New Techniques for Compiling Data-Parallel Languages

Guhan Viswanathan

Under the supervision of Associate Professor James R. Larus

at the University of Wisconsin — Madison

Data-parallel programming languages provide a portable, high level abstraction to support rapid parallel program development. Data-parallel languages are widely applicable, and considerable research has been devoted to compiling them for efficient execution. Much of this work has focused on implementing a shared address space on a distributed-memory machine, particularly for programs with regular communication patterns. Compiler support for non-regular applications has been limited to iterative irregular applications, with dynamic repetitive communication patterns.

This thesis describes three new compiler, language and run-time system techniques targeting data-parallel programs with adaptive and dynamic communication patterns. These techniques have been implemented in a compiler for the language C**, and include:

Implementing conflict-free data access C**'s semantics avoid data access conflicts between parallel functions. We explore and compare two complementary implementations of conflict-free access. For parallel functions with reg-

ular data access patterns, the compiler inserts code in the program to maintain copies. When compiler data access analysis is imprecise, the compiler relies on a Loosely-Coherent Memory system to create transparent fine-grain copies.

User-defined reductions Most data-parallel languages restrict reductions, which combine values from parallel operations, to a predefined set of reduction operators. User-defined reductions extend reductions in two ways, by allowing new combining operations, and by applying reductions to user-defined data types. This thesis motivates the need for user-defined reductions, and describes the design and efficient implementation of reductions in C** with only message-passing support.

Compiler-directed shared-memory communication This thesis describes how a compiler and a predictive cache-coherence protocol can implement shared-memory communication efficiently for iterative adaptive applications. The compiler uses data-flow analysis to identify points in a program where *potential* repetitive communication patterns exist. An incremental predictive protocol builds a communication schedule for one iteration and utilizes a schedule to pre-send data in subsequent iterations. The protocol reduces the number of remote data requests, and the total remote access latency.

New Techniques for Compiling Data-Parallel Languages

by

Guhan Viswanathan

A dissertation submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

(Computer Science)

at the

University of Wisconsin — Madison

1996

New Techniques for Compiling Data-Parallel Languages

by

Guhan Viswanathan

A dissertation submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

(Computer Science)

at the

University of Wisconsin — Madison

1996

New Techniques for Compiling Data-Parallel Languages

by

Guhan Viswanathan

A dissertation submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

(Computer Science)

at the

University of Wisconsin — Madison

1996

© Copyright by Guhan Viswanathan, 1996

All Rights Reserved

Abstract

Data-parallel programming languages provide a portable, high level abstraction to support rapid parallel program development. Data-parallel languages are widely applicable, and considerable research has been devoted to compiling them for efficient execution. Much of this work has focused on implementing a shared address space on a distributed-memory machine, particularly for programs with regular communication patterns. Compiler support for non-regular applications has been limited to iterative irregular applications, with dynamic repetitive communication patterns.

This thesis describes three new compiler, language and run-time system techniques targeting data-parallel programs with adaptive and dynamic communication patterns. These techniques have been implemented in a compiler for the language C**, and include:

Implementing conflict-free data access C**'s semantics avoid data access conflicts between parallel functions. We explore and compare two complementary implementations of conflict-free access. For parallel functions with regular data access patterns, the compiler inserts code in the program to maintain copies. When compiler data access analysis is imprecise, the compiler relies on a Loosely-Coherent Memory system to create transparent fine-grain copies.

User-defined reductions Most data-parallel languages restrict reductions, which combine values from parallel operations, to a predefined set of reduction operators. User-defined reductions extend reductions in two ways, by allowing

new combining operations, and by applying reductions to user-defined data types. This thesis motivates the need for user-defined reductions, and describes the design and efficient implementation of reductions in C** with only message-passing support.

Compiler-directed shared-memory communication This thesis describes how a compiler and a predictive cache-coherence protocol can implement shared-memory communication efficiently for iterative adaptive applications. The compiler uses data-flow analysis to identify points in a program where *potential* repetitive communication patterns exist. An incremental predictive protocol builds a communication schedule for one iteration and utilizes a schedule to pre-send data in subsequent iterations. The protocol reduces the number of remote data requests, and the total remote access latency.

Acknowledgments

My many years at Madison have been enriched by interactions with a bunch of wonderful people.

I have learnt a tremendous amount from my advisor, Jim Larus. Jim sets a wonderful example of how to do successful research, and I hope I have picked up some of it. I am also very grateful for his advice, and his constant support and encouragement.

I would also like to thank the other members of my committee, Mark Hill, Charles Fischer, Susan Horwitz, and Sang-tae Kim for taking the time to read the thesis and for their comments.

Much of this work would not have been possible without software developed as part the Wisconsin Wind Tunnel project. I've enjoyed interacting with many people in the group, particularly the numerous discussions with Brad, my office (and project) partner of many years, and the talks in the hall with Satish and Zhichen.

On a more personal note, I'm much indebted to my wife Vidya, both for her support, and for all the fun times together. I'd also like to thank some of my friends for enlivening those few non-working hours: Bajji, Viji and Swami,

Praveen and Ranju, Madhus and Sridevi, Ranga, Guru, Bharat and Madhav.

Thanks all.

GUHAN VISWANATHAN

University of Wisconsin — Madison

September 1996

Contents

Abstract i

Acknowledgments iii

List of Figures ix

List of Tables xii

1 Introduction 1

1.1 Implementing Conflict-free Data Access 5

1.2 User-defined Reductions 7

1.3 Compiler-Directed Shared-Memory Communication 8

1.4 Contributions 10

1.4.1 Generality of our Techniques 11

1.5 Thesis Structure 12

2 Background - Data Parallelism, C, and Tempest** 13

2.1 Data Parallelism, Data-Parallel Languages 14

2.1.1 Parallelism - Data-Parallel Operations 15

2.1.2 The Shared Address Space 17

2.2 C** 19

2.2.1 Data Collections - Aggregates 19

2.2.2 Data-Parallel Operations - Parallel Functions 20

2.2.3 Reductions in C** 22

2.3 Tempest 22

2.3.1 Tempest Implementations 23

3 Implementing Conflict-free Data Access for Parallel Functions 25

3.1 Related Work 27

3.2 Compiler Copying 28

3.2.1 No Conflicts 28

3.2.2 Single Writer, Multiple Readers 29

3.2.3 Zero or One Writer 30

3.2.4 Multiple Unknown Writers 31

3.3 LCM 32

3.4 Performance Comparison 37

3.5 Summary 39

4 User-defined Reductions for Efficient Communication 40

4.1 Related work 43

4.2 Reductions in Data-Parallel Languages 44

4.2.1 User-defined Reductions 46

4.3 A Motivating Example 48

4.3.1 DSMC 49
4.3.2 Particle Movement using Parallel Prefix 50
4.3.3 Particle Movement using a Predefined Reduction 52
4.3.4 Particle Movement with User-Defined Reductions 52
4.4 Semantics of User-Defined Reductions 53
4.4.1 Data races 54
4.4.2 Reordering Combining Operations 54
4.5 Implementing Reductions 55
4.5.1 Basic Reductions 55
4.5.2 Bulk reductions 56
4.5.3 Local Combining 57
4.6 Application Comparisons 57
4.6.1 DSMC 59
4.6.2 Barnes 60
4.6.3 EM3D 61
4.6.4 Moldyn 63
4.6.5 Discussion 65
4.7 Summary 66

5 Compiler-Directed Shared-Memory Communication for Iterative Applications 68

5.1 Related Work 71
5.2 A Predictive Protocol for Repetitive Communication Schedules . . 73
5.2.1 The Stache Shared-Memory Protocol 74

5.2.2 Inefficiencies in a Write-invalidate Protocol 75

5.2.3 Building Communication Schedules in the Predictive Protocol 76

5.2.4 Using Communication Schedules to Presend Data 77

5.3 Identifying Potentially Repetitive Patterns 78

5.3.1 Parallel Function Analysis - Identifying Access Patterns . . 79

5.3.2 Compiler Analysis to Place Directives 81

5.4 Measuring the Optimizations 84

5.4.1 Adaptive 85

5.4.2 Barnes 87

5.4.3 Water 89

5.4.4 Discussion 90

5.5 Summary 91

6 Conclusion **93**

A C Benchmarks** **98**

List of Figures

2.1 Interleaved sequential and parallel phases in a data-parallel program's execution 15
2.2 Whole-array stencil operation in HPF 15
2.3 Example showing data distribution and communication (shaded elements) in a 4-point stencil implementation 18
2.4 MAX reduction operation in HPF 19
2.5 Aggregate definition syntax in C** 20
2.6 Example showing conflict-free data access semantics guaranteed by C** 21
2.7 Aggregate definition syntax in C** 22

3.1 4-point stencil 29
3.2 4-point stencil with compiler-generated copying 30
3.3 Schematic representation of compiler-copying for Stencil 31
3.4 4-point threshold stencil 32
3.5 Threshold stencil 33
3.6 `unstructured` - Parallel function with unstructured accesses . . . 34

3.7 Compiler-generated pseudo-code for `unstructured` 35

3.8 Compiler-generated pseudo-code for `unstructured` using LCM . . 36

3.9 Relative execution speed for compiler-copying and LCM versions of 3 benchmarks — Stencil, Threshold and Adaptive 38

4.1 Sum reduction assignment 45

4.2 Minimum location user-defined reduction 47

4.3 Two ways of visualizing a reduction of two values to a target: Combining and Update, or Synchronized Accumulation 48

4.4 Schematic of the DSMC application 49

4.5 Schematic of DSMC data structures and the problem of synchronized addition 50

4.6 Schematic of DSMC synchronization with parallel prefix 51

4.7 Schematic of DSMC synchronization with `APPEND` 52

4.8 Schematic of particle movement with user-defined reduction `add_particle` 53

4.9 Schematic representation of basic reductions (left) and bulk communication (right) for DSMC 56

4.10 Log-log scale graphs showing execution speeds of 2 or more versions of DSMC on the CM-5 and COW. Numbers adjoining the curves are speedups relative to a sequential version. 59

4.11 Log-log scale graphs showing execution speeds of 2 or more versions of EM3D on the CM-5 and COW. Numbers adjoining the curves are speedups relative to a sequential version. 62

4.12 Log-log scale graphs showing execution speeds of 2 or more versions of Moldyn's force computation phase on the CM-5 and COW. Numbers adjoining the curves are speedups relative to a sequential version. 64

5.1 Unstructured mesh update in C** 80

5.2 Control flow graph for the main sequential loop in Barnes-Hut. CFG (a) is annotated with parallel function access patterns. CFG (b) is annotated with runtime phase directives for the predictive protocol. 82

5.3 Execution time for 4 C** versions of **Adaptive** — C** versions with and without optimized communication at 2 different cache block sizes. Numbers in parentheses indicate cache block sizes. . . 86

5.4 Execution time for 5 versions of **Barnes** — C** versions with and without optimized communication at 2 different cache block sizes, and hand-optimized SPMD. Numbers in parentheses indicate cache block sizes. 88

5.5 Execution time for 3 versions of **Water** — C** with and without optimized communication, and shared-memory Splash. Numbers in parentheses indicate cache block sizes. 89

List of Tables

4.1 Comparing three approaches to particle movement 53

4.2 High-level application description and data sets. The larger data sets are for COW runs. 58

5.1 Benchmark applications 85

A.1 A list of benchmarks in C** 99

Chapter 1

Introduction

Parallel computers exploit the processing power of multiple CPUs connected by a network to reduce the time it takes to execute a program. As CPU processing speeds reach their limit, parallel processing will provide an increasingly important and scalable way to utilize available processing power. Additionally, parallel computers can run applications with larger data sets than are possible on uniprocessors. Parallel processing now encompasses a wide range of platforms varying in cost and performance from multiprocessors to networks of workstations. Unfortunately, rapid advances in hardware technology have left parallel software behind. Writing programs for parallel computers is widely acknowledged to be a much more difficult task than writing sequential programs. For example, the programmer must keep track of multiple asynchronous threads of control that interact in a non-deterministic fashion.

Improved parallel programming languages reduce the difficulty of programming parallel computers by making parallel programs easy to specify, less error

prone, and less machine specific. One promising approach is data-parallel languages, such as HPF [19], C* [41], or NESL [7], which provide a portable, high level abstraction to support rapid parallel program development. These languages implement the data-parallel programming paradigm, which expresses parallelism through simultaneous operations on large sets of data [21]. Synchronization is implicit in the division of a program into sequential and data-parallel phases. In addition, a global name space of variables relieves a programmer of the burden of distributing data and managing communication. The data-parallel programming model is widely applicable — in a survey of 84 scientific applications, Fox [16] found that 85% of those programs could be expressed easily using this model — "the source of parallelism is essentially always domain decomposition or data parallelism".

The popularity of the data-parallel programming model has led to the development of a large number of data-parallel programming languages, including High Performance Fortran (HPF) [25], Fortran 90 [1], NESL [7] and pC++ [32]. In addition to language development, considerable research has been devoted to compiling high-level data-parallel programs for efficient execution, and in particular the problem of transforming shared address space operations into communication primitives of a distributed-memory machine. Much of this work has focused on *regular* programs, which exhibit regular patterns of communication, i.e., static communication patterns that can be identified at compile time using static compiler analysis techniques [3, 22, 40].

Compiler support for non-regular data-parallel applications on distributed-

memory machines has been limited to iterative irregular applications that fit the Inspector-Executor model. These applications specify unpredictable communication patterns that cannot be analyzed at compile time, but the patterns, once built, remain unchanged over a number of iterations. For each loop, the compiler generates an inspector phase in the loop header, which builds a communication schedule. The loop contains the executor phase, which uses the schedule to transfer data required in the iteration efficiently with messages, followed by the loop's original computation [12, 24, 46]. Since the pattern varies infrequently, the overhead of the inspector is amortized over multiple executor phases. The Inspector-Executor paradigm works well for mostly-static communication patterns, but does not extend to dynamic communication patterns.

This thesis presents three new compiler, language and run-time system techniques that enable efficient execution of a larger class of data-parallel programs, specifically, programs with dynamic adaptive communication patterns. These techniques are:

1. Compiler and memory-system support to enforce conflict-free data access

2. User-defined reductions for efficient communication, and

3. Compiler-directed shared-memory communication for iterative applications

A novel feature of techniques 1 and 3 is that they rely on and exploit the flexibility of user-level control over shared-memory coherence policies in a cache-coherent distributed shared-memory (DSM) system, which implements a shared-address space on distributed-memory hardware using a combination of hardware and software

techniques [29, 38].User-defined reductions (technique 2) and compiler-directed shared-memory communication (technique 3) both optimize communication in dynamic data-parallel applications, but target distinct communication patterns, and their effect is cumulative. User-defined reductions optimize many-to-one communication with combining, whereas compiler-directed shared-memory communication optimizes one-to-many producer-consumer or migratory sharing patterns.

User-customizable DSMs enable a compiler and run-time system to tailor coherence policies to applications, both to implement higher-level functionality and to improve performance [39]. A number of such systems exist (e.g., Tempest [38], FLASH [29], TreadMarks [4], and Sequent's STiNG [34]), and they are becoming more popular. Some other compiling approaches also target flexible DSMs [8, 22], but focus on efficient DSM support for regular applications.

This thesis demonstrates that our techniques are effective. Using a suite of popular benchmark applications consisting of irregular and adaptive applications, we show that programs written in a high-level data-parallel language can be compiled to execute as fast as equivalent hand-optimized code on the CM-5, a distributed-memory multiprocessor. Our benchmark suite does not include regular applications, although the communication optimization techniques also apply to those applications. Furthermore, these three techniques focus on only two facets of data-parallel program compilation for irregular and dynamic programs, i.e., conflict-free data access and efficient communication. This thesis does not tackle other important problems, such as automatic load balancing or efficient data distributions.

We implemented these techniques in a compiler for the language C** [31] that targets the Tempest interface. C** is a data-parallel language that provides high-level language features like a global name space and parallelism through simultaneous operations on data. C** provides coarse-grain parallelism in the form of user-defined parallel operations. Tempest [38] is a programming interface for a distributed-memory multiprocessor that provides the mechanisms to implement fine-grain cache-coherent shared memory in addition to message-passing primitives. Tempest combines good features of both message passing and shared memory; a compiler can use the fine-grain shared memory mechanisms to implement a global name space, and use custom coherence protocols or low-level communication facilities to optimize known communication patterns [28].

1.1 Implementing Conflict-free Data Access

The first technique targets the implementation of conflict-free data access in C**'s parallel operations [30]. C** supports coarse-grain user-defined parallel functions (Section 2.2) and a global name space of variables, which together give rise to the possibility of data access conflicts or data races. A data race occurs when two distinct parallel tasks access a single global data item, and one of the accesses is a write (Section 2.2.2). Data races are problematic because different temporal access orderings could generate different results, causing errors that are extremely difficult to reproduce and to correct.

C** clearly defines the semantics of conflicting memory accesses to avoid data-access conflicts. When a data-parallel operation modifies a global data item, it

receives a private copy of that data item, which is not visible to other data-parallel operations. When all operations complete, the private copies are merged into the global state. Lack of conflicts translates into nearly-deterministic execution, which is a desirable feature of some data-parallel programming languages. We explore two ways in which a compiler and run-time system can use copy-on-write to implement the high-level semantics of C**. For parallel functions with regular data-access patterns that a compiler can analyze, the compiler inserts code in the program to maintain copies. For functions with accesses that a compiler cannot identify precisely (e.g., accesses through pointers), compiler-copying results in excessive and potentially expensive run-time checks. Instead, the compiler relies on a Loosely-Coherent Memory (LCM) system to create transparent fine-grain copies. LCM is driven by runtime compiler directives (e.g., to identify modified global data items) and uses a custom cache coherence protocol to create copies of cache blocks at the same global address. The LCM protocol provides controlled inconsistency of global data items during parallel execution, which matches the semantic requirements of C**.

Using performance data from three variants of mesh relaxation codes, we show that these two techniques complement one another. Compiler-copying is efficient when compiler data access analysis is precise, and LCM works well when the analysis is imprecise. The benefit of providing two alternatives is that a compiler can choose the efficient alternative based on the precision of its data access analysis, and even use both in a program.

1.2 User-defined Reductions

The second part of the thesis describes the design and implementation of user-defined reductions. Data-parallel languages typically allow reductions to combine values from independent parallel operations (Section 2.1.2). Reductions are extremely common in parallel applications, and can be implemented efficiently in parallel. However, most data-parallel languages restrict reductions to a predefined set of reduction operators, typically arithmetic operations on basic types. We demonstrate that parallel languages need not and should not arbitrarily limit reductions in this way. User-defined reductions extend reductions in two dimensions. First, they allow new operations to combine values, such as building a list from colliding values. Second, they allow reductions to apply to user-defined data types.

This thesis demonstrates the advantages of user-defined reductions, focusing on the benefits of extending reductions to user-defined data types. To motivate user-defined reductions, we describe and compare three real-life data-parallel implementations of inter-cell particle movement in a particle-in-cell code. The first method, which uses a parallel prefix operation, is both cumbersome to specify and inefficient in practice. The second method, which uses a predefined `APPEND` reduction, works well, but is not flexible, and does not directly specify a producer-consumer pattern. The third method, user-defined reductions, allows the programmer to intuitively specify particle movement, and leads to an efficient implementation.

We also present a simple implementation of user-defined reductions in C**

to show that they can be implemented efficiently with message-passing support. The basic implementation uses messages to transfer reduction data, and two well-known optimizations, message vectorization and local combining, can be applied to improve reduction performance. We compared the execution time of four C** applications (with reductions) against equivalent hand-optimized versions. On a 32-node CM-5 and a 16-node Cluster of Workstations (COW), both versions were comparable on 3 out of 4 benchmarks, all of which had dynamic communication patterns. On the fourth benchmark, which exhibited a repetitive static communication, the message-passing hand-optimized version was considerably faster. C**'s reduction implementation does not optimize static reduction patterns, using, for example the Inspector-Executor compiling approach.

1.3 Compiler-Directed Shared-Memory Communication

The third technique describes how a data-parallel language compiler and a custom cache-coherence protocol can implement shared-memory communication efficiently for applications with unpredictable but repetitive communication patterns. This technique applies to a large class of scientific applications that are iterative — each iteration simulates the evolution of a physical system over time. For example, in static mesh calculations, nearest-neighbor communication is repeated in each iteration. In some irregular problems, such as molecular dynamics codes [46], communication changes infrequently, perhaps once every 20-30 iterations. In

adaptive problems, communication changes frequently, but incremental changes between iterations are small. For example, structured adaptive meshes gradually add mesh nodes for greater accuracy in each iteration [27], and gravitational N-body problems represent bodies in a quad-tree, which undergoes small structural changes between iterations.

This section shows that a compiler for a data-parallel language can cooperate with a predictive cache-coherence protocol in a distributed shared-memory (DSM) system to optimize shared-memory communication for applications with dynamic, but repetitive communication patterns. The compiler uses data-flow analysis to identify points in the program where *potential* repetitive communication patterns exist. A predictive protocol in the runtime system augments the default shared-memory protocol to build a communication schedule for one iteration and utilize a schedule to pre-send data to satisfy data requests in following iterations. As a result, the predictive protocol reduces the number of shared-memory data requests that cannot be satisfied locally, and the total remote memory access latency.

Compiler-directed shared-memory communication uses a combination of two techniques — a predictive cache coherence protocol, and simple compiler analysis — for optimizing shared-memory communication. The predictive protocol builds dynamic incremental communication schedules — new requests not satisfied by the pre-send phase are added to the schedule for subsequent iterations. This approach has the advantage that it can be applied to adaptive applications with repetitive dynamic communication patterns that a compiler cannot analyze. Simple compiler analysis automatically applies the predictive protocol for applications with

repetitive producer-consumer sharing patterns for which a sequentially-consistent memory coherence protocol would incur large overheads [9]. By contrast, compilers targeting message-passing machines must identify and fully analyze run-time communication patterns in applications. Our simple analysis only identifies program points at which potentially repetitive communication takes place, but need not identify the patterns themselves.

We measured the benefits of compiler-directed shared-memory communication by comparing the execution time of optimized and non-optimized versions of three applications (Adaptive, Water, and Barnes) on a 32-processor CM-5. In all cases, the optimized version had significantly less remote latency. For Adaptive and Water, the optimized version was faster than the best non-optimized version. For Barnes, which shows excellent spatial locality, the optimized and non-optimized versions are comparable.

1.4 Contributions

The important contribution of this thesis is to show that, for a larger class of scientific applications that includes irregular and dynamic applications, programs written in a high-level data-parallel language can be compiled to run as efficiently as hand-optimized code on distributed-memory multiprocessors. Ideally, we would have liked to compare the performance of data-parallel programs against equivalent message-passing versions, which typically demonstrate the best performance. However, message-passing versions are extremely difficult to develop, and we settled for the best parallel version of each application that was available to us.

Towards this goal, we present three new techniques for compiling data-parallel programs, including compiler-copying and LCM to prevent data-access conflicts, user-defined reductions for efficient communication and compiler-directed shared-memory communication for iterative applications. We describe each technique in detail, and demonstrate that they produce executable programs that run as efficiently as equivalent hand-tuned codes.

We also show that these techniques apply to a wider class of applications, i.e., those with irregular and dynamic communication patterns, than have been previously considered. For example, our suite of benchmarks includes adaptive structured mesh codes and a data-parallel implementation of the Barnes-Hut algorithm, a dynamic hierarchical gravitational N-body code (Table A.1).

1.4.1 Generality of our Techniques

Although we have developed these techniques in the context of C** and Tempest, they should readily extend to other data-parallel languages and DSM platforms.

All the compiler techniques developed in this thesis are also applicable to other data-parallel languages with coarse-grain data parallelism, including popular languages such as HPF [25]. Compiler copying for conflict-free data access (Chapter 3) is already required in a limited form for array assignment statements by the HPF standard. The upcoming HPF-2 standard is considering allowing user-defined reductions in HPF's coarse-grain `DO INDEPENDENT` loops [14]. Since C**'s parallel functions and HPF's coarse-grain `DO INDEPENDENT` loops provide very similar functionality, our reduction implementation (Chapter 4) should di-

rectly extend to supporting HPF programs. Finally, compiler analysis for optimized shared-memory communication (Chapter 5) is designed for languages with coarse-grain data parallelism.

Two of the techniques presented in this thesis (LCM and the predictive protocol), rely on the ability to customize cache-coherence protocols in a DSM system. Many recent DSM systems provide this flexibility, although to varying degrees. Tempest and TreadMarks provide maximum flexibility by implementing all coherence actions in user-level software. By contrast, FLASH and STiNG use programmable engines to implement protocol actions, allowing the moderate level of protocol customization that is required by our techniques.

1.5 Thesis Structure

This thesis contains six chapters. Chapter 2 introduces background material explaining the data-parallel model, C**, and Tempest. Chapter 3 explores and contrasts two possible implementations of conflict-free data access. Chapter 4 describes the implementation and performance of user-defined reductions in C**. In Chapter 5, we show how a data-parallel language compiler can cooperate with a custom protocol to implement shared-memory communication efficiently for iterative parallel applications. Chapter 6 concludes the thesis.

Chapter 2

Background - Data Parallelism, C**, and Tempest

This thesis presents three new techniques for compiling data-parallel programs for efficient execution. To understand these techniques, it is necessary to understand the context in which they apply, in particular, the language input to the compiler, and the target machine. In this section, we describe these two interfaces for the compiler that we developed as part of this thesis. The compiler takes as input programs written in the data-parallel language C** and generates executable programs targeting the Tempest parallel programming interface on distributed-memory multiprocessors.

This section describes C** and Tempest in some detail, specifically from a language implementor's point of view. It begins with a brief description of the data-parallel programming model and data-parallel programming language features, both to fit C** in the larger data-parallel context, and to show that although we

have developed these techniques in the context of C**, they also apply to other popular data-parallel programming languages.

2.1 Data Parallelism, Data-Parallel Languages

The distinguishing feature of the data-parallel programming model is in how a programmer specifies parallelism: a data-parallel program is organized around simultaneous operations on collections of data, in contrast to control-parallel programming's view of tasks, communication and synchronization [21]. In this model, the programmer builds a (typically large) data collection and invokes a single parallel operation on the collection — the compiler and run-time system ensure that the operation is replicated and applied in parallel on all elements of the collection.

The data-parallel programming model is popular because it provides a common abstraction underlying a large number of scientific applications [16]. As a result, a number of programming languages are based on the data-parallel model (Connection Machine Lisp [49], parts of HPF [19], ZPL [33], etc). In addition to support for data collections and data-parallel operations, data-parallel languages offer other high-level abstractions (e.g., a global name space) and potentially deterministic execution to support rapid high-level parallel program development.

This section outlines salient features of the data-parallel programming model and data-parallel programming languages. More detailed descriptions can be found elsewhere [17].

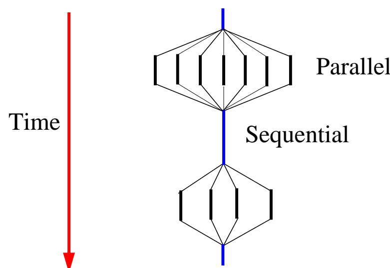

Figure 2.1: Interleaved sequential and parallel phases in a data-parallel program's execution

```
A(1:N,1:N) = (A(0:N-1,1:N)+A(2:N+1,1:N)+A(1:N,0:N-1)+A(1:N,2:N+1))/4.
```

Figure 2.2: Whole-array stencil operation in HPF

2.1.1 Parallelism - Data-Parallel Operations

A data-parallel program explicitly specifies parallel execution by invoking a data-parallel operation on a data collection. Invoking a data-parallel operation creates multiple operation invocations, one for each element of the collection. The parallel operation completes only when all individual invocations complete, clearly dividing a program's execution into distinct sequential and parallel phases (Figure 2.1). Whole-array operations in HPF provide a good example of data-parallel operations. For example, the 4-point stencil in Figure 2.2 extends the primitive addition operator to apply elementwise on array data aggregates. An analogous 4-point stencil in C** is outlined in Figure 2.2.2.

Data-parallel operations provide high-level abstractions for parallel processing. Parallelism and synchronization are implicit in the execution of a data-parallel

operation, and the language provides no other primitives for synchronization (such as locks or mutual exclusion) that are commonly available in the Single Program Multiple Data (SPMD) style of programming.

Data-parallel operations in data-parallel languages can be broadly characterized as fine-grain or coarse-grain, depending on the maximum granularity of each parallel operation (i.e., the amount of work that can be done between synchronization points). Fine-grain data-parallel operations originally evolved from SIMD machines, such as the Connection Machine CM-2 [20], which execute individual instructions in lockstep on multiple processing units. Fine-grain data-parallel operations directly reflect this execution model in a data-parallel operation's semantics, limiting a data-parallel operation's granularity to a primitive language operation (e.g., addition, assignment). Fine-grain data-parallel operations inherit both the advantages and disadvantages of the hardware model. The model offers the simplicity of a single thread of execution and the absence of data races, but suffers from the inefficiencies of conditional statements and from the synchronization necessary to execute fine-grain operations on MIMD processors.

By contrast, coarse-grain parallel operations allow arbitrary user-defined code to execute as part of a parallel operation between synchronization points. HPF's `DO INDEPENDENT` loops [19] and pC++'s parallel member functions [32] are good examples of coarse-grain parallel operations. Coarse-grain parallel operations map easily to MIMD execution, but asynchronous execution in a global address space allows data races, which lead to errors that are difficult to reproduce and therefore extremely difficult to debug.

Finally, the distinction between fine-grain and coarse-grain operations is meaningless for purely functional data-parallel languages, such as NESL [7], which do not allow imperative updates to global variables.

Implementing Parallel Functions When a data-parallel operation is invoked, the compiler and run-time system are responsible for replicating the operation and applying it in parallel on each element of the data collection. On a multiprocessor machine, this task can be divided into two parts, both of which are conceptually simple. First, the total work for the parallel operation is partitioned among the processors, usually following the data distribution for the data collection. Second, each processor executes a loop iterating the data-parallel operation over all elements of the data collection that it owns.

2.1.2 The Shared Address Space

Data-parallel languages provide a global name space of variables and data-parallel programs specify communication through read and write accesses to variables in this name space, just as shared-memory parallel programs do. For example, for a 4-point stencil operation on a 4x4 array distributed blockwise among 4 processors, all interior elements of the array must be communicated between neighboring processors (Figure 2.3).

Implementation Considerations On a distributed-memory multiprocessor, the compiler and run-time system must implement a shared-address space, i.e. they must implement a global name space by distributing data among processors,

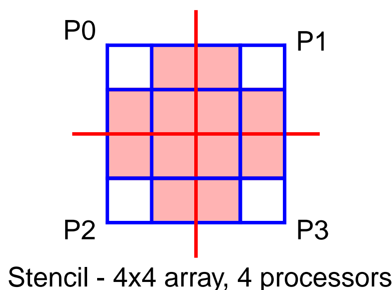

Figure 2.3: Example showing data distribution and communication (shaded elements) in a 4-point stencil implementation

and translate implicit communication through global variable access into explicit communication directives. Our implementation relies on a distributed shared-memory (DSM) machine, which transparently implements a shared address space, but may not do so efficiently. Efficient communication is vital to achieving good parallel application performance, and Chapters 4 and 5 explore techniques to implement implicit communication efficiently on DSMs.

Reductions

Communication in a shared-address space only suffices for one-to-one or one-to-many communication. Many-to-one communication causes data conflicts or collisions, when multiple values are stored in a location. Data-parallel languages typically use binary reduction operators to combine colliding values into a single value that can be stored in a location. Reductions are extremely common in parallel applications, even those written in languages that do not provide first-class support for these operations. For example, the `MAX` reduction shown in Figure 2.4 captures the maximum value in 2-dimensional array `A` in variable `max`.

```
max = MAXVAL(A, 2)
```

Figure 2.4: MAX reduction operation in HPF

Most data-parallel languages restrict reduction operators to a limited set of predefined reduction functions — typically, the associative arithmetic and logical operations.

We explore the implementation of reductions in more detail in Chapter 4.

2.2 C**

C** is a data-parallel programming language based on C++ [31]. It includes a number of desirable features of data-parallel programming languages — data collections (called Aggregates), data-parallel operations (called parallel functions), a global name space of variables, and sophisticated reduction operations. This section introduces examples to briefly illustrate the syntax and semantics of C** programs.

2.2.1 Data Collections - Aggregates

Data collections in C** are called Aggregates, and form the basis for parallelism. An Aggregate declaration, which uses different syntax from class and array declarations in C++, specifies an ordered collection of values, much like a multidimensional array of objects, that can be operated on by a data-parallel operation. For example, Figure 2.5 declares a two-dimensional collection of floating point values.

```
class Grid(float) [][]) {   /* Member functions */   };
```

Figure 2.5: Aggregate definition syntax in C**

```
void stencil (parallel Grid A) parallel
{
    A[#0][#1] = (A[#0-1][#1] + A[#0+1][#1] +
                 A[#0][#1-1] + A[#0][#1+1]) / 4.0;
}
```

The size of an Aggregate may be specified in the Aggregate declaration, or at runtime when an Aggregate object is created.

2.2.2 Data-Parallel Operations - Parallel Functions

C** provides coarse-grain data-parallel operations called *parallel functions*. A parallel function is identified by the keyword `parallel`, and includes a *parallel argument*, which indicates the Aggregate object to which the parallel function is applied. Figure 2.2.2 specifies a stencil computation on the `Grid` Aggregate. Since the data-parallel function is replicated and applied on all elements of the two-dimensional grid, the pseudo variables `#0` and `#1` identify row and column positions for the grid element allotted to an invocation of the parallel function.

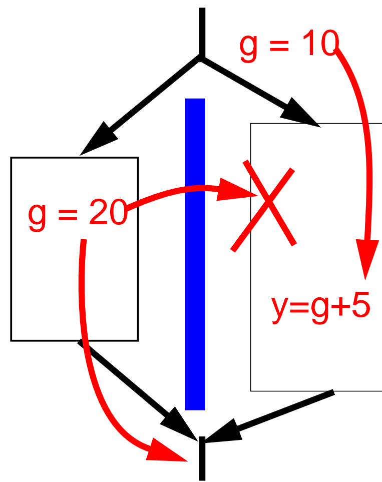

Figure 2.6: Example showing conflict-free data access semantics guaranteed by C**

C's Parallel Function Semantics**

C** allows coarse-grain parallel functions, but clearly defines the semantics of conflicting memory access to avoid non-deterministic data races. C** specifies that all parallel function invocations start from the same global state, and incur no conflicts. When an invocation updates a global value, the new value is visible only to that invocation until the data-parallel operation completes (Figure 2.6). At that point, all changes are merged into a single consistent global state. In the stencil parallel function (Figure 2.2.2), C**'s semantics specify that each point in the stencil averages *old* values of neighboring points, because a neighbor's changes are not visible until the operation itself completes.

```
void max_grid(parallel Grid A) parallel
{
    max =%> A[#0][#1];
}
```

Figure 2.7: Aggregate definition syntax in C**

2.2.3 Reductions in C**

C**, like other imperative data-parallel languages, provides reduction assignments to combine colliding values. For example, Figure 2.7 assigns the maximum of a two-dimensional grid `A` to variable `max`. Reductions use slightly different syntax from other assignments, and also have slightly different semantics. In C**, the result of a reduction is available at the target of the assignment only when all data-parallel operations complete. In other words, C** does not make intermediate reduction results visible to the programmer. This design choice is explored in more detail in Section 4.2.

2.3 Tempest

Tempest is a parallel programming substrate that provides a common programming interface across a range of distributed-memory multiprocessors [38]. A key feature of Tempest is that it provides the mechanisms to implement fine-grain user-level distributed shared memory (DSM) on top of a message-passing machine, in addition to message-passing primitives. Support for shared-memory mechanisms

on distributed memory multiprocessors is becoming increasingly popular [29].

A distributed shared-memory system implements transparent access to a shared address space which is distributed among individual processors' local memories. The system identifies all accesses to remote shared data items, and includes a memory coherence protocol that obtains and caches local copies of remote data items to allow the computation to continue.

Tempest provides a DSM interface with two additions. First, sharing in Tempest is fine-grain, with sizes between 32 and 128 bytes. Fine-grain sharing reduces false sharing, and smaller block sizes improve performance for some applications (Section 5.4). Second, and perhaps more important, Tempest allows user-level control of memory coherence protocols, i.e., it allows memory coherence actions (which obtain and release copies of remote shared data) to be implemented by the user, or in our case, by the compiler. The default memory coherence protocol in Tempest, called Stache, implements sequentially-consistent transparent shared memory using a write-invalidate protocol. One of the techniques presented in this thesis (Chapter 5) exploits user-level control over coherence protocols to augment Stache to improve performance for iterative parallel applications.

2.3.1 Tempest Implementations

Tempest is designed to be a portable parallel programming substrate. Currently, two implementations of Tempest exist, both named Blizzard, one for the Thinking Machines CM-5 [45], and one for a Cluster Of Workstations (COW) connected by an off-the-shelf Myrinet network [44].

Blizzard-CM5 implements the Tempest interface on the distributed-memory CM-5, using error correcting codes in memory to implement fine-grain access control [45]. The CM-5 uses 33 MHz Sparc processors connected by a custom network, which is optimized for small messages. Satisfying a remote shared-memory access for a 32-byte cache block on the CM-5 takes approximately 200 microseconds, including protocol handler time and network round-trip time.

Blizzard-COW implements Tempest on a Cluster of Workstations (COW) connected by a Myrinet [44]. Each COW node is a dual-processor SPARCStation 20, which includes two 66 MHz ROSS HyperSPARC chips, a Myrinet interface chip, and a T0 hardware add-on board to implement fine-grain access control. Round-trip network latencies for small messages on the Myrinet are on the order of 50 microseconds, and the remote shared-memory access latency for small cache blocks (64 bytes) takes approximately 77 microseconds.

Chapter 3

Implementing Conflict-free Data Access for Parallel Functions

C** provides a number of high-level data-parallel programming language features, including a global name space and coarse-grain data-parallel functions. The interaction between these two features — unsynchronized data access in a shared address space and multiple asynchronous threads of execution — allows the possibility of data access conflicts, which are an undesirable feature of parallel program execution [1].

Data access conflicts, or data races, arise when two distinct threads of execution (or, in our case, two distinct data-parallel operations) access a single global datum, and at least one of the accesses is a write access which updates the da-

[1]This thesis does not consider synchronization races, which arise from different temporal orderings of synchronization events, since data-parallel languages do not allow explicit synchronization

tum. In a program with data access conflicts, different temporal orderings of the conflicting accesses during different runs of the program can generate different results. Data races often lead to program errors that are extremely difficult to find, primarily because the errors can only be reproduced if the same temporal access ordering is maintained across different program runs.

Unlike most other data-parallel languages that provide few mechanisms to address data access conflicts, C** defines a clear semantics for conflicting data accesses in data-parallel operations. C**'s semantics mandate that multiple parallel function invocations (Section 2.2.2) appear to execute instantaneously and simultaneously, so that global accesses cannot conflict. All invocations start from the global state in effect at the beginning of the parallel operation. When an invocation updates a global data item, the change is visible only to that invocation until all data-parallel operations complete. At that point, all changes are merged into the global state. In effect, each parallel function invocation receives its own copy of any modified global data items, and all copies are reconciled when the data-parallel operation completes. For example, in the 4-point stencil in Figure 2.2.2, each `stencil` invocation only sees *old* values of neighboring elements.

High-level parallel language features like conflict-free data access in C** are very useful, but are unlikely to be widely used unless they can be implemented efficiently. In this chapter, we explore two different ways of implementing conflict-free data access using two variants of a copy-on-write scheme with reconciliation. Section 3.2 describes how a compiler can analyze a program to identify conflicting data items and insert code to create and reconcile copies. Compiler copying

works well for programs that a compiler can analyze, but could be expensive for dynamic programs for which compiler analysis is imprecise and may lead to unnecessary copying. In that case, Section 3.3 shows how a compiler can rely on a Loosely-Coherent Memory (LCM) system to implement fine-grain copy-on-write with reconciliation. Section 3.4 compares these two approaches and their performance on four mesh relaxation codes and shows that they complement one another. Compiler copying is more efficient for programs that permit precise analysis, and LCM incurs less overhead for programs where data access analysis is imprecise. The compiler can choose either technique to implement conflict-free data access depending on the precision of its analysis, and even use both techniques in a program. Section 3.1 compares our approaches with related work, and Section 3.5 summarizes the chapter.

3.1 Related Work

Data-parallel languages handle data races in a variety of ways. Fine-grain data-parallel languages, such as C*, use a SIMD execution model to avoid read-write conflicts, and provide combining operations for write-write conflicts. Hatcher et al. [18] have looked at ways to compile fine-grain languages for MIMD machines by increasing the grain of parallelism. By contrast, C** directly allows large-grain parallelism, but specifies a clear semantics for conflicting operations. Functional data-parallel languages, such as NESL [7] do not allow side effects, and avoid the problem entirely. Whole-array operations in HPF [19] require all input operands to be read before any output is written, providing semantics similar to C** for

a restricted subset of the language, and are usually implemented with compiler-implemented copying. Finally, some coarse-grain parallel languages, such as Paralation Lisp [42] or HPF's DO INDEPENDENT loops [19] require the programmer to specify conflict-free parallel operations and leave the semantics undefined if conflicts occur.

The Myrias machine implemented a copy-and-reconcile operation similar to the one in LCM [5]. However, copying and reconciliation were implemented in hardware at the page granularity, and followed a fixed coherence policy. Munin [6], like LCM, allows the programmer or compiler to adapt coherence policies to data structures in an application. Unlike LCM, Munin only provides a fixed set of policies, each tailored for a specific pattern.

3.2 Compiler Copying

In many simple parallel applications, a compiler can precisely identify all global data accesses within a data-parallel program, and insert extra code to make copies of global values on updates if necessary. A number of compiler-copying alternatives exist depending on the precision of compiler data access analysis, and the number of distinct invocations that read or write each global data item.

3.2.1 No Conflicts

The simplest case occurs when the data-parallel function invocations are independent and each invocation updates a distinct portion of the global space. In other

```
class Grid(float) [][]) {   /* Member functions */   };

void stencil (parallel Grid A) parallel
{
    A[#0][#1] = (A[#0-1][#1] + A[#0+1][#1] +
                 A[#0][#1-1] + A[#0][#1+1]) / 4.0;
}
```

Figure 3.1: 4-point stencil

words, no invocation attempts to read values that are possibly updated by other invocations. In this case, no action from the compiler is necessary to maintain conflict-free access.

3.2.2 Single Writer, Multiple Readers

Consider the 4-point stencil in Figure 3.1 that updates each interior point in a 2-dimensional grid with the average of its four neighboring points. Each grid point is updated by only one invocation, and read by four invocations operating on neighboring grid points.

Figure 3.2 lists a simplified version of the compiler-generated SPMD code for the stencil function. To maintain conflict-free data access the compiler maintains two copies of the grid, satisfying all read accesses from the old copy and updating values in the new copy (schematically described in Figure 3.3). The reconciliation phase, which is invoked by the compiler after the parallel phase, copies values

```
void stencil_SPMD (Grid new_A, Grid A)
{
    for all points (#0, #1) assigned to me do
    {
        new_A[#0][#1] = (A[#0-1][#1] + A[#0+1][#1] +
                         A[#0][#1-1] + A[#0][#1+1]) / 4.0;
    }
    barrier();                 /* Before reconciliation phase */
    for all points (#0, #1) assigned to me do
        A[#0][#1] = new_A[#0][#1];
}
```

Figure 3.2: 4-point stencil with compiler-generated copying

from the new copy into the old copy.

If the compiler can also guarantee that *all* points in the Grid are updated, the reconciliation can be accomplished with a simple pointer swap between `A` and `new_A`.

3.2.3 Zero or One Writer

If only some fraction of all points in the grid is updated (e.g., in the threshold stencil in Figure 3.4), the compiler must keep track of the updated points for reconciliation (e.g., using a bit-vector), or conservatively propagate points that do not satisfy the threshold value to the new copy (Figure 3.5). The conservative

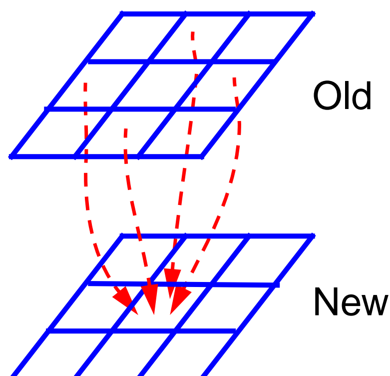

Figure 3.3: Schematic representation of compiler-copying for Stencil

copy can then be optimized using the pointer swap trick of Section 3.2.2.

3.2.4 Multiple Unknown Writers

If the compiler cannot analyze data accesses within parallel functions precisely, it must create copies for individual data items (rather than entire arrays), and include run-time checks to access the correct copy within a parallel function using a technique similar to run-time resolution [40]. Figure 3.6 shows a parallel function with unstructured accesses (i.e., through indirection arrays). In function `unstructured` (Figure 3.6), the compiler must conservatively assume that the two accesses to `A` through indirection arrays may map to the same location.

Figure 3.7 describes how a compiler can maintain per-invocation copies for a parallel function with unstructured accesses. The compiler creates and maintains a mapping of copies using support routines for allocation, lookup and unmapping. The allocation routine, `alloc_and_map`, creates a new copy of a global data item and updates it. The lookup routines, `is_mapped` and `lookup_map`, determine

```
void threshold_stencil (parallel Grid A) parallel
{
    if (diff(A[#0][#1], A[#0-1][#1], A[#0+1][#1],
             A[#0][#1-1], A[#0][#1+1]) > THRESHOLD)
    {
        A[#0][#1] = (A[#0-1][#1] + A[#0+1][#1] +
                     A[#0][#1-1] + A[#0][#1+1]) / 4.0;
    }
}
```

Figure 3.4: 4-point threshold stencil

whether a particular location is mapped and return its value respectively. The unmap routine, `unmap`, removes mappings for per-invocation copies in preparation for starting a new invocation. The reconciliation phase identifies all created copies (using function `lookup_alloced`) and merges them into the global state.

3.3 LCM

Loosely-Coherent Memory (LCM) exploits program level control of memory coherence protocols (Section 2.3) to implement fine-grain copy-on-write and reconciliation to help implement C**'s language semantics [30]. LCM relies on directives from the C** compiler to detect the need to copy shared data, and, at runtime, creates transparent per-invocation copies at the cache block granularity that share

```
void threshold_stencil_SPMD (Grid new_A, Grid A)
{
    for all points (#0, #1) assigned to me do
    {
        if (diff(A[#0][#1], A[#0-1][#1], A[#0+1][#1],
                 A[#0][#1-1], A[#0][#1+1]) > THRESHOLD)
        {
            new_A[#0][#1] = (A[#0-1][#1] + A[#0+1][#1] +
                             A[#0][#1-1] + A[#0][#1+1]) / 4.0;
        }
        else
            new_A[#0][#1] = A[#0][#1];
    }
    barrier();                  /* Before reconciliation phase */
    for all points (#0, #1) assigned to me do
        A[#0][#1] = new_A[#0][#1];

}
```

Figure 3.5: Threshold stencil

```
void unstructured (parallel Grid A, int ind[][]) parallel
{
    A[ind[#0]][ind[#1]] = f(A[#0][#1]);

    if (A[ind[#0+1]][ind[#1 -1]] > THRESHOLD)
    {
        ...;
    }
}
```

Figure 3.6: `unstructured` - Parallel function with unstructured accesses

the global address of the original data item. LCM was developed by Brad Richards and is described in detail in his thesis [39].

LCM provides the compiler with three directives. The first, `mark_modification`, marks a global data item that will potentially be modified and needs to be copied. When the directive is executed, it causes LCM to create a per-invocation copy of the global data item at the same global address. The second directive, `flush_copies` removes per-invocation copies before starting a new invocation. The final directive, `reconcile_copies`, implements a synchronized global reconciliation phase and merges all changes into the global state. Figure 3.8 shows how a compiler would use LCM directives to implement conflict-free data access for the `unstructured` parallel function described in Figure 3.6.

LCM transparently creates a per-invocation copy at the same address as the

```
void unstructured (Grid A, int ind[][])
{
    int tmp;

    for all points(#0, #1) assigned to me do {
        alloc_and_map_float(&A[ind[#0]][ind[#1]]) = f(A[#0][#1]);

        /* Personal copy or old value? */
        if is_mapped(ind[#0+1], ind[#1-1])
            tmp = lookup_map(&A[ind[#0+1]][ind[#1-1]])
        else
            tmp = A[ind[#0+1]][ind[#1 -1]];
        if (tmp > THRESHOLD)  { ...; }
        unmap(&A[ind[#0]][ind[#1]]);
    }
    barrier();     /* Before reconciliation phase */
    for all points (i, j) in A do {
        if (alloced(i, j))
            A[i][j] = lookup_alloc(&A[i][j]);
    }
}
```

Figure 3.7: Compiler-generated pseudo-code for `unstructured`

```
void unstructured (Grid A, int ind[][])
{
    for all points(#0, #1) assigned to me do
    {
        mark_modification(&A[ind[#0]][ind[#1]]);
        A[ind[#0]][ind[#1]] = f(A[#0][#1]);

        if (A[ind[#0+1]][ind[#1-1]] > THRESHOLD)
        {
            ...;
        }
        flush_copies();
    }
    reconcile_copies();
}
```

Figure 3.8: Compiler-generated pseudo-code for `unstructured` using LCM

original data item. As a result, the compiler can refer to global variables directly, and need not insert run-time checks to differentiate per-invocation copies from the global state. It offers greater benefits for programs with dynamic behavior by removing potentially expensive run-time checks [43].

3.4 Performance Comparison

We compared the performance of compiler copying and LCM with three mesh relaxation programs written in C** with varying degrees of dynamic behavior. *Stencil* performs an iterative, regular 4-point stencil for 50 iterations on a 1024x1024 mesh (Figure 3.1). *Threshold* performs a similar 4-point stencil computation for 50 iterations over a 512x512 mesh, but does not modify all mesh elements in each item (Figure 3.4). It modifies only points whose values has changed by more than a threshold. *Adaptive* is also a stencil computation over a structured mesh, but the mesh evolves over time to capture finer detail at points in the mesh where the gradient is steep. Adaptive uses dynamically allocated quad trees at mesh points to simulate subdivision of space for greater accuracy. In Adaptive, as in Stencil, all interior points in the mesh are updated in each iteration.

Figure 3.9 displays and compares the relative execution speed of compiler-copying and LCM versions of Stencil, Threshold and Adaptive. The copying versions use two copies of the entire data set, and use a pointer swap to switch between new and old versions (Section 3.2.3). Copying versions were generated automatically by the compiler for Stencil and Threshold, and by hand for Adaptive. All programs were run on a 32-processor CM-5, using the Blizzard-E version of

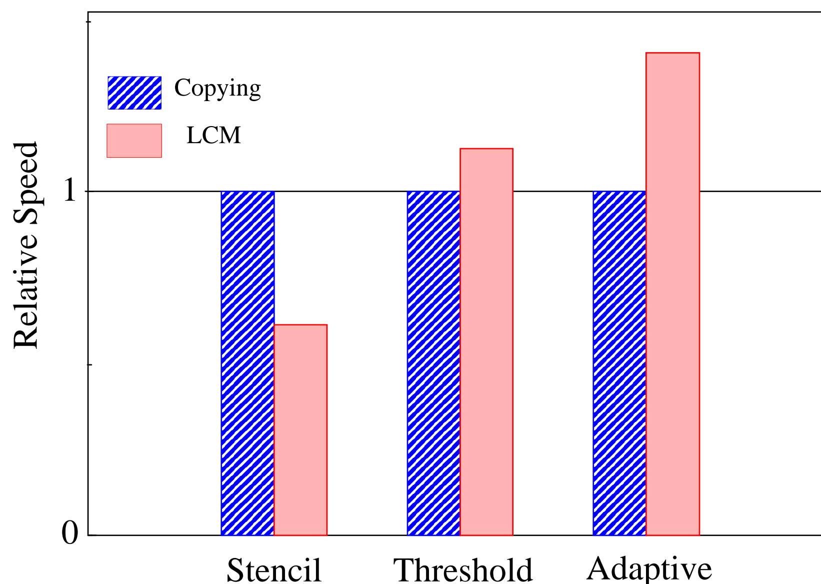

Figure 3.9: Relative execution speed for compiler-copying and LCM versions of 3 benchmarks — Stencil, Threshold and Adaptive

Blizzard-CM5 [45]. The compiler-copying versions used Blizzard's default Stache memory coherence protocol [38].

Stencil is a good example of a regular program on which precise compiler data access analysis is possible. In this case, LCM provides no benefit; the copying version is 1.5x faster than the LCM version. Compiler-copying is more conservative for Threshold than for Stencil and copies some data items unnecessarily (Section 3.2.3). LCM only creates copies of data items that are actually modified and improves the execution speed by a factor of 1.18x. LCM improves performance significantly over copying for Adaptive, which includes data accesses through pointers and dynamically allocated data for which compiler analysis is imprecise. Adaptive-LCM is faster than Adaptive-copying by a factor of 1.36x.

3.5 Summary

This chapter presented two different techniques, compiler-implemented copying and LCM, that implement C**'s semantics of conflict-free data access, with the aim of avoiding data races which lead to hard-to-find errors in parallel programs. Both techniques implement C**'s semantics using variants of copy-on-write schemes with reconciliation.

Compiler-implemented copying inserts code in the program to create and maintain explicit copies of global data items. For regular programs with data access patterns that a compiler can analyze, compiler copying provides an efficient alternative to implement conflict-free data access. LCM, on the other hand, relies on compiler directives to detect the need to copy shared data, and creates copies that share the same global address as the original item. LCM implements conflict-free access more efficiently for programs with dynamic data access behavior by removing the necessity for run-time checks to distinguish between copies.

LCM and compiler-copying complement one another, thus enabling the C** compiler to use the best of both worlds. The compiler can generate copying code in parts of the program where its data access analysis is precise, and rely on LCM to maintain C**'s semantics efficiently in other parts of the program that include dynamic behavior.

Chapter 4

User-defined Reductions for Efficient Communication

Communication in imperative data-parallel languages occurs through read or write operations in the shared address space. Although convenient and portable, communication in a shared address space only suffices for one-to-one or one-to-many communication. Many-to-one communication causes write-write conflicts or collisions, when multiple values are stored in a location. Data-parallel languages typically use binary reduction operators to combine colliding values into a single value that can be stored in a location. Reductions, which specify both communication and combining, are extremely common in parallel applications, even those written in languages that do not provide first-class support for these operations. Unfortunately, most data-parallel languages limit reductions to a small set of predefined reduction functions — typically, the associative arithmetic and logical operations.

The topic of this paper is user-defined reductions, which are a natural generalization of reductions in data-parallel languages. User-defined reductions extend reductions in two important ways. First, they allow new ways to combine values that may not have been anticipated by a language designer. For example, user-defined reductions can implement the tournament or location reduction [17, 25], or build a list from colliding values (e.g., `sendToQueue` [23]) without additional language support. Second, user-defined reductions allow a programmer to specify reductions on user-defined data types.

This paper demonstrates the advantages of user-defined reductions in a data-parallel language, focusing on the benefits, both to the programmer and to the compiler, of extending reductions to user-defined data types. The programmer benefits from being able to specify combining operations directly on structured data types. By contrast, predefined reductions require the programmer to specify combining in three steps — map structured data types to primitive types, combine primitive types, map results back to structured data types — (e.g., see Section 4.3.3). We compare three ways in which data-parallel programmers typically specify particle movement in a particle-in-cell code, including parallel prefix operations, predefined list-building reductions and user-defined reductions. In comparison to the other two methods, user-defined reductions offer the benefits of a simple, intuitive specification of direct producer-consumer particle movement. Moreover, with simple support from the run-time system, a compiler can translate reduction operations into direct producer-consumer data transfers on structured data types that can be implemented efficiently using bulk messages on a message-

passing machine.

This paper also presents a simple implementation of user-defined reductions in the coarse-grain data-parallel language C**, although the results should directly apply to other languages such as HPF [25] or pC++ [32]. Our reductions implementation relies on message-passing support, and applies traditional message-passing optimizations to reduce overhead. We analyze the performance of user-defined reductions by comparing the execution time of two versions each of four benchmark applications (Table 4.2), one in a data-parallel language with user-defined reductions and the other SPMD code optimized with application-specific communication for the reduction pattern on two hardware platforms, a 32-processor CM-5 and a 16-node Cluster of Workstations (COW) connected by an off-the-shelf network. For applications with dynamic communication patterns, the C** versions were faster by up to 25%. For applications with mostly static communication patterns, the C** version was up to 2.6x slower than the SPMD version.

User-defined reductions also allow the programmer to specify new reduction operations. The obvious benefit of this extension, which we do not explore in this chapter, lies in allowing powerful combining operations, such as the location reduction [25], or to build a list or tree with multiple values. A drawback of user-defined reductions is that they allow arbitrary user-defined code to execute in a reduction operator, which allows the possibility of non-deterministic results either due to data races, or due to reordered execution of non-commutative operations. Section 4.4 explores these language design problems in more detail, and proposes

solutions to some of them.

This chapter is organized as follows. Section 4.2 briefly describes communication and reductions in C**, and shows how they can be extended to user-defined reductions. Section 4.3 uses the example of a particle-in-cell code to contrast three different specifications of particle movement in a data-parallel language, including parallel prefix operations, predefined list-building reductions and user-defined reductions. Section 4.4 explores the consequences of extending reductions with user-defined operators in a data-parallel language. Section 4.5 describes C**'s reduction implementation. Section 4.6 describes our benchmarks and presents detailed performance results. Section 4.1 covers related work and Section 4.7 summarizes the chapter.

4.1 Related work

Many previous papers have recognized the need for powerful reduction operators. For example, Dataparallel C adds a tournament operator [17] to locate the position of the maximum value in a list of elements. In comparing the message-passing and data-parallel paradigms, Klaiber et al. [23] proposed the `sendToQueue` operator to remedy the inefficiency of expressing a list-building reduction in C*. Sharma et al. [46] proposed a similar `APPEND` operator for DSMC, a particle-in-cell application. User-defined reductions subsume these specialized reduction operations under a common framework, and do not require compiler or run-time system changes to implement these operators.

Mukherjee et al. [35] noted that reductions form the dominant communica-

tion pattern in many irregular applications. However, they used a variety of application-specific protocols and implemented custom protocols for each application to improve communication performance. The C** system uniformly handles reduction patterns by implementing them using vectorized messages and is able to run several of their applications as efficiently as the hand-tuned code.

Several applications in the HPF-2 motivating applications suite [14] note their requirement for user-defined reductions. In addition, some languages allow user-defined functions for reduction operations (e.g., Connection Machine Lisp [49], Paralation Lisp [42], and Fortran D [15]). However, we are unaware of papers describing implementations of user-defined reductions on parallel machines.

4.2 Reductions in Data-Parallel Languages

Recall from Section 2.1.2 that reductions augment assignment statements in a parallel operation with a combining or reduction operator. When the reduction assignment executes, the statement's combining operator combines colliding values and updates the left-hand-side with the result. For example, Figure 4.1 uses a reduction assignment to sum the values of a `Grid` object.

Reduction Result Availability

In coarse-grain data-parallel languages, a task may continue after executing a reduction assignment. The language must specify the value of a reduction target (e.g., `g` in parallel function `sum` in Figure 4.1) between the reduction assignment and the end of the parallel function. Three approaches are possible:

```
float g;

void sum(parallel Grid &A) parallel
{
    g =%+ A[#0][#1];
    ...;
}
```

Figure 4.1: Sum reduction assignment

1. The language may prohibit accesses to g, except as a reduction target, as does Fortran D [15]. Erroneous accesses can be identified syntactically. This approach allows the runtime system to defer updating the target. However, syntactic analysis may not identify all erroneous accesses, particularly those involving arrays or pointers.

2. The language may retain the old value of g after a reduction. When the data-parallel operation completes, the colliding values can be combined and used to update reduction targets. We call this approach **deferred reductions**.

3. The language may defer combining, but update the local copy of g by merging contributions from the local task. This approach is suitable for a language like C**, which mandates local copies to enforce independence. However, other data-parallel languages do not make such a clear distinction between local and global values and are better off using deferred reductions.

4.2.1 User-defined Reductions

In C**, *user-defined reduction assignments* required a minor syntax extension to allow function names as reduction operations. For example, Figure 4.2 shows a location reduction. Given an array of numbers, the location reduction identifies the minimum (or maximum) value in the collection, along with its location in the array. The location reduction is used, for example, in a parallel implementation of Dijkstra's algorithm, to find the next node with the shortest path. The `find_min` parallel function pairs the node's distance and its position and applies the user-defined `min_pos` combining function to compute the minimum value and its position.

Note that the reduction function `min_pos` in C** is not a symmetric binary operator with type `PosVal` × `PosVal` → `PosVal`, but `(PosVal *)` × `PosVal` → `void`, and uses the first parameter for both input and output.

Combining and Update in Reductions

A reduction assignment comprises two actions, combining and update (Figure 4.3). In many cases, both actions are identical and can be specified with a single function (e.g., in the sum reduction in Figure 4.1, the colliding values are added together and the resulting sum is added to `g`). It is sometimes useful to separate the two operations, especially for user-defined reductions, to allow different types for the target and colliding values, or when the combining operation itself is expensive.

One such example involves building an array from colliding values. Each value carries a position field and a data field, and values that map to the same array

```
struct PosVal {          /* Data type for position reductions */
   int value, position;
};

void min_pos(PosVal *lhs, PosVal rhs) /* Location reduction function */
{
    if (lhs->value > rhs.value)
        (*lhs) = rhs;       /* RHS is the winner */
}


void find_min(parallel Vector &V) parallel
{
    PosVal result = { V[#0].value, #0};
    min =%min_pos  result;
}
```

Figure 4.2: Minimum location user-defined reduction

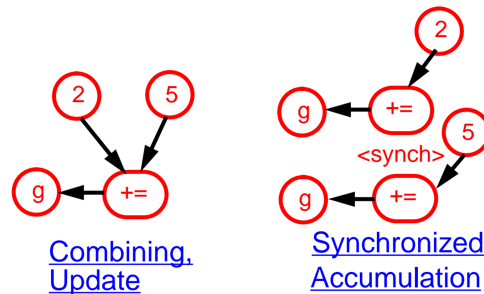

Figure 4.3: Two ways of visualizing a reduction of two values to a target: Combining and Update, or Synchronized Accumulation

position must be added in the result [51]. One way to specify this pattern uses an array addition reduction. Input values are entered into distinct zero-filled arrays, which are combined using array addition. Clearly, this approach is expensive. An alternative approach starts with a zero array, and specifies a *reduction update* function, which merges values one-by-one into the array. This pattern, which we call *synchronized accumulation* (Figure 4.3), is also applicable in cases when the combining operator just collects colliding values (e.g., particle movement in Section 4.3). C** allows the programmer to specify both combining and update functions as part of a reduction.

4.3 A Motivating Example

An important aspect of user-defined reductions is that they extend reductions to apply to user-defined data types. This section explores the benefit of this extension from the programmer's point of view, i.e., simple and direct specification of communication and combining. In this section, we demonstrate this benefit

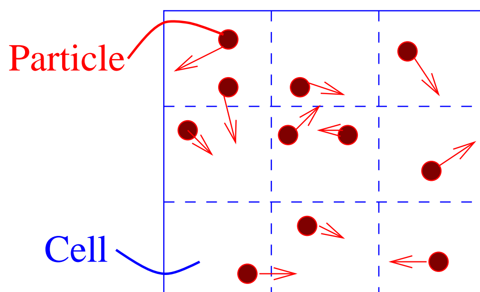

Figure 4.4: Schematic of the DSMC application

by comparing three real-life data-parallel specifications of particle movement in a particle-in-cell code called Discrete Simulation Monte Carlo (DSMC). First, we briefly describe DSMC, its implementation in a data-parallel language, and the problem of particle movement in DSMC. We describe and contrast three approaches to specifying particle movement in a data-parallel language — using parallel prefix operations, with a predefined `append` operation and with user-defined reductions.

4.3.1 DSMC

DSMC simulates particle movement and collision in a three dimensional domain using a Discrete Simulation Monte Carlo method [46]. DSMC divides the domain into cells in a static Cartesian grid and distributes molecules among cells (Figure 4.4). Each time step of the algorithm consists of three phases for each cell, the move phase, which moves molecules according to their velocities, the addition phase, which adds new molecules, and the collision phase, which simulates molecule collisions.

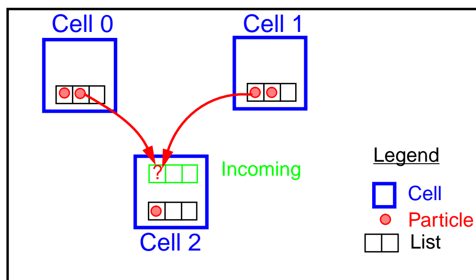

Figure 4.5: Schematic of DSMC data structures and the problem of synchronized addition

The move phase updates each molecule's position using its current velocity. In the process, a molecule may move from one cell to another, which may require communication if the cells are mapped to distinct processors. Recall that all communication in an imperative data-parallel language occurs through global reads and writes (Section 4.2). To transfer a molecule between two cells, the source cell must write the molecule to a global location (typically, an "incoming" area for the destination molecule), which is then read by the destination molecule. Furthermore, multiple molecules entering a cell (e.g., from different neighbors) must be synchronized (Figure 4.5).

Since data-parallel languages do not provide low-level synchronization primitives such as locks or mutual exclusion primitives, we look at other methods of specifying particle movement and synchronization.

4.3.2 Particle Movement using Parallel Prefix

Parallel prefix operations provide one solution to synchronizing multiple additions to a cell (Figure 4.6). For each destination cell, all source cells enter a negotiation

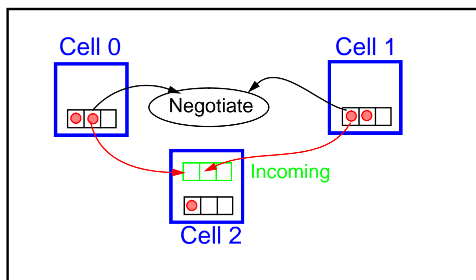

Figure 4.6: Schematic of DSMC synchronization with parallel prefix

phase to partition the `Incoming` list to prevent write-write conflicts. The negotiation phase is itself implemented as a parallel prefix (or scan) operation that adds the number of molecules from each source. In essence, this method uses a lower-level reduction operation (the parallel prefix) to implement a higher-level reduction pattern.

The negotiation phase is carried out for each destination cell, making this approach cumbersome to specify. The implementation is also likely to be inefficient on MIMD machines because it involves a negotiation phase (which is usually not supported on MIMD hardware), followed by a transfer phase.

The negotiation phase must be specified for each destination cell, making this approach cumbersome to specify. The implementation is also likely to be inefficient on MIMD machines because it involves a negotiation phase (which is usually not supported on MIMD hardware) followed by the source-destination transfer.

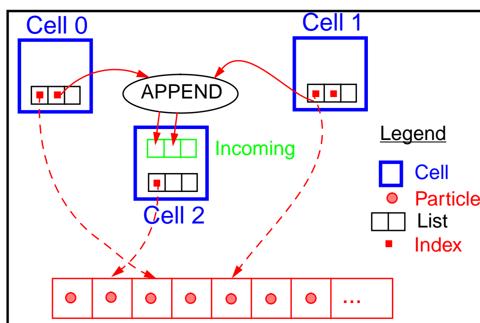

Figure 4.7: Schematic of DSMC synchronization with `APPEND`

4.3.3 Particle Movement using a Predefined Reduction

The second approach uses a predefined `APPEND` reduction operator that builds a list from colliding integer values [46] (Figure 4.7). In this approach, each cell contains a list of indices into the global particle array where particles are stored (squares in Figure 4.7).

The predefined `APPEND` operator combines negotiation and communication. However, it separates particles from cells, and specifies communication indirectly — the moving particle is written to the global array by a source cell and read in a subsequent iteration by a destination cell.

4.3.4 Particle Movement with User-Defined Reductions

The final approach utilizes a user-defined reduction operator (`add_particle`), which implements synchronized accumulation (Section 4.2.1) using knowledge of particle and cell data types (Figure 4.8). This approach does not need an `Incoming` list because the execution of reductions is deferred (Section 4.2).

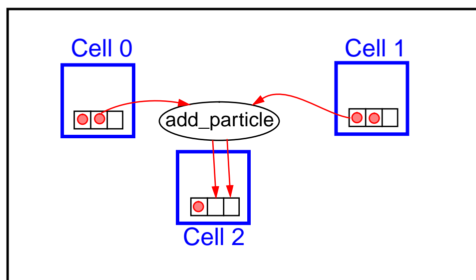

Figure 4.8: Schematic of particle movement with user-defined reduction `add_particle`

Approach	**Specification**	**Flexible?**	**Direct transfer?**
Parallel Prefix	Cumbersome	No	Yes
`APPEND`	Simple	No	No
User-defined Reduction	Simple	Yes	Yes

Table 4.1: Comparing three approaches to particle movement

User-defined reductions are more flexible than the other two approaches, and specify direct producer-consumer data transfer. Table 4.1 summarizes the three approaches.

4.4 Semantics of User-Defined Reductions

User-defined reductions add expressive power to a data-parallel language, but the combination of parallel execution and arbitrary user-defined operators gives rise to the possibility of non-deterministic execution in two ways: data access conflicts and non-associativity.

4.4.1 Data races

In most data parallel languages, the built-in, primitive reduction functions are side-effect free and cause no data access conflicts. However, user-defined reduction functions need not share this property. User-defined reductions consist of arbitrary user-defined code, which may execute in parallel with access to data in a global address space. Chapter 3 shows how this combination allows data access conflicts, which can undermine a data-parallel language's semantic guarantees.

We see three approaches to avoiding data access conflicts with user-defined reductions. First, user-defined reductions can be restricted to use only well-behaved functions that a compiler ensures are side-effect free. Although plausible, this rule is too restrictive because of limited compiler analyses in languages that support pointers and aliasing. Second, a language may permit compiler directives, such as HPF's INTENT directive, that assert properties of user-defined reductions that a compiler is unable to prove. This approach opens the door to difficult-to-find errors if a directive is incorrect. Third, a language may allow general functions, but require a run-time system to identify data access conflicts, as in Steele's Parallel Scheme [48]. Run-time conflict identification can be expensive and complex. The C** compiler relies on programmer guarantees that user-defined functions are safe.

4.4.2 Reordering Combining Operations

Another issue is that user-defined reductions may not be commutative or associative, so that different combining orders lead to non-deterministic results. This is

not a problem for two reasons. First, user-defined reductions are typically effectively associative [42] functions, for which the absence of associativity does not affect a program's result. For example, for the combining function `append`, the result list usually represents a set so that the order of elements is unimportant. Second, a programmer can collect all values into a list, sort them, and then combine to force a specific combining order. Mandating a combining order for all reductions unnecessarily restricts language implementors and imposes overhead on applications that do not require ordering.

4.5 Implementing Reductions

The C** compiler implements user-defined reductions with a small amount of runtime support. This section describes how the compiler and runtime system implement basic and update reductions (Section 4.5.1), exploit the deferred reduction model to vectorize messages (Section 4.5.2), and combine values locally to reduce message traffic (Section 4.5.3).

4.5.1 Basic Reductions

A reduction assignment updates its target with the result of combining right-hand-side values. The C** implementation involves two processors: the processor that executes the reduction assignment (processor A) and the processor that owns the target location (processor B). Processor A, which executes the reduction, sends processor B a message containing three items: the right-hand-side value,

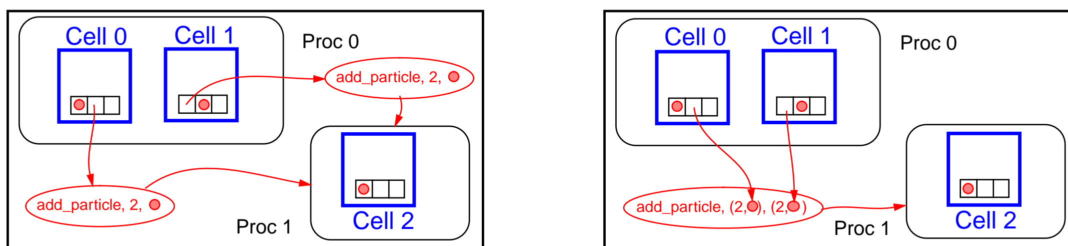

Figure 4.9: Schematic representation of basic reductions (left) and bulk communication (right) for DSMC

the combining function descriptor, and the target location pointer. At the end of a parallel phase, Processor B collects incoming reduction messages, combines colliding values and updates target locations. For example, Figure 4.9 shows the basic implementation of communication for DSMC (Section 4.3).

The "owner-updates" model is simple, and depends on the runtime system to identify the home location of a target data item, which is usually available in implementations that provide a global name space.

4.5.2 Bulk reductions

During a data-parallel operation, a processor may execute multiple reductions for two reasons. First, the number of data-parallel tasks is usually much larger than the number of processors, so each processor runs multiple tasks. Second, each coarse-grain data-parallel task may execute multiple reduction assignments. The deferred reduction model (Section 4.2) allows the compiler to defer sending reduction messages until the end of the parallel phase. This permits several messages to the same destination processor to be bundled into a single message (Figure 4.9),

which is typically far more efficient to send and receive. Message vectorization is essential when the application program uses reductions to communicate large amounts of data. As the graphs in Section 4.6 show, this optimization improved program performance 6.76x for EM3D on the CM-5.

4.5.3 Local Combining

If a processor executes multiple reductions to the same target, the values can be combined locally before being sent for global combining. Local combining requires the runtime system to identify common targets locally, for which the C** system uses a hash table of target addresses. Probing this table increases the overhead of the reduction, but allows for a decrease in communication costs. This is a good example of an optimization that trades off worse sequential performance for better communication (and therefore parallel) performance.

On Moldyn, an application that benefited from local combining, (Section 4.6), this optimization improved performance by 2.32x.

4.6 Application Comparisons

This section compares four parallel applications (DSMC, Barnes, Moldyn, EM3D) written in C** with user-defined reductions against hand-optimized alternatives written in an SPMD style. Table 4.2 describes the benchmarks, and their the input data sets. The SPMD codes were previously written (by others) using a hybrid of shared-memory and message-passing techniques, and used as the best

Program	Scientific Domain	Application of reductions	Data sets
DSMC	Particle-in-cell	Moving molecules between cells	9,720 cells, 20 iterations initially 48,600 particles finally 72,500 particles
EM3D	Electro-magnetics	Accumulating edge interactions	32,000/320,000 nodes, 20% remote edges degree 5, 100 iterations
Moldyn	Molecular dynamics	Accumulating interaction forces	16,384/55,296 mols, 30 steps
Barnes	Hierarchical N-body	Inserting bodies into a quad-tree	16,384 bodies, 4 iterations

Table 4.2: High-level application description and data sets. The larger data sets are for COW runs.

examples of hybrid coherence protocols in several published papers [13, 35]. These programs use transparent shared memory as a basis, but communicate crucial data structures through application-specific shared-memory or message-passing protocols. User-defined reductions optimize the same communication patterns as the application-specific protocols, but provide a simpler semantic model and use a simple, unified implementation using messages. Mukherjee et al. demonstrated that custom protocols compare favorably with the CHAOS library [12] for some irregular applications, including DSMC and Moldyn [35]. We compared these programs on both the CM-5 and the COW using Blizzard.

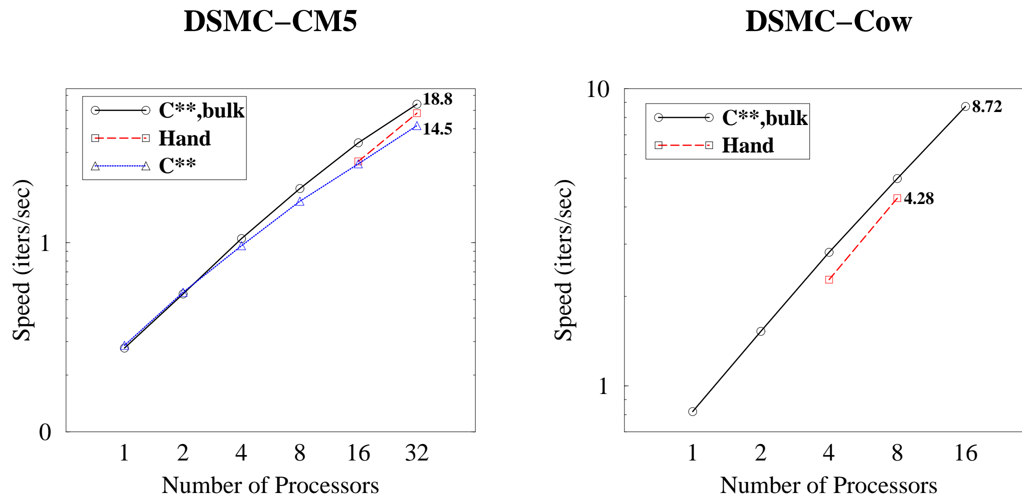

Figure 4.10: Log-log scale graphs showing execution speeds of 2 or more versions of DSMC on the CM-5 and COW. Numbers adjoining the curves are speedups relative to a sequential version.

4.6.1 DSMC

DSMC, and its implementation in C** with user-defined reductions, are described in detail in Section 4.3.

The hand-optimized implementation also uses bulk messages to move molecules to neighboring processors, but uses single messages for molecules moving to distant processors. Mukherjee et al. showed that the hand-optimized implementation of DSMC ran slightly faster than a DSMC version implemented with the well-known communication library CHAOS [12] on the CM-5 [35] .

Figure 4.6.1 plots the speed of the hand-optimized version (DSMC-Hand) and C**'s unoptimized and bulk reduction versions. On both platforms, DSMC-C** is faster than DSMC-Hand (1.11x on CM-5 and 1.16x on COW), primarily because

DSMC-Hand vectorizes messages only to neighboring processors (the common case), since vectorization is tedious to apply by hand in all cases.

4.6.2 Barnes

Barnes [47] simulates the movement of bodies in a gravitational system over time. The bodies are modeled as point masses that exert gravitational forces on other bodies. The algorithm computes forces between bodies which are used to update body positions in each time step. Rather than computing all N^2 forces, Barnes approximates the force exerted by a distant collection of bodies by that of a point mass at the center of mass of the collection.

Barnes uses an oct-tree to represent bodies in 3-dimensional space. Each node in the tree represents a region in space, with a child representing one octant of its parent's space. The tree is unbalanced and deeper in regions of high body density. To calculate the force on a body, the algorithm performs a depth-first traversal of the tree. If an interior node is sufficiently far away from the body, the bodies in that region are approximated by a point mass. Otherwise, the algorithm continues to traverse the subtrees.

User-defined reductions target the tree-rebuild phase, which occurs in every time step. In the SPMD implementation, processors insert bodies into the tree in parallel, using locks to synchronize accesses to tree nodes. Data-parallel languages lack explicit synchronization, which disallows a similar tree build method. The C** data-parallel implementation of Barnes uses multiple pipelined parallel phases to build the tree. To start with, bodies are inserted at the root of the (empty)

tree. In each subsequent pipeline stage, bodies move one level down the tree (extending the tree as necessary). The build phase ends when all bodies have settled at the leaves of the tree. The C** implementation utilizes user-defined reductions to move bodies between tree nodes, just as the DSMC implementation moves particles between cells (Section 4.3). Subsequent iterations of the tree-build algorithm are further optimized by starting from the tree structure generated in the previous iteration with the bodies removed.

For the parameters described in table 4.2, the SPMD tree building phase takes approximately 15% of the total execution time of the entire application (12.6 seconds out of 73.7). The data-parallel C** implementation takes 9.5 seconds for all tree builds, not counting 9.5 seconds for the first tree build, which starts from an empty tree.

The object of this section is to show that user-defined reductions help efficiently implement an application that is typically thought to be "asynchronous". The data-parallel tree-build phase is competitive with the SPMD implementation and a first step towards an efficient implementation of Barnes-Hut in a data-parallel language.

4.6.3 EM3D

EM3D is an unstructured graph application that models the propagation of electromagnetic waves through objects in three dimensions [11]. The problem is formulated as a bipartite graph of H nodes representing magnetic fields and E nodes representing electric fields, with directed edges between H nodes and E nodes.

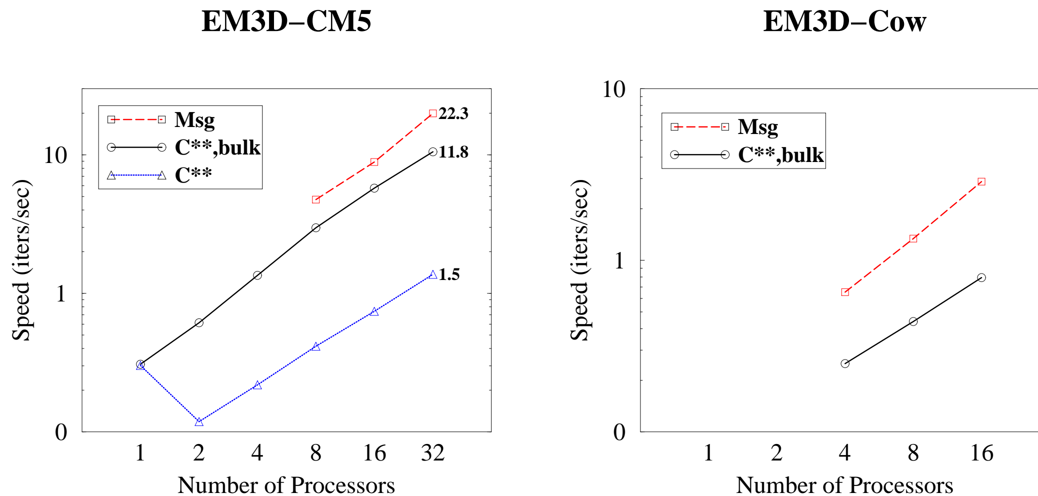

Figure 4.11: Log-log scale graphs showing execution speeds of 2 or more versions of EM3D on the CM-5 and COW. Numbers adjoining the curves are speedups relative to a sequential version.

Each time step consists of two parts: first, each H node accumulates the effects of neighboring E nodes, and then each E node accumulates the effects of neighboring H nodes.

The message-passing version of EM3D efficiently implements the producer-consumer sharing pattern in EM3D.

In C**, values are accumulated with primitive sum reductions. In the first part of a time step, E nodes send their values (using reductions) to H nodes, where they are collected and combined. The reduction implementation mimics the producer-consumer data movement pattern of the program, which is essential to good performance [13].

Figures 4.11 plots the speed of the message-passing version (EM3D-MP) and

the simple and bulk-reduction C** versions. On both platforms, EM3D-MP is significantly faster than EM3D-C** (1.9x on CM-5 and 2.6x on COW). There are two reasons for this slowdown, both related to the fact that EM3D has a static communication pattern. First, EM3D-C** transfers 1.5x more data than EM3D-MP. EM3D-MP exploits the static communication pattern to only transfer node values. C**'s reduction implementation is not optimized for static patterns, and transfers a target address (4 bytes) along with each node value (8 bytes). Second, EM3D-C** incurs the overhead of collecting reductions dynamically into bulk messages, and testing for buffer overflow.

4.6.4 Moldyn

Moldyn is a well-known molecular dynamics code used to model macromolecular systems [35]. Molecules are initially distributed uniformly in a cuboidal region with a Maxwellian distribution of initial velocities, and exert forces on other molecules within a cut-off radius. In Moldyn, interacting molecule pairs are maintained on an interaction list which is updated infrequently. Evaluating an interaction involves reading the positions of two molecules, computing the resulting force, and updating each molecule with the resultant force. The force on a molecule is the sum of force increments from a number of interacting molecules, and forms the reduction pattern in the application.

The hand-optimized implementation [35] uses a reduction implementation that minimizes time at the expense of space. It stores local copies of force increments for *all* molecules on *each processor*. Interactions modify the local copy, and the

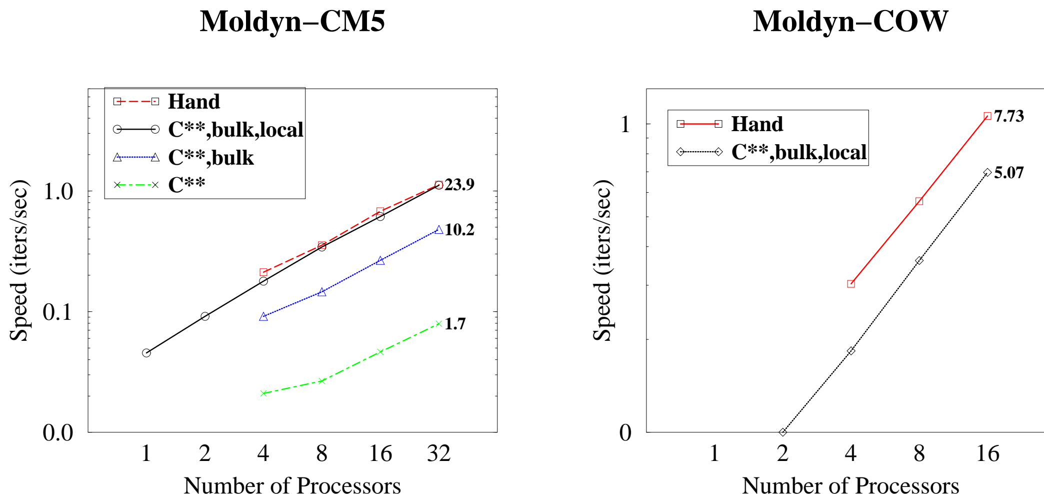

Figure 4.12: Log-log scale graphs showing execution speeds of 2 or more versions of Moldyn's force computation phase on the CM-5 and COW. Numbers adjoining the curves are speedups relative to a sequential version.

reduction phase combines local copies in an efficient ring reduction using messages.

C** uses the bulk reduction implementation with local combining to implement Moldyn's reduction pattern. C** only allocates space for reductions actually executed on a processor, but incurs the overhead of building and maintaining a hash table to identify candidates for local combining.

Figure 4.12 compares the speed of Moldyn's force computation phase for the hand-optimized version (Moldyn-Hand) and C** versions at three levels of optimization (simple reductions, bulk reductions and bulk reductions with local combining). We compare only the force computation phase in Moldyn; the C** version used a faster algorithm for building the interaction list and is faster overall.

On the CM-5, Moldyn-C** is only 1% slower than Moldyn-Hand. Bulk re-

duction provides a 6.04x improvement in execution time and local combining improves further by 2.32x. On COW, Moldyn-Hand is 1.5x faster than the best C** version, because Moldyn-C** incurs significant overheads for creating and mapping local copies. On the CM-5, the small 32K cache on each processor node penalizes per-processor local copies of the entire force array, making both versions comparable.

4.6.5 Discussion

The C** compiler provides a general reduction implementation for data-parallel programs. As such, it works well for programs with dynamic communication patterns (e.g., DSMC, Barnes), but incurs unnecessary data transfer and processing overheads for programs with static communication patterns (e.g., Moldyn, EM3D). In the case of EM3D, these overheads dominate execution, resulting in significant slowdowns over a message-passing version. For static or mostly-static communication patterns, the Inspector-Executor compiling paradigm [24], which we have not implemented, should be successful in reducing reduction overheads. The inspector phase identifies the reduction pattern in one iteration, and preallocates buffer space and initializes a communication schedule for subsequent executor phases. We have not implemented this optimization.

4.7 Summary

Data-parallel languages mitigate the difficulty of parallel programming by providing high-level abstractions for concurrency, synchronization and communication. In data-parallel languages, reductions are the primary feature for expressing many-to-one communication patterns by combining multiple colliding values using a binary reduction operator. Unfortunately, many data-parallel languages limit reductions to a pre-defined set of operations. This chapter demonstrates that user-defined reductions are a useful addition to a data-parallel language.

User-defined reductions are useful because they generalize reductions in two dimensions. First, they support powerful combining operations (e.g., location reductions and list building) in a familiar framework. Second, they generalize reductions to user-defined data types, with significant benefits, as we show in this paper. The programmer benefits from being able to apply combining operations directly to structured data types. The compiler can translate reductions into direct producer-consumer data transfers that can be implemented efficiently with messages. This paper also considers some drawbacks of this extension, such as possible data access conflicts, and proposes some solutions.

This chapter describes the design and implementation of user-defined reductions in the coarse-grain data-parallel language C**.It also describes a simple, yet general, reduction implementation that uses messages to communicate reduction data. Two simple and well-known optimizations — message vectorization and local combining — significantly improve the execution speed of applications using reductions. We present performance results comparing C** versions of 4 applica-

tions using reductions with equivalent, hand-optimized SPMD programs running on a 32-node CM-5 and a 16-node COW. On 2 applications with dynamic communication patterns, the C** versions were between and 15% and 25% faster. On 2 applications with mostly-static communication pattern, the C** versions were up to 2.5x slower. The C** implementation is amenable to Inspector-Executor style optimization, which we have not implemented. Given the complexity and effort in tuning the SPMD codes, the C** programs are far more attractive.

Chapter 5

Compiler-Directed Shared-Memory Communication for Iterative Applications

Many scientific applications are iterative with each iteration simulating the evolution of a physical system along one dimension of the problem domain (typically time). Each iteration of a problem is usually divided into multiple phases of parallel execution separated by synchronization. Communication within a parallel phase may include both structured communication (e.g., nearest-neighbor communication in Jacobi iteration) and unstructured communication (e.g., using indirection arrays or pointer dereferences). Many of these applications have communication patterns that show little or no change **between** iterations. As a result, even for irregular programs, for which static analysis is imprecise, a run-

time system can detect communication patterns in one iteration and use them to predict communication in the subsequent iterations.

There are many examples of programs with repeated patterns of communication. In static mesh calculations, nearest-neighbor communication is repeated in each iteration. In some irregular problems, such as molecular dynamics codes [46], communication changes infrequently, perhaps once every 20-30 iterations. In adaptive problems, communication changes frequently, but incremental changes between iterations are small. For example, structured adaptive meshes gradually add mesh nodes for greater accuracy in each iteration [27], and gravitational N-body problems represent bodies in a quad-tree, which undergoes small structural changes between iterations.

This chapter shows that a compiler for a data-parallel language can cooperate with a predictive cache-coherence protocol in a DSM system to implement shared-memory communication efficiently for applications with dynamic, but repetitive communication patterns. The compiler uses static analysis to place protocol directives at points in a program at which potentially repetitive communication takes place. A two-part predictive protocol in the runtime system extends the standard memory coherence protocol. The first part of the protocol identifies communication patterns at runtime and builds a communication schedule. The second part uses a schedule to pre-send data that anticipates data requests in subsequent iterations. As a result, the protocol can reduce the number of remote data requests, total remote memory access latency, and program execution time. The predictive protocol optimizes communication for repetitive producer-consumer or migratory

patterns; it does not target other sharing patterns (e.g., reductions, for which high-level language support is available in data-parallel languages).

This chapter describes this combination of two techniques — a predictive cache coherence protocol, and simple compiler analysis — for optimizing shared-memory communication. The predictive protocol relies on and exploits customizable cache-coherence protocols in a cache-coherent DSM system to build dynamic incremental communication schedules — new requests not satisfied by the pre-send phase are added to the schedule for subsequent iterations. This approach has the advantage that it can be applied to adaptive applications with repetitive dynamic communication patterns that a compiler cannot analyze.

The second technique, simple compiler analysis, automatically applies a predictive protocol for applications with repetitive producer-consumer sharing patterns for which a sequentially-consistent memory coherence protocol would incur large overheads [9]. By contrast, compilers targeting message-passing machines must identify and fully analyze run-time communication patterns in applications. Our simple analysis only identifies program points at which potentially repetitive communication takes place, but need not identify the patterns themselves.

We analyze the performance of cooperative communication optimization by comparing the performance of optimized and non-optimized C** versions of three applications on a 32-processor CM-5. They include Adaptive, an adaptive structured mesh relaxation, Barnes, a gravitational N-body code and Water, a molecular dynamics code. In all cases, the predictive protocol reduced total remote access latency. In two cases, the optimized version is faster than the best non-optimized

version (1.5x for Adaptive and 1.07x for Water). For Barnes, which shows excellent spatial locality, the optimized and non-optimized versions are comparable.

This chapter proceeds as follows. Section 5.1 compares the approaches in this chapter to previous work. Section 5.2 describes C**'s predictive protocol for communication optimization, building on an outline of Blizzard's default Stache coherence protocol. Section 5.3 describes compiler analysis for C** programs to identify repetitive communication patterns and place runtime system directives. Section 5.4 shows how these optimizations can be applied to improve the performance of three different applications. Section 5.5 concludes the chapter.

5.1 Related Work

Related work for repetitive communication support falls into four broad categories: libraries, compilers, memory coherence protocols, and hardware.

A number of run-time libraries provide communication support for specific classes of applications, and require explicit programmer actions to structure the application using abstractions provided by the library. LPARX [26] and its adaptive-mesh extension [27] provide a software infrastructure to support structured static and adaptive mesh methods on message-passing machines. By contrast, our approach implements automatic communication optimization for programs written in a data-parallel language running on customizable cache-coherent DSMs (which also run on message-passing machines).

The most closely related work is the compiler-based Inspector-Executor approach that targets irregular communication patterns using the CHAOS [12] com-

munication library. For each parallel loop that specifies irregular communication (e.g., using indirection arrays), the compiler generates an inspector and an executor. The inspector identifies non-local accesses at runtime and builds a communication schedule, which the executor uses to transfer data before executing the loop. A number of optimizations attempt to reduce the cost of the inspector phase, which is typically expensive, and must be executed whenever the indirection array changes. Ponnusamy et al. [36] note that if indirection arrays do not change between iterations, the communication schedule need not be rebuilt. Agrawal et al. [2] describe two optimizations that apply to distinct parallel loops whose schedules overlap: coalescing, which merges the two schedules, or incremental schedules, which subtracts the common part from the second schedule. Our work differs from the Inspector-Executor approach in three significant ways. First, CHAOS targets message-passing multiprocessors while C** targets customizable cache-coherent DSMs, which again can run on message-passing machines. Second, our approach requires no separate inspector and executor code, because the default protocol handles the problem of obtaining a copy of remote data which is absent when the loop is executed. Third, our approach includes incremental communication schedules, which are necessary for adaptive applications. Although the CHAOS group has looked at means to build incremental schedules, we are unaware of published descriptions of their approach or results.

Many DSM systems provide mechanisms to control a memory system to provide better support for parallel applications. Falsafi et al. [13] show that application-specific protocols can significantly improve application performance, especially for

repetitive producer-consumer sharing patterns for which write-invalidate policies are inefficient. Their implementation included hand-written custom protocols for each application. By contrast, C** uses a single protocol that is automatically invoked by compiler directives.

Ramachandran et al. [37] propose additional hardware coherence primitives (e.g, update and prefetch) to help the programmer optimize common sharing patterns. Their SEL_WRITE primitive provides functionality very similar to our predictive protocol. Our approach adds compiler analysis for automatic predictive protocol usage.

5.2 A Predictive Protocol for Repetitive Communication Schedules

Blizzard's default Stache coherence protocol provides sequentially consistent, transparent shared memory using a write-invalidate protocol [38]. Shared memory provides a high level of abstraction, which makes compiler development easier, but the write-invalidate policy incurs large overheads for producer-consumer sharing patterns (which occur repeatedly in many iterative applications).

The C** predictive protocol optimizes shared-memory communication for repetitive producer-consumer and migratory sharing patterns in data-parallel programs. It augments Stache to build communication schedules in one iteration and to presend data using a schedule in subsequent iterations. If the application's communication pattern is repetitive, the predictive protocol reduces the number of

high-latency, non-local shared data accesses. The predictive protocol builds incremental communication schedules — new requests not anticipated previously are identified through access faults and are added to the schedule for subsequent iterations. The predictive protocol was developed using Teapot, a domain-specific language that reduces the complexity of specifying and developing cache-coherence protocols [10].

This section describes two parts of C**'s predictive protocol, the first part that builds a communication schedule, and the second part that pre-sends data. Before describing the predictive protocol, we outline Stache's mechanisms and policies, and briefly describe why a write-invalidate protocol is inefficient for producer-consumer sharing patterns.

5.2.1 The Stache Shared-Memory Protocol

Stache implements sequentially-consistent shared memory using a directory-based write-invalidate protocol[38]. Stache is built on Tempest, which is a parallel programming substrate that supports fine-grain access control, i.e., at the cache block granularity (32–128 bytes). Each cache block may be in one of three states: **Invalid**, **ReadOnly**, or **ReadWrite**. Inappropriate accesses to a block (e.g., a read access to an **Invalid** block) generate faults that are vectored to a user-level handler in the Stache protocol.

Each shared-memory cache block in the system is mapped to its **home** node, where it resides initially. The home node also maintains a block's directory information, which lists multiple readers or a single writer, and is used to maintain

consistency.

A read access to an invalid block invokes a user-level Stache fault handler, which sends a message to the home node requesting a copy of the block. The home node updates its directory information and sends a read-only block back to the requesting processor. On a write access to an invalid or read-only block, the home processor invalidates all outstanding read-only copies (to maintain sequential consistency) and sends a writable block to the requestor.

5.2.2 Inefficiencies in a Write-invalidate Protocol

It is widely known that write-invalidate protocols are inefficient for iterative producer-consumer communication patterns (see, for example, [9]). Each data transfer between producer and consumer involves four messages if a data item's home location is different from the producer and consumer:

1. The consumer requests a readable copy from the home node
2. The home node requests the producer to invalidate its copy
3. The producer returns its copy to the home node
4. The home node sends the consumer a readable copy

The producer follows a similar protocol to acquire a writable copy when it generates new values.

When producer-consumer sharing patterns can be identified in an application, a write-update protocol can transfer a data item with one or two messages [13].

However, update protocols do not ensure sequential consistency and cannot be used in general.

5.2.3 Building Communication Schedules in the Predictive Protocol

C**'s predictive protocol augments Stache to collect communication information within a parallel phase. The protocol identifies, for each cache block requiring communication due to faulting accesses, whether the block was read or written (and the processors that read or wrote the block). The protocol relies on the compiler to demarcate parallel phases in the program (Section 5.3).

Since all requests to a block are routed through the home node, the predictive protocol augments Stache handlers at the home node. At run time, when the home node receives a read (or write) request from a remote node for a cache block, the augmented handler updates the communication schedule to mark the block as read (or written) in that phase. If a block is read and written within the same phase, it is marked as a "conflict" block. This can occur if there is false sharing (i.e., when two processors access distinct parts of the block), or if parallel tasks conflict.

The predictive protocol builds schedules incrementally, starting from an empty schedule. During the first iteration, the protocol identifies faulting cache block accesses and extends the schedule. In subsequent iterations, changes in the communication pattern may cause faulting accesses to additional blocks, which are identified and added to the original schedule. This allows the protocol to track

evolving sharing patterns characteristic of adaptive applications.

The predictive protocol works well for incremental additions to a schedule, but does not track deletions. When a processor no longer accesses a block, the protocol transfers the block unnecessarily. For applications whose pattern changes include a significant number of deletions, the schedule must be rebuilt often by flushing the old schedule and building a new one.

5.2.4 Using Communication Schedules to Presend Data

At the beginning of a subsequent iteration of the parallel phase, compiler directives invoke the pre-send phase of the predictive protocol on all processors to transfer data according to the communication schedule. The goal of the pre-send phase is to anticipate block requests and execute anticipated actions early.

Each processor executes one of two actions for blocks in the communication schedule for which it is the home node. For a block marked "read", the processor sends invalidations to any current writer, and forwards readable copies to all processors marked as readers. For a block marked "write", the processor invalidates current readers or writers, and forwards a writable copy to the marked writer. Currently, there is no action for blocks marked "conflict", since they occur very rarely in programs with independent parallel threads of execution. One possible action for such blocks is to anticipate the first stable block state (read or write) before the conflict occurred.

Pre-sent copies are cached at remote nodes with appropriate access control tags (**ReadOnly** or **ReadWrite**). Accesses to cached copies are handled transparently

by Tempest, usually at full hardware speeds, without invoking the protocol or other software intervention.

After all blocks in the schedule have been transferred, the protocol enforces a global barrier synchronization to ensure that all protocol cache blocks states are stable and match those expected by the default protocol. For efficiency, the predictive protocol coalesces neighboring blocks and transfers them using bulk messages to amortize message startup costs.

5.3 Identifying Potentially Repetitive Patterns

The predictive protocol relies on directives from the C** compiler to identify points in the program where potentially repetitive communication patterns exist. In C**, as in other data-parallel languages, data-parallel operations clearly divide a program's execution into sequential and parallel phases (Section 2.1.1). The C** compiler uses data-flow analysis to identify repetitive parallel phases that require communication, and augments these phases with directives invoking the predictive protocol.

Our simple compiler analysis is optimistic and conservative and does not attempt to identify actual patterns of communication in the program (e.g., nearest-neighbor communication), or even that the pattern is really repetitive in the sense that data items requested in a previous iteration will be requested again in a following iteration. While such analysis is routine for programs with mostly static communication patterns, it is infeasible for programs with dynamic communication patterns such as adaptive applications. Our analysis can wrongly identify

a non-repetitive pattern as a repetitive one, leading to slower (but still correct) execution of the program with the predictive protocol.

This section describes our data-flow analysis, which proceeds in two phases. First, parallel functions are analyzed to broadly classify their access patterns. Second, the sequential part of the program (which includes calls to parallel functions) is analyzed to identify where annotations for parallel phases must be placed.

5.3.1 Parallel Function Analysis - Identifying Access Patterns

Calling a parallel function on an Aggregate creates multiple function invocations, one for each element of the Aggregate. Each parallel function invocation "owns" the element of the Aggregate on which it operates. In addition to its "own" element, each invocation may also access neighboring Aggregate elements or elements from other global Aggregates. For example, the parallel function `update` in Figure 5.1 implements a simple unstructured mesh update on a bipartite mesh (partitioned into `primal` and `dual`). The edge descriptors (and their corresponding transfer coefficients) are stored with each mesh element.

The parallel function `update` in Figure 5.1 includes unstructured accesses to the `dual` mesh, some of which require inter-processor communication. For each parallel function, the C** compiler uses context-insensitive analysis to compile a list of all Aggregate member accesses that potentially require communication. Each access is (conservatively) categorized as a **Home** access (for example, access to the "own" element), or a **Non-Home** access (for all other accesses). For

```
class Node {
    double value;
    int edges[MAX_EDGES];  double coeff[MAX_EDGES];
};

class Mesh(Node) [])  {   /* Member functions */    };

void update(parallel Mesh &primal, Mesh &dual) parallel
{
    /* Loop over all in-edges */
    for (int i = 0; i < primal[#0].in_degree; i ++)
        primal[#0].value -=
             dual[primal[#0].edges[i]].value * primal[#0].coeff[i];
}
```

Figure 5.1: Unstructured mesh update in C**

example, the summary access list of function `update` in Figure 5.1 contains two elements, (primal, Write access, Home), and (dual: Read access, Non-Home).

5.3.2 Compiler Analysis to Place Directives

The second step analyzes the sequential portion of the program which includes calls to parallel functions. First, the compiler builds a flow graph of the sequential program, mapping parallel function data access lists back to function call sites. As our compiler currently does not support inter-procedural analysis, the sequential portion is restricted to the main function. For example, Figure 5.2 displays the control flow graph (CFG) for the main loop in the sequential portion of Barnes-Hut (Section 5.4.2) annotated with access lists.

We perform data-flow analysis on the sequential section of the program to determine, for each Aggregate at each program point, whether cached copies of Aggregate elements may exist on remote processors due to unstructured read or write accesses. If these copies cannot exist, a single copy of each element is present on its home processor, created by an owner write access. Analogous to reaching definitions, we define the reaching unstructured accesses property, which is true whenever cached copies of an Aggregate element may exist on remote processors.

The compiler uses a forward-flow, any-path data-flow analysis to compute reaching unstructured accesses for each Aggregate at each program point, using a framework identical to the reaching-definition problem. There are three transfer functions for parallel function data accesses:

1. Owner write accesses kill reaching unstructured accesses, because the remote

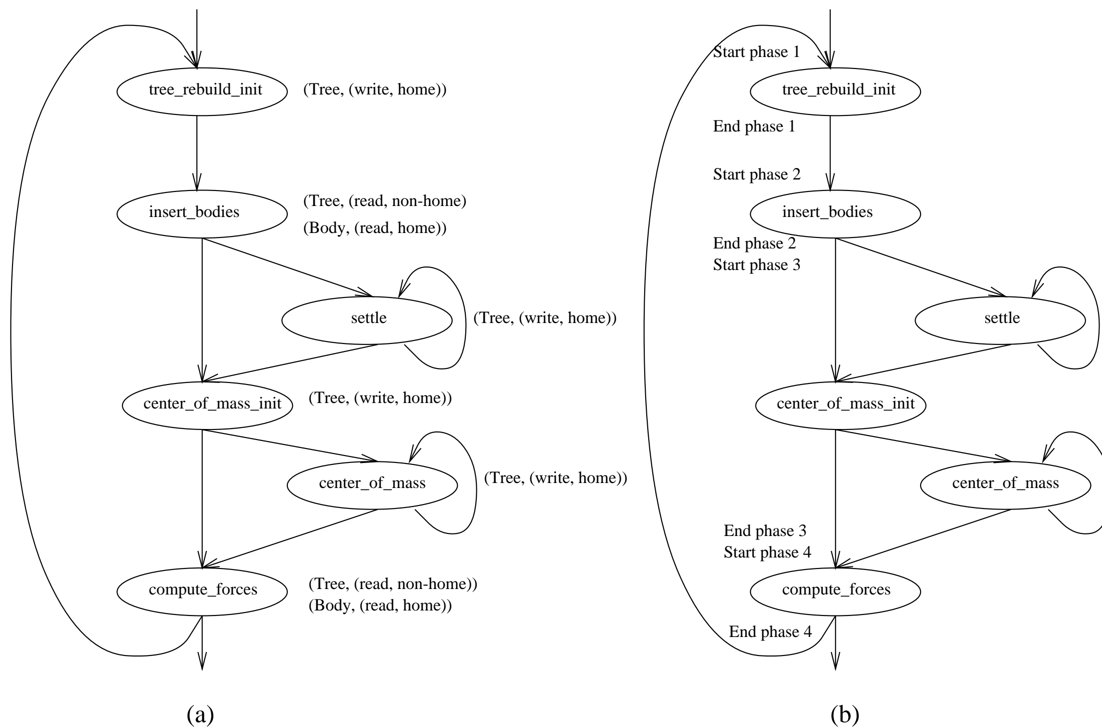

Figure 5.2: Control flow graph for the main sequential loop in Barnes-Hut. CFG (a) is annotated with parallel function access patterns. CFG (b) is annotated with runtime phase directives for the predictive protocol.

copies are invalidated.

2. Unstructured write accesses kill reaching unstructured accesses, but generate potentially new unstructured accesses.

3. Unstructured read accesses do not kill reaching unstructured accesses (because the protocol allows multiple readers), and generate unstructured accesses.

The compiler computes reaching unstructured accesses using an iterative bit-vector based data-flow computation on the sequential control flow graph.

Results of the reaching unstructured access data-flow computation direct the placement of run-time protocol directives. A parallel function call requires a communication schedule and preceding predictive protocol phase if, for any Aggregate

1. The call is reached by unstructured accesses and includes owner write accesses, or

2. The call includes unstructured accesses itself, whether the reaching property includes unstructured accesses or not.

The placement algorithm also includes one optimization to coalesce multiple communication schedules. The compiler uses an inside-out pass on the CFG to coalesce neighboring phases that include only home accesses, and moves schedules out of loops that contain only home accesses (e.g., function `center_of_mass` in Figure 5.2). This optimization is analogous to communication schedule coalescing in the inspector-executor model [2], and amortizes the overhead of the predictive

protocol over multiple parallel functions. In Figure 5.2, this optimization allowed a single directive for phase 3.

5.4 Measuring the Optimizations

In this section, we measure the effect of compiler-directed shared-memory communication on three iterative data-parallel scientific applications (Adaptive, Barnes, and Water) which are described briefly in Table 5.1. Adaptive and Barnes have dynamic repetitive communication patterns, and Water demonstrates a static repetitive communication pattern. All three applications spend a non-trivial fraction of execution time in remote access latency (Figures 5.3, 5.4, 5.5). We briefly outline the algorithm for each application, and compare the performance of C** versions with and without optimized communication. For Barnes, we also compare both versions against a hand-optimized SPMD version (written by others) that uses an application specific protocol for efficiency [13]. For Water, we compare both versions against the Splash-2 version [50] that is optimized for transparent shared memory.

Each performance graph compares the execution time of two or more versions of each benchmark application relative to the fastest version of that application. All execution times were measured on a 32-processor Thinking Machines CM-5 with Blizzard [45]. Each bar in the graph is divided into three sections:

Remote data wait Time spent waiting for non-local memory accesses to complete

Program	**Brief Description**	**Data set**
Adaptive	Structured adaptive mesh	128x128 mesh, 100 iterations
Barnes	Hierarchical N-body	16384 bodies, 3 iterations
Water	Molecular dynamics	512 molecules, 20 iterations

Table 5.1: Benchmark applications

Predictive protocol Time spent in the pre-send phase of the predictive protocol

Compute+Synch Time spent in computation and synchronization. This portion of the execution time varies between different versions of the same program because of differences in synchronization time.

We also experimented with different cache block sizes for each application. In general, the predictive protocol worked best for small cache blocks (the smallest being 32-bytes), while the unoptimized or hand-tuned SPMD codes were able to exploit larger cache blocks effectively. In addition to the 32-byte block comparison between unoptimized and optimized codes, we also present execution times using programs with larger cache block sizes which minimized execution time for unoptimized or hand-optimized codes.

5.4.1 Adaptive

Adaptive is a structured mesh calculation that computes electric potentials in a box. The program imposes a mesh over the box and computes the potential at each point by averaging its four neighbors. At points where the gradient is steep,

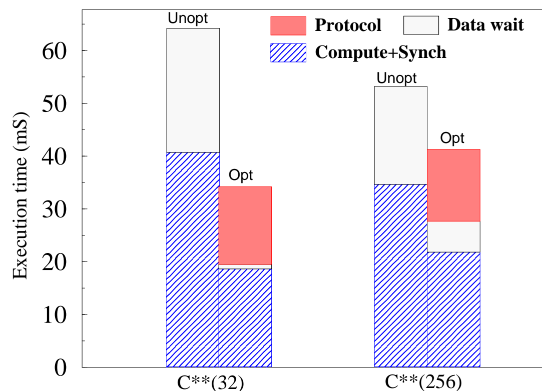

Figure 5.3: Execution time for 4 C** versions of **Adaptive** — C** versions with and without optimized communication at 2 different cache block sizes. Numbers in parentheses indicate cache block sizes.

finer detail is necessary and the program subdivides the cell into four child cells. This process iterates until the mesh relaxes. Initially, the mesh is represented by a two-dimensional array, and dynamically allocated quad trees capture cell subdivision. Each iteration of the program consists of a red-black sweep over the mesh computing averages. Within each sweep, each cell updates values in its quad tree, reading values from neighboring points. The predictive protocol optimizes data movement from neighbor reads in the quad tree.

Figure 5.3 shows that the predictive protocol successfully reduces shared-data wait time by pre-sending data. The protocol also indirectly reduces synchronization time in Adaptive, resulting in significantly lower total execution time. Synchronization time is reduced because load imbalance in Adaptive implies that the shared-data wait time is distributed unevenly among processors, and differences in wait time contribute to synchronization time on lightly loaded processors. At a

larger cache block size of 256 bytes (the best case for the unoptimized program), the predictive protocol is less effective because it transfers larger amounts of data, some of which may be redundant. The best optimized version of Adaptive is 1.56x faster than the best unoptimized version.

5.4.2 Barnes

Barnes [47] simulates the movement of bodies in a gravitational system over time. The bodies are modeled as point masses that exert gravitational forces on other bodies. The algorithm computes forces between bodies which are used to update body positions in each time step. Rather than computing all N^2 forces, Barnes approximates the force exerted by a distant collection of bodies by that of a point mass at the center of mass of the collection.

Barnes uses an oct-tree to represent bodies in 3-dimensional space. Each node in the tree represents a region in space, with a child representing one octant of its parent's space. The tree is unbalanced and deeper in regions of high body density. To calculate the force on a body, the algorithm performs a depth-first traversal of the tree. If an interior node is sufficiently far away from the body, the bodies in that region are approximated by a point mass at the tree node. Otherwise, the algorithm "opens" up the interior node and traverses its subtrees. If the force computation encounters a body at the leaf of the tree, it computes interactions with that body.

The Barnes algorithm in C** includes unstructured accesses to tree nodes during two phases, the force computation phase, and the tree-build phase, with

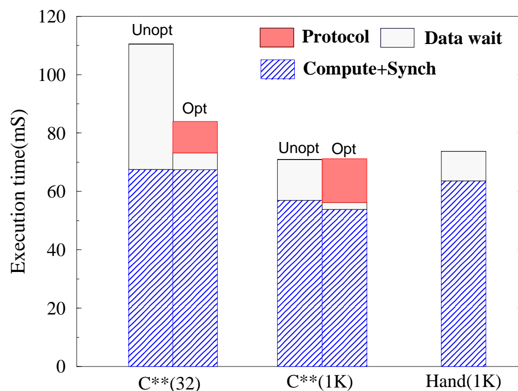

Figure 5.4: Execution time for 5 versions of **Barnes** — C** versions with and without optimized communication at 2 different cache block sizes, and hand-optimized SPMD. Numbers in parentheses indicate cache block sizes.

the center-of-mass calculation in between. The compiler inserts directives for 4 parallel phases in the program where transitions between non-home and home accesses occur (Figure 5.2).

Figure 5.4 shows that communication optimization reduces shared-memory wait time significantly for 32-byte cache blocks. However, Barnes shows good spatial locality and the unoptimized version benefits significantly from 1024-byte blocks making it marginally faster than the optimized version. Both 1024-byte versions are slightly faster than a hand-optimized SPMD version of Barnes [13] that uses a write-update protocol for efficient shared-memory communication on the CM-5.

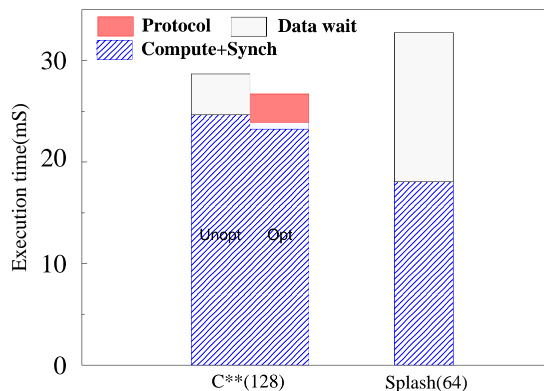

Figure 5.5: Execution time for 3 versions of **Water** — C** with and without optimized communication, and shared-memory Splash. Numbers in parentheses indicate cache block sizes.

5.4.3 Water

Water [47] evaluates forces and potentials in a system of water molecules over a number of time steps. The potential of the system includes inter-molecular potentials arising from interactions between molecules. The program computes interactions between all pairs of molecules that lie within a spherical cutoff range equal to half the length of the box enclosing all molecules. In the data-parallel implementation of Water, each molecule potentially computes interactions with half the remaining molecules following it in the ordered data set.

Compiler-directed communication optimizations target the interaction computation phase which uses a static repetitive producer-consumer sharing pattern — a molecule's position updated in one iteration is read by $\frac{n}{2}$ other molecules in the following iteration.

Communication optimization reduces shared-memory wait time for Water (Fig-

ure 5.5), but results in small execution time improvements overall (1.05x). The optimized version is 1.2x faster than the Splash version [50], which is optimized for transparent shared memory, and does not utilize custom protocols or message-passing primitives for communication. The cache block sizes were chosen to show the best case for each version.

5.4.4 Discussion

The predictive protocol decreases remote memory access latency at the cost of an extra pre-send phase and the cost of building communication schedules in augmented protocol handlers. This technique is beneficial on multiprocessor machines with significant remote memory access latency, such as Blizzard on the CM-5 (200 microseconds average remote access latency), or networks of workstations without hardware support for shared memory. The tradeoff is likely to be different for shared-memory multiprocessors or hardware-assisted DSMs, which have smaller remote access latencies.

The predictive protocol also coalesces neighboring cache blocks in the pre-send phase to amortize message startup costs over large messages. The benefits of this optimization should extend uniformly to all classes of distributed-memory multiprocessor machines, possibly with better results than on the CM-5 network which is optimized for small messages.

5.5 Summary

Many scientific applications simulate physical systems using iterative parallel computations. Typically, these applications involve communication patterns that are also repetitive. This chapter demonstrates that cooperation between a data-parallel language compiler and a predictive protocol in a cache-coherent DSM can automatically improve shared-memory communication for repetitive producer-consumer or migratory communication patterns. The compiler uses simple static analysis to identify points in the program where potentially repetitive communication patterns exist. The predictive protocol augments the default shared-memory protocol to use communication schedules generated in one iteration to pre-send data in subsequent iterations.

The combination of compiler-analysis and memory system support gives this approach two advantages. First, dynamic run-time support from the memory system allows our approach to optimize adaptive problems whose reference patterns cannot be analyzed by a compiler and which incur large overheads in compiler-implemented shared-memory approaches. Second, communication pattern analysis in the compiler enables automatic custom protocol usage. This approach inherits some of the advantages of application-specific protocols, but is far simpler for a programmer.

Experiments with three applications show that pre-sending data with this approach effectively reduces the amount of time spent waiting for shared data when compared to the request-response model of a write-invalidate coherence protocol. In two cases, the optimized program was 1.05x and 1.50x faster than

the unoptimized version. In the third case, the unoptimized program was able to exploit a larger cache block size to run slightly faster than the optimized program.

Chapter 6

Conclusion

Data-parallel programming languages provide a promising solution to the problem of developing portable parallel applications. They include a number of high-level abstractions, such as parallelism on data and global variable name space, that support rapid parallel program development. The large number of data-parallel programming languages (e.g., HPF [25], NESL [7] and pC++ [32]) is a good testament to their popularity. However, languages by themselves are not useful unless they can be compiled for efficient execution. Considerable research has been devoted to compiling high-level data-parallel programs for scalable distributed-memory parallel machines. Much of this work has focused on communication optimization for programs that exhibit regular patterns of communication, which a compiler can analyze precisely and transform to an efficient executable with optimized communication.

This thesis presents three new techniques that enable efficient execution of a larger class of data-parallel programs, specifically, programs with irregular and

dynamic communication patterns. Two of these techniques exploit user-level cache-coherence protocols in a cache-coherent distributed shared-memory system. These techniques were developed for the data-parallel language C** [31] targeting the Tempest interface [38]. Tempest provides both shared memory and message-passing mechanisms on distributed-memory machines, allowing a compiler to implement a shared address space easily, and optimize communication for specific communication patterns.

The first technique targets the implementation of conflict-free data access in C**'s parallel operations [30]. C** clearly defines the semantics of conflicting memory accesses in coarse-grain parallel tasks to avoid data-access conflicts, and allow nearly-deterministic execution. We explored two ways in which a compiler and run-time system can use copy-on-write to implement the high-level semantics of C**. For parallel functions with regular data-access patterns that a compiler can analyze, the compiler can insert code in the program to maintain copies. For functions with access that a compiler cannot identify precisely (e.g., through pointers), the compiler cooperates with a Loosely-Coherent Memory (LCM) system that extends the coherence protocol to create copies at a fine granularity. Using performance data from three variants of mesh relaxation codes, we show that these two techniques complement one another. Compiler-copying is efficient when compiler data access analysis is precise, and LCM works well when the compiler cannot precisely identify data accesses in a program. The benefit of providing two alternatives is that a compiler can choose the efficient alternative based on the precision of its data access analysis, and even use both in a program.

The second technique, user-defined reductions, describes how and why data-parallel languages must extend reductions to allow user-defined operators. Reductions use a binary operator to combine multiple values that collide when assigned to a target. Reductions are extremely common in parallel applications, but most parallel languages only provide a predefined set of reductions. User-defined reductions extend reductions in two important ways: they allow powerful combining operations, such as location reductions or building a list of colliding values, and they extend reductions to user-defined data types. This thesis demonstrates the advantages of user-defined reductions in a data-parallel language, focusing on the benefits of extending reductions to user-defined data types. We show that they provide a simple intuitive specification of particle movement in a particle-in-cell code when compared to two other well-known methods. We also present a simple implementation of user-defined reductions in a coarse-grain data-parallel language, which relies on simple message-passing support from the run-time system. We analyze the performance of user-defined reductions by comparing the execution time of two versions each of four benchmark applications (Table 4.2), one in a data-parallel language with user-defined reductions and the other SPMD code optimized with application-specific communication for the reduction pattern. For applications with dynamic communication patterns, the execution times of the C** and SPMD programs differed by at most 10%. The C** version of EM3D, which exhibits a static repetitive reduction pattern, was up to 2.6x slower than the SPMD version, but could be optimized using the Inspector-Executor compiling paradigm.

The third technique describes how a data-parallel language compiler and a predictive cache-coherence protocol can implement shared-memory communication efficiently for applications with unpredictable but repetitive communication patterns. The compiler uses data-flow analysis to identify points in the program where *potential* repetitive communication patterns exist. A predictive protocol in the runtime system augments the default shared-memory protocol to build a communication schedule for one iteration and utilize a schedule to pre-send data to satisfy data requests in following iterations. As a result, the predictive protocol reduces the number of shared-memory data requests that cannot be satisfied locally, and the total remote memory access latency. The predictive protocol is incremental and applies to adaptive applications. Simple compiler analysis automatically applies the predictive protocol for applications with repetitive producer-consumer sharing patterns for which a sequentially-consistent memory coherence protocol would incur large overheads [9]. We show that compiler-directed shared-memory communication is effective at reducing remote latency on three applications (Adaptive, Water, and Barnes) on a 32-processor CM-5. For Adaptive and Water, the optimized version was faster than the best-optimized version. For Barnes, which shows excellent spatial locality, the optimized and non-optimized versions are comparable.

All three techniques presented in this thesis target efficient execution of non-regular applications that a compiler finds difficult to analyze. The variety of benchmarks presented in this thesis shows that these techniques are effective, and widens the class of data-parallel programs to include irregular and dynamic

applications. Thus, this thesis takes one more step towards the goal of providing portable, efficient, and high-level languages to help develop parallel applications.

Appendix A

C** Benchmarks

Name	**Brief Description**	**Speedup**	
		CM-5	**COW**
Stencil	Structured mesh		
Adaptive	Structured adaptive mesh	5.7	
EM3D	Unstructured mesh	11.8	
Moldyn	Molecular dynamics	23.9	5.1
Water	Molecular dynamics	13.1	
DSMC	Particle-in-cell	18.8	8.7
FFT	Fast Fourier Transform	8.3	
Barnes	Hierarchical N-body	21.7	
FMM	Hierarchical N-body		
QCD	Quantum chromodynamics		

Table A.1: A list of benchmarks in C**

Bibliography

[1] Jeanne C. Adams, Walter S. Brainerd, Jeanne T. Martin, Brian T. Smith, and Jerrold L. Wagener. *Fortran 90 Handbook*. McGraw-Hill, 1992.

[2] Gagan Agrawal and Joel Saltz. Interprocedural Compilation of Irregular Applications for Distributed Memory Machines. In *Proceedings of Supercomputing '95*, San Jose, CA, November 1995.

[3] Gagan Agrawal, Alan Sussman, and Joel Saltz. Compiler and Runtime Support for Structured and Block Structured Applications. In *Proceedings of Supercomputing '93*, pages 578–587, 1993.

[4] Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwanepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[5] Monica Beltrametti, Kenneth Bobey, and John R. Zorbas. The Control Mechanism for the Myrias Parallel Computer System. *Computer Architecture News*, 16(4):21–30, September 1988.

[6] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence. In *Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 168–176, February 1990.

[7] Guy E. Blelloch. NESL: A Nested Data-Parallel Language (Version 2.6). Technical Report CMU-CS-93-129, Department of Computer Science, Carnegie Mellon University, April 1993.

[8] Satish Chandra and James R. Larus. HPF on Fine-Grain Distributed Shared Memory: Early Experience. In Utpal Banerjee, Alexandru Nicolau, David

Gelernter, and David Padua, editors, *Proceedings of the Ninth Workshop on Languages and Compilers for Parallel Computing.* August 1996.

[9] Satish Chandra, James R. Larus, and Anne Rogers. Where is Time Spent in Message-Passing and Shared-Memory Programs? In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 61–75, October 1994.

[10] Satish Chandra, Brad Richards, and James R. Larus. Teapot: Language Support for Writing Memory Coherence Protocols. In *Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation (PLDI)*, May 1996.

[11] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel Programming in Split-C. In *Proceedings of Supercomputing '93*, pages 262–273, November 1993.

[12] Raja Das, Mustafa Uysal, Joel Saltz, and Yuan-Shin Hwang. Communication Optimizations for Irregular Scientific Computations on Distributed Memory Architectures. *Journal of Parallel and Distributed Computing*, 22(3):462–479, September 1994.

[13] Babak Falsafi, Alvin Lebeck, Steven Reinhardt, Ioannis Schoinas, Mark D. Hill, James Larus, Anne Rogers, and David Wood. Application-Specific Protocols for User-Level Shared Memory. In *Proceedings of Supercomputing '94*, pages 380–389, November 1994.

[14] High Performance Fortran Forum. HPF-2 Scope of Activities and Motivating Applications, November 1994. Available at ftp://hpsl.cs.umd.edu/pub/hpf_bench/hpf2.ps.

[15] Geoffrey Fox, Seema Hiranandani, Ken Kennedy, Charles Koelbel, Ulrich Kremer, Chau-Wen Tseng, and Min-You Wu. Fortran D language specification. Technical Report CRPC-TR900749, Centre for Research on Parallel Computation, Rice University, December 1990.

[16] Geoffrey C. Fox. What Have We Learnt from Using Real Parallel Machines to Solve Real Problems. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, volume 2, pages 897–955, Pasadena, CA, January 1988.

[17] Philip J. Hatcher and Michael J. Quinn. *Data-Parallel Programming on MIMD Computers*. MIT Press, 1991.

[18] Phillip J. Hatcher, Michael J. Quinn, Anthony J. Lapadula, Bradley K. Seevers, Ray J. Anderson, and Robert R. Jones. Data-Parallel Programming on MIMD Computers. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):377–383, July 1991.

[19] High Performance Fortran Forum. High Performance Fortran Language Specification. Version 1.0, May 1993.

[20] W. Daniel Hillis. *The Connection Machine*. MIT Press, Cambridge, Mass., 1985.

[21] W. Daniel Hillis and Guy L. Steele, Jr. Data Parallel Algorithms. *Communications of the ACM*, 29(12):1170–1183, December 1986.

[22] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiling Fortran D for MIMD Distributed-Memory Machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[23] Alexander C. Klaiber and James L. Frankel. Comparing Data-Parallel and Message-Passing Paradigms. In *Proceedings of International Conference on Parallel Processing*, pages II–11–II–20, August 1993.

[24] Charles Koelbel and Piyush Mehrotra. Compiling Global Name-Space Parallel Loops for Distributed Execution. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):440–451, October 1991.

[25] Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele Jr., and Mary E. Zosel. *The High-Performance Fortran Handbook*. MIT Press, 1994.

[26] Scott R. Kohn and Scott B. Baden. Irregular coarse-grain data parallelism under LPARX. *Journal of Scientific Programming*. To appear.

[27] Scott R. Kohn and Scott B. Baden. A Parallel Software Infrastructure for Structured Adaptive Mesh Methods. In *Proceedings of Supercomputing '95*, San Jose, CA, November 1995.

[28] David Kranz, Kirk Johnson, Anant Agarwal, John Kubiatowicz, and Beng-Hong Lim. Integrating Message-Passing and Shared-Memory: Early Experience. In *Fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 54–63, May 1993.

[29] Jeffrey Kuskin et al. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 302–313, April 1994.

[30] James R. Larus, Brad Richards, and Guhan Viswanathan. LCM: Memory System Support for Parallel Language Implementation. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 208–218, October 1994.

[31] James R. Larus, Brad Richards, and Guhan Viswanathan. Parallel Programming in C**: A Large-Grain Data-Parallel Programming Language. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming Using C++*, chapter 8, pages 297–342. MITP, 1996.

[32] Jenq Kuen Lee and Dennis Gannon. Object Oriented Parallel Programming, Experiments and Results. In *Proceedings of Supercomputing '91*, pages 273–282, Albuquerque, NM, November 1991.

[33] Calvin Lin and Lawrence Snyder. ZPL: An Array Sublanguage. In *Languages and Compilers for Parallel Computing (Proceedings of the Sixth Internationa Workshop)*, pages 96–114. Springer-Verlag, 1994.

[34] Tom Lovett and Russell Clapp. STiNG: A CC-NUMA Computer System for the Commercial Marketplace. In *Proceedings of the 23rd International Symposium on Computer Architecture*, pages 308–317, Santa Margherita Ligure, Italy, June 1995.

[35] Shubhendu S. Mukherjee, Shamik D. Sharma, Mark D. Hill, James R. Larus, Anne Rogers, and Joel Saltz. Efficient Support for Irregular Applications on Distributed-Memory Machines. In *Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 68–79, July 1995.

[36] Ravi Ponnusamy, Joel Saltz, and Alok Choudhary. Runtime-Compilation Techniques for Data Partitioning and Communication Schedule Reuse. In

Proceedings of Supercomputing '93, pages 361–370, Portland, Oregon, November 1993.

[37] Umakishore Ramachandran, Gautam Shah, Anand Sivasubramaniam, Aman Singla, and Ivan Yanasak. Architectural Mechanisms for Explicit Communication in Shared-Memory Multiprocessors. In *Proceedings of Supercomputing '95*, San Jose, CA, November 1995.

[38] Steven K. Reinhardt, James R. Larus, and David A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 325–337, April 1994.

[39] Bradley Eric Richards. *Memory Systems for Parallel Programming.* PhD thesis, University of Wisconsin-Madison, 1996.

[40] Anne Rogers and Keshav Pingali. Process Decomposition Through Locality of Reference. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation (PLDI)*, pages 69–80, June 1989.

[41] John R. Rose and Guy L. Steele Jr. C*: An Extended C Language for Data Parallel Programming. In *Proceedings of the Second International Conference on Supercomputing*, pages 2–16, Santa Clara, California, May 1987.

[42] Gary W. Sabot. *The Paralation Model: Architecture-Independent Parallel Programming.* MIT Press, 1988.

[43] Joel H. Saltz, Ravi Mirchandaney, and Kay Crowley. Run-Time Parallelization and Scheduling of Loops. *IEEE Transactions on Computers*, 40(5):603–612, May 1991.

[44] Ioannis Schoinas, Babak Falsafi, Mark D. Hill, James R. Larus, Christopher E. Lucas, Shubhendu S. Mukherjee, Steven K. Reinhardt, Eric Schnarr, and David A. Wood. Implementing Fine-Grain Distributed Shared Memory On Commodity SMP Workstations. Technical Report 1307, Computer Sciences Department, University of Wisconsin–Madison, March 1996.

[45] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David A. Wood. Fine-grain Access Control for Distributed Shared Memory. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 297–307, October 1994.

[46] Shamik D. Sharma, Ravi Ponnusamy, Bongki Moon, Yuan-Shin Hwang, Raja Das, and Joel Saltz. Run-time and Compile-time Support for Adaptive Irregular Problems. In *Proceedings of Supercomputing '94*, pages 97–106, November 1994.

[47] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared Memory. *Computer Architecture News*, 20(1):5–44, March 1992.

[48] Guy L. Steele Jr. Making Asynchronous Parallelism Safe for the World. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 218–231, January 1990.

[49] Guy L. Steele Jr. and W. Daniel Hillis. Connection Machine LISP: Fine-Grained Parallel Symbolic Processing. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, pages 279–297, August 1986.

[50] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, Philadelphia, PA, May 1996.

[51] Bwolen Yang, Jon Webb, James M. Stichnoth, David R. O'Halloran, and Thomas Gross. Do&Merge: Integrating Parallel Loops and Reductions. In Utpal Bannerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, pages 169–183, Portland, Oregon, August 1993. Springer-Verlag.