INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Fibers: A High Performance MPMD Mechanism for Integrated Asynchronous Data and Control Transfer

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

by

Aloke Majumdar

November 1994

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

UMI Number: 9523199

Copyright 1994 by
Majumdar, Aloke
All rights reserved.

UMI Microform Edition 9523199
Copyright 1995, by UMI Company. All rights reserved.

This microform edition is protected against unauthorized
copying under Title 17, United States Code.

UMI

300 North Zeeb Road
Ann Arbor, MI 48103

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Abstract

Fibers: A High Performance MPMD Mechanism for Integrated Asynchronous Data and Control Transfer

Aloke Majumdar

Yale University

1994

Existing MPMD data transfer mechanisms for MPP architectures involve a compromise between expressiveness and efficiency. The use of higher-level blocking mechanisms produces code with simple control flow but sub-optimal performance due to the presence of buffer copying or unnecessary idle time. Performance can be improved by using lower-level asynchronous mechanisms, but only at the cost of a considerable increase in the complexity of program control flow.

This dissertation makes the case for a language mechanism which provides support for data transfers coordinated with asynchronous execution. It introduces a new program composition mechanism, the *fiber*, whose semantics are designed for this purpose. The use of the fiber constructs permits the specification of asynchronous data transfers which nevertheless result in relatively simple program control flow.

Incorporating the data and control transfer mechanism in a language construct produces other benefits as well. It makes possible the use of compilation technology to produced optimized transfers which would be impossible to hand-code or would be non-portable and error-prone if so coded. It also results in greater portability of source code and, if the language semantics are appropriately designed, more robust programs. The fiber mechanism provides all these benefits.

The properties that make the fiber construct a useful mechanism for specifying user programmed applications also make it a suitable target language for an HPL compiler.

We begin with a detailed analysis of all the costs associated with data transfer. The cost structure that is thus revealed serves as the basis for an exhaustive list of optimizations which can be used to reduce these costs. This is followed by an examination of the semantic requirements for an integrated asynchronous data and control transfer mechanism.

The results of this analysis were used to guide the design of the fiber mechanism. We present the syntax and semantics of this mechanism and demonstrate its expressiveness and safety characteristics. We then show how the fiber constructs can be efficiently (and portably) implemented in general, as well as how compiler optimizations can produce superior performance in special cases. The results of experiments confirm the promise of this approach, showing performance comparable or superior to that of existing mechanisms on individual transfers.

Finally, we apply the fiber mechanism to improve the performance of a real MPMD program. The use of the fiber constructs results in a significant improvement in the performance of a fairly typical MPMD program without a concomitant increase in the complexity of the code.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Acknowledgments

I would like to thank my advisor Marina Chen for many years of support, encouragement and pragmatic advice.

The members of my committee, Marina Chen, Young-Il Choo and Geoffrey Fox, have generously taken the time out of their busy schedules to comment on this dissertation. Sanjay Ranka made me aware of the LogP model at an early stage of writing. Satish Pai and Dong-Yuan Chen carefully read early drafts of this thesis and provided valuable feedback. Thorsten von Eicken and Lok-Tin Liu helped with CMAM at various points of the implementation and John Holm generously provided access to his application code.

I would like to mention the various graduate students, past and present, who provided years of collegiality and cameraderie: Jingke Li, Michel Jacquemin, Lee-Chung Lu, Joe Rodrigue, Allan Yang, Cheng-Yee Lin, Satish Pai, Janet Wu, Dong-Yuan Chen, Rob Bjornson, Scott Fertig, Shakil Ahmed, Bhaskar Ghosh, David Kaminsky, Alok Dutt, Page Starr and Susanne Hupfer. I remain grateful to Abhiram Ranade for taking the time to talk to a very green graduate student.

It would be remiss not to acknowledge the outstanding researchers who taught me by example (although they may dispute that I have learned anything). Interacting with them proved to be the most valuable part of the experience of being a graduate student. Young-Il Choo taught me a healthy skepticism, to look deeper into problems and to constantly search for better answers. From Nicholas Carriero I learned how to identify the real problem that needed to be solved and to focus on its relevant aspects. Stan Eisenstadt remains an inspiration in being able to look at a problem in the context of the big picture and identifying its significance (or lack thereof). From Marina Chen I learned perhaps the most important thing: the pragmatics of going from a good concept to a useful piece of research.

It is also a pleasure to acknowledge more formative influences. Prof. P.C.P. Bhatt of IIT Delhi gave advice whose full import dawned upon me years later. Prof. Emeritus J. Roy of ISI Calcutta, provided many years of guidance and inspiration.

I am grateful to my friends Sashi Reddi and Srikanth Srinivasan for their support and level-headed perspective over the years.

Most of all I thank my family. I could always count on my dada Anindo for his patience and understanding. Finally, my amma and baba, Grace and Ashutosh, have provided decades of support and encouragement. This work is dedicated to them.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

© Copyright by Aloke Majumdar 1994

All Rights Reserved

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Contents

List of Tables iv

List of Figures v

1 Introduction 1

1.1 Motivation 1

1.2 Overview of Present Work 3

1.2.1 Data Transfer Costs 3

1.2.2 Transfer Coordination 4

1.2.3 The Fiber Abstraction 5

1.2.4 Fiber Implementation 5

1.2.5 Fiber Application 6

1.3 Connections to Related Work 7

1.3.1 Cost Model 7

1.3.2 Transfer Protocols & Asynchronous Execution 7

1.3.3 Language Semantics 8

1.3.4 HPL Compiler Target 9

2 Data Transfer Costs 10

2.1 Compute Model 11

2.1.1 MPP Architectural Abstraction 11

2.1.2 System Model 12

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2.1.3 Programming Model 12
2.2 Data Transfer Model 13
2.2.1 Single Packet Transfer 13
2.2.2 Multi Packet Transfer 15
2.2.3 Abstract Transfer Primitives 17
2.3 Data Transfer Costs 17
2.3.1 Cost Structure 18
2.3.2 Magnitude of Cost Components 20
2.4 Optimizing Data Transfer 24
2.4.1 Pure Transfer 25
2.4.2 Protocol Overhead 25
2.4.3 Transfer Related 26
2.5 Semantics, Overlapped Computation & Compilation 27

3 Transfer Coordination **29**
3.1 The I-BAD-C Framework 29
3.2 Transfer Protocols 31
3.3 Example: Global Array Assignment 32
3.4 Preliminaries 34
3.5 Blocking MP 34
3.5.1 I-BAD-C Characteristics 36
3.6 Nonblocking MP 39
3.6.1 I-BAD-C Characteristics 43
3.7 Active Message 44
3.7.1 I-BAD-C Characteristics 47
3.8 Complexity, Efficiency & Safety 48

4 The Fiber Abstraction **49**
4.1 Design Goals & Constraints 49

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.2 Fiber Computation 50
4.2.1 Asynchronous Execution 50
4.2.2 Data Transfer 51
4.2.3 Usage Note 52
4.3 C-Eli 52
4.3.1 `eli_main` 52
4.4 Fiber Definition 53
4.4.1 Fiber Parameters 53
4.4.2 `userbound` Parameters & Transfer Binding Block 54
4.5 Fiber Invocation & Coordination 55
4.5.1 Intra-Node 55
4.5.2 Inter-Node 57
4.5.3 Argument Passing 58
4.6 Array Assignment Example 59
4.7 I-BAD-C Characteristics 60
4.8 Implementing Common Idioms 61
4.8.1 Synchronous Communication 61
4.8.2 Fuzzy Barrier 62
4.8.3 Fuzzy Global Reduction 64
4.9 Fiber Expressiveness & Safety 64

5 Fiber Implementation **66**
5.1 Implementation Challenge 66
5.2 Invocation & Coordination 67
5.2.1 Fiber Compilation 67
5.2.2 Fiber Scheduling 69
5.2.3 Execution Cost 70
5.3 Data Transfer 71
5.3.1 General Protocol 71

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

5.3.2 Execution Cost 73
5.4 Compiler Optimizations 75
5.4.1 Invocation & Coordination 75
5.4.2 Data Transfer 76
5.5 CM5 Implementation 78
5.5.1 Invocation & Coordination 80
5.5.2 Data Transfer 80
5.5.3 Compiler Optimizations 81
5.5.4 Performance 82
5.6 Fiber Efficiency 84

6 Fiber Application **85**
6.1 User Code 86
6.1.1 QR Factorization 86
6.1.2 Parallelization 87
6.1.3 MP Realization 89
6.1.4 Converting to C-Eli 92
6.1.5 Comparative Performance 95
6.2 HPL Target 95
6.3 Fiber Utility 96

7 Conclusion **98**
7.1 Future Directions 98
7.1.1 Semantic Refinement 98
7.1.2 Transfer Patterns & Aggregation 99
7.1.3 Flow Analysis 99

Bibliography **100**

A C-Eli Syntax **104**

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

List of Tables

2.1 **Pure Transfer:** CM5 21

2.2 Inferring **Protocol Overhead:** CM5 21

2.3 **Buffer Management & Binding:** CM5 22

2.4 **Network Service / Protocol Coordination:** CM5 22

2.5 Inferring **Pure Transfer & Protocol Overhead:** DELTA 23

2.6 **Protocol Coordination:** DELTA 24

5.1 Fiber Invocation & Coordination Performance 82

5.2 Data Transfer Performance 83

6.1 QR Factorization: Comparative Performance 95

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Ex.1122 / IPR2025-00916 / Page 12 of 119
Intel Corporation v. Advanced Cluster Systems, Inc.

List of Figures

2.1 MPP Architectural Abstraction 11
2.2 Transfer Timeline: NB to NB 14
2.3 Transfer Timeline: Memory to Memory 16

3.1 Block Partitioned Destination Array A_g 32
3.2 Column Partitioned Source Array B_g 32
3.3 Array Assignment $A_g = B_g$ 33
3.4 Types & Functions for Manipulating Node Addresses 35
3.5 Initialization for Array Partitioning & Chunking 35
3.6 Top Level Code: Blocking MP 36
3.7 $A_g = B_g$ on Source: Blocking MP 37
3.8 $A_g = B_g$ on Destination: Blocking MP 38
3.9 Top Level Code: Nonblocking MP 39
3.10 $A_g = B_g$ on Source (I): Nonblocking MP 40
3.11 $A_g = B_g$ on Source (II): Nonblocking MP 41
3.12 $A_g = B_g$ on Destination: Nonblocking MP 42
3.13 Top Level Code: Active Message 45
3.14 $A_g = B_g$ on Source (I): Active Message 45
3.15 $A_g = B_g$ on Source (II): Active Message 46
3.16 $A_g = B_g$ on Destination: Active Message 47

4.1 Top Level Code: C-Eli 55

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.2 $A_g = B_g$ on Source: C-Eli 56
4.3 $A_g = B_g$ on Destination: C-Eli 56
4.4 Synchronous Communication in C-Eli 62
4.5 Fuzzy Barrier in C-Eli 63
4.6 Fuzzy Global Reduction in C-Eli 65

5.1 Compiled fiber 68
5.2 Fiber scheduling 69
5.3 General Transfer Protocol 72

6.1 Supporting Variables & Functions 89
6.2 The P_k^j Transformation 90
6.3 QR Factorization: Blocking MP 91
6.4 QR Factorization: C-Eli 93

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 1

Introduction

1.1 Motivation

Massively Parallel Processing (MPP) architectures are a reality. The interesting questions regarding these architectures are no longer whether they are practical or whether specific problems can be efficiently parallelized on them. Real problems exist which are too large to be practically attempted on other classes of architectures. The challenge now is to provide support for programming these machines and to bring down the costs of software development.

At this time, the majority of applications on these architectures are written for the Single Program Multiple Data (SPMD) (or in some cases, the more general Multiple Program Multiple Data (MPMD)) paradigm. Such programs are typically written in a sequential programming language, such as C or Fortran, augmented with communication calls for data transfer and coordination between processors.

Traditionally, explicit MPMD programming has been accomplished by using Message Passing (MP) primitives of the **send/receive** type in conjunction with simple synchronization operations. It is now widely recognized that such primitives, although quite portable, do not map well on to the hardware architecture that is typical of modern MPP machines.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Greater efficiency may be obtained by using the Active Message (AM) primitives, which correspond very closely to the functionality provided by MPP hardware. Not surprisingly, however, these primitives are quite low-level. They can be viewed as the MPP equivalent of a portable assembly language.

This dissertation examines the requirements for the efficient, expressive, safe and portable programming of MPP architectures within the MPMD paradigm. Efficiency is viewed from the perspective of total execution latency, i.e. wall-clock time. Expressiveness relates to ease of specification of a desired computation. Safety refers to the robustness of the programming paradigm and its susceptibility to error. Portability is taken to mean the ability to transfer applications between MPP architectures, without significant change to the performance characteristics, by simply recompiling the source code.

We find that existing transfer paradigms involve a tradeoff between efficiency and expressiveness. Expressive paradigms tend to have sub-optimal performance characteristics. Conversely, the lower-level, less expressive, paradigms allow better performance. The performance improvements arise chiefly from the elimination of buffer copying and the overlapping of data transfer and computation.

We recommend a fundamentally different approach to explicit MPMD programming: language-based constructs for data transfer and coordination. We define a new program composition mechanism, the fiber, and supporting constructs for binding and sequence control. These constructs are encapsulated in a language extension, Eli, which can be implemented on top of standard sequential programming languages. The Eli extensions permit the high-level specification of asynchronous execution units and the structured transfer of data and control between them.

Such an approach has several benefits. It allows expressiveness without loss of performance. At the same time, embedding the constructs in a language forces the use of compiler technology to generate low-level transfer operations. This makes it possible to apply optimizations which would either be impossible to hand-code or would be error-prone and non-portable if so coded. A language mechanism also has

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

superior safety and portability characteristics.

The characteristics of such a language mechanism make it a suitable MPMD target code representation for a High Performance Language (HPL) compiler.

1.2 Overview of Present Work

Chapters 2 and 3 examine transfer costs and coordination mechanisms in detail. They make the case for a language-based data and control transfer mechanism and provide guidelines for the design of such a mechanism.

Chapters 4 and 5 present the fiber program composition mechanism for integrated data transfer and control coordination.

Chapter 6 shows the utility of the fiber construct by applying it to the specification of a real MPMD program.

1.2.1 Data Transfer Costs

Chapter 2 takes a close look at the costs which arise from inter-node data transfer and examines ways in which these might be reduced.

We begin by outlining the system assumptions underlying our data transfer model, specifically the MPP architecture and operating system abstractions and the programming paradigm. This is followed by a description of the sequence of operations required to perform a data transfer.

The execution model which is thus developed permits us to perform a detailed analysis of the cost structure of data transfer mechanisms. We identify only those costs which result in execution latency. These costs are grouped into **Pure Transfer** costs which comprise the minimal cost of a data transfer, **Protocol Overhead** costs which are specific to particular transfer mechanisms, and **Transfer Related** costs which arise indirectly from the transfer process. Experimental results from Thinking Machines Corporation's CM5 and Intel's DELTA architecture provide an indication

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

of the relative magnitude of these costs.

The remainder of the chapter considers ways of reducing or eliminating individual cost components. We find that the elimination of buffer copying and the implementation of efficient interrupt-driven data transfer require appropriately defined transfer semantics. Specification of overlapped computation requires a mechanism to support asynchronous execution. We identify specific optimizations which can reduce costs, such as buffer alignment, instruction scheduling, smart buffer management and protocol customization.

Taken together, these cost reduction techniques indicate the need for a compilable data transfer mechanism in conjunction with appropriately defined semantics to support asynchronous execution.

1.2.2 Transfer Coordination

Chapter 3 makes the case for a language mechanism which integrates data transfer with asynchronous execution.

We first introduce an analytical framework called I-BAD-C to examine the safety and coordination characteristics of transfer protocols. We then examine several existing transfer mechanisms in the context of a simple but realistic example. A study of the code and I-BAD-C characteristics provides insight into the weak points of these mechanisms. It reveals the source of the expressiveness/efficiency tradeoff and identifies some semantic pitfalls which permit non-robust code.

The complexity of expressing overlapped computation is seen to arise from the need to specify the definition and invocation of continuations which encapsulate asynchronous computations, and the need to coordinate such invocations with data transfers. In the absence of appropriate language support, these requirements result in convoluted code.

There are basically two semantic loopholes in existing transfer mechanisms, the possibility of inadvertent overloading of transfer identifiers, and the undisciplined

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

extents of these identifiers.

These conclusions point in the direction of a program composition mechanism which permits the construction and invocation of continuation structures for asynchronous execution coupled with integrated support for inter-node data transfers.

1.2.3 The Fiber Abstraction

Chapter 4 presents the Eli extensions. The fiber execution model is introduced. This is followed by a description of the syntax and semantics of the fiber mechanism. The concepts are illustrated in the context of the example used in Chapter 3. The Eli implementation of several common MPMD idioms display the expressiveness of the fiber construct.

The Fiber is an abstraction mechanism analogous to the procedure. Unlike a procedure, a fiber may be asynchronously invoked. There are specific statements, called sequencers, for controlling the invocation and coordination of fiber execution. Arguments may be passed to a fiber invocation in a manner analogous to procedure call. Each fiber invocation corresponds to an asynchronous execution continuation.

Data transfer is integrated into the fiber model in a straightforward way, by means of a sequencer for inter-node fiber invocation. The arguments to this sequencer comprise the data to be transferred. Coordination with transfers is achieved by means of other sequencers. A novel parameter binding mechanism is introduced to permit the specification of unbuffered transfers.

The semantics of inter-node fiber invocation are carefully defined to avoid the pitfalls identified in Chapter 3.

1.2.4 Fiber Implementation

The fiber mechanism was designed with efficient implementation as a design constraint. Chapter 5 demonstrates the efficiency of the fiber constructs.

The general implementation methodology for fiber definition and invocation is

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

described. We then go into the details of compiling inter-node invocations into lower-level data transfer primitives. We show that, in the general case, performance of coordination and transfer are both close to optimal for our MPP computational model.

We then describe the optimizations that may be performed on Eli code. In principle, fiber semantics permit the application of all the optimizations that are identified in Chapter 2.

The current C-Eli implementation for the CM5 is described. We present performance results of this implementation. The performance of general case transfer is seen to be comparable to existing Message Passing transfer costs. Optimization of special cases is shown to result in an order of magnitude improvement. The cost of fiber invocation and coordination is small.

1.2.5 Fiber Application

The utility of the fiber constructs in improving the wall-clock time of real MPMD codes is demonstrated in Chapter 6.

A fairly typical MPMD code, QR Factorization, is used as the test bed. We describe the synchronous algorithm used by our sample program. The Blocking MP implementation of the code is then presented.

It is shown how a relatively straightforward transformation to the fiber mechanism has the potential to improve the performance of the program. Experimental results confirm this: the fiber constructs decrease execution time by $\approx$ 25%–30%.

We also show how the properties of the fiber construct make it a suitable target language for an HPL compiler.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

1.3 Connections to Related Work

1.3.1 Cost Model

The data transfer model developed in Section 2.2 resembles the more abstract LogP model described in [Culler *et al.* 93]. The detailed breakdown of transfer costs presented in Section 2.3, which includes specific system overheads present in actual implementations is completely new.

1.3.2 Transfer Protocols & Asynchronous Execution

There has been relatively little work done on integrating MPMD data transfer and asynchronous execution.

The standard Message Passing transfer protocols ([MPIF 94], [TMC 93]), and more recent Active Message transfer primitives ([von Eicken *et al.* 92], [Liu & Culler 94], [Brewer & Kuszmaul 94]) allow low-level specification of asynchronous execution, but provide no threading functionality. Coordination has to be explicitly programmed by the user. As pointed out in [Bae & Ranka 94], this buys efficiency at the cost of greater program complexity.

In principle, a user-level thread mechanism as described in [Cooper & Draves 88], [Stein & Shah 92] or [Anderson *et al.* 92], could be used in conjunction with existing transfer mechanisms. However, such an approach would not integrate data transfer with thread coordination. The semantics for transfer coordination would be ad-hoc, thereby introducing new possibilities for non-robust usage. Furthermore, thread packages provide greater functionality and consequently, have larger overhead, than is required for data transfer. Finally, compiler optimization of data transfer requires an integrated coordination mechanism whose semantics are known to the compiler.

In any case, the semantics and coordination issues identified in Chapter 3 are relevant to any attempt to integrate threaded functionality with data transfer. We anticipate that other such efforts will result in semantics substantially similar to those

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

of Eli.

Some recent work in operating systems such as [Bershad *et al.* 90] and [Draves *et al.* 91] is similar in spirit to the approach taken in this dissertation. The focus there is on optimizing data transfer costs between user and kernel address spaces, rather than between MPP nodes. Not surprisingly, the notion of execution continuations plays a part in these optimizations. Although similar in spirit, these approaches cannot be directly applied to the inter-node data transfer problem.

Some efforts have been made in the area of run time systems which provide functionality for task creation, migration and scheduling, for example [Kale 90], [Berryman *et al.* 90]. This functionality is too rich and expensive for our problem.

Several architectures provide hardware support for context switching and threads in conjunction with data transfer such as those described in [Whitby-Stevens 85], [Papadopoulos & Culler 90], [Dally *et al.* 87]. However, these architectures are specialized for specific language paradigms and not suited for MPMD programs.

The message driven threads model described in [Holm *et al.* 94] is similar to the Eli object code in some operational aspects. This model appears to be intended primarily as a run time system. There seem to be no explicit coordination primitives.

1.3.3 Language Semantics

The fiber mechanism was inspired in large measure by the coroutine construct.

Traditional coroutines are intended for the expression of interleaved execution ([Dahl & Hoare 72], [Marlin 80], [Gentleman 72], [Skordalakis & Papakonstantinou 78], [Haynes *et al.* 86]). They allow the construction and invocation of deterministic execution continuations. The intra-node fiber constructs may be considered an asynchronous extension to the coroutine mechanism.

Coroutine semantics have also been applied to the abstract expression of "true" parallelism ([Kahn 74], [Kahn & MacQueen 77], [Dennis 71], [de Bruin & Bohm 85]). There is no explicit mapping onto any concrete computer architecture. Furthermore,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

"inter-process" communication is performed by means of a *port* construct rather than the argument passing mechanism used in Eli.

Other language mechanisms for integrated transfer and control coordination such as those defined by [Hoare 78], [Chandy & Taylor 89] and [Chandy & Kesselman 92] do not fall into the MPMD framework.

1.3.4 HPL Compiler Target

Of late there has been a great deal of interest in developing High Performance Languages (HPFortran, HPC++) and their compilers for MPP machines.

As pointed out in [Fox *et al.*], there is much overlap between these languages in their support for parallelism. The report cogently argues for a common framework for supporting different program transformation and runtime components.

One part of such a framework is the intermediate or target representation of MPMD code within the HPL compiler. We believe that the Eli extensions form a suitable prototype for such a target language.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 2

Data Transfer Costs

In this chapter we take a close look at data transfer costs and means for reducing them.

In order to precisely define such costs, we have to specify the underlying execution model. The first section details the assumptions that are made about MPP architecture, operating system and programming model. The following section describes the data transfer model implied by the architecture.

The execution model is then used to decompose the cost of data transfer into its constituent parts in section three. This section also provides measurements of the cost components on two architectures, the CM5 and the DELTA.

The final section describes ways of optimizing data transfer by reducing or eliminating individual cost components. In the process, we make an argument for

- a compiled data transfer mechanism,
- a means for specifying asynchronous execution to overlap computation with data transfer.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

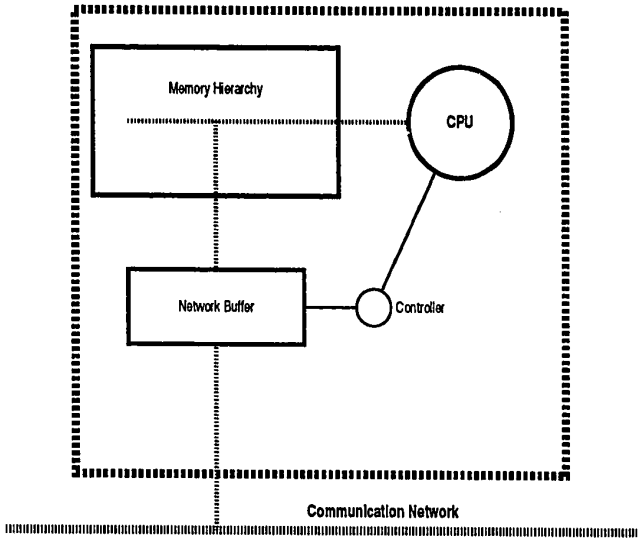

Figure 2.1: MPP Architectural Abstraction

2.1 Compute Model

2.1.1 MPP Architectural Abstraction

An MPP architecture is composed of a number N of homogeneous nodes. Each node consists of a CPU and its local memory hierarchy, i.e. registers, internal and external cache, RAM and possibly a storage device. The nodes are connected to each other by a scalable network interconnect. Each node accesses the network through a Network Buffer (NB), which has data paths into the memory hierarchy. We abstract away architecture specifics such as

- the precise data path between NB and memory,
- whether the NB has multiported access to the network or memory, and
- whether a DMA style network spooler exists.

Figure 2.1 is an abstract schematic of the control and data paths on a single node of such an architecture.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Individual machine nodes interact by transferring packets over the network. A transfer moves packets from an originating or source node to a terminal or destination node. Our model requires only the existence of a point-to-point transfer mechanism, no assumption is made about the presence of hardware supported multicast or combining primitives.

2.1.2 System Model

A physical MPP may be space partitioned into several virtual MPPs. In our model, each of these is treated as an independent MPP.

The MPP has a simple OS kernel which provides basic memory management, process scheduling and space partitioning support. In particular, the OS does not need to provide data transfer support. All data transfer is presumed to occur in user space by directly reading or writing the NB. If data transfer is interrupt driven, interrupts are assumed to be disabled during OS calls. Thus, there are never any kernel traps during communication which cross the boundary between the user and OS address spaces.

In any given time interval, an MPP is executing a single user process on every node. Time sharing is ignored in the model, because when present, it occurs at a much coarser time-scale than communication.

2.1.3 Programming Model

We restrict our attention to the MPMD programming model. The model is well suited for expressing data-parallel computations and has the benefit of having a straightforward mapping to the hardware architecture.

For purposes of exposition, we restrict ourselves to the slightly simpler SPMD model. We do not thereby lose generality, since the SPMD model extends easily with the addition of rules to specify transfer linkage between different executables. Nor do we lose relevance, for the majority of MPP applications have been developed on this

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

model.

In the SPMD model, a program consists of a single executable P. During execution, N copies of P are distributed onto the machine nodes $1..N$ of the MPP. Each copy has a private address space, which is mapped onto the local memory of the node it executes upon. The name space of the program is thus replicated on all nodes. Each name that has external linkage has the same binding on every node. There is a distinguishing variable, hereunder called `selfAddress`, which has a distinct constant value (generally in the interval $1..N$) in each copy. This variable identifies the copy of the executable in all data transfers. Control flow in each copy is independent of, and is assumed to occur asynchronously with respect to, control flow in other copies.

A data transfer is an operation which copies values from a source area in the address space of one node to a destination area in the address space of another. The specifics of data transfer and associated control flow are dependent on the particular transfer and coordination protocol(s) being used.

2.2 Data Transfer Model

2.2.1 Single Packet Transfer

At the hardware level, data transfer occurs in units of single packets of machine dependent size. The packet size is small enough that the entire packet can be held in the NB. A packet transfer moves a packet from the NB of the source node to the NB of the destination node.

The packet starts at the NB of the source node. The NB injects this package into the network. The packet moves through the network and is ejected into the NB on the destination node. The source processor is notified when injection is complete and the NB can be written. Likewise the destination processor is notified when a packet is ejected and the NB must be read. This notification is accomplished either by setting a status flag which can be tested, or by signaling a hardware interrupt. If the NB on

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

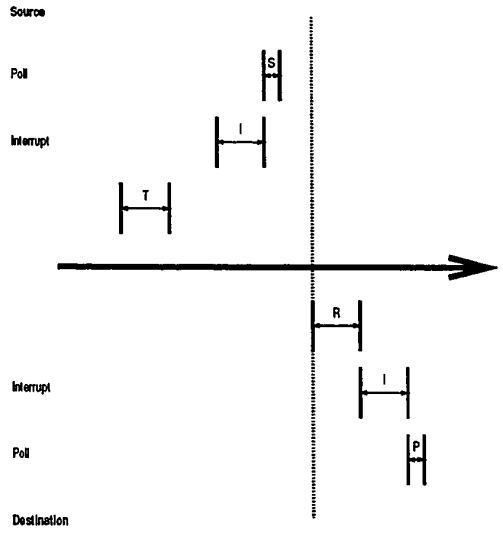

Figure 2.2: Transfer Timeline: NB to NB

the destination node is not read, network flow control logic comes into operation and eventually blocks all further traffic to the same destination, or between the specific pair of nodes.

Figure 2.2 is a timing diagram which shows the sequence of events on source and destination. At T the packet is injected into the network. When the injection is complete, one of two things happens:

- a status bit gets set to indicate NB availability. At some later point S, the processor queries this status bit.
- a hardware interrupt I is signaled to indicate NB availability. This interrupt must be serviced by the source processor.

At R the network ejects the packet into the NB. At this point, one of two scenarios unfold:

- a status bit gets set to indicate the presence of data in the NB. At some later

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

point P, the processor queries the status bit.

- a hardware interrupt is signaled to indicate the presence of data in the NB. The destination processor must service this interrupt.

Packet transfer is assumed to be reliable, i.e. the network hardware and firmware guarantee delivery to the destination once a packet is accepted.

Data transfer on a wide variety of existing parallel architectures, for example the CM5, DELTA and iWARP, conform to this model[1]. The chief differences between machines are in the areas of packet size, precise data paths, NB controller functionality and interconnect topology.

Thus far, we have not said anything about how data moves to (from) the NB on the source (destination) node, nor on how transfers larger than the packet size are to be decomposed into individual packet transfers. These are aspects of the data transfer process that are under software control. Different protocols can be used at this level. Each such protocol hides the lower-level details of packet transfers and presents the programmer with a data transfer paradigm.

2.2.2 Multi Packet Transfer

A general data transfer is one which moves a known but arbitrarily large amount of data from source to destination. Without loss of generality, we assume that the data is to be copied from a contiguous area of the source address space to a contiguous area of the destination address space. We assume that the location and extent of the source and destination areas are prespecified.

At minimum, a general transfer is a series of packet transfers from source node to destination node. Figure 2.3 is a timing diagram which displays this minimal transfer. As is readily seen by a comparison with Figure 2.2, it is composed of a number of

[1]On some machines, such as the DELTA, the node OS protects the network hardware from the user program. In principle, this protection can be disabled, leaving a transfer mechanism like the one described here.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

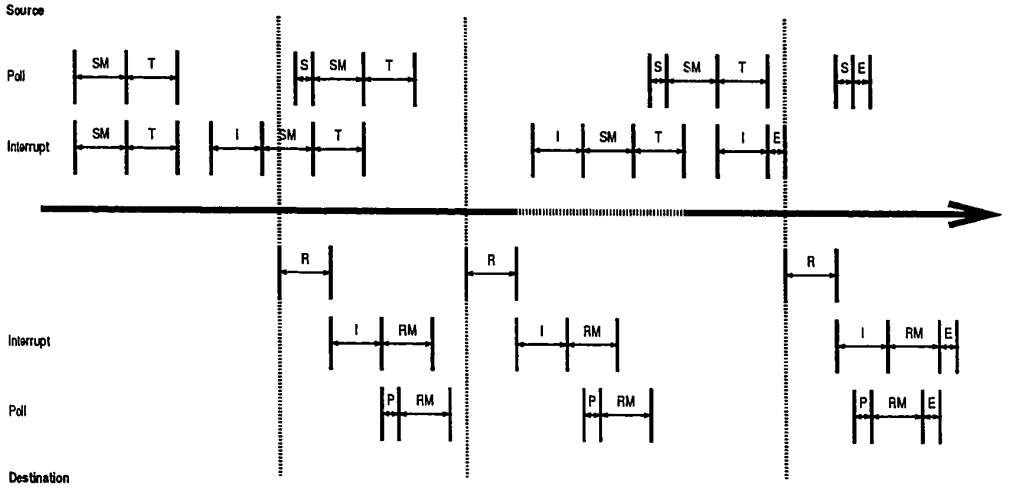

Figure 2.3: Transfer Timeline: Memory to Memory

packet transfers. The diagram shows the first two and the last packet transfers. On the source side, each transfer, ending at I or S, is followed by SM which copies the next packet of the contiguous area from the memory hierarchy to the NB. When the entire transfer is complete, end of transfer processing is carried out at E. On the destination side, each transfer, ending at I or P, is followed by RM which copies the packet from the NB into the next packet position in the destination area. If the relative ordering of individual packet transfers between source and destination is not guaranteed by the network, it will be necessary for each packet to carry information about its offset within the destination area. End-of-transfer processing occurs at E.

These operations do not necessarily have to be scheduled by the CPU. It is possible to have a controller which performs spooling between memory and NB, analogous to a DMA transfer. Some architectures have such controllers built in, for example the iWARP.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2.2.3 Abstract Transfer Primitives

The transfer mechanism described in the previous sections can be encapsulated by defining the following abstract operations.

- `AMPacket (<node>,<packet>,<handler-function>)` sends a packet from the NB on the source to the NB on the destination `<node>`. When the packet is received, `<handler-function>` is called, with access to the packet.

- `channelID (<start-address>,<handler-function>,<data-ptr>)` returns a channel identifier (a field that is generally one word wide). This channel identifier is used in conjunction with the `AMLong` primitive described next. When a message is sent to this channel identifier, the contents are stored starting at memory location `<start-address>`. When the message has been fully received, the function `<handler-function>` is invoked, with access to `<start-address>` and an auxiliary area `<data-ptr>` which can be used to store other transfer related information.

- `AMLong (<node>,<channel-id>,<start-address>,<size>)` sends a message to the `<channel-id>` on the destination `<node>`. The content of the message is the data block of size `<size>` starting at `<start-address>`.

`AMPacket` corresponds directly to single packet transfer with a function call on the destination. `AMLong` is simply a multi packet transfer with a function call at the end of the transfer. It can be implemented by a stateless, asynchronous, self-scheduled protocol. These operations can be regarded as an abstract definition of the Active Message transfer primitives.

2.3 Data Transfer Costs

The execution costs of an SPMD computation can be broadly divided into compute costs and transfer related costs. We are interested in the latter. However, we wish to

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

identify only those costs which contribute directly to *execution latency*, i.e. the costs that use up processor cycles on the source or destination nodes.

2.3.1 Cost Structure

The compute model and minimal general data transfer mechanism outlined in the previous sections reveal the following cost structure for data transfer.

Pure Transfer costs are the lower bound costs of moving data from the memory hierarchy of the source node to the memory hierarchy of the destination node.

- **Network Access.** CPU time taken to move data from (to) memory to (from) NB and inject (eject) data in to (out of) the interconnect, i.e. time interval $T_{\mathsf{SM}} + T_{\mathsf{T}}$ ($T_{\mathsf{RM}} + T_{\mathsf{R}}$), on the timeline in Figure 2.3. This cost may be visible even if the transfer is spooled by a network controller, since the corresponding memory bus cycles will be "stolen" from the CPU.

- **Network Service.** CPU time taken to poll or catch an interrupt. These are identified by the intervals T_{S}, T_{P} and T_{I} on the timeline.

- **Buffer Copy.** If data is buffered on source or destination, the CPU time required to copy data between user space and a protocol buffer.

The extra costs incurred by specific transfer protocols are grouped into **Protocol Overhead** costs.

- **Buffer Management.** CPU time required to manage—allocate and free—the data transfer buffers used by the protocol.

- **Buffer Binding.** CPU time required to bind a buffer to a specific data transfer operation, for example the cost of the `channelID` operation described in Section 2.2.3.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- **Protocol Data.** CPU time spent in transferring bookkeeping data specific to a protocol. The cost of each such transfer will be composed of its own **Pure Transfer** costs.

- **Protocol Control.** CPU time spent in executing protocol code other than that required for the categories above.

There are other costs which are indirectly connected to data transfer. Several of these arise from the need to coordinate computation with inter-node transfer. All these costs are grouped together as being **Transfer Related.**

- **Network Delay.** CPU time, if any, spent waiting for the NB to become available for the next transfer, i.e. the interval between T and I on the timeline. It is visible on the source node and is determined by the characteristics of the network and by the volume and pattern of message traffic.

- **Idle Time.** CPU time, if any, spent idling while a program is waiting for a data transfer to complete in order to proceed with computation. This is visible on the destination node and is determined by the overall control structure of the program, the ability to overlap computation with data transfer, and the load balance between different nodes.

- **Protocol Coordination.** CPU time spent executing protocol specific primitives which are used to coordinate data transfer with computation.

- **(Mis)Alignment Penalty.** The relative alignment of the source and destination areas in memory can have a high impact on transfer performance. This is due to the fact that modern processor architectures generally have more efficient memory access operations which operate on multiple words at a time. Although this is not a direct transfer cost its dramatic impact on transfer performance merits its inclusion in this list.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2.3.2 Magnitude of Cost Components

To obtain an indication of the relative magnitude of the cost components, we ran some test programs on the CM5 and the DELTA. The results of these experiments are summarized here.

CM5

The CM5 provides a compute model very close to the one defined in Section 2.1. It has a polling based Active Message implementation—CMAM—which provides a good approximation to the minimal transfer primitives described in Section 2.2.3. It also has a Message Passing implementation CMMD, which never buffers data. In addition, the CM5 also permits the user program direct access to a hardware microsecond timer. For these reasons, it is possible to make accurate measurements of a number of the cost components enumerated in Section 2.3.1.

Transfer timings for both Active Message and Synchronous MP were measured with nodes pre-synchronized and using a ring topology. Data was transferred from each node to its neighbor on the right. The cost figure is extracted by averaging over all nodes.

Table 2.1 shows the timings of a **Buffer Copy** on a single node and the combined **Network Access**, **Service** and **Delay** on source and destination. Timings were done with processors synchronized and on buffers which had the same alignment relative to a double-word boundary, i.e. the highest performance alignment. The buffer copy timings were obtained from a tightly coded assembler loop. CMAM was used to obtain the **Network** timings. It was not possible to measure accurately either the **Access** and **Delay** timings separately, or the components of **Access** on source and destination. Under the conditions of the experiment the **Delay** timings are expected to be at a minimum.

The costs of both **Buffer Copy** and **Network Access** go up if the source and target areas do not have identical alignments relative to a double-word boundary.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

	800 B	4000 B
Network: Access + Service + Delay	170 μs	760 μs
Buffer Copy	40 μs	200 μs

Table 2.1: **Pure Transfer:** CM5

	0 B	4000 B
Active Message	3.3 μs	763.3 μs
Synchronous MP	98.7 μs	1004.7 μs

Table 2.2: Inferring **Protocol Overhead:** CM5

For identical alignment relative to only a single word boundary the execution cost goes up by a factor of ≈ 1.5.

Row 2 in Table 2.2 shows timings for a Synchronous MP **send-receive** on the CM5. Row 1 is the minimum transfer Active Message cost. The 0 byte cost in row 1 is the cost of a single packet transfer. The difference between the timings on row 2 and row 1 correspond closely to **Protocol Overhead**[2] for Synchronous MP. The 0 byte message timing on row 2 must roughly be equal to a lower bound on **Protocol Overhead**. This implies a minimum protocol overhead of ≈ 95 μs.

Table 2.3 presents timings for **Buffer Management & Binding.** Static allocation uses a compile-time allocated buffer. Binned allocation specializes the allocation to use predetermined memory pools when the object type or size is known in advance. Dynamic allocation is `malloc`. The `malloc` timing was obtained by allocating a buffer immediately after it had been `free`'d. It therefore represents a lower bound on `malloc`. The **Buffer Binding** cost is the cost of a `CMAM_open_segment` operation, which is semantically similar to the `channelID` primitive defined in Section 2.2.3.

Table 2.4 displays several timings. The first is for the simplest generalized state

[2]This is not strictly true, since `CMMD` is implemented upon an alternative commercial implementation of Active Messages, `CMAML`.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Static Allocation	0 μs
Binned Allocation	$O(1)$ μs
Dynamic Allocation	$O(10)$ μs
Buffer Binding	10 μs

Table 2.3: **Buffer Management & Binding**: CM5

State Save	26 μs
Assembler Poll	1 μs
Nonblocking MP Poll	3.4 μs

Table 2.4: **Network Service / Protocol Coordination**: CM5

save operation on the CM5—a non-local jump in `C`. This can be taken as the lower-bound for a generalized interrupt servicing mechanism. The second timing is for a negative assembler-level poll of the network, using `CMAM_poll`. The last timing measures a negative Nonblocking MP poll for a pending message. This was timed when there were no outstanding message transfers.

Several points are noteworthy.

- Alignment choices, relative to word or double-word boundaries, of source and destination areas can change the transfer cost by 30%-50% or more.
- The cost of a single **Buffer Copy** is a significant fraction ($\approx$ 25%–30%) of the minimum inter-node transfer cost, i.e. **Network Access + Network Service.**
- The minimum **Protocol Overhead** for an optimized commercial protocol is an order of magnitude greater (95 μs) than the cost of a packet transfer (3.3 μs).
- The costs of static and binned buffer allocation are comparable to the cost of a packet transfer. Dynamic allocation, even in the best case, is more expensive.
- Generalized state save is expensive at best.
- Interrupt driven data transfer is potentially quite expensive because of the high cost of generalized state save. Although a single assembly-level polling operation

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

	0 B	**800 B**	**1600 B**	**2400 B**
Receive after **Send**	354 μs	537 μs	741 μs	1004 μs
Send after **Receive**	265 μs	426 μs	621 μs	769 μs
Force Message	268 μs	355 μs	529 μs	635 μs

Table 2.5: Inferring **Pure Transfer** & **Protocol Overhead**: DELTA

is inexpensive, frequent negative polls can add up.

- Control coordination based on state saving is potentially expensive.
- Even **Buffer Binding** has a nontrivial execution cost.

DELTA

The DELTA unfortunately does not allow such accurate measurements of different cost components. The NB on the DELTA is protected and can only be accessed by making Message Passing calls to the node OS `NX/M`. In principle, all transfers are buffered except when a *force* message type is used. `NX/M` uses binned buffer management internally.

The memory architecture of the DELTA is quite complex, and we did not have the expertise to write an optimized buffer copying routine. Our goal on the DELTA was to run an experiment which could be used to infer the magnitude of **Buffer Copy**, **Access** plus **Delay**, and **Protocol Overhead**.

Table 2.5 shows the cost of executing Nonblocking MP **send** and **receive** operations for different sized messages. The first row shows the timings when all **sends** preceded **receives**. In this situation, buffering may be expected to take place on both source and destination. The next row shows the timing when all **receives** preceded **sends**. In this case, buffering may be expected to occur on source but not on destination. The final case, using *force* message types prevents buffering on both source and destination. Note that each transfer is kernel mediated and therefore involves some number of kernel traps which cross the user-kernel address space boundary.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Nonblocking MP Poll	30 μs – 50 μs

Table 2.6: **Protocol Coordination**: DELTA

The timings on row 3 correspond roughly to **Network: Access + Delay + Service**[3] **+ Protocol Overhead**. Row 2 corresponds roughly to row 3 plus a single **Buffer Copy**. Row 1 corresponds to row 2 plus another **Buffer Copy**. The 0 byte message timing on row 3 is a rough indication of the lower bound on **Protocol Overhead**, i.e. ≈ 270 μs. Row 3 can also be used to infer the minimum transfer cost of **Network: Access + Delay + Service**, by subtracting column 1 (size 0 transfer) from the other columns.

Table 2.6 shows a negative Nonblocking MP poll for a pending message. This was timed when there were no outstanding message transfers. Since the poll is a kernel operation, it too involves a kernel trap.

Note the following.

- The inferred cost of a single **Buffer Copy** is a huge fraction (from $\approx 35\%$ to over 100%) of the minimum inferred inter-node transfer cost.
- The **Protocol Overhead** for an optimized commercial protocol is large.
- The cost of **Protocol Coordination** is relatively high, probably due to the kernel trap and concomitant state save.

2.4 Optimizing Data Transfer

Having identified the different components of the cost of data transfer and their relative magnitudes, we now prescribe ways in which these costs can be reduced. A comprehensive optimization plan must work at different levels: transfer and coordination semantics, code transformations and hardware support. This combination points to the need for language and compiler support for data transfer and control

[3]may include kernel traps.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

coordination.

2.4.1 Pure Transfer

Nothing can be done to reduce the cost of **Network Access**. It is completely determined by the hardware.

Network Service costs depend on the implementation of multi packet transfer.

Polling. The total cost is determined by the frequency of polling. It is not clear how best to optimize this cost.

Interrupt Driven. The cost of interrupt driven transfer is potentially very high, because of the high cost of a generalized state save. If interrupt handling can be made cheap, however, it is potentially much less expensive than polling.

An interrupt handler which is stateless (i.e. does not require the state of the application to be saved) and can execute in the user context, on the user stack, should be no more expensive than a function call. An example of such a handler on the CM5 would be one which could be compiled into a *leaf procedure* on the SPARC architecture. The semantics of the transfer protocol determine the kind of handler which can be used.

Buffer Copy can, in principle, be eliminated. However the need for buffer copying is determined by the semantics of the transfer protocol and its associated control coordination mechanism. Transfer semantics which eliminated buffer copying would result in a large performance increase for individual transfers.

2.4.2 Protocol Overhead

Buffer Management costs can be reduced by static or binned allocation of buffers. To perform either kind of allocation, the size of the buffers needs to be determined in advance, at compile time. This points to the need for compiler support of a data transfer protocol.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Buffer Binding, **Protocol Data** and **Protocol Control** costs are all determined by the specific protocol being used. These costs can be reduced by specializing the protocol to each specific transfer. Thus the protocol for a small amount of data would be different than that for a large amount of data and might not require any **Buffer Binding**. Any number of specialized protocols could be used. This kind of specialization is a perfect opportunity for compilation, providing more support for the idea of compiled data transfers.

2.4.3 Transfer Related

Both **Network Delay** and **Idle Time** can be reduced by *overlapping computation with transfer.* Different mechanisms must be used for the two, reflecting the difference in their origins.

Network Delay is observed on the source node of the transfer, for instance when an `AMLong` operation is in progress. Optimization requires that instructions be scheduled to "hide" the network delay. This can be done either at compile-time or at run-time.

Compile-Time. The code for `AMLong` needs to be transformed so that useful computation is hoisted into the "network delay slot" as it were. In general, this would involve unrolling the loop of packet transfers inside the `AMLong` operation. This could not even be attempted without a compiler.

Run-Time. This would require hardware support, either in the form of an interrupt driven transfer mechanism, or a network controller which was capable of memory to NB spooling. Either of these mechanisms would bring the **Network Delay** cost down to zero.

For either approach to be useful, there must be non-transfer related code which can be executed while a transfer is outstanding. This requires some mechanism to specify

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

code which can be executed asynchronously with the transfer. The compiler must know the semantics of this mechanism.

This requirement is equally necessary for reducing **Idle Time**. This cost can be reduced if there is any computation which can use the processor on the destination side till the transfer completes.

The specification of asynchronous execution is therefore critical to reducing these transfer related costs. The mechanism for asynchronous execution must permit coordination with other computations as well as with data transfers. The coordination with data transfers is generally provided by primitives for **Protocol Coordination**. If a unified mechanism were provided for coordination with computation as well as with data transfers, a compiler could be used to generate specialized code to reduce the cost or frequency of **Protocol Coordination**.

The last of the **Transfer Related** costs is the **Alignment Penalty**. The alignment of variables is generally generated by a compiler. A compiler that supported inter-node transfer could generate identical alignments for source and destination buffers. This would result in a large performance increase for individual transfers.

2.5 Semantics, Overlapped Computation & Compilation

The previous section underscores the need for several levels of support for efficient data transfer.

- The elimination of buffer copying and efficient interrupt-driven transfer require appropriate protocol semantics.
- Masking of network latency by overlapped computation requires a mechanism to specify asynchronous computation.
- Reduction of the misalignment penalty, instruction scheduling to reduce network delay, smart buffer management and protocol specialization all point to the need

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

for compiled data transfers.

In conjunction, these requirements seem to indicate a need for a compilable language mechanism with appropriately defined semantics which provides (preferably integrated) mechanisms for specifying data transfers and asynchronous computation.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 3

Transfer Coordination

This chapter makes the case for an integrated language mechanism to support the coordination of data transfer with asynchronous execution.

We begin by introducing an analytical framework called I-BAD-C (an acronym for Identification-Binding-Availability-Definition-Continuation) which allows us to deconstruct the coordination characteristics of arbitrary transfer protocols.

We then look at several existing transfer protocols in the context of a simple, but realistic, example. Viewing these protocols within the I-BAD-C framework allows us to isolate their deficiencies.

There are principally two problems which are thus identified:

- unsafe semantics for the inter-node coordination of transfers, and
- complicated control structure in the coordination of transfer with computation.

The final section summarizes the specific deficiencies of existing mechanisms and suggests why a language based mechanism might prove superior.

3.1 The I-BAD-C Framework

Recall from Section 2.1.3, that a data transfer is an operation which copies values from the address space of a source node to that of a destination node. Consider the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

informational requirements of such an operation.

The source and destination nodes must asynchronously identify the specific transfer operation that is intended. This implies that in principle each individual transfer operation must be unambiguously **identified** by a *transfer identifier* which the source and destination agree upon. In practice, this identifier may be split into two parts, an inter-node identifier and an intra-node identifier.

The next informational requirement is that the specific addresses on the source and destination node be specified. We say that the address(es) or location(s) are **bound** to the transfer operation.

It is also necessary to be able to detect when a transfer is complete and specify the computation that follows transfer completion.

Source. Upon completion of transfer, the address(es) involved are **available** to be written to. The computation that follows, the **availability continuation**, may write into this address.

Destination. On transfer completion, the value(s) in the address(es) are **defined**. The computation that follows, the **definition continuation**, may read from the address(es).

Finally, it may be desirable to specify the execution of some computation that overlaps asynchronously with the transfer. This computation, the **overlap continuation**, should not write to (read from) the address(es) bound to the transfer on the source (destination).

There are thus 3 coordination issues:

1. coordinating the transfer on source and destination by means of a transfer identifier,
2. coordinating the computation of the availability (definition) continuation with the end of transfer on the source (destination), and
3. coordinating the computation of the overlap continuation with the computation of the availability (definition) continuation on the source (destination).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Several questions arise naturally with regard to the semantics of transfer identification and location binding. How is a transfer identification declared? What serves as the transfer identifier? How is a binding declared?

It turns out that the most important semantic issue with regard to identification is one of extent or lifetime. The interactions between the extent of the data transfer (points in execution between the start and end of transfer on a node), the extent of the transfer identifier (points in execution where the identifier is accessible) and the extents of locations (points in execution where the location is well-defined) determine the robustness of the transfer mechanism and its susceptibility to programming errors.

3.2 Transfer Protocols

We will examine the following widely used transfer protocols.

Blocking Message Passing (MP). This protocol uses the **send** and **receive** operations on source and destination respectively. The **send** does not return until one of two things happens

1. The transfer is completed by the invocation of the corresponding **receive**, or
2. The data is copied into a protocol buffer.

The **receive** does not return until the transfer is complete

Nonblocking MP. An extension to MP to allow the specification of overlapped computation. The **send** and **receive** operations start the data transfer on source and destination, returning a **message-id** which identifies the transfer. The **poll** operation can test a **message-id** to determine if the transfer is complete. The **wait** operation blocks on a **message-id** until the transfer is complete.

Active Message. Strictly speaking, this is not a protocol. The Active Message paradigm provides low-level primitives which can be used to build different protocols.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

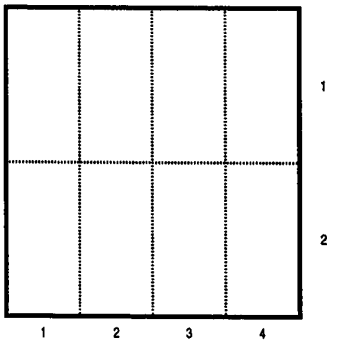

Figure 3.1: Block Partitioned Destination Array A_g

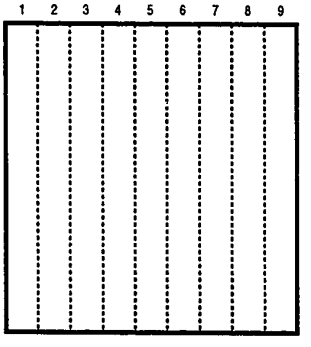

Figure 3.2: Column Partitioned Source Array B_g

The actual protocol is user specified. For our purposes, the abstract primitives specified in Section 2.2.3 can be considered the Active Message specification.

3.3 Example: Global Array Assignment

We now look at a simple task and examine its realization under different transfer mechanisms.

The problem is simple. We are given two $\mathtt{N} \times \mathtt{N}$ arrays, A_g (Figure 3.1) and B_g

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

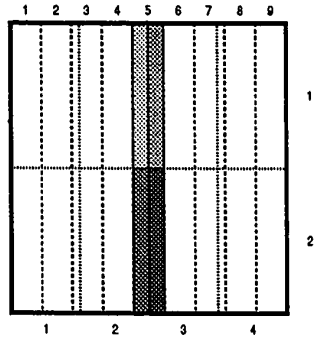

Figure 3.3: Array Assignment $A_g = B_g$

(Figure 3.2) , and we have to produce SPMD code to perform the array assignment $A_g = B_g$. Such an operation will come up in virtually any scientific computation, and could be written in a single line of HPFortran (with a couple of extra lines for declarations).

The only complexity lies in the partitioning of the arrays. B_g is column partitioned onto a linear topology of nodes, and A_g is block partitioned onto a mesh topology of nodes. There are assumed to be more columns in the linear topology than there are in the mesh topology.

Figure 3.3 shows the arrays superposed. Notice the shading of different parts of the array B_g which reside on the node with linear numbering #(5). The elements of B_g on #(5) must be assigned to elements of A_g on nodes with mesh numberings #(1,2), #(1,3), #(2,2) and #(2,3). The shades indicate the chunks that will be assigned to the same node.

The algorithm for assignment is simple, each node

1. computes the chunk boundaries for the node-local sub-array B of B_g, by rows (may be many) and columns (at most one), and
2. transmits the data to the appropriate destination node by rows, starting at row 1 (a row might be split into 2 chunks), and

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3. receives the rows that are being sent to it and assigns them to the appropriate elements of the node-local sub-array `A` of A_g.

3.4 Preliminaries

All code is presented in this dissertation is in `C` with a few minor "extensions" for the sake of brevity

- variables are allowed in an array declaration,
- variables are allowed in an aggregate initializer, and
- "//" is used to delimit end-of-line comments.

and @ at the beginning of a line of code are exogenous labels used to refer to specific portions of the code from the main text.

There are certain common functions and data that are used in all examples.

Data types for 1D and 2D node addresses and functions for translating between them are presented in Figure 3.4.

Figure 3.5 shows the initializations for variables which control the partitioning of the two arrays into their node-local sub-arrays, the destination node column(s) and the chunk splitting column if there is one.

3.5 Blocking MP

Our first example uses the blocking variant of Message Passing (MP). This protocol has the simplest coordination structure and consequently the code is the easiest to understand. All MP operations used in the code examples in this chapter conform to the syntax and semantics of the MPI standard.

Figure 3.6 shows the high-level structure of the implementation. The function `Source` implements the sending of array B_g on each node. The function `Destination` implements the reception of B_g and assignment to A_g on each node. The function `Rest` corresponds to the rest of the program.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
#define LinearUB 12     // # nodes in linear topology
#define RowUB 4         // # rows in mesh topology
#define ColumnUB 3      // # cols in mesh topology
typedef int Single;     // 1D Address for Node
typedef struct {        // 2D Address for Node
  int x; int y;
} Double;

Double S2D (Single s) {// Conversion from 1D Address to 2D Address
  Double d = {s / ColumnUB, s % ColumnUB};
  return d;
}
Single D2S (Double d) {// Conversion from 2D Address to 1D Address
  Single s = d.x * ColumnUB + d.y;
  return s;
}
Single myS = selfAddress ();
Double myD = S2D (myS);
```

Figure 3.4: Types & Functions for Manipulating Node Addresses

```
// Local Dope Vector for B_g
int mBlockSize = ceil (N / LinearUB);
int mLower = mBlockSize * (myS - 1);
int mUpper = myS != LinearUB ? (mBlockSize * myS) - 1 : N;
// Local Dope Vector for A_g
int xBlockSize = ceil (N / RowUB);
int xLower = xBlockSize * (myD.x - 1);
int xUpper = (myD.x != RowUB) ? (xBlockSize * myD.x) - 1 : N;
int yBlockSize = ceil (N / ColumnUB);
int yLower = yBlockSize * (myD.y - 1);
int yUpper = myD.y != ColumnUB ? (yBlockSize * myD.y) - 1 : N;

// Node-Local Section of Arrays A_g & B_g
float A [xUpper - xLower + 1][yUpper - yLower + 1]
float B [N][mUpper - mLower + 1];

// Mesh Column of Destination Node for First Transfer
int destColumn = mLower / yBlockSize + 1;

// Array Column at which Row is Split
int split = destColumn * yBlockSize - mLower;

// Assign Columns of Destinations for First and Second Transfers
Double dest1 = {0, destColumn}, dest2 = {0, destColumn + 1};
```

Figure 3.5: Initialization for Array Partitioning & Chunking

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
void Transfer () {
 Source ();
 Destination ();
 Rest ();
}
```

Figure 3.6: Top Level Code: Blocking MP

The code for `Source` is shown in Figure 3.7. The code loops through all rows in the node-local sub-array `B`, transferring each row to the appropriate destination node. There are two cases depending on whether the row is to be transferred to a single destination or split and transferred to 2 destinations. In either case, the appropriate part of the row (locations starting at `B[row][0]` and `B[row][split]`) is transferred as a block using a **send** with message type `LB`. Before transferring the row data, the code transfers the row and column indices of `A` on the destination at which the data is to be assigned. This is done with a gather **send** using message type `LBRC`.

Figure 3.8 shows the code for `Destination`. The code first computes the number of messages it will get, then enters a loop which iterates that number of times. Within the loop it blocks until it **receive**s a message of type `LBRC` specifying the row and column position. It then issues a blocking **receive** on message type `LB` which specifies the starting location `A[row][column]` at which the row data must be assigned. Note that the two messages cannot be combined into one scatter message, since the information from the first is used to determine the location for the second. Since there may be more than one node sending messages of type `LB`, the code must query for the sender of the `LBRC` message and ensure that the `LB` message is from the same sender.

3.5.1 I-BAD-C Characteristics

The semantics of identification for Blocking MP are simple. An identification is declared by the operation, either **send** or **receive**, which performs the transfer. The transfer is identified by a message type—a compile time determined constant integer

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
void Source () {
 int row, locations[2], sizes[2] = {sizeof (int)};
 locations [0] = &row; locations [1] = &mLower;// setup gather

@1 for ( row = 0; row < N; row++ ) {
    dest2.x = dest1.x = row / xBlockSize + 1;
    if ( split >= mBlockSize ) {// entire row sent to 1 destination
      // send row and column on destination with gather
      MPI_GSEND (MPI_BLOCKING, 0, D2S (dest1), LBRC, sizes, locations, 2);
      // send row data
#1a   MPI_SSEND (MPI_BLOCKING, &B[row][0], D2S (dest1), LB,
          sizeof (float), sizeof (float), mBlockSize);
    } else {                    // row split, sent to 2 destinations
      // send 1st part of row
      locations [1] = &(mLower + split);
      MPI_GSEND (MPI_BLOCKING, 0, D2S (dest1), LBRC, sizes, locations, 2);
#2    MPI_SSEND (MPI_BLOCKING, &B[row][0], D2S (dest1), LB,
          sizeof (float), sizeof (float), split);
      // send 2nd part of row
@2    locations [1] = &mLower;
      MPI_GSEND (MPI_BLOCKING, 0, D2S (dest2), LBRC, sizes, locations, 2);
#1b   MPI_SSEND (MPI_BLOCKING, &B[row][split], D2S (dest2), LB,
          sizeof (float), sizeof (float), mBlockSize - split);
    }}}
@1a
```

Figure 3.7: $A_g = B_g$ on Source: Blocking MP

value. The extent of the identifier is therefore unlimited, since constant integers may be used anywhere in a program. The extent of the transfer is from start to finish of the operation which declares it. There is no intra-node identifier.

A binding is similarly declared by the transfer operation. The extent of a binding is the same as the extent of the transfer.

On-node coordination issues are simply resolved. There is no overlap continuation. Since transfer operations block until completion, the coordination between the transfer and the availability (definition) continuation is straightforward: sequential composition. Thus, for the example in Figure 3.7, the availability continuation—labeled "@1"—of the **sends** labeled "#1a" and "#1b" is invoked by sequencing. Similarly the definition continuation—labeled "@3"—of the **receive** labeled "#3" in

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
void Destination () {
 int locations[2], sizes[2] = {sizeof (int)};
 int source = yLower / mBlockSize;

 // compute number of messages to be received
 for ( int numSources = 1;
       source < yUpper;
       source += mBlockSize, numSources++ );
 // setup scatter
 locations [0] = &row; locations [1] = &column;

@3 for ( int messages = 0;
        messages < numSources * (xUpper - xLower + 1);
        messages++ ) {
    // receive (with scatter) row and column at which to assign data
#4  MPI_GRECV (MPI_BLOCKING, 0, ANYNODE, LBRC, sizes, locations, 2);
@4  int source = MPI_INFOS ();
    // compute local row & column
    row = row % xBlockSize; column = column % yBlockSize;
    // receive data
#3  MPI_SRECV (MPI_BLOCKING, &A[row][column], source, LB,
        sizeof (float), sizeof (float), MAXSIZE);
  }}
@3a
```

Figure 3.8: $A_g = B_g$ on Destination: Blocking MP

Figure 3.8 is invoked by sequencing.

For all MP variants, the mechanism for coordination of inter-node identification is incomplete and error-prone.

1. Since a transfer is identified by an executable statement, it is possible for a transfer to be initiated which is never identified (i.e. declared) on the other side.

2. It is possible for **sends** and **receives** to become mismatched if the same transfer identifier is used for more than one **send** (or **receive**) in the code.

3. For the same reason it is difficult to create MP libraries. The identifications in the library can conflict with those in the application code if transfer identifiers are inadvertently duplicated.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
int row, sndmsgids[2], first, reusePossible = 0;
int slocations[2], ssizes[2] = {sizeof (int)};
int rcvmsgid, totalTransfers, messages, useEnabled = 0;
int dlocations[2], dsizes[2] = {sizeof (int)};

void Transfer () {
 SourceStart ();                    // start sending
 DestinationStart ();               // start receiving
 OverlappedComputation ();          // overlap
 while ( ! useEnabled               // have all rows been received?
         || ! reusePossible ) {     // have all rows been sent?
   if ( MPI_STATS (rcvmsgid) )
     DestinationNextMessage ();
   if  ( MPI_STATS (sndmsgids [0])
      && MPI_STATS (sndmsgids [1]) )
     SourceNextMessage ();
 }
 Rest ();
}
void OverlappedComputation () {
 ........;
 while ( MPI_STATS (rcvmsgid) )         // Receive Completed?
   DestinationNextMessage ();
 ........;
 while ( MPI_STATS (sndmsgids [0])
      && MPI_STATS (sndmsgids [1]) )    // Send Completed?
   SourceNextMessage ();
 ........;
}
```

Figure 3.9: Top Level Code: Nonblocking MP

3.6 Nonblocking MP

The performance of the simple Blocking MP realization described in Section 3.5 can potentially be improved upon. Two areas of inefficiency exist.

1. Buffer copying may take place on source, destination, or both.
2. The execution of the program can block at each **send**[1] and **receive**, thereby wasting processor cycles which could be used by overlapped computation.

Nonblocking MP operations are intended for use in situations such as these.

[1]Even with a buffer copying implementation, a **send** can block for an arbitrary length of time when the buffer is full due to a backed up network connection.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
void SourceStart () {
 slocations [0] = &row; slocations [1] = &mLower;
 row = 0; first = 0;
 dest2.x = dest1.x = row / xBlockSize + 1;
 if ( split >= mBlockSize ) {
   sndmsgids [0] = MPI_GSEND (MPI_NONBLOCKING, 0, D2S (dest1), LBRC,
       sizes, slocations, 2);
   sndmsgids [1] = MPI_SSEND (MPI_NONBLOCKING, &B[row][0], D2S (dest1), LB,
       sizeof (float), sizeof (float), mBlockSize);
 } else {
   slocations [1] = &(mLower + split);
   sndmsgids [0] = MPI_GSEND (MPI_NONBLOCKING, 0, D2S (dest1), LBRC,
       sizes, slocations, 2);
   sndmsgids [1] = MPI_SSEND (MPI_NONBLOCKING, &B[row][0], D2S (dest1), LB,
       sizeof (float), sizeof (float), split);
   first = 1;
 }}}
```

Figure 3.10: $A_g = B_g$ on Source (I): Nonblocking MP

The Blocking MP implementation can be transformed to one which uses Nonblocking MP. The basic idea is to replace a Blocking MP operation with its nonblocking equivalent. The code can then perform other possibly unrelated computation, periodically testing to see if the outstanding operation has completed. We assume that the application allows the possibility of such unrelated computation.

Figure 3.9 shows the high level structure of this implementation. The `Source` function has been replaced by 2 functions, `SourceStart` and `SourceNextMessage`. Likewise, the `Destination` function has been replaced by `DestinationStart` and `DestinationNextMessage`. The computation begins by starting off the first source **sends** of type `LBRC` and `LB` and the first destination **receive** of type `LBRC`. The function `OverlappedComputation` is then called to perform other useful computation (left unspecified as ellipses), periodically testing to see if the last outstanding **send** or **receive** operation has completed, and starting a new transfer if it has. At the end of this function, the computation loops until all messages have been **send** and **received**.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
void SourceNextMessage () {
 if ( ! reusePossible ) {
   if ( first ) {
     slocations [1] = &mLower;
     sndmsgids [0] = MPI_GSEND (MPI_BLOCKING, 0, D2S (dest2), LBRC,
         sizes, slocations, 2);
     sndmsgids [1] = MPI_SSEND (MPI_BLOCKING, &B[row][split], D2S (dest2), LB,
         sizeof (float), sizeof (float), mBlockSize - split);
     first = 0;
   } else {
     if ( ++row < N ) {
       dest2.x = dest1.x = row / xBlockSize + 1;
       if ( split >= mBlockSize ) {
         sndmsgids [0] = MPI_GSEND (MPI_NONBLOCKING, 0, D2S (dest1), LBRC,
             sizes, slocations, 2);
         sndmsgids [1] = MPI_SSEND (MPI_NONBLOCKING, &B[row][0], D2S (dest1), LB,
             sizeof (float), sizeof (float), mBlockSize);
       } else {
         slocations [1] = &(mLower + split);
         sndmsgids [0] = MPI_GSEND (MPI_NONBLOCKING, 0, D2S (dest1), LBRC,
             sizes, slocations, 2);
         sndmsgids [1] = MPI_SSEND (MPI_NONBLOCKING, &B[row][0], D2S (dest1), LB,
             sizeof (float), sizeof (float), split);
         first = 1;
       }
     } else
       reusePossible = 1;
   }}}
```

Figure 3.11: $A_g = B_g$ on Source (II): Nonblocking MP

The essence of the transformation to Nonblocking MP is to expose the continuation after the call to the Blocking MP operation.

This is easier to see in the somewhat simpler code for the destination node. Figure 3.12 shows the 2 functions `DestinationStart` and `DestinationNextMessage`. `DestinationStart` performs the initialization up to the first **receive** of type `LBRC` in `Destination` from Figure 3.8. `DestinationNextMessage` performs all subsequent **receives**. It represents the continuation labeled "@4" in the code for `Destination`. The continuation labeled "@3a", which follows all **receives**, is not replaced by any specific code. Rather, the status variable `useEnabled` is set when the continuation can

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
void DestinationStart () {
 int source = yLower / mBlockSize;
 for ( int numSources = 1;
       source < yUpper;
       source += mBlockSize, numSources++ );
 totalTransfers = numSources * (xUpper - xLower + 1);
 messages = 0;

 dlocations [0] = &row; dlocations [1] = &column;
 // Post Receive for First Row & Column Binding
 msgid = MPI_GRECV (MPI_NONBLOCKING, 0, ANYNODE, LBRC, sizes, dlocations, 2);
}
void DestinationNextMessage () {
 if ( ! useEnabled ) {
   int source = MPI_INFOS ();
   row = row % xBlockSize; column = column % yBlockSize;
   // Receive Row Data
   MPI_SRECV (MPI_BLOCKING, &A[row][column], source, LB,
       sizeof (float), sizeof (float), MAXSIZE);
   if ( ++messages < totalTransfers )
     // Post Receive for Next Row & Column Binding
     msgid = MPI_GRECV (MPI_NONBLOCKING, 0, ANYNODE, LBRC, sizes, dlocations, 2);
   else
     useEnabled = 1;
 }}
```

Figure 3.12: $A_g = B_g$ on Destination: Nonblocking MP

be invoked. This is the variable that is tested to see if all **receives** have completed.

A similar transformation has been carried out on the `Source` function. This code, displayed in Figure 3.10 and Figure 3.11, is more complicated since there are now 2 continuations of interest—those labeled "@1" and "@2". `SourceNextMessage` represents the continuations labeled "@1" and "@2", using the status variable `first` to distinguish between the two. The status variable `reusePossible` is used in proxy for continuation "@1a". It is tested for completion of all **sends**.

In order to restrict attention to the transformation required to achieve nonblocking code, we have kept the example simple. In the process, we have not taken the conversion to nonblocking code as far as is possible. For instance, we have retained the sequentialization of successive row **sends** and **receives** to prevent proliferation

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

of **message-ids** to be tested. We have also left the **receive** operation of type LB as a blocking receive which saves us the complexity of having to expose the additional continuation "@4".

Although the transformation methodology is fairly easy to understand, the resulting code is not. In general, code that is written using Nonblocking MP suffers from the problem of convoluted control flow.

3.6.1 I-BAD-C Characteristics

An identification in Nonblocking MP is declared by the operation, either **send** or **receive**, which starts the transfer operation. The inter-node identifier, as in all MP operations, is a compile time constant integer message type. Thus the extent of the inter-node identifier is unlimited. There is also, an intra-node identifier—the **message-id**. The extent of this identifier is the extent of the variable holding the **message-id**.

A binding is declared by the start-of-transfer operation. Its extent is the extent of the transfer.

These semantics lead to additional complexity, since the extent of the transfer is no longer a sub-interval of the extent of the location bound to the transfer. Additionally, there is now the extent of the intra-node identifier, which is a variable. Thus for instance

- control can exit the scope and therefore extent of the location bound to a transfer, *before* the transfer is complete, or
- the variable containing the **message-id** identifying the transfer can have greater extent than the transfer itself and can therefore be referenced *after* the transfer is complete.

On-node coordination is nontrivial. As the example shows, control flow is convoluted. This happens for several reasons:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

1. Coordination between the overlap continuation and the data transfer must be explicitly specified. This entails calls to the **poll** operation and explicit invocation of the appropriate continuation, e.g. the call to `SourceNextMessage` in `OverlappedComputation`.

2. Coordination between the end of transfer and the availability (definition) continuation must also be explicitly specified. This entails calls to a **poll** operation, tests of global status variables and the invocation of the appropriate continuation, e.g. the call to `DestinationNextMessage` in `Transfer`.

3. The only mechanism available for representing general continuations is the procedure. This makes construction of the appropriate continuation quite tedious, involving as it does, the exposing of layers of sequenced continuations, e.g. the structure of `SourceNextMessage`.

Coordination can be further complicated by the fact that message order between a source-destination pair is not preserved.

3.7 Active Message

The Active Message (AM) paradigm offers another alternative to improving performance over that of Blocking MP. This section presents the AM realization of our example. We use the syntax of the CM5 implementation of AM—`CMAM`, although we use the semantics of the primitive presented in Section 2.2.3, ignoring certain technical restriction placed on the CM5 implementation.

Figure 3.13 shows the high level structure of our AM realization. This code is structurally similar to the code for Nonblocking MP. The major difference lies in the fact that there is no explicit test to check for end of successive outstanding transfers, since a function call is automatically scheduled at the end of a transfer. Instead one needs to test only the status variable which indicates the end of all transfers and

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
int row, sndmsgids[2], split, first, reusePossible = 0;
int slocations[2], ssizes[2] = {sizeof (int)};
int rcvmsgid, totalTransfers, messages, useEnabled = 0;
int dlocations[2], dsizes[2] = {sizeof (int)};

void Transfer () {
  DestinationStart ();            // start receiving
  ........;
  SourceStart ();                 // start sending
  OverlappedComputation ();       // overlap
  while ( ! useEnabled            // have all rows been received?
          || ! reusePossible)     // have all rows been sent?
  Rest ();
}
void OverlappedComputation () {
  ........;
}
```

Figure 3.13: Top Level Code: Active Message

```
void SourceStart () {
  row = 0; first = 0;
  dest2.x = dest1.x = row / xBlockSize + 1;
  if ( split >= mBlockSize )
    CMAM_4 (D2S(dest1), &Destination(), myS, row, mLower,
        mBlockSize * sizeof (float));
  else
    CMAM_4 (D2S(dest1), &Destination(), myS, row, mLower, split * sizeof (float));
}
```

Figure 3.14: $A_g = B_g$ on Source (I): Active Message

the consequent validity of the appropriate continuation. Another difference is that `DestinationStart`, which initializes certain transfer variables, must be completed on all nodes before any node starts a transfer.

Each transfer occurs in 3 phases.

1. The source sends an active message corresponding to LBRC in the MP implementation to the destination node. This is done by `SourceStart` in Figure 3.14 and `SendComplete` in Figure 3.15.

2. On reception of the active message, the destination allocates a `segment` which

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
void SendComplete (int segment) {
  if ( first )
    CMAM_xfer (D2S (dest2), segment, &B[row][split],
        (mBlockSize - split) * sizeof (float));
  else if ( split >= mBlockSize )
    CMAM_xfer (D2S (dest1), segment, &B[row][0], mBlockSize * sizeof (float));
  else
    CMAM_xfer (D2S (dest1), segment, &B[row][0], split * sizeof (float));
  if ( ++row < N ) {
    dest2.x = dest1.x = row / xBlockSize + 1;
    if ( split >= mBlockSize )
      CMAM_4 (D2S(dest1), &Destination(), myS, row, mLower,
          mBlockSize * sizeof (float));
    else if ( first ) {
      CMAM_4 (D2S(dest1), &Destination(), myS, row, mLower,
          split * sizeof (float));
      first = 0;
    } else {
      CMAM_4 (D2S(dest2), &Destination(), myS, row, mLower,
          (mBlockSize - split) * sizeof (float));
      first = 1;
    }
  } else
    reusePossible = 1;
}
```

Figure 3.15: $A_g = B_g$ on Source (II): Active Message

points to the location in A into which the row has to be transferred. It then sends an AM to the source identifying the allocated segment. This is done by Destination in Figure 3.16.

3. The source sends an AM containing the data directly into the allocated segment. This is also done by SendComplete.

When all transfers are complete, the status variables useEnabled and reusePossible are set.

SendComplete is similar to SendNextMessage in structure and complexity. Taken together, Destination and ReceiveComplete provide roughly the same functionality as DestinationNextMessage. However the AM code is perhaps slightly simpler. This should not be surprising, since in effect we have created a custom protocol to match

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
void DestinationStart () {
  int source = yLower / mBlockSize;
  for ( int numSources = 1; source < yUpper; source += mBlockSize, numSources++ );
  totalTransfers = numSources * (xUpper - xLower + 1);
  messages = 0;
}
void Destination (int source, int row, int column, int bytecount) {
  int segment = CMAM_open_segment (&A[row % xBlockSize][column % yBlockSize],
      bytecount, &ReceiveComplete(), 0);
  CMAM_4 (source, &SendComplete, segment);
}
int ReceiveComplete () {
  if ( ++messages == totalTransfers )
    useEnabled = 1;
  return 0;
}
```

Figure 3.16: $A_g = B_g$ on Destination: Active Message

the structure of this example.

3.7.1 I-BAD-C Characteristics

In the Active Message paradigm each long data transfer `CMAM_xfer` is identified by a `segment`, which is a structure containing pointers to the data buffer and to a handler function. The identification is declared by the transfer operation. The extent of the identifier is the extent of the `segment` variable which holds it.

A binding is declared on the source by the `CMAM_xfer` operation. Its extent is the extent of the transfer. The binding on the destination is declared by the `CMAM_open_segment` operation which creates the `segment`.

In principle Active Messages suffer from the problems of potentially overlapping transfer, location and identifier extents which afflict the Nonblocking MP protocol. In practice, it may be slightly safer, since the `segment` is usually a computed value.

On-node coordination is slightly simpler than in the case of Nonblocking MP. Since the `segment` can be customized to include a partial continuation in its function address field, there does not have to be explicit coordination between the overlap continuation

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

and the data transfer. However, the other problems of coordination remain.

3.8 Complexity, Efficiency & Safety

This chapter establishes several points.

1. Overlapped computation requires asynchronous execution, which must be coordinated both with the transfer and with other computation.

2. Expressing this coordination with existing control structures results in complex, convoluted code. This is largely the result of having to encapsulate and invoke what are essentially asynchronous continuations with insufficient linguistic support.

3. The coordination mechanisms provided by existing transfer protocols are error-prone, due to the undisciplined extent of transfer identifiers.

These shortcomings of existing protocols seem to indicate the need for a language-based transfer mechanism. Some program composition mechanism, hence language extension, is clearly required to support the construction, invocation and coordination of the asynchronous continuation structures required for overlapped computation and transfer. This mechanism would have to be integrated with the data transfer mechanism. A language based mechanism could also enforce safety by semantically disallowing error-prone usages.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 4

The Fiber Abstraction

This chapter presents Eli, a set of language extensions which provide support for data transfer and asynchronous execution continuations.

The chapter begins with a summary of the design considerations—goals and constraints—for the Eli extensions. We then provide an overview of the fiber execution model, which is the centerpiece of the Eli design.

The syntax and semantics of fibers are summarized in the following sections. We go back to the global array assignment example of Section 3.3, providing an Eli realization. The resulting code is simpler and safer than the other realizations presented in Chapter 3. The I-BAD-C analysis, in the following section, provides insight into the reasons for this.

We then present Eli implementations of some common SPMD programming idioms. These examples provide further support for the expressive power of the fiber abstraction.

4.1 Design Goals & Constraints

We observed in Section 3.8 that overlapped computation results in asynchronous execution continuations, which are not directly supported by appropriate abstraction,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

program composition and coordination mechanisms. This motivated our search for language extensions which would provide such support. In addition, these extensions had to provide semantically safe coordination with inter-node data transfers.

The design of the data transfer mechanism was also important. The analysis of transfer cost optimization outlined in Section 2.4 indicates the benefits of compiler (and therefore language) support for data transfer. We wanted our transfer semantics to permit the full range of such optimizations. In particular, we wished to provide semantics which could eliminate buffer copying.

We also had a pragmatic constraint. We wanted to extend an existing sequential programming language. We therefore had to design constructs whose semantics and implementation were orthogonal to those of standard sequential constructs. Our choice of initial base language was based upon the twin criteria of widespread use and availability of compiler sources. We picked C.

4.2 Fiber Computation

Eli's approach to the specification of asynchronous computation is a new abstraction mechanism, the *fiber*.

4.2.1 Asynchronous Execution

A fiber is a new kind of execution module, analogous to procedures and functions. Like these more familiar modules, it is composed of declarations and executable statements which specify variables and computations upon them. Unlike procedures or functions, fibers have an asynchronous execution model.

A fiber is not called in the conventional sense, nor does it return. Rather, a fiber is *activate*d by fiber *invocation*. A fiber activation eventually gains control of the processor. At this point, it performs some computation and finally *deactivate*s itself. Each subsequent activation of the fiber resumes execution at the point of last

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

deactivation.

At any point in time, there may be several fiber activations. Only one of these activations has control of the processor. Corresponding to each fiber is a compile-time determined enumerated constant *fiber priority*. Fiber activations gain control of the processor in order of decreasing fiber priority. If there are two activations with equal fiber priority, the activations gain control in order of invocation.

Fibers are activated and deactivated by means of explicit *sequence control* statements or *sequencers*. These statements serve as the coordination mechanism for asynchronous execution on a single node.

A fiber invocation, like a procedure call, may include arguments to an activation. These arguments are available to the activation when it eventually gains control of the processor.

A fiber thus serves as an encapsulation mechanism for a group of related entry points or continuations. These continuations are composed via the standard composition mechanisms available in the underlying sequential programming language. Fiber invocation semantics then allow these continuations to be individually and asynchronously invoked.

4.2.2 Data Transfer

One of the major innovations of the Eli design is to use fiber invocation and the argument passing mechanism for inter-node data transfer. There is a specific invocation statement which can be used to activate a fiber on another node with arguments from the address space of the invoking node. The underlying intuition is that a data transfer serves the purpose of enabling a specific (definition) continuation on the destination node. This is exactly what fiber invocation accomplishes.

In this fashion, data transfer is integrated into a high level program composition mechanism. The coordination of data transfer with computation relies upon the same sequence control statements which are used for coordinating asynchronous

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

computations on a single node.

The argument passing semantics for fiber invocation also incorporate a novel binding mechanism that can be used to suppress buffer copying.

4.2.3 Usage Note

The motivating force for the fiber design was the simple and efficient specification of inter-node data transfer and coordination of overlapping computation. We expect it to be used in situations where asynchronous computation is used primarily to mask transfer latency. Although fibers could be used to specify arbitrary asynchronous parallel computation on a single processor, this usage was not a priority in our design considerations.

4.3 C-Eli

The language extensions required to implement fibers are called Eli. We chose to base the first Eli implementation on the `C` programming language. Consequently the abstract syntax of Eli, which is used throughout this dissertation, is based upon the syntax of `C`. A formal syntax for `C`-Eli may be found in Appendix A.

The next sections introduce the syntax and semantics of `C`-Eli. Figure 4.1, Figure 4.2 and Figure 4.3 show the `C`-Eli realization of the the data transfer example of Section 3.3. The code serves the dual purposes of introducing `C`-Eli and presenting a concrete example of its expressiveness.

4.3.1 `eli_main`

Just as a `C` program has the `main` entry point, a `C`-Eli program has the special fiber `eli_main`. Fiber computation begins at the first call to `eli_main`[1]. All other fibers

[1]It is the only time that a fiber is called rather than being invoked.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

are directly or indirectly invoked by `eli_main`. Eventually `eli_main` will finish its computation and return[2].

4.4 Fiber Definition

In C-Eli, fiber declaration and definition are analogous to function declaration and definition. C-Eli introduces a new type specifier `fiber`, which is used as the return type of fibers. It is similar to the `void` return type for functions.

Figure 4.2 is an example of a fiber definition. The body of a fiber definition is quite similar to that of a procedure definition. It consists of a number of variable declarations followed by executable statements. The scope of variables declared in a fiber body is the fiber body. However, the extent of variables is unlimited, i.e. they behave much like `static` variables in functions.

Note that `fiber` in Figure 4.2 is qualified by the keyword `source`. This indicates to the compiler that the fiber can perform an inter-node fiber invocation. Similarly the fiber definition in Figure 4.3 is qualified by the keyword `target`. This indicates to the compiler that the fiber will be the target of an inter-node fiber invocation. A fiber can be both a `source` and a `target`.

The fiber in Figure 4.3 is also qualified by the keyword `high`. The keywords `high`, `medium` and `low` specify the *priority* of fiber activation. The default priority of a fiber is `medium`.

4.4.1 Fiber Parameters

Fibers, like functions, are defined with formal parameters. The scope of a formal parameter is the fiber body. The extent of a parameter is a single activation of the fiber[3], i.e. between successive deactivations.

[2]It is the only fiber which "returns" like a procedure call.

[3]There is one exception, a fiber invocation with no arguments. In this situation, the arguments from the previous activation are used.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

C syntax and semantics have been modified for array and pointer parameters to target fibers. In C these can be used interchangeably in a formal parameter list. In C-Eli, pointers, fixed size arrays and variable size arrays are distinct when used as formal parameters. This change is useful for specifying block data transfers. A size of 0 indicates a variable length array, for example the parameter Vector of the fiber Destination in Figure 4.3 is a variable length array.

4.4.2 userbound Parameters & Transfer Binding Block

C-Eli introduces an innovative parameter binding mechanism for use with data transfers. In C-Eli certain parameters' bindings can be computed instead of being automatically generated by the compiler. This mechanism allows the direct specification of a user buffer on the destination node of a transfer. This means that a transfer can move data directly into the appropriate destination area without a buffer copy.

The qualifier userbound when applied to the parameter of a target fiber indicates that the parameter has a computed binding. The declaration of all non-userbound parameters must precede that of the first userbound parameter in the parameter list. The parameter Vector of fiber Destination in Figure 4.3 is userbound.

Every fiber that has one or more userbound parameters must also provide a so called *transfer binding block*. This is a block of C code delimited by "<" and ">", which appears between the parameter list and the fiber body. All non-userbound parameters are visible in the body of the transfer binding block. All userbound parameters are visible but have an lvalue type, rather than the parameter type. Each parameter declared userbound must be bound in the transfer binding block code. This is done by assigning an appropriately typed lvalue to the parameter. Figure 4.3 provides an example of a transfer binding block.

The transfer binding block for an activation is executed before the data transfer for the corresponding userbound parameter is initiated, and by implication, before the actual point of activation. References to a userbound parameter in the body

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
fiber Transfer () {
  q_continue Source ();
  q_continue Overlap ();
  continue Destination ();
}
low fiber Overlap () {
  ........;
  relinquish;
  ........;
  continue Rest ();
}
fiber Rest () {
  suspend;
  suspend;
  ........;
}
```

Figure 4.1: Top Level Code: C-Eli

of the fiber are automatically dereferenced to refer to the appropriate user specified buffer.

4.5 Fiber Invocation & Coordination

Fiber activation and deactivation are controlled by fiber sequence control statements or sequencers. There are several such statements for intra-node fiber invocation and one for inter-node invocation. Sequencers can only occur inside the body of a fiber, for example they may not occur inside function bodies. Control transfer to a fiber activation can occur only during the execution of a sequencer.

4.5.1 Intra-Node

The following sequencers are defined for intra-node fiber invocation and coordination:

- `continue <fiber> <arguments>`. This statement invokes the named `<fiber>` with the argument list `<arguments>`. The invoking fiber activation is deactivated and the invoked fiber is activated.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
source fiber Source () {
  for ( int row = 0; row < N; row++ ) {
    relinquish;
    int destRow = row / xBlockSize + 1;
    dest2.x = dest1.x = row / xBlockSize + 1;
    if ( split >= mBlockSize )
      remote_continue D2S (dest1)
        Destination (row, mLower, nocopy (B[row][0] @ mBlockSize));
    else {
      remote_continue D2S (dest2)
        Destination (row, mLower + split, nocopy (B[row][split] @ split));
      relinquish;
      remote_continue D2S (dest1)
        Destination (row, mLower, nocopy (B[row][0] @ mBlockSize - split));
    }
  }
  continue Rest ();
}
```

Figure 4.2: $A_g = B_g$ on Source: C-Eli

- `q_continue <fiber> <arguments>`. The named fiber `fiber` is activated with the argument list `<arguments>`. The invoking fiber activation remains activated. This can loosely be thought of as "forking" the fiber.

- `relinquish`. This sequencer has no effect if there is no higher priority fiber activation at the time of execution. If there is, control is transferred into it. The fiber activation that executes this sequencer remains activated.

```
target high fiber Destination (int row, int column, userbound float Vector [0]) <
  Vector = &A[row % xBlockSize][column % yBlockSize];
> {
  int source = yLower / mBlockSize;
  for ( int numSources = 1; source < yUpper; source += mBlockSize, numSources++ );
  for ( int messages = 0;
        messages < numSources * (xUpper - xLower + 1);
        messages++ )
    suspend;
  continue Rest ();
}
```

Figure 4.3: $A_g = B_g$ on Destination: C-Eli

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Note that a higher priority activation can only exist if a remote invocation created it since the last transfer of control.

The purpose of the `relinquish` operation is to prevent a fiber from monopolizing the processor while higher priority activations accumulate due to remote invocation. The presence of this sequencer should not change the semantics of correctly written fiber code.

- `suspend`. The fiber activation that executes this sequencer is deactivated.
- `reset`. This sequencer reinitializes the fiber. The next activation of the fiber will resume execution at the beginning of the fiber body.
- `exit`. Execution leaves the fiber body. Attempting to invoke the fiber again is an error. There is an implicit `exit` after the last statement of every fiber definition.

The execution of `continue`, `suspend`, `reset` and `exit` always leads to a transfer of control. `q_continue` and `relinquish` will lead to a control transfer only if an activation with priority higher than the priority of the currently executing fiber exists. When control transfer occurs, the fiber with the highest priority or (in the case of more than one fiber of equal priority) earliest invocation gains control of the processor.

When executed from a `source` fiber, the effect of the sequencers `continue`, `reset` and `exit` are delayed until all outstanding data transfers started from the fiber are completed. The delay serves to ensure that the transfer is completed before an availability continuation is invoked. A control transfer will still occur immediately if there are other previously activated fibers at the time.

4.5.2 Inter-Node

There is one sequencer meant specifically for inter-node fiber invocation. It can occur only inside the body of a `source` fiber.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- `remote_continue <node-#> <fiber> <arguments>`. This sequencer invokes the fiber named `<fiber>` with argument list `<arguments>` on the node numbered `<node-#>`. The invoking fiber activation remains activated. `<fiber>` must be declared a `target` fiber. The node on which the sequencer is executed is the source node, and `<node-#>` is the destination node.

  Note that the semantics allow the actual data transfer to take place asynchronously. The semantics only guarantee that the transfer will be finished before the execution of the next `continue`, `reset` or `exit` sequencer. The compiler has considerable leeway on the precise transfer protocol that is used.

  A `remote_continue` which specifies its own node number in `<node-#>` behaves identically to a `q_continue`.

4.5.3 Argument Passing

Argument passing in fiber invocation has almost the same semantics as in function call. Parameters are passed by value.

The only twist arises in the arguments passed to a `remote_continue` invocation. In order to suppress buffer copying on the source node, a new expression type is introduced. A `nocopy (<arg>)` expression when applied to an argument in a fiber invocation, indicates that the corresponding argument `<arg>` is not to be copied on the source node. A `nocopy` expression can only be substituted for a parameter that is declared to be `userbound`. Local variables of a `source` fiber may not be used in a `nocopy` expression.

There is also an operator `<num-elements> @ <start>` which is used to specify a start address `<start>` and number of elements `<num-elements>` of an array argument to a fiber invocation.

An example of the `nocopy` expression and `@` operator can be found in Figure 4.2.

The argument passing mechanism also provides an extra measure of safety for data transfers—type checking. A fiber must be invoked with either

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- exactly the same number and type of arguments as specified in its declaration, in which case the new arguments are used when control is transferred to it, or
- no arguments, in which case the old arguments are used when control is transferred to it.

The second case, of invocation with no arguments, is intended to allow

- the execution of initialization code which does not depend on the parameters to the fiber.
- coordination of local computation with the remote activation of a target fiber.

4.6 Array Assignment Example

Figure 4.1 shows the high level structure of the C-Eli code for the array assignment example.

The `Transfer` fiber invokes the `source` fiber `Source` and the fiber `Overlap`. Neither of these activations gains control since they do not have a higher priority than `Transfer`. Transfer then invokes the `Destination` fiber. At this point, the `Destination` activation has the highest priority of the three activations and gains control.

The first activation of the `Destination` fiber (Figure 4.3), which may arise from an inter-node invocation, rather than the intra-node invocation, computes the number of row transfers. The fiber then enters a loop in which it suspends that many times. Each time an inter-node invocation occurs, the transfer binding block code executes, using the row and column indices to set up the appropriate row address for the transfer. The data is transferred directly into the correct locations before the fiber is activated. Upon activation, the fiber immediately suspends if there are still rows to be transferred. When all rows have been received, it invokes the `Rest` fiber.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Whenever the `Destination` fiber first suspends, the activation of the `Source` fiber (Figure 4.2) gains control, since it has higher priority than the `Overlap` fiber. This activation starts off each of the row transfers individually. In between transfers, it periodically `relinquishes` in case there are `Destination` activations waiting to execute. It then invokes the `Rest` fiber.

Note that the fiber implementation allows the row and column information to be specified in the same transfer operation as the row data block.

At this point, the `Overlap` activation gains control of the processor, periodically relinquishing control to ensure that `Destination` activations execute. When the overlapped computation is done, it too invokes the `Rest` fiber.

The `Rest` fiber suspends two times at the beginning. This requires that it be invoked by all three fibers `Source`, `Destination` and `Overlap` before it can continue with useful computation. The `continue` sequencer in the `Source` fiber will only invoke `Rest` after all the transfers are completed. Thus, when `Rest` is invoked, all row transfers and overlapped computation are done.

4.7 I-BAD-C Characteristics

Remote fiber invocation has more disciplined I-BAD-C characteristics than any existing transfer mechanism.

An identification is declared by the `remote_continue` sequencer. On the source, the specific execution of the sequencer is itself the identifier and is unique. The extent of the identifier is the extent of the (asynchronous) transfer. On the destination, the specific invocation of the transfer binding block of `target` fiber is the identifier and is unique. The identifier has asynchronous extent, but ceases to exist at some point before the fiber activation is created.

On the source, locations are bound to the argument list of the `remote_continue` sequencer. On the destination, locations are bound to the parameter list of the invoked `target` fiber.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The dynamic extent of the identification does not cause problems in this case. On the source side, the extent of the bound location is always a super-interval of the extent of a `source` fiber[4], which is always a super-interval of the extent of any single `remote_continue` transfer. On the destination side, the extent of the bound location is always a super-interval of the extent of the transfer binding block of the `target` fiber.

The definition, availability and overlap continuations are all encapsulated in fibers. These continuations are invoked by the `continue`, `q_continue` and `remote_continue` sequencers. Coordination is performed by explicit execution of the `suspend` and `relinquish` sequencers. The resulting code, as seen in this example, is simple and structured.

4.8 Implementing Common Idioms

We now display several examples which use C-Eli to implement some common SPMD programming idioms.

4.8.1 Synchronous Communication

Figure 4.4 shows the C-Eli code which implements synchronous communication.

The fiber `start` begins the synchronous send by invoking the `source` fiber `send` with arguments that specify the destination node, start address and number of elements of the area to be `send`.

This `send` fiber remotely invokes the `recv` fiber with the contents of the source area. The use of the `nocopy` expression signals the compiler to suppress buffer copying on the source node. The fiber then invokes the availability continuation `sourceFinish` with the `continue` sequencer. The corresponding activation is delayed until the transfer is complete.

[4]except in the case of a fiber argument used in a `nocopy` expression. This case can be handled correctly by a compiler.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
double S[ 100 ]; double R[ 100 ];

fiber start () {
  ........;
  continue send (5, S, 100);
}
fiber destinationFinish () {
  ........;
}
target high fiber recv (userbound double Message [0]) <
  Message = & R;
> {
  continue destinationFinish ();
}
fiber sourceFinish () {
  ........;
}
source fiber send (int node, double * address, int size) {
  remote_continue node recv (nocopy (address @ size));
  continue sourceFinish ();
}
```

Figure 4.4: Synchronous Communication in C-Eli

The fiber `recv` on the destination receives the transfer and utilizes the transfer binding block to direct it into the appropriate user buffer. When the fiber is activated, it invokes the definition continuation `destinationFinish` via a `continue` sequencer.

This transfer does not perform buffer copying on source or destination. Execution of the availability continuation does not begin until the transfer is complete on the source. Execution of the definition continuation does not begin until transfer is complete on the destination. It therefore provides the functionality of synchronous communication.

4.8.2 Fuzzy Barrier

The next example in Figure 4.5 uses C-Eli to implement fuzzy barrier synchronization.

The integers from 0 to `numNodes` represent the nodes in the barrier. The fiber

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
int numNodes = 5;

fiber start () {
  ........;
  q_continue enterBarrier ();
  continue Overlap ();
}
fiber finish () {
  suspend;
  ........;
}
low fiber Overlap () {
  ........;
  relinquish;
  ........;
  continue finish ();
}
source fiber enterBarrier () {
  for ( int i = 0; i < numNodes; i++ )
    remote_continue i decrement ();
}
target high fiber decrement () {
  for ( int i = 1; i < numNodes; i++ )
    suspend;
  continue finish ();
}
```

Figure 4.5: Fuzzy Barrier in C-Eli

`start` enters the barrier by invoking the `source` fiber `enterBarrier`. It then invokes `Overlap` and deactivates itself.

The `enterBarrier` activation gains control next, since it has higher priority than `Overlap`. It loops over the node addresses of all nodes in the barrier. Within the loop `remote_continue` is used to invoke the `target` fiber `decrement` on each of these nodes.

When `enterBarrier` is done, the `Overlap` fiber gains control and performs overlapping computation. It periodically `relinquishes` control to permit remotely invoked `decrement` activations to gain control. At the end of this computation, it invokes `finish`.

The `decrement` fiber iterates `numNodes - 1` times. On each iteration, it `suspends`.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

When it has been invoked `numNode` times, i.e. when all node have entered the barrier, it invokes `finish` via the `continue` sequencer.

The `finish` fiber `suspends` on its first invocation. It therefore needs to be invoked twice, from `Overlap` and `decrement`, before it can perform useful computation.

This code provides a fuzzy barrier between `start` and `finish` with `Overlap` providing the overlapped computation.

4.8.3 Fuzzy Global Reduction

The example in Figure 4.6 implements fuzzy global sum reduction in C-Eli.

The structure of this computation is similar to that of the fuzzy barrier in Figure 4.5. The integers from 0 to `numNodes` represent the nodes participating in this reduction. The `start` fiber invokes the `source` fiber `startReduce` with the value `local` which is the local component of the reduction. It then invokes `Overlap` and deactivates itself.

`startReduce` remotely invokes the `target` fiber `reduce` with the local value as argument. The invocation is performed on all participating nodes.

The `reduce` fiber is invoked `numNodes` times with each of the local values from the other nodes. These are added to the variable `value`. When all reductions have been performed, the `finish` fiber is invoked with the resulting global value.

`Overlap` is invoked and operates in a manner similar to the previous example.

4.9 Fiber Expressiveness & Safety

The expressiveness of the fiber mechanism stems from the linguistic support for asynchronous execution continuations, the unique binding mechanism and the semantics of sequencers.

The fiber provides a natural way to specify the relative ordering of related continuations which are asynchronously interleaved with data transfers or other computa-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
int numNodes = 5;

fiber start () {
  double local;
  ........;
  q_continue startReduce (local);
  continue Overlap ();
}
fiber finish (double global) {
  suspend;
  ........;
}
low fiber Overlap () {
  ........;
  relinquish;
  ........;
  continue finish ();
}
source fiber startReduce (double l) {
  for ( int i = 0; i < numNodes; i++ )
    remote_continue i reduce (l);
}
target high fiber reduce (double l) {
  double value = 0;
  for ( int i = 1; i < numNodes; i++ ) {
    value += l;
    suspend;
  }
  value += l;
  continue finish (value);
}
```

Figure 4.6: Fuzzy Global Reduction in C-Eli

tions. The `continue`, `q_continue` and `suspend` sequencers, allow for the intra-node computation coordination. In conjunction with the `remote_continue` sequencer, they can be used to coordination computation with asynchronous data transfer.

The parameter binding mechanism made possible by `userbound` parameters and the transfer binding block permit the safe expression of transfers which are not buffered on the destination. The `nocopy` expression in conjunction with the semantics of sequencers in `source` fibers allow the safe expression of transfers which are not buffered on the source.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 5

Fiber Implementation

In this chapter we demonstrate that the fiber constructs can be efficiently implemented.

We begin by outlining our criteria of efficiency. We then describe the general compilation framework for fiber invocation, coordination and data transfer on general purpose hardware. An analysis of the execution costs reveals that they meet our criteria for efficiency. Next, we describe specific compiler optimizations which can be used to customize special cases to further reduce execution costs.

Finally, we describe the CM5 C-Eli implementation and present performance data. The cost figures confirm our performance projections. They also demonstrate the dramatic impact of compiler optimization on data transfer performance as well as fiber switching costs.

5.1 Implementation Challenge

Performance was a major criterion of the Eli design. The C-Eli constructs were designed to allow compilation into efficient code on standard hardware architectures.

The implementation challenge was threefold:

1. compile code for fast fiber invocation in the general case.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2. compile efficient data transfer code for the general case `remote_continue` sequencer.
3. use compilation technology to recognize and optimize special cases.

Our criteria for efficiency in the general case were:

Data Transfer. The absolute computational cost of data transfer be within an additive constant or small factor of the cost of the minimum transfer described in Section 2.2.2. On a comparative basis, it should perform no worse than existing transfer protocols.

Control Transfer. The cost of fiber invocation should be small, not more than some small constant number of machine cycles. In particular, it should not impose high additional overhead to the cost of a single packet transfer.

5.2 Invocation & Coordination

5.2.1 Fiber Compilation

Each fiber is compiled in a manner analogous to the compilation of a procedure. The compiler expands the fiber body into a block of assembly code. The local variables of the fiber are bound to a *frame*, similar to a procedure frame. Fiber parameters are bound to an *activation record.* The entry point to a fiber is a *fiber descriptor.*

Figure 5.1 is a pictorial representation of the most general form of the compiled code for a fiber. The fiber descriptor contains a pointer to the current fiber continuation. It also contains a pointer to the activation record of the current activation of the fiber.

Variables in the fiber body are dereferenced through the frame and the pointer to the activation record. `userbound` variables in the fiber body have an extra level of dereferencing through a field in the activation record. `userbound` variables in the transfer binding do not have this extra level of dereferencing.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

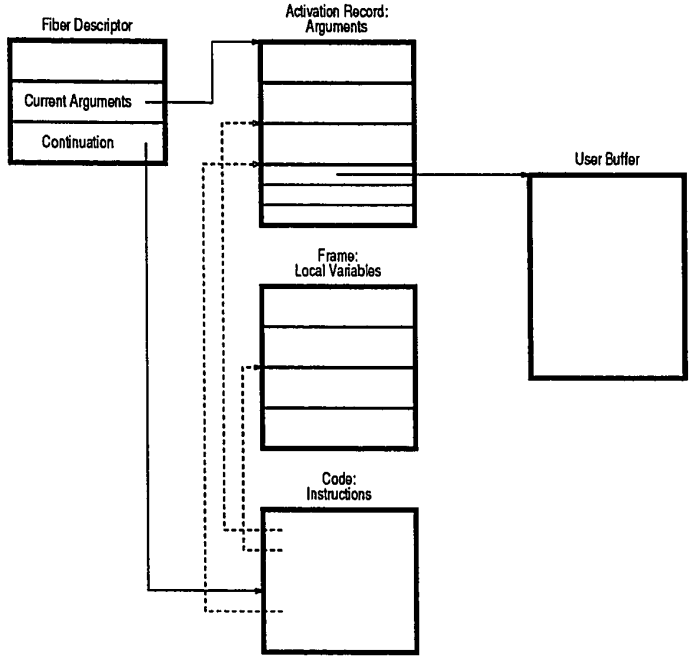

Figure 5.1: Compiled fiber

Control enters a fiber by means of the continuation field in the fiber descriptor. This is initialized to point at the beginning of the executable code. When a fiber deactivates itself, this continuation is set to point to the instruction after the sequence control statement which deactivated the fiber.

Each sequencer is compiled into code which performs one or more of the following:

1. activates a fiber, i.e. creates a new activation record and initializes it with the arguments to the invocation,
2. deactivates itself and resets the continuation in the fiber descriptor,
3. saves the internal state of the invoking fiber, and
4. transfers control—by means of the fiber descriptor—to the first activation of the highest priority fiber which is currently activated.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

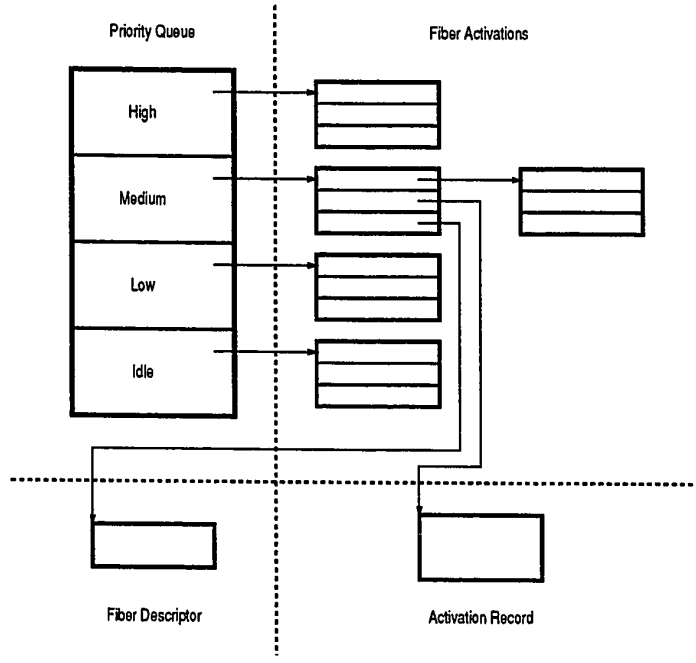

Figure 5.2: Fiber scheduling

5.2.2 Fiber Scheduling

In general, the determination of the highest priority activation must be left until execution time. This requires run-time support in the form of a fiber scheduler.

Figure 5.2 shows the abstract structure of such a scheduler. There is a queue of fiber activations which correspond to each of the priority levels. A fiber activation has a pointer to the associated fiber descriptor and one to an activation record.

When control is transferred into a fiber, the fiber activation is dequeued and the fiber descriptor is updated to point at the new activation record. Execution continues at the continuation specified in the fiber descriptor.

Unlike ordinary procedure call, neither the frame nor the activation record is stack allocated. The frame is statically allocated and the activation record is generally heap

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

allocated. This has one important consequence. Since the semantics permit fiber sequencing only in the body of a fiber, the stack is empty when control is transferred to another fiber. This permits a single stack implementation with attendant performance benefits

- fewer register window saves or equivalently fewer cache invalidations,
- "stack-handoff" during control transfer, reducing transfer time,
- simple, and therefore fast, run-time fiber scheduling, and
- reduced start-up costs and memory utilization.

5.2.3 Execution Cost

The execution cost of the general intra-node fiber mechanism, can be split into five components:

Allocating Activation Record. This is potentially the highest cost component, since the activation record is heap allocated in general. However, the size of the record is known at compile time, so binned allocation techniques can be used.

Marshalling Arguments. Argument marshalling is almost identical to procedure call and should be no more expensive in general. The cost is proportional to the number and size of arguments.

State Saving. The cost of state saving depends on the effectiveness of the compiler.

Fiber Scheduling. Fiber scheduling only inserts activations at the beginning or end of a queue and extracts activations from the head of the queue. Scheduling is therefore a constant time operation.

Dereferencing Variables. Each fiber argument is accessed through one or more levels of indirection.

Only the first four of these are direct costs of fiber invocation. If allocation can be made sufficiently cheap, fiber invocation is more expensive than procedure call by the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

cost of fiber scheduling and state saving.

5.3 Data Transfer

5.3.1 General Protocol

The efficient implementation of a `remote_continue` invocation of a general `target` fiber `foo`, involves a multi-phase transfer protocol. This protocol (very slightly simplified) is shown in Figure 5.3. We use the abstract transfer primitives defined in Section 2.2.3 in the specification.

Phase 1. The `remote_continue` creates an activation record on the source node, of type `fooACTVREC`, for `foo`. The activation record is initialized with the arguments to the invocation. The source also creates a transfer dope vector, of type `fooDOPEREC`, for transfer bookkeeping. A channel identifier is created which binds to the dope vector. This channel identifier and some transfer bookkeeping information is sent in a packet to the destination handler `fooHandler1`.

Phase 2. `fooHandler1` creates an activation record and a transfer record (which contains a dope vector and a pointer to the activation record) on the destination. It creates a channel identifier which binds to the activation record. This channel identifier and some bookkeeping information is sent in a packet to the source handler `fooHandler2`.

Phase 3. `fooHandler2` sends the activation record on the source directly into the activation record on the destination, using the appropriate channel identifier. `fooHandler3` is called on the destination when transfer is complete.

Phase 4. `fooHandler3` executes the *transfer binding block* code defined for the fiber. This creates the user bindings which identify the locations to which data has to be transferred. A channel identifier is then bound to each of the locations of the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Source

```
// remote_continue
 PACKET pkt;
 fooACTVREC * fAR   = // allocate
 fooDOPEREC * fDR   = // allocate
 fDR.actvr          = fAR;
 fDR.dest           = dest;
 ..; // Initialize fAR with arguments
 fooHandler0 (fDR);

void fooHandler0 (fooDOPEREC * fDR) {
 CHANID tID         = channelID (
     fDR, & fooHandler4, 0);
 pkt.src            = selfAddress;
 pkt.sTID           = tID;
 pkt.sDR            = fDR;
 AMPacket (dest, pkt, & fooHandler1);
}
```

Destination

```
void fooHandler1 (PACKET pkt) {
 PACKET pkt1;
 fooACTVREC * fAR   = // allocate
 fooTRANS * fTR     = // allocate
 fTR.doper.src      = pkt.src;
 fTR.actvr          = fAR;
 fTR.doper.sTID     = pkt.sTID;
 CHANID tID         = channelID (
     fTR, & fooHandler3);
 pkt1               = pkt;
 pkt1.dTID          = tID;
 AMPacket (pkt.src, pkt1, & fooHandler2);
}
```

```
void fooHandler2 (PACKET pkt) {
 fooDOPEREC * fDR   = pkt.sDR;
 AMLong (fDR.dest, pkt.dTID,
     fDR.actvr, sizeof (..));
}
```

```
void fooHandler3 (
 fooTRANS * fTR, void * v) {
 int * ubTransCount = // allocate
 fooACTVREC * fAR   = fTR.actvr;
 ..; // execute TRANSFER BINDING BLOCK code
 for (int i = 0; i < numUserBindings; i++)
   fTR->doper.b[ i ]= channelID (
       userbinding[ i ],
       & fooHandler5, uBTransCount);
 uBTransCount       = numUserBindings;
 AMLong (fTR.doper.src, fTR.doper.sTID,
     & fTR.doper, sizeof (..));
}
```

```
void fooHandler4 (
 fooDOPEREC * fDR, void * v) {
 for (int i = 0; i < numUserBindings; i++)
   AMLong (fDR.dest, fDR->doper.b[ i ],
     data1, sizeof (..));
 ..; // enable sequencers and next transfer
}
```

```
void fooHandler5 (
  void * userbinding, int * uBTransCount) {
 if ( --ubTransCount > 0 )
   return;
 else {
   ..; // transfer complete, activate fiber
 }}
```

Figure 5.3: General Transfer Protocol

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

`userbound` parameters. These channel identifiers are stored in a dope vector and then sent to the appropriate channel identifier which calls `fooHandler4`. A counter is created and initialized to the number of user bindings in the transfer.

Phase 5. `fooHandler4` initiates the unbuffered transfer to each of the channel identifiers (which correspond to the `userbound` variables) identified in the dope vector. As each of these transfers completes, it calls the destination handler `fooHandler5`.

When all transfers are complete, end of transfer processing occurs on the source. Each end of transfer must coordinate with any `continue`, `reset` or `exit` sequencers which may have been called from `source` fiber which executed this `remote_continue` sequencer. These sequencers can then finish execution.

The end of transfer may also be used to coordinate successive data transfers. It may be desirable to sequentialize transfers so that only one or two are outstanding at any time. The next transfer may be asynchronously started by the end of this transfer.

Phase 6. `fooHandler5` decrements the counter each time a transfer is complete. When the counter reaches zero, all `userbound` variables have been initialized. At this point a fiber activation for `foo` is created and inserted into the fiber scheduler.

All the code required to implement this protocol can be compiler generated from the fiber definition.

5.3.2 Execution Cost

In terms of the transfer overheads identified in Section 2.3.1, this protocol has the following cost structure.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Pure Transfer

- **Network Service.**
  - **Polling.** Costs depend on the frequency of polling and cost of a single hardware poll.
  - **Interrupt Driven.** Notice that each protocol handler has a very small stack frame, i.e. few and small local variables. Due to fiber semantics, the non-frame data that is accessed is either static, heap allocated or on the *single* user stack. These handlers can therefore execute in the user context, on the user stack without significant state-save costs, thereby making interrupt driven transfer a practical proposition.
- **Buffer Copying.** There is *no* buffer copying in this protocol.

Protocol Overhead

- **Buffer Management.** The transfer protocol allocates an activation and dope record. This is potentially expensive, but as in the case of fiber invocation, the sizes of these records are known at compile time and binned allocation techniques can be used.
- **Buffer Binding.** The buffer bindings for the `userbound` variables would be present in any transfer protocol. The protocol therefore performs 2 extra buffer bindings.
- **Protocol Data.** There are 3 transfers over and above the **Pure Transfer** requirements. These are the 2 packet transfers at the end of Phases 1 and 2, and the long transfer at the end of Phase 4. The long transfer sends a small constant amount of data plus a small amount for each `userbound` variable.
- **Protocol Control.** The control overhead for the protocol is 4 function calls plus a function call and a few assembly statements for each `userbound` variable.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Aside from allocation, the execution cost for this protocol is the cost of a minimal transfer + a small constant buffer binding overhead + a small constant pure transfer cost + a small constant protocol data overhead + (a small constant overhead + a small constant pure transfer cost) per `userbound` parameter.

5.4 Compiler Optimizations

Aside from expressiveness, optimization is the single most important benefit of having linguistic support for data and control transfer. During the process of compilation, a great deal of information is available about variables, control flow, and data movement. This information can be used to generate code that is customized for each data and control transfer and specialized to the target architecture.

5.4.1 Invocation & Coordination

Allocating Activation Record

The execution cost for allocation of activation records and bindings can be substantially reduced by

- using a binned allocator specialized to each fiber, and
- preallocating a certain fixed number of activation records.

This can bring down the allocation execution cost to a small constant.

In certain cases, the cost can be brought down to zero, if desired, by statically allocating an activation record. For example, any time that a fiber is invoked, other than in the body of a looping construct, this can be done.

Marshalling Arguments

Many of the same techniques that work for procedure call will work as well for fiber invocations, for instance, passing arguments in registers.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

State Saving

The compiler has complete information about the fiber frame and register usage. This information can be used to save the minimal required state when switching control out of a fiber.

Dereferencing Variables

All parameters are dereferenced via a pointer to the activation record. A smart compiler will cut down the cost of parameter dereferencing by loading the base address of the activation record and doing offset addressing. The cost over ordinary variable reference will thus be low (essentially loading the base address of the activation record), since the whole activation record will tend to be in cache. When the activation record is statically allocated, variable reference should be even (albeit slightly) faster.

In instances where the user binding in the transfer binding block can be determined to be a compile time constant, the user binding can be replaced by a constant binding.

5.4.2 Data Transfer

A compiler has information about the type, size, alignment, scope and extent of each variable that is declared in a program. If the language semantics include inter-node transfers, the compiler also has information relevant to transfers, node addresses, target architecture, etc.

Protocol Overhead

- **Buffer Management.** The strategy of binned allocation plus preallocation works for inter-node code as well as for intra-node code, with similar performance benefits.

  It is somewhat harder to determine when static allocation can be safely used on the destination node. There are such cases however, for instance, if the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

target fiber does not contain any `suspends`, non-terminal `continues`, or loops, its activation record can be statically allocated. Such a fiber structure might occur, for instance, in the implementation of a permutation-like communication pattern.

- **Protocol Specialization.** The protocol described in the previous section is a general protocol. Specialized protocols, with fewer phases and/or lower overhead can be used depending on the type and size of variables. For instance, a small activation record that will fit in a single packet transfer can be transferred with a 2 phase protocol. The first phase starts the transfer on the source. The second executes on the destination on packet reception, creates and initializes the activation record and inserts an activation into the scheduler.

- **Architecture Specialization.** The straightforward compilation of a sequence of `remote_continues` would generate a sequence of data transfers, with perhaps only 1 or 2 outstanding at any time to prevent network congestion.

  This compilation behavior can be modified to take advantage of special features of the architecture. For instance.

  - If the NB on the architecture has multi-ported access to the network, it may be advantageous to have more transfers outstanding at any time. The compiler can use knowledge of the architecture to guide code generation.
  - There will be situations when `remote_continue` is executed a number of times with the same arguments, but different destination addresses, either sequentially or from a loop. If the architecture has a broadcast or multicast primitive, a somewhat different broadcast protocol can be generated in place of the point-to-point protocol.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Transfer Related

- **Network Delay.** Recall from Section 2.4 that the general approach to optimizing this cost is instruction scheduling.

  - **Compile-Time.** This requires "unrolling" the loop of packet transfers in `AMLong` and "hoisting" other code into the "network delay slots".

    Unfortunately, the protocol handlers shown in Figure 5.3 execute asynchronously to the rest of the program. This implies that the only code that can be statically hoisted into the appropriate slots is code from the same handler. There is no such suitable code for the `AMLong` operation in `fooHandler3`. However the code for successive `AMLong` operations in the loop in `fooHandler4` could be interleaved. The efficacy of doing so would depend on the performance characteristics of the network architecture.

    Protocols customized for special usages may offer more opportunities for scheduling.

  - **Run-Time.** Fiber semantics permit both memory to DMA spooling and (because of the nature of the protocol handlers) efficient interrupt driven transfers.

- **Alignment Penalty.** The performance of both intra-node and inter-node transfer can depend heavily on data alignment. If safe, a compiler can modify the alignment of variables, so that the source and destination locations of inter-node transfers are identically aligned.

5.5 CM5 Implementation

The first `C`-Eli implementation was targeted to Thinking Machines Corporation's CM5.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Each CM5 node consists of a SPARC processor, main memory and an NB. Two features of the CM5 architecture make it a significant test bed for validating C-Eli performance.

- The packet size on the CM5 is very small, being a mere 4 words long.
- Generalized state-save is very expensive on the SPARC processor.

It thus simultaneously requires compiler generated state saves for fiber scheduling and imposes a tough test on the overhead of remote fiber invocation.

At the time of implementation, the only practical low level transfer mechanism available on the CM5 was the CMAM Active Message implementation. This implementation is polling based. For reasons having to do with deadlock avoidance, CMAM handlers are restricted in the transfer operations they can invoke. In particular, handler to handler transfer is not permitted to a depth greater than 1. A handler can transfer to another handler, but the latter cannot perform any transfers. For any greater depth of handler invocation, it is necessary to explicitly schedule execution of handlers in software.

We implemented C-Eli by modifying gcc `v.2.2.2` to compile the fiber constructs into SPARC assembly language for computation and CMAM `v.2.5` for data transfer.

Due to the use of CMAM we were also restricted to host-node style SPMD programming (only the node programs use C-Eli constructs), rather than the newer hostless style.

One of our concerns was to be completely backward compatible with C, i.e. not affect the compilation of non-fiber code. This was no mean task, since fiber implementation required changes not just to the parser, but to the code generator, some parts of the optimizer and instruction scheduler as well. Roughly 40,000 out of a total of approximately 200,000 lines of code were potentially affected. We succeeded to the extent that `eli-gcc` can compile itself.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

5.5.1 Invocation & Coordination

The CM5 implementation of intra-node code generation is close to that outlined in Section 5.2 above. The fiber scheduler is written in 150 lines of `C`. 10 lines of `SPARC` assembler are required to implement the actual fiber switch.

The implemented language is slightly different from the language described in this thesis. The definition of a `source` fiber was conceived after the implementation. The addition improves the I-BAD-C characteristics of the language. Fortunately, this change leaves the compilation and performance characteristics of fibers largely unaffected.

5.5.2 Data Transfer

The inter-node code does not use the protocol described in Section 5.3.1. We use a 4 phase protocol instead.

The reason for this is historical. Because of the handler to handler transfer constraints built into `CMAM`, we initially tried to implement a 3 phase protocol. This turned out to be not general enough, but was easily extended to the 4 phase protocol which the current implementation uses.

The 4 phase protocol differs from the 6 phase protocol by collapsing `fooHandler0` and `fooHandler2` into a single handler which sends the activation record directly in a message that is always buffer copied on the destination. `fooHandler1` is eliminated. The other phases remain approximately the same. The 4 phase protocol performs a **Buffer Allocation** and **Buffer Copy** not performed by the 6 phase protocol. On the other hand, the 6 phase protocol performs one more **Buffer Binding**. On the whole, we expect the cost of the 4 phase protocol to exceed the cost of the 6 phase protocol. For fibers with a few small-sized arguments, the difference should be small.

The implementation also has the extra overhead of explicit handler scheduling forced by `CMAM`'s constraint on handler to handler communication. This scheduling is performed by 50 lines of `C` code.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

There is one area in which the inter-node compilation can be improved. Experience suggests that it is better to restrict the number of outstanding simultaneous transfers, by delaying invocation of subsequent transfers until the prior ones have completed. This can be accomplished fairly simply by a compiler. The current implementation does not do this.

5.5.3 Compiler Optimizations

Implementation of compiler optimizations was constrained both by the optimization machinery built into `gcc v. 2.2.2` and the project time frame.

Thus, for instance, optimizations such as global alignment analysis and modification were not feasible because `gcc` did not have the machinery to perform inter-procedural analysis, and time did not permit us to implement such a facility from scratch.

Other optimizations, which can in principle be implemented on top of `gcc`, were not implemented because of lack of time. The compiler was built in 5 man-months (also equal to 5 months) from start to finish.

The purpose of the implementation was to establish the efficacy of the fiber mechanism as a low overhead approach to specifying overlapped computation and data transfer. We therefore decided to concentrate our optimization efforts on the most difficult case, rather than the most frequent. Short messages are intrinsically the most expensive to integrate with scheduling, since the computational overheads are proportionally greatest.

The optimizations that we did include were:

- Minimal state saving for fiber switch, i.e. save only live registers.
- Binned buffer allocation with a limited number of preallocated buffers. This approach can be taken much further.
- A specialized 2 phase single packet protocol for `remote_continue` executions with no arguments. This protocol makes no extra memory accesses during data

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

suspend	4.2 μs
relinquish	1.4 μs
poll	1.1 μs

Table 5.1: Fiber Invocation & Coordination Performance

transfer, i.e. marshalls its arguments in registers.

- Double word alignment for all activation records and other fiber related storage which participate in data transfer.

5.5.4 Performance

The performance of the C-Eli coordination mechanisms are presented in Table 5.1.

suspend gives a good estimate of the bare costs of custom State Saving and fiber scheduling—4.2 μs. Compare this with the cost of the minimal generalized state save (Table 2.4)—26 μs. The performance benefits accrue from the combination of single stack semantics and compiled register saves.

The cost of a negative relinquish operation is 1.4 μs. Recall from Section 4.5 that a relinquish operation can be viewed as a check for completion of a transfer on the destination. The cost can therefore be compared to the cost of a negative Nonblocking MP poll (Table 2.4)—3.4 μs. The relinquish operation is actually cheaper.

The poll[1] operation is an assembly poll of the network interface (CMAM_poll) combined with the handler scheduling mechanism. The cost of a negative poll is 1.1 μs. Comparison with the cost of an assembly poll (Table 2.4)—1 μs, reveals an overhead of 0.1 μs for the scheduling mechanism with no handlers.

Table 5.2 shows the performance of remote_continue. These measurements were obtained using a ring topology. Each node performed a number of remote_continue invocations on the neighbor to its right. When all invocations were complete, an

[1]While not a part of the core C-Eli semantics, the poll operation is necessary in polled transfer implementations.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

	0 B	**4 KB**
`remote_continue`	12.5 μs	978 μs

Table 5.2: Data Transfer Performance

end-of-test fiber was activated. The time interval between the beginning of the initial `remote_continue` and the start of execution of the end-of-test fiber was averaged over all nodes. This gives a measurement for the combined cost on source and destination nodes of a single `remote_continue` operation.

Compare this to the cost of minimal transfer (Active Message) and Synchronous MP shown in Table 2.2.

For single packet messages (column 1), the C-Eli cost is the same order of magnitude as the minimal transfer Active Message cost. This cost of 12.5 μs includes the cost of scheduling—4.2 μs. The **Protocol Overhead** therefore is at most 5 μs. Consider the analysis of the general protocol presented in Section 5.3.2. The **Protocol Overhead** of the general protocol must be at least the cost of 2 **Buffer Bindings**—20 μs (from Table 2.3)—which is quite a bit larger than the minimal packet transfer cost of 3.3 μs. Compiler customization of the transfer protocol proves itself in this case. In particular, the C-Eli cost is much lower than that of a short Synchronous MP message.

For long transfers the C-Eli cost is comparable to the CMMD cost. It should be mentioned that CMMD preallocates transfer resources, whereas the C-Eli implementation can definitely be improved in this area. In addition, the C-Eli transfers were not started with synchronized processors (very difficult to do since C-Eli transfers are asynchronous). This could be expected to lead to a greater **Network Delay**, which can not be measured and factored out.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

5.6 Fiber Efficiency

This chapter has shown that the fiber constructs can be efficiently implemented on architectures conforming to our compute model (Section 2.1).

In the general case, this efficiency stems from several sources:

- The single stack semantics of the fiber execution model, make possible very rapid control transfers between asynchronous execution units.
- The single stack semantics also makes it practical to use interrupt driven data transfers.
- Each `remote_continue` sequencer can be compiled into code which causes very little execution latency.
- It is possible to safely specify unbuffered data transfers.
- Asynchronous data transfers have the potential to greatly reduce idle time.

The appropriate application of compiler technology further improves fiber performance:

- Aggressive static and binned allocation of resources improves the performance of both control and data transfer.
- Compilation of fiber sequencers results in minimal state saving, thus improving control transfer performance.
- Adjustment of the alignment of source and destination areas improves data transfer performance.
- Protocol specialization can significantly improve the performance of individual transfers.
- Compilation of data transfers leaves open the possibility of "filling network delay slots".
- Various architecture specific data transfer optimizations are possible.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 6

Fiber Application

This chapter demonstrates the practical utility of the fiber mechanism in improving the execution latency of real SPMD code. We also consider the benefits of using the fiber mechanism as the target language for an HPL compiler.

Thus far we have demonstrated that the performance of individual data transfers compiled from `remote_continue` fiber invocations is comparable or better in practice and superior in theory to that of existing high-level transfer mechanisms.

We would like to show that

- the fiber design also leads to real improvement in the execution time of entire programs, and
- that such improvements can be achieved without a lot of programming effort.

These points are best demonstrated in the context of real programs, i.e. existing code written by resource-constrained programmers using prevailing programming and algorithmic practices.

In particular, we wished to answer the following questions:

1. Are there real SPMD programs whose performance can be significantly improved by the use of an asynchronous data and control transfer mechanism?
2. Can such programs be converted to the fiber model fairly easily?
3. How much of a performance benefit, if any, would such a conversion yield?

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

In general, we would expect the fiber construct to have the greatest impact on the performance of programs which spend a significant fraction of execution time

- performing buffered or short data transfers, or
- idling

Do such applications codes exist in practice? *A priori*, one might expect that the natural candidates for the fiber construct are applications which are asynchronous (or loosely synchronous) and have irregular data decomposition. If this were indeed the case, the usefulness of fibers would be severely limited, since such applications form a small fraction of all SPMD codes.

We looked instead for more typical SPMD codes, i.e. synchronous algorithms with a regular data decomposition. In this chapter, we present our experience with one such program. Based on this example, the answers to our questions are quite favorable. The program we present performs matrix factorization using the QR algorithm[1].

6.1 User Code

6.1.1 QR Factorization

The QR factorization algorithm factors an $N \times N$ matrix A into an $N \times N$ orthogonal matrix Q and an $N \times N$ upper triangular matrix R such that $A = QR$.

The algorithm works by successively applying a series of transformations P^j to A. This results in a sequence of transformed matrices A^j ($A^0 = A$). The same sequence of transformations applied to the identity matrix I, results in a sequence of transformed matrices I^j. The matrix A^j has the property that the elements $A^j_{m,n} \mid m \leq j \wedge n > m$ are zero. The sequence results in matrices $A^N = R$ and $I^N = Q$ which have the desired properties.

The transformation P^j affects rows $j+1$ through N of A^{j-1}. It is composed of a

[1]We gratefully acknowledge the assistance of John Holm and Prithviraj Banerjee in making Holm's QR factorization code available to us.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

sequence of rotations $P_k^j \mid 1 \leq k \leq N-j$. P_k^j works on a pair of neighboring rows A_{N-k}^{j-1} and A_{N-k+1}^{j-1}. It has the effect of zeroing out the element $A_{N-k+1,j}^{j-1}$, without affecting the zeros already present in columns $A_{N-k+1,1:j-1}^{j-1}$ and $A_{N-k,1:j-2}^{j-1}$.

6.1.2 Parallelization

When the transformations are viewed as matrix operations, the data dependences force strict sequentiality, i.e. P_m^i cannot be applied before P_l^h when $(h < i) \vee (h = i \wedge l < m)$.

The potential for parallelization exists when the transformations are viewed as operations on row pairs. The constraints now are:

1. P_m^j cannot be applied before P_l^j when $l < m$,
2. P_k^i cannot be applied before P_k^h when $h < i$, and
3. P_k^{j+1} cannot be applied before P_{k+1}^j.

Any pair of transformations which lie outside these constraints may be applied in parallel, e.g. P_l^{j+1} can be applied in parallel with P_m^j when $m > l+1$.

Row operation parallelization is the strategy used in the QR factorization code.

The rows of matrix A are cyclically partitioned onto an M (even and > 2) node MPP with a ring topology. Row N lies on processor 0, row $N-1$ on processor $1, \ldots,$ row $N-n$ on processor $n \bmod M, \ldots$. Neighboring rows always lie on neighboring nodes. In general, node m has rows $N-m, N-m-M, N-m-2M, \ldots, N-m-K_mM$ where $K_m = \lfloor \frac{N-m-1}{M} + 1 \rfloor$.

The program uses a multistep synchronous algorithm. Each step consists of a sequence of substeps. Each substep consists of a sequence of the P_k^j transformations.

For example on node m, step 1^m consists of the substeps:

$1_1^m.\ P_{N-m}^1$

$1_2^m.\ P_{N-m}^2$

$\ldots$

$1_{\frac{M}{2}}^m.\ P_{N-m}^{\frac{M}{2}}$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

step 2^m consists of the substeps:

2^m_1. $P^{\frac{M}{2}+1}_{N-m}$, P^1_{N-m-M}

2^m_2. $P^{\frac{M}{2}+2}_{N-m}$, P^2_{N-m-M}

...

$2^m_{\frac{M}{2}}$. P^M_{N-m}, $P^{\frac{M}{2}}_{N-m-M}$

and so on.

In general on node m, step h^m consists of the substeps:

h^m_1. $P^{(h^m-1)\frac{M}{2}+1}_{N-m}$, $P^{(h^m-2)\frac{M}{2}+1}_{N-m-M}$, ..., $P^1_{N-m-i_{h^m}M}$

h^m_2. $P^{(h^m-1)\frac{M}{2}+2}_{N-m}$, $P^{(h^m-2)\frac{M}{2}+2}_{N-m-M}$, ..., $P^2_{N-m-i_{h^m}M}$

...

$h^m_{j_{h^m}}$. $P^{(h^m-1)\frac{M}{2}+j}_{N-m}$, $P^{(h^m-2)\frac{M}{2}+j}_{N-m-M}$, ..., $P^j_{N-m-i_{h^m}M}$

where

$$i_{h^m} = \begin{cases} h^m - 1 & \text{if } h^m < K_m + 1 \\ \min(K_m, k_{h^m} - 1) & \text{otherwise} \end{cases}$$

$$j_{h^m} = \min(k_{h^m}, \frac{M}{2})$$

and

$$k_{h^m} = \lfloor N - m - 1 - \frac{(h^m - 1)M}{2} \rfloor$$

The algorithm on node m terminates after step h^m in which $j_{h^m} = 1$.

Since each pair of neighboring rows is partitioned onto neighboring nodes, a single P^j_k transformation must be carried out in two parts, one on each of the neighboring nodes. Thus, when we say that node m performs a transformation P^j_k, we mean that it initiates the transformation. Half of the transformation must actually be performed on node $m + 1 \bmod M$. Operationally, each node transfers a copy of its row to the appropriate (left or right) neighbor and then updates its own row in accordance with the transformation.

Each node participates in all the transformations of substep h_i of its predecessor (left neighbor) before performing the transformations in its own substep h_i. Given

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
#define MATU_T        101
#define MATL_T        102

int M = CMMD_partition_size ();
int N /* Number of rows (or columns) in the matrix */
int ownRows = N / CMMD_partition_size ();
int size = (2 * N + 1) * sizeof(double);
int diag = N - nodeID;
double RQ[ ownRows ][ 2 * N + 2 ];
int nRows = 1;
int nZeros = 0;
BOOL flag = 1;
int nR;
int origin = (nodeID == 0) ? 1 : 0;

int pred (int n) { return (n - 1) % M; }
int succ (int n) { return (n + 1) % M; }
```

Figure 6.1: Supporting Variables & Functions

the synchronous nature of the algorithm, this leads to the following alternating communication pattern: a phase E where node pairs $(m, m+1 \bmod M) \mid m \bmod 2 = 0$ are communicating, alternating with a phase O where node pairs $(m, m+1 \bmod M) \mid m \bmod 2 = 1$ are communicating.

6.1.3 MP Realization

Figure 6.1, Figure 6.2 and Figure 6.3 display a (slightly cleaned up) real program which implements this strategy using the CM5 CMMD Message Passing library.

Figure 6.1 shows the supporting functions and variables used by the code.

The functions pred and succ are used to implement the ring topology. M is the machine size and N is the array size.

Each row of the array RQ holds a node-local row of the matrix A^j in columns 0 to N - 1 and the corresponding row of I^j in columns N to 2 * N - 1. Column 2 * N + 1 holds the index k of the row, and column 2 * N holds the value i of the latest P^i_{k+1} transformation performed on the row.

The variable nZeros on node m holds the value j of the latest P^j_{N-m} transforma-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
void updateRowU (double own[], double other[]) {
  int index = other[ 2 * mSize ] - 1;
  double a = own[ index ];
  double b = other[ index ];
  double k = sqrt (a * a + b * b);
  double c = a / k;
  double s = b / k;
  for (int i = index; i < 2 * mSize; i++)
    own[ i ] = own[ i ] * c + other[ i ] * s;
}

void updateRowL (double own[], double other[]) {
  int index = own[ 2 * mSize ] - 1;
  double a = other[ index ];
  double b = own[ index ];
  double k = sqrt (a * a + b * b);
  double c = a / k;
  double s = b / k;
  for (int i = index; i < 2 * mSize; i++)
    own[ i ] = own[ i ] * c - other[ i ] * s;
}
```

Figure 6.2: The P_k^j Transformation

tion performed on row $N - m$. The variable nRows on node m holds the value i_{h^m} defined in the algorithm description.

Instead of explicitly performing the computations for h^m, i_{h^m}, j_{h^m} and k_{h^m} used in the algorithm description, the code uses the values of nZeros, nRows, QR[][2 * N] and QR[][2 * N + 1] to test for the required conditions and achieve the same execution behavior.

Figure 6.2 shows the 2 procedures updateRowL and updateRowU which perform the 2 halves of the transformation P_k^j on rows k (lower row on node m) and $k-1$ (upper row on node succ(m)) respectively.

Figure 6.3 shows the Blocking MP implementation of the algorithm. On a node m the computation consists of a main loop. Within the main loop are two loops. The first loop implements node m's half of the transformations in substep $h^{\mathtt{pred}(m)}{}_{k-1}$ and the second initiates the transformations in substep h_k^m. A change in the value of nRows within the main loop corresponds to the end of a step h_m.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
double bbb[ 2 * N + 1 ];

void QR (void) {
  while ( nRows ) {
    if ( nRows == ownRows ) flag = 0;
    nR = nRows;
    for ( int i = origin; i < nRows; i++ ) {
#3    CMMD_receive_block (CMMD_ANY_NODE, MATU_T,  bbb,  size);
#4    CMMD_send_noblock (pred (nodeID), MATL_T, RQ[ i ],  size - sizeof(double));
      if ( bbb[ 2 * N ] == RQ[ i ][ 2 * N + 1 ] )
        nR -= 1;
      else
        RQ[ i ][ 2 * N ] = bbb[ 2 * N ];
      updateRowU (RQ[ i ], bbb);
    }
    nRows = nR;
    if ( nodeID == 0 )
      RQ[ 0 ][ 2 * N ] = nZeros + 1;
    for ( i = 0; i < nRows; i++ ) {
#1    CMMD_send_noblock (succ (nodeID), MATU_T,  RQ[ i ],  size);
#2    CMMD_receive_block (CMMD_ANY_NODE, MATL_T,  bbb,  size - sizeof(double));
      updateRowL (RQ[ i ], bbb);
    }
    nZeros += 1;
    if ( flag && nZeros % (M / 2) == 0 ) nRows += 1;
    if ( nodeID == 0 && nZeros == diag - 1 ) nRows = 0;
  }}
```

Figure 6.3: QR Factorization: Blocking MP

The transfer of rows required to carry out a transformation P_k^j is accomplished by the Blocking MP operations within the inner loops. The transfer is initiated on node m by **sending** a copy of its row (line "#1"). As soon as succ(m) **receives** this row (line "#3"), it **sends** its own row (line "#4"), which node m **receives** (line "#2"). Both **receives** are blocking operations. The bbb array is used on both nodes to store a temporary copy of the non-local row.

After the non-local row has been transferred, the transformation is performed in parallel by calling updateRowL and updateRowU on m and succ(m) respectively.

The main loop continues until the termination condition is satisfied.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

6.1.4 Converting to C-Eli

This program is an interesting test case for the practicality of the fiber construct. The algorithm, as described, earlier is synchronous and the blocking behavior is required to satisfy the data dependences. The cyclic row partitioning also assures a good *a priori* load balance. Short of reordering the P_k^j transformations, which would essentially change the algorithm, there seems to be limited potential in terms of finding independent computations to overlap with data transfer.

Potential Performance Slack

However, this program does perform a lot of communication. Each transformation requires the transfer of 2 rows between neighboring nodes. Improving the performance of these transfers should have a sizable impact on the overall execution latency.

A closer inspection of the communication routines reveals a potential inefficiency in the blocking **receive**s on lines "#2" and "#3". Whenever this **receive** is reached *after* the matching **send**, an extra time cost is potentially incurred, i.e. the time taken for a buffer copy (in a copying implementation) or the time taken for the entire transfer (in a non-copying implementation) [2].

A priori, the delay is most likely to occur on the **receive** in the first iteration of an inner loop (thereafter, the communicating nodes are more likely to be synchronized). Beyond such qualitative statements, it is difficult to characterize theoretically the total performance impact of this delay.

This delay could be totally eliminated if the entire transfer could be completed asynchronously. The only extra cost here would be that incurred by the control transfer mechanism. This delay is the performance slack targeted by the fiber implementation.

[2] CMMD is a non-copying implementation of Message Passing, so the entire transfer cost is incurred. If CMMD provided a buffer copying implementation, this cost would, in theory, be lower by a factor of $\approx$ 3–4 (Table 2.1). In practice, the cost would likely be higher than the theoretical minimimum, since more than the bare minimum buffer copying could be expected to occur. In any case, even a factor of 4 reduction would not make a significant delay into an insignificant one.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
double bbbU[ 2 * N + 1 ];
double bbbL[ 2 * N ];

source target high fiber upper (userbound double row[ 0 ]) <
  row = bbbU
> {
  while ( 1 ) {
    for ( int i = origin; i < nRows; i++ ) {
#3    suspend;
#4    remote_continue pred (nodeID) (nocopy (2 * N @ RQ[ i ]);
      if ( bbbU[ 2 * N ] == RQ[ i ][ 2 * N + 1 ] )
        nR -= 1;
      else
        RQ[ i ][ 2 * N ] = bbbU[ 2 * N ];
      updateRowU (RQ[ i ], bbbU);
    }
    continue lower ();
  }}

source target high fiber lower (userbound double row[ 0 ]) <
  row = bbbL
> {
  while ( 1 ) {
    nRows = nR;
    if ( nodeID == 0 )
      RQ[ 0 ][ 2 * N ] = nZeros + 1;
    for ( i = 0; i < nRows; i++ ) {
#1    remote_continue succ (nodeID) upper (nocopy (2 * N + 1 @ RQ[ i ]);
#2    suspend;
      updateRowL (RQ[ i ], bbbL);
    }
    continue QR ();
  }}

fiber QR (void) {
  while ( nRows ) {
    if ( nRows == ownRows ) flag = 0;
    nR = nRows;
    continue upper ();
    nZeros += 1;
    if ( flag && nZeros % (M / 2) == 0 ) nRows += 1;
    if ( nodeID == 0 && nZeros == diag - 1 ) nRows = 0;
  }}
```

Figure 6.4: QR Factorization: C-Eli

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Fiber Realization

Figure 6.4 shows the C-Eli implementation of the QR factorization code.

The conversion to fiber code is extremely straightforward. The first loop is encapsulated in the fiber `upper`. The continuation of the first loop until the end of the second loop is encapsulated in the fiber `lower`. The main loop is encapsulated in the fiber `QR`.

Control flow is identical to that in the Blocking MP version. `QR` invokes `upper`, which carries out its transformations and invokes `lower` which eventually returns control to the main loop in `QR`. The data transfers are initiated by the `remote_continue` sequencers and blocking is implemented by the `suspend` sequencer. The transfers are specified to eliminate buffer copying on both source and destination. The fiber code blocks in exactly the same places as the Blocking MP code. The numbered lines "#1", "#2", "#3" and "#4" correspond in functionality to the identically numbered lines in Figure 6.3.

Exactly the same variables are used to control execution behavior. The only change is the replacement of the array `bbb` with 2 arrays, `bbbU` and `bbbL`. The transfers in the first loop are made into `bbbU` and the transfers in the second loop are made into `bbbL`. This makes possible the simultaneous transfer of data from (to) the preceding (succeeding) node, thus eliminating the extra delay that exists in the Blocking MP implementation.

Ease of Conversion

As may be anticipated by an examination of the fiber code, the conversion to C-Eli was extremely straightforward. It took about 4 hours to complete the fiber implementation including the time taken to "untar" the original code.

In fact, the original conversion was done without an understanding of the details of the algorithm. The transformations were made on the basis of the synchronization and coordination structure of the program.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

	$T_{\text{Blocking MP}}$	$T_{\text{C-Eli}}$	$\frac{T_{\text{C-Eli}}}{T_{\text{Blocking MP}}}$
32	35.6224 s	25.3449 s	71.15 %
64	18.5806 s	13.6686 s	73.57 %
512	4.5289 s	3.3225 s	73.36 %

Table 6.1: QR Factorization: Comparative Performance

The conversion was made simple by the fact that the algorithm uses blocking primitives. The fiber bodies simply encapsulate the control structure of the relevant parts of the original program within an infinite loop.

6.1.5 Comparative Performance

Table 6.1 displays the performance results of the 2 programs. The programs were executed on 32, 64 and 512 node machines. The timings reported are elapsed times for the QR computation. I/O and array initialization were not timed. The timings correspond to an $N = 512$ sized array A initialized with random numbers (deterministic seed on each node) in the range -0.5 to 0.5. The programs were executed in the absence of other time-shared processes and the timings were made with the `CMMD_node_timer` calls. All times are in seconds.

The results are very good. The fiber implementation executed in 25%–30% less time than the Blocking MP implementation for all machine sizes.

6.2 HPL Target

The fiber construct could also prove to be useful as the target language for an HPL compiler. Such a target language should have at least the following characteristics:

- it should be capable of expressing any computation that is possible on the target hardware,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- it should provide a representation which can be easily manipulated by the compiler to implement optimizing transformations,
- the primitives should be efficient,
- it should be portable to a variety of MPP architectures, *without a change in performance characteristics*, and
- the primitives should allow for architecture specific optimizations.

The past few chapters have shown that:

- the asynchronous data and control transfer execution model of fibers is closely matched to the capabilities of the target hardware,
- encapsulated asynchronous execution continuations are a natural way to represent the coordinated data and control transfers which occur in SPMD computation,
- the fiber primitives are efficient,
- the fiber primitives can be implemented on a variety of MPP architectures and have similar *execution latency* characteristics on all of them.
- the primitives allow further optimization which are architecture or program specific.

Thus the fiber construct satisfies some of the requirements for an HPL target language. The fiber semantics could be refined to make it even more suitable for this purpose. For now, the fiber mechanism and its semantics provide a concrete prototype for such a target language.

6.3 Fiber Utility

We have affirmed the utility of the fiber abstraction mechanism. In response to the questions posed in the beginning of this chapter, we have found

1. There are real SPMD codes which can benefit from an asynchronous data and control transfer model. In fact, and perhaps surprisingly, *synchronous* codes, which form the great majority of SPMD codes, can so benefit.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2. The conversion of such codes to the fiber model can be accomplished quite easily in cases where blocking primitives are used.
3. Even a straightforward structural conversion, with no detailed understanding of the program, can result in substantial performance improvements.

In the light of these results and the findings of previous chapters, the fiber mechanism is also seen to be a suitable target language for HPL compilers.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 7

Conclusion

We have made a case for the use of a language mechanism for the specification of MPMD data transfer on MPP architectures. This mechanism provides superior expressiveness, safety and portability when compared to existing transfer paradigms without an accompanying loss of performance. Indeed, in many cases, it provides improved performance through the use of compiler optimization. The application of this mechanism results in wall clock speedups for a typical MPMD code. The mechanism also seems suited to serve as a prototype for the internal or target representation of MPMD code for an HPL compiler.

7.1 Future Directions

7.1.1 Semantic Refinement

The C-Eli coordination semantics are not definitive in the sense of constituting the last word on the subject. In the course of our project we made one major and several minor changes to the definition of C-Eli. At several points in the process we made what were essentially arbitrary decisions when faced with a choice between alternative constructs or semantic interpretations.

As we accumulate experience with asynchronous data transfer, it might become

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

necessary to reevaluate those choices. We expect the fiber semantics to be refined as such data becomes available.

7.1.2 Transfer Patterns & Aggregation

The C-Eli semantics already permit the specification of scatter-gather type transfers, through the use of fiber parameters.

Other transfer patterns, such as strided transfers, could be supported by the addition of suitable syntax in the declaration and instantiation of fiber parameters. The compilation of such specifications is relatively straightforward.

A more interesting challenge is to provide, within the fiber framework, language and compiler support to schedule and aggregate transfers in the manner described in [Wang *et al.* 93], [Ranka *et al.* 94]. Although some of these transformations cannot be performed at compile-time, others might be attainable through the use of appropriate compiler technology.

Such support would also make the fiber framework a more convenient HPL target language.

7.1.3 Flow Analysis

A compiler for the fiber constructs must be capable of transforming the fiber code to implement optimizations. This capability is even more important for an HPL compiler which uses fibers as an intermediate or target representation for MPMD code.

One development which would greatly increase this capability would be a flow analysis framework for fibers. The data and control flow characteristics of fiber code need to be systematized. A calculus of transformations on such flow graphs should then be built up.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Bibliography

[Anderson *et al.* 92] T.E. Anderson, B.N. Bershad, E.D. Lazowska, and H.M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Transactions on Computer Systems*, pages 53–79, February 1992.

[Bae & Ranka 94] Seungjo Bae and Sanjay Ranka. A Comparison of Different Message-Passing Paradigms for the Parallelization of Two Irregular Applications. Submitted to the Journal of Supercomputing, July 1994.

[Berryman *et al.* 90] Harry Berryman, Joel Saltz, and Jeffrey Scroggs. Execution Time Support for Adaptive Scientific Algorithms on Distributed Memory Machines. Technical Report 90-41, ICASE, May 1990.

[Bershad *et al.* 90] B.N. Bershad, T.E. Anderson, E.D. Lazowska, and H.M. Levy. Lightweight Remote Procedure Call. *ACM Transactions on Computer Systems*, 8(1), February 1990.

[Brewer & Kuszmaul 94] E.A. Brewer and B.C. Kuszmaul. How to Get Good Performance from the CM-5 Data Network. In *International Parallel Processing Symposium*, 1994.

[Chandy & Kesselman 92] K.M. Chandy and C. Kesselman. CC++: A Declarative Concurrent Object Oriented Programming Language. Technical Report CS-TR-92-01, Computer Science Department, California Institute of Technology, 1992.

[Chandy & Taylor 89] K. Mani Chandy and Stephen Taylor. The Composition of Parallel Programs. In *Proceedings of Supercomputing '89*, November 1989.

[Cooper & Draves 88] Eric C. Cooper and Richard P. Draves. C Threads. Technical Report CMU-CS-88-154, Department of Computer Science, Carnegie Mellon University, 1988.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[Culler *et al.* 93] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, May 1993.

[Dahl & Hoare 72] O.J. Dahl and C.A.R. Hoare. Hierarchical program structures. In O.J. Dahl, E.W.Dijkstra, and C.A.R. Hoare, editors, *Structured Programming*, pages 175–220. Academic Press, 1972.

[Dally *et al.* 87] W.J. Dally, L. Chao, A. Chein, S. Hassoun, W. Horwat, J. Kaplan, P. Song, B. Totty, and S. Wills. Architecture of a Message Driven Processor. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pages 189–196, 1987.

[de Bruin & Bohm 85] Arie de Bruin and Wim Bohm. The Denotational Semantics of Dynamic Networks of Processes. *ACM Transactions on Programming Languages and Systems*, 7(4):656–679, October 1985.

[Dennis 71] J.B. Dennis. Coroutines and Parallel Computation. In *Proceedings Fifth Annual Princeton Conference on Information Sciences and Systems 1971*, pages 293–294, 1971.

[Draves *et al.* 91] Richard P. Draves, Brian N. Bershad, Richard F. Rashid, and Randall W. Dean. Continuations: Unifying Thread Management and Communication in Operating Systems. Technical Report CMU-CS-91-115, Computer Science Department, Carnegie Mellon University, March 1991.

[Fox *et al.*] Geoffrey C. Fox, Sanjay Ranka, Jim Browne, Marina C. Chen, Alok Choudhary, Thomas Cheatham, Jan Cuny, Rudolf Eigenmann, Amr Fahmy, Ian Foster, Dennis Gannon, Tom Haupt, Mike Karr, Carl Kesselman, Chuck Koelbel, Wei Li, Monica Lam, Thomas LeBlanc, Allen D. Malony, Jim Openshaw, David Padua, Constantine Polychronopoulos, Joel Saltz, Michael Scott, Alan Sussman, Gil Weigand, and Kathy Yelick. Common runtime support for high-performance parallel languages. anon ftp: pcrc/pcrc-001@minerva.npac.syr.edu. Parallel Compiler Runtime Consortium.

[Gentleman 72] W.M. Gentleman. A Portable Coroutine System. In C.V. Freiman, editor, *Proceedings IFIP Congress 71*, pages 419–424. North Holland, 1972.

[Haynes *et al.* 86] Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand. Obtaining Coroutines with Continuations. *Computer Languages*, 11(3/4):143–153, 1986.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[Hoare 78] C. A. R. Hoare. Communicating Sequential Processes. *CACM*, 21:666–677, 1978.

[Holm *et al.* 94] John Holm, Antonio Lain, and Prithviraj Banerjee. Compilation of Scientific Programs into Multithreaded and Message Driven Computation. In *Scalable High-Performance Computing Conference*, pages 518–525, May 1994.

[Kahn & MacQueen 77] G. Kahn and D. MacQueen. Coroutines and Networks of Parallel Processes. In B. Gilchrist, editor, *Proceedings IFIP Congress 77*, pages 993–998. North Holland, 1977.

[Kahn 74] G. Kahn. The Semantics of a Simple Language for Parallel Programming. In J.L. Rosenfeld, editor, *Proceedings IFIP Congress 74*, pages 471–475. North Holland, 1974.

[Kale 90] L.V. Kale. The Chare-Kernel Parallel Programming Language and System. In *Proceedings of the International Conference on Parallel Processing*, volume 2, pages 17–25, 1990.

[Kernighan & Ritchie 88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice-Hall, 2nd edition, 1988.

[Liu & Culler 94] Lok T. Liu and David E. Culler. Measurements of Active Messages Performance on the CM-5. Technical report, Department of Electrical Engineering and Computer Science, University of California, Berkeley, 1994.

[Marlin 80] Christopher D. Marlin. *Coroutines: A Programming Methodology, a Language Design and an Implementation.* Lecture Notes in Computer Science 95. Springer-Verlag, 1980.

[MPIF 94] MPIF. MPI: A Message-Passing Interface Standard, May 1994.

[Papadopoulos & Culler 90] G.M. Papadopoulos and D.E. Culler. Monsoon: An Explicit Token-Store Architecture. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 82–90, May 1990.

[Ranka *et al.* 94] Sanjay Ranka, Jhy-Chun Wang, and Geoffrey Fox. Static and Runtime Algorithms for All-to-Many Personalized Communication on Permutation Networks. *IEEE Transactions on Parallel and Distributed Systems*, 1994. to appear.

[Skordalakis & Papakonstantinou 78] E. Skordalakis and G. Papakonstantinou. Coroutines in FORTRAN. *ACM SIGPLAN Notices*, 13(9):76–84, 1978.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[Stein & Shah 92] D. Stein and D. Shah. Implementing lightweight threads. In *USENIX*, 1992.

[TMC 93] TMC. Cmmd Reference Manual: Version 3.0, May 1993.

[von Eicken *et al.* 92] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *19th Intl. Symp. on Comp. Arch.*, 1992.

[Wang *et al.* 93] Jhy-Chun Wang, Tseng-Hui Lin, and Sanjay Ranka. NICE: Non-uniform Irregular Communication Exchange on Distributed Memory Systems. Technical Report SCCS-546, NPAC, Syracuse University, June 1993.

[Whitby-Stevens 85] C. Whitby-Stevens. The Transputer. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 292–300, 1985.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Appendix A

C-Eli Syntax

This appendix presents the abstract syntax of C-Eli. The grammar is based on that for ANSI-C presented in [Kernighan & Ritchie 88, pp. 234–239]. In keeping with the notation used in the original, we present non-terminal symbols as *italicized* text. Terminal symbols are presented in a `typewriter` font. An *opt* subscript indicates an optional non-terminal. Alternative productions of a non-terminal are displayed on successive lines.

Instead of reproducing the entire grammar of C, we have only reproduced those non-terminal symbols which are modified, or new in, C-Eli.

The following non-terminal symbols of C are modified for C-Eli.

function-definition:

  declaration-specifiers$_{opt}$ *declarator declaration-list*$_{opt}$ *binding*$_{opt}$ *compound-statement*

type-specifier: one of

  `void char short int long float double`

  `signed unsigned` *struct-or-union-specifier enum-specifier typedef-name*

  `fiber source target high medium low userbound`

statement:

  labeled-statement

  expression-statement

  compound-statement

  selection-statement

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

iteration-statement

jump-statement

sequencer-statement

newbinding-statement

multiplicative-expression:

cast-expression

multiplicative-expression * *cast-expression*

multiplicative-expression / *cast-expression*

multiplicative-expression % *cast-expression*

multiplicative-expression @ *cast-expression*

unary-expression:

postfix-expression

++ *unary-expression*

-- *unary-expression*

unary-operator cast-expression

sizeof *unary-expression*

sizeof (*type-name*)

nocopy (*cast-expression*)

The following non-terminal symbols have been introduced in C-Eli.

binding:

< *declaration-list*$_{opt}$ *statement-list*$_{opt}$ >

sequencer-statement:

continue *identifier* (*argument-expression-list*$_{opt}$) ;

q_continue *identifier* (*argument-expression-list*$_{opt}$) ;

suspend ;

relinquish ;

remote_continue *primary-expression identifier* (*argument-expression-list*$_{opt}$) ;

reset ;

exit ;

poll ;

newbinding-statement:

newbinding *primary-expression* ;

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.