is determined statically and no task stealing is possible. In contrast, the COOL version distributes the blocks (and tasks) in a simple round-robin fashion and depends on dynamic task-stealing to improve load-balance. Furthermore,

```
class block_c {
  . . .
  int remainingUpdates;
  condH ready;
public:
  // update this block by the given source blocks
  // Make this function parallel for the destination-based scheme.
  mutex condH* updateBlock (block_c* src1, block_c* src2) {
    . . . update the block . . .
    if (−−remainingUpdates == 0)
      completeBlock ();
  }

  parallel condH* completeBlock () {
    . . . perform internal completion . . .
    // I am now ready. Signal others that may be waiting.
    ready.broadcast ();
    // Generate updates that use this block.
    for (all blocks 'b' above 'this' in the same column) {
      // Wait for that block to become ready.
      block[b].ready.wait ();
      dest = block updated by 'this' and 'b';
      block[dest].updateBlock (this, block+b));
    }
  }
} *block;

main () {
  int i;
  for (i=0; i<B; i++)
    migrate (block+i, i);
  . . .
  waitfor {
    for (all blocks 'b' that are initially ready)
      block[b].completeBlock ();
  }
}
```

Figure 5.16: The Block Code.

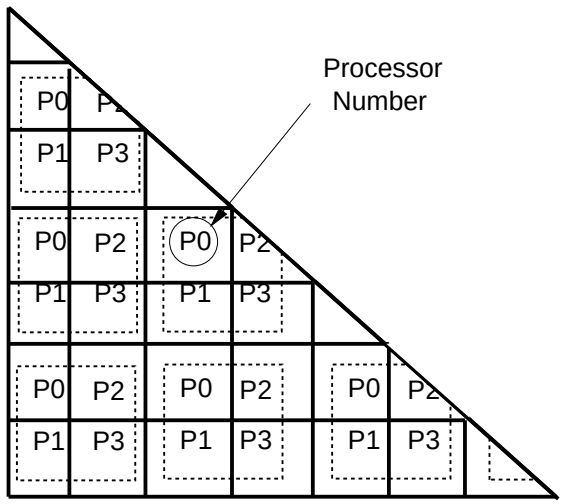

Figure 5.17: A 2-D round-robin distribution of blocks used in the ANL code (shown for four processors).

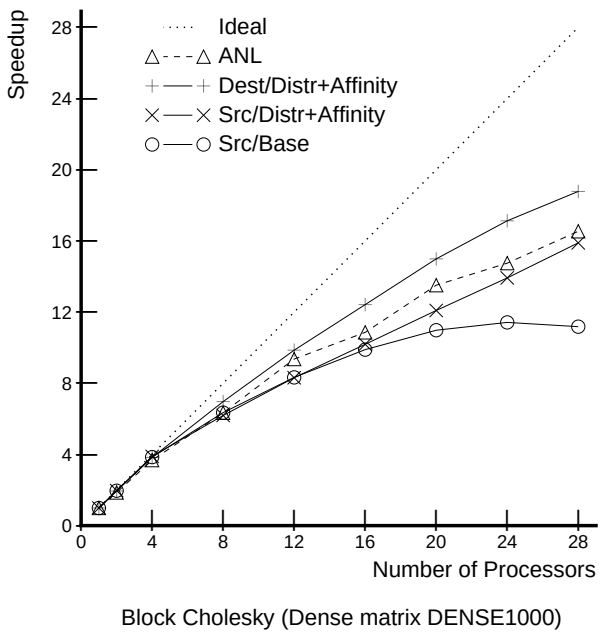

Figure 5.18: Performance improvement with affinity hints for Block (DENSE1000).

the ANL code must maintain per-processor task-queues and perform all task distribution optimizations in an explicit fashion.

We present performance results on a dense matrix (of size 1000x1000) in Figure 5.18, and on a sparse matrix in Figure 5.19. The cache miss statistics for the two inputs are presented in Figures 5.20 and 5.21 respectively. The sparse matrix is BCSSTK15 from the Boeing-Harwell set of sparse matrix benchmarks [32]. This matrix has 3948 columns organized into 3661 blocks. The DENSE1000 matrix takes about 38 seconds, while

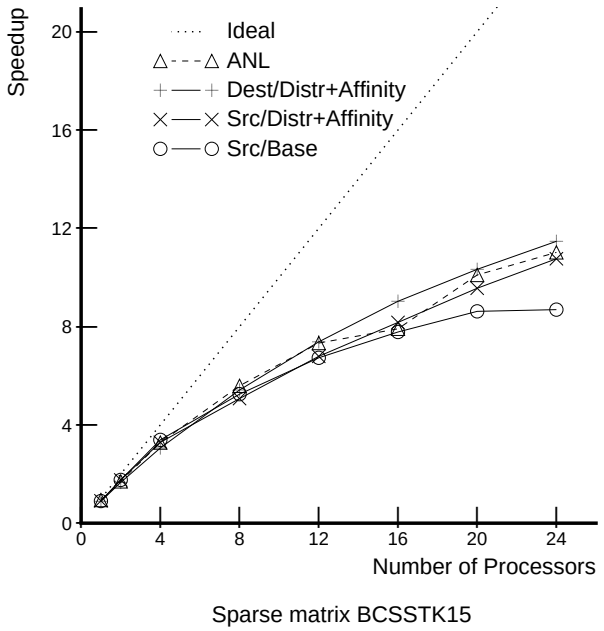

Figure 5.19: Performance improvement with affinity hints for Block (BCSSTK15).

BCSSTK15 takes about 28 seconds on a single processor. We present results for three variations of the program: the first is the base source-based scheme in which data is allocated on a first-touch basis and tasks are scheduled based on default affinity; the second is the source-based scheme with the blocks distributed across processors and tasks scheduled based on affinity for the source block; and finally the destination-based scheme with both block distribution and affinity scheduling. The speedups are much higher on the dense matrix compared to the sparse matrix, but the relative performance of the different scheduling schemes is similar. Block distribution and affinity scheduling provide substantial performance improvements over the base version of the program. The improvements in the cache miss behavior, however, are small, suggesting that the improvement are likely due to better memory bandwidth with the distribution of data. Exploiting locality on the destination block (which is being modified) is clearly more beneficial than the source-based scheme. Performance improves significantly, and there are fewer cache misses with more of the misses serviced in local memory. Finally, we also present the performance of the destination-based scheme coded using the ANL macros. Although the two codes (COOL and ANL) perform block distribution and scheduling optimizations, the ANL code uses a "cookie-cutter" distribution of blocks as described above while the COOL code has a simple round-robin distribution of blocks. In the range of machine-sizes that we consider, the round-robin distribution scheme exhibits better load-distribution. Hence the COOL code outperforms the ANL program, as shown in the Figure.

To summarize, this application offered several potentially interesting optimizations. We could easily express both the source and destination based schemes within the same computational structure of the program. Optimizations such as object distribution for

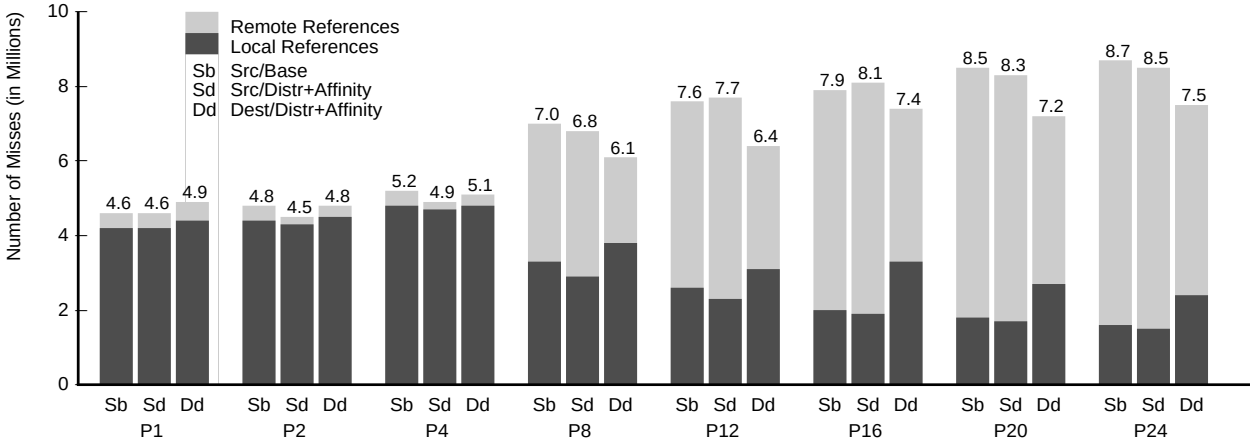

Figure 5.20: Cache miss statistics for Block Cholesky (Dense matrix DENSE1000).

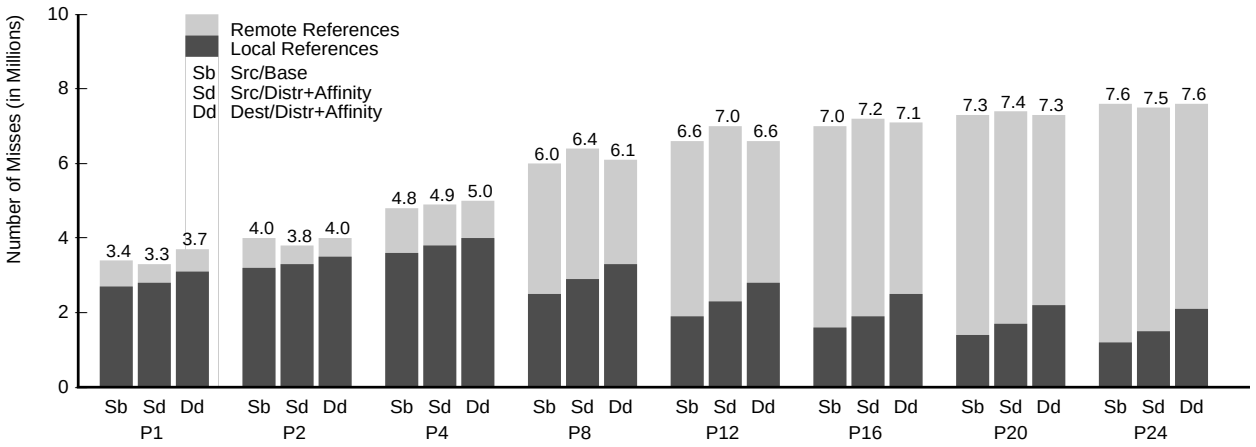

Figure 5.21: Cache miss statistics for Block Cholesky (Sparse matrix BCSSTK15).

better memory bandwidth, and task distribution for better memory locality were naturally expressed within the object structure of the program and the scheduling defaults. The default affinities scheduled the tasks appropriately, and the programmer only needed to distribute blocks across processors. It may be possible to automate even the block distribution based on the default object distribution strategy that we have described before. In contrast, the ANL version of the code had to perform all the task management and scheduling chores explicitly.

5.7 Summary

Having studied the performance of the various applications in COOL in detail, the most important observation is that COOL is efficient. The performance of the COOL versions of the applications is comparable to that of the hand-tuned codes written using the ANL macros (and often better). We can therefore conclude that the constructs can be implemented efficiently and are suitable for high-performance parallel programming.

Let us now consider the questions posed in the beginning of this chapter regarding the affinity hints provided in COOL. In all the applications that we studied, both data locality and load-balancing were critical factors in achieving high performance. The applications required a range of optimizations—Ocean, Panel, and Block Cholesky required both task and data distribution to exploit cache and memory locality, while Water, LocusRoute, and Barnes-Hut required affinity scheduling for cache reuse. In all cases, we found that the affinity hints in COOL provided a sufficient degree of control for exploiting the desired locality—whether in the cache, in local memory, or both. Supplying the affinity hints resulted in substantial performance improvements, ranging from 60-135%. More importantly, the programs tuned using the affinity hints in COOL performed as well as the corresponding programs hand-optimized using the ANL macros.

Supplying the hints clearly requires programmer effort, including an understanding of the application structure and the underlying memory hierarchy. The real benefit from our approach is that the scheduling mechanisms for exploiting locality are built within the COOL runtime system, allowing the programmer to control the optimizations using the locality abstractions. As we have shown, the abstractions are easily supplied as annotations on parallel functions or as distribution of objects. Furthermore, they do not affect the semantics of the program, allowing programmers to easily experiment with different scheduling optimizations while tuning their program, as shown by the Barnes-Hut and Block Cholesky applications. Finally, actually performing the optimizations was very simple and required only a few lines of additional code in each application.

To summarize, the locality support provided in COOL is as effective in achieving high performance as explicitly hand-coding the optimizations but is a whole lot easier for the programmer. It seems difficult to entirely automate these optimizations—for instance, in applications such as LocusRoute the optimizations were based on an understanding of the application semantics. However, as our experience with applications such as Ocean, Panel, and Block Cholesky suggests, smart default strategies for object distribution and task scheduling can significantly reduce the burden on the programmer.

Chapter 6

Related Work

The design of new languages for parallel programming has been an area of active research for over two decades. A large variety of parallel languages have been proposed in the literature. Apart from languages like COOL for exploiting task-level parallelism, there are languages for exploiting data-level parallelism in dense matrix computations (e.g., High Performance Fortran (HPF) [78] and DataParallel C [53]), languages for exploiting parallelism in symbolic Lisp [60] programs (e.g., Multilisp [46] and Qlisp [38]), languages for parallel logic programming (e.g., Parlog [42]), and languages for writing distributed client-server style applications (e.g., Concert C [10]). Furthermore, this list is by no means exhaustive and only serves to demonstrate the diversity of parallel language research.

Given the variety of work in this area, we narrow our discussion of related work based on the two primary aspects of COOL—the support for expressing parallelism and the support for improving locality in a program's execution. Therefore, of the various parallel languages in the literature, we focus on those languages designed to exploit task-level parallelism for achieving high-performance. In particular, we compare COOL with other monitor-based languages that are similar in terms of the individual constructs that they provide but differ in other important aspects such as the flexibility of the monitor mechanism or the integration of the constructs with the underlying object structure of the program. We then contrast the COOL approach for improving data locality and load-balancing with some other techniques, including automatic techniques such as operating system scheduling and page migration, and explicit techniques such as those provided by the ANL macros or by programming languages such as HPF [78].

6.1 Other Parallel Languages

The individual constructs in COOL are by no means novel—for instance, monitors and condition variables were proposed by Hoare in 1974 [55] and have since appeared in a multitude of languages such as Turing [58], Concurrent Euclid [29, 57], PRESTO [14], Emerald [15, 63, 85], and Capsules [40] (to name a few). COOL differs from these languages in different aspects of the overall design which we evaluate below.

First, we compare COOL with some of the early monitor-based languages such as Concurrent Pascal [47, 52] and Mesa [70, 83]. These languages were initially proposed for programming operating systems; we explore their suitability for writing parallel applications.

Next, we evaluate the tradeoffs in exploiting parallelism using the same basic constructs as COOL, but with the constructs provided through runtime libraries rather than integrated into the base language [13, 14, 28, 31].

Third, we discuss monitor-based languages that provide additional synchronization features over the basic monitor construct [19, 35, 40, 84]. We examine the utility of these features versus the simpler monitors of COOL.

Fourth, recall that while the COOL constructs are integrated with the C++ class mechanism, the programmer can violate the modularity of objects when desirable for programmability or performance reasons. We contrast this aspect of COOL with the *strict* object approach taken by languages such as Emerald [15, 63, 85] and Orca [11, 100], where the modularity of individual objects is rigorously enforced by the language.

Finally, we compare COOL with languages that exploit task-level parallelism while maintaining the sequential execution paradigm of the original uniprocessor program.

In our discussion in this section we focus on the language support for expressing concurrency, communication, and synchronization. COOL is distinct from these languages in the support that it provides for performing locality related optimizations. We discuss other approaches to improving data locality in the following section.

6.1.1 Early Monitor-based Languages

Monitors were originally proposed by Hoare [55] as a synchronization mechanism for coordinating concurrent processes on a uniprocessor. The original monitor mechanism is quite similar to that provided in COOL: in a language with abstract data types, the programmer can declare a shared data type (e.g., a C++ class) to be a monitor. A monitor object (an instance of a monitor class) has the property that only one operation (member function of that class) can execute on the object at any one time, thereby serializing multiple simultaneous operations on the object. This ensures exclusive access to the object for the individual operations. Event synchronization is provided through instances of a special type called condition variables, that provide operations to wait and signal on the event. COOL monitors are very close to Hoare's monitors, except that they allow

somewhat greater flexibility—multiple reader methods can execute concurrently on a monitor object, and the programmer has the flexibility to bypass the abstraction offered by the synchronizing monitor operations to access the shared object directly.

Languages like Concurrent Pascal [47, 48, 52], Mesa [70, 83], and Modula [106] have integrated monitors with support for data abstraction in the language. Concurrent Pascal was amongst the first monitor-based languages; we therefore describe it in detail and compare it with COOL.

Concurrent Pascal extends Pascal with the notion of processes, monitors, and queues (or condition variables). Concurrency is specified through the notion of a process, which corresponds to a separate thread of execution along with a set of data structures that are private to the process. A process cannot operate on the private data of another process, and monitors are the only mechanism for inter-process communication. Therefore any shared data must be specified to be a monitor and is operated upon only by monitor methods that are guaranteed exclusive access to the object while executing. Furthermore, the only permissible parameters to a process or a monitor method are constants or monitor objects. This, coupled with the restriction that both a process and a monitor method cannot access anything other than local data or a parameter, ensures that all inter-process communication is through monitors alone. Monitors in Concurrent Pascal are strict and allow only one method within a monitor at any time. Concurrent Pascal also provides queues that support operations such as waiting for an event and releasing the monitor, or signaling an event.

The primary focus of languages such as Concurrent Pascal and Mesa has been concurrent programming on a uniprocessor, rather than parallel programming on a multiprocessor. For instance, as recently as 1980, Mesa [70] contemplated implementing mutual exclusion simply through a non-preemptive scheduler, although this solution was ultimately rejected for several reasons, one of them being the "possibility of a multiprocessor execution environment." This bias towards operating systems programming is further reflected by several aspects of the design of Concurrent Pascal—for instance, concurrency is provided through heavy-weight operating system processes that execute in their own private address space; the communication mechanisms are designed for coarse-grained communication between processes since they do not share data directly but communicate through global monitors in the program; and finally, the synchronization mechanisms are designed only to support mutual exclusion for shared resources such as a disk buffer, whereas parallel applications also need control synchronization such as a barrier or the COOL waitfor construct.

Clearly, therefore, these languages were not originally intended for parallel programming. Furthermore, based on the analysis presented below, we find that several aspects of their design make them unsuitable for exploiting parallelism. The primary drawback is the low degree of coupling between multiple processes, with all inter-process communication restricted to be through shared monitor objects. Since monitor objects are strict in Concurrent Pascal and can only be operated upon by a monitor method, it becomes difficult to express computation that refers to more than one shared object. For instance, routing a wire in the LocusRoute application [89] requires the task to access both the wire

object as well as the global CostArray (see Chapter 2.4.3). Or, the inter-grid operations in the Ocean code [95] require a task to reference multiple grids simultaneously while processing the inter-grid operation (see Chapter 2.4.2). Finally, the update operations in both Panel and Block Cholesky require access to both the source and destination panel/block being updated.

The second drawback of Concurrent Pascal is that the synchronization model is quite strict, with all accesses to shared data synchronized through a monitor operation. The programmer therefore cannot bypass the synchronization when desirable for programmability or performance reasons. Examples where this flexibility is useful include unsynchronized accesses to the CostArray in LocusRoute (see Chapter 2.4.3), relaxed SOR computation in the Ocean code (Chapter 2.4.2), and unsynchronized access to objects during certain phases in each of the other four applications. In contrast, these accesses to shared data would all have to be performed through expensive monitor operations.

The third drawback of Concurrent Pascal for exploiting parallelism is that synchronization support is restricted to monitors and queues, with no control synchronization such as a waitfor (as in COOL) or a barrier. Recall that each of the six applications studied in Chapter 2 used the waitfor construct to wait for a set of tasks to complete; this synchronization would need to be explicitly built in Concurrent Pascal.

Finally, the concurrency abstraction in Concurrent Pascal is similar to that provided in the ANL macros, i.e., heavy-weight processes executing in their own private address space. This execution model requires the programmer to explicitly schedule work across the processes. This can be quite cumbersome in practice as well as having other drawbacks discussed in Chapter 2.1.

In a follow-up to Concurrent Pascal, Brinch Hansen designed the Edison [49, 50, 51] language that addressed some of the above concerns by building in greater flexibility into the language. Rather than the heavy-weight processes of Concurrent Pascal, Edison allows the programmer to dynamically specify concurrent execution using the structured cobegin-coend construct. Rather than communicating only through monitor objects as in Concurrent Pascal, these parallel tasks communicate through shared variables in the program. Finally, there are no monitors or condition variables in Edison; instead, it provides the *when* construct (a simpler form of the conditional critical region [54]). Given a conditional expression and a statement list, the 'when' construct ensures that (a) the statements are executed only after the condition evaluates to true and (b) only one 'when' statement list executes at any one time in the entire program. The former property allows the programmer to provide a high-level expression, rather than explicitly managing the low-level queueing of processes using condition variables (at the expense of efficiency, however). The latter property is used to express mutual exclusion, although it synchronizes over the entire program rather than a particular monitor object. Overall, Edison deliberately compromises some of the safety properties of Concurrent Pascal in return for the flexibility provided by the new constructs. Some of these features are also similar to constructs such as Capsules [40], discussed later in Section 6.1.3.

To summarize, while the early monitor-based languages offered similar constructs as COOL, several aspects of their design were directed towards concurrent programming on

uniprocessors making them unsuitable for exploiting parallelism.

6.1.2 Parallel Runtime Libraries

Several researchers have proposed support for exploiting parallelism through runtime libraries, such as C-threads [28], Presto [14], Brown threads [31], and the ANL [18]/Parmacs [13] macros from the Argonne National Lab. A typical runtime library consists of multiple light-weight tasks that execute in the same address space, communicate through shared memory, and synchronize through a variety of constructs such as locks, monitors, condition variables, barriers, and atomic integers.

Providing these constructs through a runtime library has the advantage of not requiring any changes to the compiler, and the constructs can be quickly and easily prototyped by implementing them within the runtime system. However, simply leaving the constructs as library routines rather than integrating them within the base language can adversely affect the readability, security, and efficiency of the language, as we describe below.

Let us first consider the impact on the readability of parallel programs written using runtime libraries. While a monitor operation in COOL is expressed by labeling a method on a class to be mutex or nonmutex, a runtime libary provides a predefined type called a monitor on which the programmer can invoke entry and exit operations. The programmer declares instances of the monitor type, and explicitly inserts calls to entry/exit operations on these instances at the appropriate synchronization points in the code. Whereas the class header in a COOL monitor clearly identifies the shared data being protected and the various operations that reference/modify it, with a runtime library all of these properties can be quite hard to infer from the program, making it difficult to understand the synchronization structure of an application.

We experienced the above problem when trying to understand the Panel and Block Cholesky applications written using the ANL macros [18]. Even though the mutual exclusion synchronization for a panel/block during an update operation was expressed through a simple lock/unlock operation, we could only guess at the data that was actually being protected. Furthermore, the corresponding pair of lock and unlock operations were invoked in different procedures, making it extremely cumbersome to trace the control flow in the program and match the calls. There were numerous implicit assumptions strewn about the program, and several sessions with the program's author were necessary to understand the synchronization structure of the program. Most of these problems are addressed by the structured constructs in COOL.

The second drawback of using runtime libraries—security—stems from factors similar to those discussed above. Since the association between a monitor instance and the shared data is implicit, it is all too easy to accidentally invoke entry/exit on the wrong monitor instance, or invoke entry on one monitor instance and exit on a different monitor instance. Second, since the entry and exit operations must be explicitly inserted into the code, it is possible to forget one of the two calls. Or, since the calls can be inserted anywhere (in particular across procedure boundaries), complex control flow between the entry and

exit operations may cause a monitor operation to inadvertently be invoked multiple times or perhaps skipped entirely. Although the structured design of COOL does not entirely eliminate this class of errors, it is decidedly more secure than the same constructs offered through runtime libraries.

A language-level design is not a panacea, of course, and convoluted programs can be written in any language, including COOL. Conversely, a good programmer can follow rigorous coding rules to write highly-structured programs using runtime libraries. However, the language abstractions should be designed to encourage good coding practices and make programs easier to understand. In this regard a language level approach like COOL is more desirable than runtime libraries.

The third benefit of COOL compared to runtime libraries relates to optimizations that can be performed by a compiler to reduce the overhead of the synchronization constructs. With the structured design of COOL, the compiler can identify the shared data being protected (monitor variables), the various operations that manipulate the shared data (monitor methods), and even the actual code within each critical section (body of a monitor method). It can therefore analyze how the object is being manipulated by the various operations and optimize their implementation to reduce the synchronization overheads. Chapter 4 presented various optimizations such as discarding the support for multiple readers for monitors with only mutex functions and implementing monitors that have small critical sections (such as atomic integers) with pure busy-wait synchronization. As we showed, these optimizations substantially reduced both storage and runtime overheads associated with the synchronization operations in COOL applications; these optimizations would not be possible with an approach based on runtime libraries.

Finally, there are several smaller benefits of the integrated approach in COOL. For instance, in a runtime library the programmer creates a task by passing it a pointer to the function $parFn$ that should be executed concurrently. Furthermore, the $parFn$ function can only take a single one-word argument, requiring the programmer to (a) marshall multiple arguments into a struct, (b) pass a pointer to this struct when creating a task, and (c) unmarshall the arguments within $parFn$. In COOL, on the other hand, the programmer can simply label a function to be parallel and invoke it like an ordinary sequential function; the implementation automatically handles the marshalling and subsequent unmarshalling of arguments. Next, the runtime libraries typically provide a barrier construct whereas the COOL waitfor construct provides greater flexibility, as illustrated by the example of synchronizing separately for two independent phases discussed in Chapter 2.2.4. Finally, the language abstractions and runtime optimizations for improving locality described in Chapter 3 exploit the integration of the COOL constructs with the underlying object structure of a program; this approach would be much less effective were these constructs provided as runtime libraries.

For these various reasons, therefore, although the basic functionality of the constructs provided in COOL and the runtime libraries is similar, providing the constructs integrated within the language offers substantial benefits for both the programmability and the efficiency of the language.

6.1.3 Variations in Monitor Design

The monitor construct in COOL provides simple mutual exclusion and is used in combination with condition variables to express general-purpose synchronization. However, several researchers have argued that condition variables are a low-level queueing mechanism that can lead to error-prone programs. Furthermore, with condition variables the synchronization code and the actual work get all mixed up together within the body of a monitor operation. Therefore, rather than providing condition variables, some languages have proposed constructs that instead provide the programmer with additional control (i.e., beyond simple mutual exclusion) over the execution of monitor operations.

Examples of such constructs include guarded-commands [30] that allow the programmer to specify a precondition that must be true before an operation can proceed, or path-expressions [20, 110] that use regular expressions to specify all legal orderings of operations on an object. These various constructs are appealing since they allow the programmer to provide a high-level specification of the desired synchronization. Languages that incorporate variations of these constructs include Capsules [40], PSather [35, 84], uC++ [19], Edison [49, 50, 51], Orca [11, 100], Enabled-sets [103], and Guide counters [66]. We focus on a detailed comparison of COOL with one of these languages, Capsules, since that incorporates both path-expression and guarded-command style constructs.

Capsules [40] are a monitor-like facility for controlling access to shared data, provided within the Concurrent C++ language [41]. (Figure 6.1 presents a simple capsule example.) A capsule is an extension of the C++ class mechanism with the keyword class replaced by the new keyword capsule. A capsule is like a monitor in that all methods in the capsule by default acquire exclusive access to the capsule instance before executing. There are no condition variables. Instead, within a capsule specification the programmer can supply a *par* and a *sync* section (see Figure 6.1). The par section is used to specify the monitor operations that can be executed in parallel. For instance, the first par specification in the figure allows multiple reads to execute concurrently, while the second specification allows a read to execute concurrently with a write that is to a different location—the latter requirement is specified by the *suchthat* clause as shown. The sync section enables the programmer to (i) specify the conditions under which a method can proceed (using the *suchthat* clause) and (ii) order the execution of multiple instances of a particular method (using the *by* clause). For instance, the first sync specification allows a read to proceed only when the buffer being read is non-empty, while the next specification allows a write to proceed only when the location being written to is empty. Furthermore, the *by* clause allows the programmer to order the execution of multiple enabled writes by accepting writes with a lower value first. As shown by the example, the conditions within the par and sync sections can reference local variables within the capsule as well as the actual parameters to the monitor operation. However, they cannot contain any side-effects since they may be repeatedly evaluated by the implementation to determine the eligibility of each method.

Applying the capsule construct to the synchronization examples discussed earlier in

```
// Definition of a capsule.
// (warning: doesn't do anything useful).
capsule dictionary {
   struct {
      double value;
      int empty;
   } *buffer;
public:
   double read (int);
   void write (int, double);

// par section: specify methods that can execute concurrently.
par:
   // Allow multiple reads to proceed in parallel.
   { read (int)* };

   // Allow reads and writes to different locations to proceed in parallel.
   { read (int i), write (int j, double) } suchthat (i != j);

// sync section: specify preconditions and orderings for methods.
sync:
   // A read can proceed only when the location is not empty.
   accept read (int i) suchthat (!buffer[i].empty);

   // Accept writes only when the location is empty.
   // The "by" clause specifies that accept the lowest valued write first.
   accept write (int i, double val) by (val) suchthat (buffer[i].empty);
};
```

Figure 6.1: Example code illustrating Capsules.

Chapter 2, we find that most of the previous applications can be easily expressed. For instance, the synchronized double (Chapter 2.3.2) accepts a read only when the object is not empty and allows multiple reads to execute concurrently. Acquiring exclusive access to multiple objects (Chapter 2.3.3) can be coded similarly to the solution in COOL, by invoking the claim operations in an ordered fashion on multiple objects and accepting the claim and surrender operations in an interleaved manner within each capsule. However, the barrier abstraction cannot be easily expressed using capsules, since it needs two synchronization points: the first to atomically update the count of the number of arrived processes and the second to wait until the rest of the processes have arrived. Capsules provide only one synchronization point, namely when a method is invoked on a capsule. Furthermore, the suchthat clauses can query the state of the capsule but cannot have any side-effects. Any updates to the number of arrived processes must be therefore done within the protection of the capsule.

Therefore, while capsules allow the synchronization conditions to specified through high-level boolean expressions, they do not allow side-effects to the synchronization state. The solutions to some synchronization problems, therefore, can become quite complex. Furthermore, the implementation of capsules requires the repeated evaluation of the boolean expressions, which is inefficient. In contrast, although condition variables require more explicit control on the part of the programmer, they provide greater flexibility in expressing different synchronizations as well as lead to more efficient programs.

6.1.4 Concurrent Object-Based Languages

Several object-based parallel languages have been proposed in the literature. The basic concepts of object-based concurrency are discussed in [3], while a survey of different approaches can be found in [109]. Examples of such languages include Emerald [15, 63, 85], Orca [11, 100], POOL-T [6], Actors [4], CST [59], and ABCL/1 [111]. In these languages, mechanisms for specifying concurrent execution include explicit processes in the program, independently executing active objects, or simply the asynchronous processing of messages between objects. The object state is usually encapsulated within the object, with both communication and synchronization through method invocations (messages) on objects. Objects may be single-threaded (similar to monitors) in that operations on the object are serialized, or they may be multi-threaded by allowing multiple operations to execute simultaneously on the object. In the latter case synchronization must be explicitly programmed through other mechanisms such as locks and semaphores.

We describe one of these languages, Orca [11, 100], in detail and compare it with COOL. Orca is an object-based parallel language in which all inter-process communication is expressed through shared objects. Concurrent execution is expressed by "forking" a procedure to execute in parallel. This procedure can be passed both input (value) and output (shared) parameters. The shared attribute attached to a formal parameter signifies that the actual argument should be passed by reference—there are no global objects in the program and shared parameters are the only communication mechanism between concurrently executing procedures. Objects are *strictly modular* and must be referenced

through methods. Orca deliberately omits pointer types to preserve the modularity of objects and enhance security. An Orca object is a strict monitor and provides guarded commands [9, 30] instead of condition variables. A method on a shared object can contain a list of condition-statement pairs (guarded commands). The method blocks upon invocation until at least one of the conditions evaluates to true, whereupon one true guard is nondeterministically selected and its set of statements executed.

A strict object model is appealing from security purposes, but, as discussed earlier in Chapter 2.4.7, can prove quite cumbersome while expressing fine-grained communication between objects. Requiring objects to be accessed through the interface methods only is appropriate for expressing coarse-grained operations on individual objects but is not suitable when some computation requires intimate access to multiple objects. We have seen instances of this in several of the applications that we studied earlier, such as Ocean, Panel, and Block Cholesky. Furthermore, Orca objects present a strict synchronization model as well, in which all methods must acquire exclusive access to a shared object before executing. In several of the applications discussed earlier (e.g., Panel/Block Cholesky, Water, LocusRoute), this results in unnecessary synchronization that could potentially be optimized away by the programmer based on an understanding of the program structure.

To summarize, there is a tradeoff between programming ease and performance on the one hand, versus security and modularity on the other hand. The flexible approach in COOL allows the programmer to make this tradeoff based on the particular application. The programmer can choose to write a strictly modular program, or when useful, bypass the method and/or the synchronization interface for certain objects in the program for efficiency reasons.

However, while the strict modularity of these language makes them cumbersome for shared-memory machines, this very same property makes these languages attractive for programming *non-shared* address space machines. Our discussion so far has assumed an architectural base of hardware shared-memory. In contrast, message passing machines such as the Intel iPSC [56], Intel Paragon [43], Intel Touchstone Delta [56], Ncube [33], and CM-5 [101], do not provide hardware support for shared memory. In these environments a processor cannot directly reference data that resides in another processor's local memory, and all communication between processors must be specified by the programmer through explicit messages. This can be very cumbersome compared to shared-memory architectures where communication and coherence on shared data is maintained automatically in hardware. A strict object model is quite attractive for programming such architectures because all inter-process communication in the program is restricted to be through interface operations on shared objects. Therefore the compiler/runtime system can automatically translate a method invocation on a shared object into some runtime code to (a) locate the object in the system, (b) invoke the method remotely, and (c) return with the return value upon completion of the method—all entirely transparent to the programmer. The programmer is provided a simple model of shared objects in the program, with all the details of communication handled automatically within the runtime system. Along with Orca, there are several other proposals of object-based systems for

programming these loosely-coupled architectures such as Jade [69, 88], Linda [5], and Emerald [15, 63, 85]. However, a detailed discussion of these software shared-object systems is not relevant for this thesis.[1]

6.1.5 Languages Based on Serial Semantics

FX-87 [62] and Jade [69, 88] are languages designed to express parallel execution while preserving the semantics of the corresponding sequential program. In Jade the programmer identifies tasks that are useful to execute concurrently in the program and specifies the data usage information for each task. This data usage information identifies the shared variables that are read or written by each task. Based on the data referenced by each task and the implicit serial order of tasks in the program, the implementation executes tasks concurrently while preserving the data dependences between tasks. Tasks synchronize for the common shared objects in program order to ensure serial semantics. In FX the side effects of expressions are specified through an *effect system* that integrates type checking and side effect analysis. This effect system is used by the compiler to determine the functions that can execute concurrently.

While Jade enables synchronization based on the serial paradigm to be expressed naturally, it is overly restrictive in other situations. For instance, many non-deterministic parallel computations (such as graph search algorithms) do not fit within the Jade paradigm. Similarly, applications that require cyclic communication cannot be expressed efficiently in Jade. Such programs must be written so that each communication cycle is instead replaced by an additional producer and consumer task. Although task creation overhead is small with light weight threads, it can become significant if the program has frequent fine-grained synchronization in this fashion. Another restriction in Jade is that tasks must be created in the *same* order in which the corresponding functions are executed in the serial algorithm. However, this can be inefficient in some situations. For instance, in an algorithm to perform the Gaussian elimination of a sparse matrix, the serial order requires that all tasks corresponding to reductions by a column be created before proceeding to the reductions due to the next column. For large input matrices the task creation can become a significant bottleneck since most of these tasks are not immediately executable. A better strategy may be to create the runnable tasks earlier, so that the processors do not idle for lack of available work.

While Jade offers only one basic synchronization paradigm (through dependence specification on objects), in COOL we provide flexible support for the user to construct different *kinds* of synchronizations specialized to the application. Based on our application experience, the efficiency gained through specialization is often worth the cost of the programming effort required.

Similar to Orca, an interesting aspect of Jade is that it is designed to be portable across

[1] Our discussion of Jade in the following section focuses on how Jade exploits concurrency within the serial execution paradigm rather than its portability to non-shared memory machines.

both shared-memory and message-passing architectures. On a shared-memory architecture the usage specifications are used for synchronization between multiple tasks. On a message-passing architecture these same usage specifications double as communication mechanisms as well. Since these usage specifications identify the objects referenced by a task, on a message-passing architecture the runtime system automatically communicates the referenced objects to the task. Thus a Jade program transparently ports across both shared-memory and message-passing architectures.

6.2 Exploiting Locality

We next discuss some of the other proposed techniques for addressing data locality and load-balancing in a parallel program. There has been relatively little work in addressing these problems, most of it consisting of either providing bare-bones primitives and then leaving it all up to the programmer, or ranging to the other extreme where the optimizations are performed automatically by the operating system or the compiler with the programmer having no control. In contrast, in the COOL approach we build a range of optimizations within the runtime system and provide a hierarchy of abstractions in the language for the programmer to influence the optimization process.

In this section we give a brief overview of these different approaches. We first discuss the automatic techniques, including operating system and compiler based approaches. We then describe an explicit approach using ANL macros. Finally we compare COOL with scheduling in an object-based system.

6.2.1 Operating System Based Approaches

An operating system can try to improve data locality through process scheduling and automatic page migration. While scheduling processes [44, 76, 105, 80, 23], the operating system tries to improve cache reuse by techniques such as (a) giving a process affinity for the processor where it last ran so that it is likely to reuse its data in the cache across scheduling time slices and (b) increasing the duration of the scheduling time slice itself. However, these scheduling optimizations help in reducing cache interference *between* different applications and are orthogonal to the task scheduling optimizations necessary to improve locality *within* an application.

Several automatic page migration schemes have been proposed in the literature [16, 23, 71, 72], wherein the operating system tries to improve memory locality by automatically migrating a page to a processor that is incurring the most cache misses to that page. However, in most machines today there is no support for counting the actual number of cache misses incurred to a page while the application is executing. Therefore these page migration techniques typically use the number of TLB misses to a page as an approximation to the number of cache misses. Future architectures (e.g., the Stanford FLASH [67]) that provide hardware support for counting the number of cache misses to each page can improve the robustness of these techniques.

Automatic page migration is attractive because it can be performed within the operating system transparent to the programmer. However, its biggest limitation is the lack of information about the application structure. This can be a particularly severe drawback in applications that exhibit a dynamically varying sharing pattern, where, by the time the operating system realizes that (i) a page needs to be migrated due to a high number of cache misses or conversely (ii) realizes that a page needs to be frozen because it is being migrated too often, the applications has moved on to a subsequent phase that exhibits different sharing behavior. In the presence of these possibilities, any automatic technique must perforce be conservative to avoid incurring unnecessary overheads. In contrast, the programmer, based upon knowledge of the application structure, can always do a better job at these scheduling and data placement optimizations.

6.2.2 Compiler Based Approaches

Compiler techniques to improve data locality consist of optimizations such as scheduling iterations of a loop to improve cache reuse and distributing individual elements of arrays across the local memories of processors for better memory locality [7, 45, 65, 77, 107, 108, 112]. Examples of this approach include compiler analysis to perform these optimizations integrated with exploiting loop-level parallelism in sequential programs. Another related approach is that taken by High Performance Fortran (HPF) [78]. In HPF the programmer writes a serial program and annotates it with data distribution specifications. A rich set of primitives is provided for this purpose and includes a variety of distribution patterns such as interleaved, round-robin, blocked, cyclic, alignment, and dynamic redistribution. HPF also provides primitives to identify concurrent computation, such as an intrinsic data-parallel array operation or an explicit *forall* statement that identifies a loop as fully parallel. Using the data distribution specifications, the compiler partitions the program into parallel activities based on a owner-computes rule, and together the compiler/runtime system manage the scheduling of the parallel computation and optimizations to reduce the communication overheads.

However, compiler-based approaches are successful primarily in the context of exploiting loop-level parallelism in regular dense matrix computations. In contrast, our focus is on exploiting task-level parallelism in programs with an irregular structure, where such compiler analysis is much harder to perform. For instance, in the LocusRoute application discussed in Chapter 5.3, the task scheduling optimizations necessary to exploit cache reuse on the CostArray required an understanding of the semantics of the application—that although a wire was being routed, most of the data misses were to the shared CostArray.

6.2.3 Explicit Approaches

In contrast to these automatic techniques, at the other extreme we have the primitive support provided in the ANL macros [18]. As we showed in Chapter 5, the only support provided in the ANL macros are primitives to (a) lock a process to a particular processor,

and (b) allocate data from the memory of a particular processor. Since the scheduling of work across parallel processes must be explicitly managed by the programmer (see Chapter 2.1), any further optimizations for either of cache or memory locality must be explicitly programmed by the user through the low-level management of the details of task scheduling. As we saw through the case-studies discussed in Chapter 5, the support in COOL provides much greater flexibility and is substantially easier to use than the ANL macros, without compromising either performance or the degree of control provided to the programmer.

6.2.4 Scheduling in Object-Based Systems

Object-based systems have the potential to perform scheduling optimizations automatically based on the objects referenced by tasks. In this section we discuss the task scheduling mechanisms in Jade [87]. (The Jade system was developed concurrently with COOL).

Recall from Section 6.1.5 that along with each task the programmer provides an *access specification* that identifies all the shared objects referenced by that task. These access specifications are used by the runtime system to synchronize the execution of the program based on the serial semantics, but they also provide the runtime system with information about the objects referenced by the task. They can therefore be used to perform locality optimizations, as described below.

The Jade implementation [87] performs optimizations for both shared-memory and non shared-memory systems; we focus on the optimizations for shared-memory systems. The Jade runtime system assumes that a task wishes to exploit locality on the *first* object in its access specification (termed the *locality object*); the programmer can use this property to control the object for which the runtime system performs locality optimizations. The programmer can also allocate an object from the local memory of a particular processor when the object is created. The runtime system tries to exploit both cache and memory locality on the locality object for a task by (a) servicing multiple tasks on this object on the same processor in an uninterrupted fashion for better cache reuse and (b) servicing these tasks on the particular processor that contains the object allocated in its local memory for better memory locality. A task is enqueued on the queue corresponding to its locality object. There is an *object task queue* for each shared object that belongs to the processor that contains the object allocated in its local memory. Each processor contains a queue for each object allocated in its local memory. It services tasks from its own queues for memory locality and further services tasks from the same queue in a back-to-back fashion for better cache reuse.

An idle processor steals tasks from other processors to improve load-balance. The runtime system provides two different scheduling heuristics—a *cache heuristic* in which an entire object queue is stolen from another processor and all tasks on the stolen executed by the previously idle processor for good cache reuse on the locality object, and a *memory heuristic* in which object queues are not stolen but always stay with the processor where the data is allocated; instead, an idle processor steals just a single task from an object

queue of another processor. The programmer must select one of these two scheduling heuristics when compiling the program.

As we can see, these scheduling optimizations are similar to those provided in the COOL, described earlier in Chapters 3 and 4. The primary difference is that COOL offers a wider range of optimizations in the runtime system and provides the programmer with a greater degree of control over these optimizations. In contrast, the Jade programmer can only specify the particular object important for locality for a task, with all the optimizations entirely encapsulated within the runtime system. For instance, in COOL the programmer can choose to exploit only cache locality on an object regardless of where the object is allocated. Furthermore, the particular locality may be different for different objects/tasks in the program. In contrast, the Jade runtime system chooses one heuristic for the entire program, that is determined when the program is compiled. This additional control in COOL also enables the programmer to exploit cache locality on one object and memory locality on a different object, as illustrated by the Panel Cholesky application discussed in Chapter 5.5.

Regarding data allocation, Jade provides constructs to allocate data from the local memory of a particular processor when the object is initially allocated, but once allocated an object cannot be moved from one processor's memory to another, as is possible in COOL. This is potentially useful when different phases of the same program require a different object distribution (we did not need to dynamically redistribute data in the COOL applications studied in Chapter 5, however). Finally, all scheduling optimizations in Jade are based on the locality object of a task; however, sometimes it is useful to distribute tasks and data directly in terms of the underlying processors, such as in the Ocean and Water applications discussed in Chapter 5.1 and 5.2 respectively. This is supported through the processor affinity abstraction in COOL, but is not possible in Jade.

Overall, therefore, while both Jade and COOL provide scheduling optimizations in the runtime system to enhance data locality, in Jade these optimizations are performed purely based on the task access specifications. In contrast, the hierarchy of abstractions in COOL provides the programmer much greater flexibility in controlling the optimization process.

6.3 Summary

As we have seen, there is a wide variety of parallel programming systems described in the literature. Compared to this body of work, the notable aspect of COOL is the tradeoffs that we make in the language design to provide the programmer full control over the parallel programming process in as simple a fashion as possible. This is apparent in comparing various aspects of the design with the alternate choices made in other languages. For instance, in our design of monitors and condition variables, we provide a simple version of the constructs that can be implemented efficiently. As a result, although the constructs are 'lower-level' compared to some of the other languages, they allow the programmer full flexibility and control in efficiently exploiting different kinds of parallelism. For

instance, the Capsule construct is more expressive in several situations, but we chose the simpler condition variables since (i) they provide greater flexibility in building different synchronizations, and (ii) they avoid the overhead of repeated evaluation of the boolean expressions. Next, while we provide the constructs integrated with the C++ class mechanism so as to exploit the object structure of the program, we are nevertheless not tied to objects in contrast to other object-based concurrent languages such as Orca. This allows the COOL programmer the flexibility to enjoy the benefits of the object-structure when useful, yet to bypass objects when they are more a hindrance than a help.

Finally, COOL is unique in providing support in the language to improve locality. Our approach is structured so as to provide the performance-oriented programmer full control over the necessary optimizations, yet at the same time allow these optimizations to be performed through simple hints in the program. This approach has proven to be very useful compared to the current extremes of either entirely automatic techniques or explicit low-level primitives.

Chapter 7

Conclusions

This research started several years ago when we initially set about exploring some simple future-like synchronization constructs for parallel programming. The constructs evolved and other features were added as they converged to the language described here. We then set about implementing the language on whatever parallel machine we could find. With a working system we could finally write and run real parallel programs. We implemented a variety of parallel applications, chosen mostly from the scientific and engineering domain. However, when we first ran the programs and looked for speedups, we did not find much—instead, we found that most of the execution time was wasting away in the memory system. So it was back to the drawing board as we added support both in the implementation and in the language to address these performance concerns. With these additions, though, we finally have a system that flies.

So what have we learned from all this work? A high-level lesson has been the special importance of *simplicity* and *efficiency* in providing support for high-performance parallel programming. These are not the sole criteria, of course, since taken to the extreme they might well argue for programming directly in assembly language. Furthermore, although these are generic goals, we have found them to be particularly important for parallel programming. Based on our programming experience in COOL what we have appreciated most about the language is that (i) the constructs are simple and easy to understand, (ii) they can directly express the requirements of most applications, and serve as flexible building blocks for expressing more complex requirements, and finally (iii) they can be implemented very efficiently. Like all explicit mechanisms, they do make it possible for programmers to "shoot themselves in the foot". However, we believe that they more than compensate in return by providing greater flexibility and control to the programmer. As a result, expressing the parallelism and the synchronization in all the applications that we considered has generally been quite straight-forward.

We have addressed several questions along the way. We explored the viability of

monitors and condition variables for parallel programming. These constructs have been around for a while, and were originally proposed for concurrent programming of operating systems on uniprocessors where the primary requirement was coarse-grained communication between operating system processes. They have subsequently been adopted by some parallel languages as well. However, the requirements of parallel applications are quite different—these applications are characterized by fine-grained sharing of data between multiple tasks executing in the same address space. Little has been reported about the usefulness of monitors, either their expressiveness or their efficiency, for real parallel applications. We incorporated them into a complete language design, developed techniques to optimize their implementation overhead, and used them to express a variety of parallel applications. Based on our experience reported in this thesis we have found them attractive on counts of both programmability and efficiency.

Another important aspect of our approach has been to exploit the object structure of programs for expressing parallelism. To this end we chose C++ as our base language and offered constructs integrated with the C++ class mechanism. We have found several advantages of exploiting objects to write parallel programs. The first benefit is improved expressiveness—as several examples have shown, the programmer can build abstractions that encapsulate the details of parallelism or synchronization within the object and provide a simple interface for their use. Given a shared object, for instance, the various methods that manipulate it and the accompanying synchronization are clearly identified by the monitor operations. Second, the synchronization operations (and the data protected by the synchronization) can be clearly identified by the compiler and optimized to run very efficiently. These optimizations were described in Chapter 4. Finally, in an object-based program tasks have a natural association with objects enabling optimizations such as task and data placement for improved locality and load-balancing.

On the other hand, being *confined* to strictly modular objects can sometimes be quite cumbersome. For instance, objects are typically expected to be accessed only through methods on the object that operate on only one object at a time. As a result programs that exhibit fine-grained sharing of data and require direct access to more than one object simultaneously can be extremely cumbersome to express. Furthermore, synchronization for these shared objects is typically restricted to be through monitor methods alone, often resulting in unnecessary synchronization overheads. In both of these regards we have appreciated the flexibility in the COOL language design that enables programmers to benefit from the object structure where useful yet allows them to bypass some of the above restrictions for either ease of programming or efficiency reasons.

Finally we provide integrated language and implementation support for improving data locality and load-balancing through task and object distribution. Most programming systems today require the programmer to hand-code these optimizations in a very machine-specific fashion. In our approach we build a variety of scheduling mechanisms within the implementation and provide a set of abstractions that enable the programmer to control and experiment with different optimizations. These abstractions are structured as hints and are easy to use, and provide a range of control over the optimization process. We have found this approach to be very effective—COOL programs optimized in this

fashion perform as well as hand-tuned programs and are a whole lot easier to optimize.

7.1 Future Directions

There are several interesting research directions to pursue based on COOL. First, the support in COOL for improving data locality can be enhanced in several ways. For instance, the affinity hints provided by the programmer are only used to determine an appropriate task distribution in our current implementation. However, these hints provide valuable information about the objects referenced by tasks and can be used for a range of further optimizations. For instance, we can build better task-stealing heuristics within the runtime system such as stealing tasks scheduled for cache locality before stealing those tasks that expect to reference the object in the local memory. Second, extensions to the affinity abstractions that specify whether the locality object is read or written by the task are also useful—for instance, if several tasks read an object then they can execute on different processors and still reference the object in the cache, which is not possible if the object is being modified (in which case perhaps those tasks should execute on the same processor). Third, it would be useful if the programmer could supply multiple affinity hints for a task and depend on runtime heuristics to make an intelligent choice amongst the several objects based upon criteria such as the size of the individual objects. Fourth, as demonstrated by some of the examples in Chapter 5 (e.g., Ocean, Panel, and Block Cholesky), a simple round-robin distribution of arrays of locality objects can be provided automatically and could be combined with default affinity scheduling to fully optimize these programs entirely automatically. A round-robin distribution of the locality objects therefore appears to be a reasonable default distribution of data, which could of course be over-ridden by the programmer through explicit distribute constructs. Finally, the runtime system could maintain the affinity hints for each task so that if a task is stolen by another processor then that processor can automatically *prefetch* those locality objects that are in the remote memory of another processor.

Besides these enhancements for improving data locality, a second possible extension of our work is to enhance the COOL programming environment with better debugging support for both correctness and performance. For instance, we have found it extremely useful (while debugging a COOL program) to generate a trace of various activities such as the start/completion of parallel function or the entering/leaving of a monitor operation during the execution of the program. Furthermore, to avoid being deluged by excess information, it should be possible to restrict the generated traces to particular parallel functions or individual monitor instances. In addition, regarding performance debugging support, the runtime system can keep track of the manner in which tasks were initially scheduled in the program, how many of them executed as scheduled, and how many were stolen for execution by another processor. Such debugging support can go a long way in enhancing both the debuggability and the performance tuning of parallel applications.

Finally, so far we have assumed that the application is running on a dedicated machine in a stand-alone fashion. However, expensive parallel machines are commonly shared

between multiple serial and parallel applications, destroying this abstraction of a dedicated machine. A useful research direction is to develop parallel programming support for multiprogrammed environments. One scheduling policy that has been explored is process control [23, 44, 104] or scheduler activations [8]. This policy is based upon the *operating point* effect, i.e., an application executes more efficiently (with better processor utilization) with fewer processors, reflecting the penalties of increased communication, synchronization, and load imbalance with increasing number of processors. In this approach the operating system partitions the machine into sets of processors, with each set executing a single parallel application. The sets are dynamically resized based on the system load. Furthermore, a parallel application dynamically adjusts its number of active processes to match the number of processors assigned to its set of processors, thereby executing at a more efficient operating point along its speedup curve.

Process control is attractive because it can be exploited entirely automatically within the runtime system of applications written using the task-queue model of parallelism. However, perturbations in the physical processors in the set or in the number of active processes make it difficult for the programmer to perform the task and data distribution optimizations that are essential for good data locality. Given the importance of these optimizations for high-performance applications, there is a need for a programming system that allows the programmer to optimize the program for good data locality on the one hand, yet at the same time provides the runtime/operating system the flexibility to exploit the operating point effect.

Bibliography

[1] A. Agarwal, B.-H. Lim, D. Kranz, and J. Kubiatowicz. April: A processor architecture for multiprocessing. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 104–114, Seattle, WA, May 1990.

[2] T. Agerwala, J. L. Martin, J. H. Mirza, D. Sadler, D. M. Dias, and M. Snir. SP2 System Architecture. *IBM Systems Journal*. To appear.

[3] G. Agha. Concurrent object oriented programming. *Communications of the ACM*, 33(9):125–141, September 1990.

[4] G. A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. The MIT Press, Cambridge, MA, 1986.

[5] S. Ahuja, N. Carriero, and D. Gelernter. Linda and friends. *IEEE Computer*, 19(8):26–34, August 1986.

[6] P. America. POOL-T: A parallel object-oriented language. In A. Yonezawa and M. Tokoro, editors, *Object Oriented Concurrent Programming*, pages 199–220. MIT Press, Cambridge, MA, 1987.

[7] J. M. Anderson and M. S. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 112–125, Albuquerque, NM, June 1993.

[8] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *Proceedings of the 13th ACM Symposium on Operating System Principles*, pages 95–109, October 1991.

[9] G. R. Andrews and F. B. Schneider. Concepts and notations for concurrent programming. *ACM Computing Surveys*, 15(1):3–43, March 1983.

[10] J. S. Auerbach, A. S. Gopal, J. R. Russell, and M. T. Kennedy. Concert/C: Supporting distributed programming with language extensions and a portable multiprotocol runtime. In *Proceedings of the 14th International Conference on Distributed Computing Systems*, pages 152–159, Pozman, Poland, June 1994.

[11] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, March 1992.

[12] F. Baskett, T. Jermoluk, and D. Solomon. The 4D-MP graphics superworkstation: Computing + graphics = 40 MIPS + 40 MFLOPS and 100,000 lighted polygons per second. In *Proceedings of COMPCON '88*, pages 468–471, Feb. 1988.

[13] B. Beck. Shared-memory parallel programming in C++. *IEEE Software*, 7(4):38–48, July 1990.

[14] B. Bershad, E. Lazowska, and H. Levy. Presto: A system for object-oriented parallel programming. *Software—Practice and Experience*, 18(8):713–732, August 1988.

[15] A. Black, N. Hutchinson, E. Jul, and H. Levy. Object structure in the Emerald system. In *Proceedings of the OOPSLA-86 Conference*, pages 78–86, Portland, OR, September 1986.

[16] W. J. Bolosky, M. L. Scott, R. P. Fitzgerald, R. J. Fowler, and A. L. Cox. NUMA policies and their relation to memory architecture. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 212–221, Santa Clara, CA, April 1991.

[17] W. J. Bolosky, M. L. Scott, R. P. Fitzgerald, R. J. Fowler, and A. L. Cox. NUMA policies and their relation to memory architecture. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pages 212–221, Santa Clara, CA, April 1991.

[18] J. Boyle, R. Butler, T. Disz, B. Glickfield, E. Lusk, R. Overbeek, J. Patterson, and R. Stevens. *Portable Programs for Parallel Processors*. Holt, Rinehart and Winston, Inc., New York, NY, 1987.

[19] P. A. Buhr, G. Ditchfield, R. A. Stroobosscher, B. M. Younger, and C. R. Zarnke. uC++: Concurrency in the Object-oriented Language C++. *Software—Practice and Experience*, 22(2):137–172, February 1992.

[20] R. H. Campbell and A. N. Habermann. The specification of process synchronization by path expressions. In G. Goos and J. Hartmanis, editors, *Lecture Notes in Computer Science, Vol. 16*, pages 89–102. Springer-Verlag, 1974.

[21] L. Cardelli, J. Donahue, L. Glassma, M. Jordan, B. Kalsow, and G. Nelson. Modula-3 report. Technical Report 31, Digital Systems Research Center, Palo Alto, CA, August 1988.

[22] M. Cekleov, D. Yen, P. Sindhu, J.-M. Frailong, J. Gastine, M. Splain, J. Price, G. Beck, B. Liencres, F. Cerauskis, C. Coffin, and Basset. SPARCcenter 2000: multiprocessing for the 90's. In *Digest of Papers of COMPCON Spring '93*, pages 345–353, San Francisco, CA, February 1993.

[23] R. Chandra, S. Devine, B. Verghese, A. Gupta, and M. Rosenblum. Scheduling and page migration for multiprocessor compute servers. In *Proceedings of the Sixth Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 12–24, San Jose, CA, October 1994.

[24] R. Chandra, A. Gupta, and J. L. Hennessy. COOL: A language for parallel programming. In D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, pages 126–148, Urbana-Champaign, IL, 1990. MIT Press, Cambridge MA.

[25] R. Chandra, A. Gupta, and J. L. Hennessy. Data locality and load balancing in COOL. In *Fourth ACM SIGPLAN Symposium on the Principles and Practice of Parallel Programming (PPoPP)*, pages 249–259, San Diego, CA, May 1993.

[26] R. Chandra, A. Gupta, and J. L. Hennessy. COOL: An object-based language for parallel programming. *IEEE Computer*, 27(8):14–26, August 1994.

[27] Convex Computer Corporation. *Convex Exemplar: System Overview*. 3000 Waterview Parkway, Richardson, TX 75083-3851, 1994. Order No. 080-002293-000.

[28] E. C. Cooper and R. P. Draves. C threads. Technical Report CMU-CS-88-154, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, 1988.

[29] J. R. Cordy and R. C. Holt. Specification of Concurrent Euclid. Technical Report CSRI-133, Computer Systems Research Institute, University of Toronto, Toronto, Canada, August 1981.

[30] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, August 1975.

[31] T. W. Doeppner Jr. Threads: A system for the support of concurrent programming. Technical Report CS-87-11, Department of Computer Science, Brown University, Providence, RI, 1987.

[32] I. Duff, R. Grimes, and J. Lewis. Sparse matrix problems. *ACM Transactions on Mathematical Software*, 15(1):1–14, March 1989.

[33] B. Duzett and R. Buck. An overview of the nCUBE 3 supercomputer. In *Fourth Symposium on the Frontiers of Massively Parallel Computation*, pages 458–464, McLean, VA, October 1992.

[34] Encore Computer Corporation. *Multimax Technical Summary*. 1986.

[35] J. A. Feldman, C.-C. Lim, and F. Mazzanti. pSather monitors: Design, Tutorial, Rationale and Implementation. Technical Report TR-91-031, International Computer Science Institute, Berkeley, CA, September 1991.

[36] J. A. Fisher. Very long instruction word architectures and ELI-512. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, page 342, Sweden, Stockholm, June 1983.

[37] J.-M. Frailong, M. Cekleov, P. Sindhu, J. Gastinel, M. Splain, J. Price, and A. Singhal. The next-generation SPARC multiprocessing system architecture. In *Digest of Papers of COMPCON Spring '93*, pages 475–480, San Francisco, CA, February 1993.

[38] R. P. Gabriel and J. McCarthy. Queue-based multiprocessing lisp. In *ACM Symposium on Lisp and Functional Programming*, pages 25–44, August 1984.

[39] M. Galles and E. Williams. Performance optimizations, implementation, and verification of the SGI Challenge multiprocessor. In *Proceedings of the Twenty-Seventh Hawaii International Conference on System Sciences, Vol. I: Architecture*, pages 134–143, Wailea, Hawaii, January 1994.

[40] N. H. Gehani. Capsules: A shared memory access mechanism for concurrent C/C++. *IEEE Transactions on Parallel and Distributed Systems*, 4(7):795–811, July 1993.

[41] N. H. Gehani and W. D. Roome. Concurrent C++: Concurrent programming with class(es). *Software—Practice and Experience*, 18(12):1157–1177, December 1988.

[42] S. Gregory. *Parallel Logic Programming in PARLOG: the Language and its Implementation*. Addison-Wesley, Reading, MA, 1987.

[43] W. Groscup. The Intel Paragon XP/S Supercomputer. In G.-R. Hoffmann and T. Kauranne, editors, *Proceedings of the Fifth ECMWF Workshop on the Use of Parallel Processors in Meteorology. Parallel Supercomputing in Atmospheric Science*, pages 173–187, Reading, United Kingdom, November 1992.

[44] A. Gupta, A. Tucker, and L. Stevens. Making effective use of shared-memory multiprocessors: the process control approach. Technical Report CSL-TR-91-475A, Computer Systems Lab, Stanford University, Stanford, CA, May 1991.

[45] M. Gupta and P. Banerjee. Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers. *Transactions on Parallel and Distributed Systems*, 3(2):179–193, March 1992.

[46] R. H. Halstead. Parallel symbolic computing. *IEEE Computer*, 19(8):35–43, August 1986.

[47] P. B. Hansen. The programming language Concurrent Pascal. *IEEE Transactions on Software Engineering*, SE-1(2):199–207, June 1975.

[48] P. B. Hansen. *The Architecture of Concurrent Programs*. Prentice-Hall Inc, Englewood Cliffs, NJ, 1977.

[49] P. B. Hansen. The design of Edison. *Software—Practice and Experience*, 11(4):363–396, April 1981.

[50] P. B. Hansen. Edison: A multiprocessor language. *Software—Practice and Experience*, 11(4):325–361, April 1981.

[51] P. B. Hansen. Edison programs. *Software—Practice and Experience*, 11(4):397–414, April 1981.

[52] P. B. Hansen. Monitors and concurrent pascal: A personal history. In *Proceedings of the Second ACM SIGPLAN Conference on the History of Programming Languages (HOPL-II)*, pages 1–35, Cambridge, MA, April 1993. Published as ACM SIGPLAN Notices, 28(3).

[53] P. J. Hatcher, M. J. Quinn, A. J. Lapadula, B. K. Seevers, R. J. Anderson, and R. R. Jones. Data-parallel programming on MIMD computers. *Journal of Parallel and Distributed Computing*, 3(2):377–383, July 1991.

[54] C. A. R. Hoare. Towards a theory of parallel programming. In *Operating System Techniques*, pages 61–71. Academic Press, 1972.

[55] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[56] R. W. Hockney. Comparison of communications on the Intel iPSC/860 and Touchstone Delta. *Parallel Computing*, 18(9):1067–1072, September 1992.

[57] R. C. Holt. *Concurrent Euclid, The Unix System and Tunis*. Addison-Wesley, Reading, MA, 1983.

[58] R. C. Holt and J. R. Cordy. The Turing programming language. *Communications of the ACM*, 31(12):1410–1423, December 1988.

[59] W. Horwat, A. A. Chien, and W. J. Dally. Experience with CST: Programming and implementation. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 101–109, Portland, OR, 1989.

[60] John McCarthy. Recursive functions of symbolic expressions. *Communications of the ACM*, 3(4):184–195, April 1960.

[61] S. C. Johnson. Yacc—yet another compiler compiler. Technical Report 32, AT&T Bell Laboratories, Murray Hill, NJ, 1975.

[62] P. Jouvelot and D. K. Gifford. Parallel functional programming: the FX project. In M. Cosnard, Y. Robert, P. Quinton, and M. Raynal, editors, *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 257–267. North-Holland, October 1988.

[63] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[64] Kendall Square Research. *KSR1 Technical Summary*. Waltham, MA, 1992.

[65] K. Knobe, J. D. Lukas, and G. L. Steele. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *Journal of Parallel and Distributed Computing*, 8:102–118, 1990.

[66] S. Krakowiak, M. Meysembourg, V. H. Nguyen, M. Riveill, C. Roisin, and X. Rousset de Pina. Design and implementation of an object-oriented, strongly typed language for distributed applications. *Journal of Object-Oriented Programming*, 3(3):11–14, Sept-Oct 1990.

[67] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH multiprocessor. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 302–313, Chicago, IL, April 1994.

[68] M. S. Lam. Software pipelining: an effective scheduling technique for VLIW machines. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 318–328, Atlanta, GA, June 1988.

[69] M. S. Lam and M. C. Rinard. Coarse-grain parallel programming in Jade. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 94–105, Williamsburg, VA, April 1991.

[70] B. W. Lampson and D. D. Redell. Experience with processes and monitors in Mesa. *Communications of the ACM*, 23(2):105–117, February 1980.

[71] R. P. Larowe and C. S. Ellis. Experimental comparison of memory management policies for NUMA multiprocessors. *ACM Transactions on Computer Systems*, 9(4):319–363, November 1991.

[72] R. P. LaRowe Jr., C. S. Ellis, and M. A. Holliday. Evaluation of NUMA memory management through modeling and measurements. *IEEE Transactions on Parallel and Distributed Systems*, 3(6):686–701, November 1992.

[73] R. P. LaRowe Jr., C. S. Ellis, and L. S. Kaplan. The robustness of NUMA memory managament. In *Symposium on Operating System Principles*, pages 137–151, Pacific Grove, CA, October 1991.

[74] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. The Stanford Dash multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[75] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. L. Hennessy. The DASH prototype: Implementation and performance. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 92–103, Gold Coast, Australia, May 1992.

[76] S. T. Leutenegger and M. K. Vernon. The performance of multiprogrammed multiprocessor scheduling policies. In *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 226–236, Boulder, CO, May 1990.

[77] J. Li and M. Chen. Index domain alignment: Minimizing cost of cross-referencing between distributed arrays. In *Frontiers90*, pages 424–432, College Park, MD, October 1990. IEEE.

[78] D. B. Loveman. High performance Fortran. *IEEE Parallel and Distributed Technology, Systems & Applications*, 1(1):25–42, February 1993.

[79] P. G. Lowney, S. M. Freudenberger, T. J. Karzes, W. D. Lichtenstein, R. P. Nix, J. S. O'Donnell, and J. C. Ruttenberg. The multiflow trace scheduling compiler. *Journal of Supercomputing*, 7(1-2):51–142, May 1993.

[80] C. McCann, R. Vaswani, and J. Zahorjan. A dynamic processor allocation policy for multiprogrammed shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 11(2):146–178, May 1993.

[81] J. D. McDonald. Particle simulation in a multiprocessor environment. In *Proceedings of the 26th Thermophysics Conference*, June 1991. AIAA Paper No. 91-1366.

[82] Assembly language programmer's guide. Technical report, MIPS Computer Systems, Inc., Sunnyvale, CA, 1986.

[83] J. G. Mitchell, W. Maybury, and R. Sweet. Mesa language manual version 5.0. Technical Report CSL-79-3, Systems Development Department, Xerox Palo Alto Research Center, Palo Alto, CA, April 1979.

[84] S. Murer, J. A. Feldman, and C.-C. Lim. pSather: Layered Extensions to an Object-Oriented Language for Efficient Parallel Computation. Technical Report TR-93-028, International Computer Science Institute, Berkeley, CA, June 1993.

[85] R. K. Raj, E. Tempero, H. M. Levy, A. Black, N. Hutchinson, and E. Jul. Emerald: A general purpose programming language. *Software—Practice and Experience*, 21(1):91–118, January 1991.

[86] B. R. Rau and J. H. A. Fisher. Instruction-level parallel processing: history, overview, and perspective. *Journal of Supercomputing*, 7(1-2):9–50, May 1993.

[87] M. C. Rinard. *The Design, Implementation and Evaluation of Jade: A Portable, Implicitly Parallel Programming Language*. PhD thesis, Department of Computer Science, Stanford University, Stanford, CA, September 1994.

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

[89] J. S. Rose. Locusroute: A parallel global router for standard cells. In *Proceedings of the 25th ACM/IEEE Design Automation Conference*, pages 189–195, June 1988.

[90] E. Rothberg. *Exploiting the Memory Hierarchy in Sequential and Parallel Sparse Cholesky Factorization*. PhD thesis, Computer Systems Lab, Stanford University, Stanford, CA, November 1992. Technical Report CSL-TR-93-555.

[91] E. Rothberg and A. Gupta. Techniques for improving the performance of sparse matrix factorization on multiprocessor workstations. In *Supercomputing '90*, pages 232–243, New York, NY, November 1990.

[92] E. Rothberg and A. Gupta. An efficient block-oriented approach to parallel sparse cholesky factorization. *SIAM Journal on Scientific Computing*, 15(6):1413–1439, November 1994.

[93] J. P. Singh. *Parallel Hierarchical N-body methods and their Implications for Multiprocessors*. PhD thesis, Computer Systems Lab, Stanford University, Stanford, CA, March 1993. Technical Report CSL-TR-93-565.

[94] J. P. Singh and J. L. Hennessy. An empirical investigation of the effectiveness and limitations of automatic parallelization. In *Proceedings of the International Symposium on Shared Memory Multiprocessing*, April 1991.

[95] J. P. Singh and J. L. Hennessy. Finding and exploiting parallelism in an ocean simulation program: Experience, results, and implications. *Journal of Parallel and Distributed Computing*, 15(1):27–48, May 1992.

[96] J. P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared memory. *Computer Architecture News*, 20(1):5–44, March 1992.

[97] M. D. Smith, M. Horowitz, and M. S. Lam. Efficient superscalar performance through boosting. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, pages 248–259, Boston, MA, October 1992.

[98] M. D. Smith, M. S. Lam, and M. Horowitz. Boosting beyond static scheduling in a superscalar processor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 344–354, Seattle, WA, May 1990.

[99] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, MA, 1986.

[100] A. S. Tanenbaum, M. F. Kaashoek, and H. E. Bal. Parallel programming using shared objects and broadcasting. *IEEE Computer*, 25(8):10–19, August 1992.

[101] Thinking Machines Corporation. *Connection Machine CM-5 Technical Summary*. 245 First Street, Cambridge, MA 02142-1264, November 1993. Order No. 700-009002.

[102] R. M. Tomasulo. An efficient algorithm for exploiting multiple functional units. *IBM Journal of Research and Development*, 11(1):25–33, January 1967. Also published in Computer Structures: Principles and Examples, by Daniel P. Siewiorek, C. Gordon Bell, and Allen Newell, McGraw Hill, 1982, Chapter 19.

[103] C. Tomlinson and V. Singh. Inheritance and synchronization with enabled-sets. In *Proceedings of the OOPSLA-89 Conference*, pages 103–112, New Orleans, LA, October 1989.

[104] A. Tucker. *Efficient Scheduling on Multiprogrammed Shared-Memory Multiprocessors*. PhD thesis, Department of Computer Science, Stanford University, Stanford, CA, November 1993. Technical Report CSL-TR-94-601.

[105] R. Vaswani and J. Zahorjan. The implications of cache affinity on processor scheduling for multiprogrammed, shared memory multiprocessors. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 26–40, Pacific Grove, CA, October 1991.

[106] N. Wirth. Modula: A language for modular multiprogramming. *Software—Practice and Experience*, 7(1):3–36, January 1977.

[107] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 30–44, Toronto, Canada, June 1991.

[108] M. J. Wolfe. *Optimizing supercompilers for supercomputers*. MIT Press, Cambridge, MA, 1989.

[109] B. B. Wyatt, K. Kavi, and S. Hufnagel. Parallelism in object-oriented languages: A survey. *IEEE Software*, 9(6):56–66, November 1992.

[110] A. Yonezawa. Comments on monitors and path expressions. *Journal of Information Processing*, 1(4), 1979.

[111] A. Yonezawa, J.-P. Briot, and E. Shibayama. Object oriented concurrent programming in ABCL/1. In *Proceedings of the OOPSLA-86 Conference*, pages 258–268, Portland, OR, September 1986.

[112] H. Zima and B. Chapman. *Supercompilers for parallel and vector computers*. Addison-Wesley, Reading, MA, 1990.