A Formal Model for Divide-and-Conquer and Its Parallel Realization

Zhijing George Mou

Research Report YALEU/DCS/RR-795

May 1990

A Formal Model for Divide-and-Conquer and Its Parallel Realization

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

by
Zhijing George Mou
May 1990

ABSTRACT

A Formal Model for Divide-and-Conquer and Its Parallel Realization

Zhijing George Mou

Yale University

1990

Divide-and-conquer (DC) has long been known to be an effective programming paradigm in sequential computing. More recently, its significance to parallel computation has also been noted; many efficient parallel DC algorithms have emerged. However, the notion of divide-and-conquer has not yet received the formal treatment it deserves. This has precluded recognition of the common structure and intrinsic constituents of many DC algorithms, as well as the definition of a parallel programming environment that supports the development of DC algorithms.

This dissertation contains the results of a research effort aimed at solving the problems above. I present (1) an algebraic model for divide-and-conquer called *pseudomorphism*, which permits DC algorithms to be designed by studying the algebraic properties of the problems; (2) a programming notation called *Divacon* which allows DC algorithms to be specified by a small set of primitives and functional forms in a way that is concise, hierarchical, and highly modular; (3) a collection of applications programs based on the formal DC model; (4) a tool developed for the analysis of Divacon programs; and (5) a prototype implementation of the model on the Connection Machine.

The DC model leads to the definition of two parallel programming constructs called *PDC* and *SDC*. Their expressiveness is demonstrated by several examples, including polynomial evaluation, matrix multiplication, sort, Gaussian elimination, and the solution of triangular systems. Furthermore, these two parallel programming constructs are powerful enough to subsume many other well-known parallel constructs such as broadcast, reduction, and scan.

Acknowledgments

I would first like to thank my advisor, Paul Hudak, for his generous support and invaluable guidance. I would also like to thank the other members of my committee, David Keyes and Bill Gropp, for their careful and timely reading of this document.

Financial support for this work was provided in part by the Department of Energy through grant #FG02-86ER25012 and in part by the National Science Foundation through grant #CCR-8809919 to Yale University.

I am very indebted to Alan Perlis. He was on my dissertation committee until his untimely death at the age of 67 in February, 1990. His insight and guidance during the dissertation research were invaluable. He also helped me to see the implication of this work and its connections to that of others.

This work benefited from the insight into parallel computation that I obtained from Marina Chen in its early stages. It also benefited from many helpful discussions with my officemate, Steve Anderson.

I would like to thank the Brandeis Computer Science faculty and staff for their interest and support; special thanks go to Jacques Cohen, Timothy Hickey, and Myrna Fox. I am also thankful to the graduate students in CS160 at Brandeis who have been contributing to the further development of this work.

Finally, I would like to thank my wife Leslie and my son Eli for their love and support. I am most grateful to my father Chuanwen and my mother Shuxiou, who provided much inspiration; this dissertation is dedicated to them.

Contents

1 Introduction **1**

1.1 Objective 1

1.2 Overview 3

1.2.1 Organization 3

1.2.2 Summary 4

1.3 Suggestions on Reading this Dissertation 20

2 Space Domain Algebras **23**

2.1 Spaces and Space Domains 23

2.2 Operations over Space Domains 26

2.2.1 Divide Operations 26

2.2.2 Combine Operations 30

2.3 Operation Induced Domain Algebras 32

3 Arrays and Array Operations **35**

3.1 Tuples 35

3.2 Index Sets 37

3.2.1 Basic Notions 37

3.2.2 Normalization 37

3.3 Arrays 38

3.3.1 Basic Notions 38

3.3.2 Vectors and Matrices 40

3.3.3 The Relative Indexing Scheme 41

3.4 Divide Operations 43

3.4.1 Basic Notions 43

3.4.2 Balanced Divisions 45

3.4.3 Dividing Higher Dimensional Arrays 47

3.4.4 Dividing Multiple Arrays 49

3.5 Combine Operations 51

3.6 Universal Array Operations 53

3.6.1 Distribution 54

3.6.2 Construction 55

3.6.3 Communication Functions 56

3.6.4 Functional Forms 60

3.6.5 Zip and Unzip 61

3.7 Divacon Notation 62

4 Parallel Divide-and-Conquer **65**

4.1 Pseudomorphism: the Basic Models 65

4.1.1 Morphisms 65

4.1.2 Postmorphisms 67

4.1.3 Premorphisms 69

4.1.4 Pseudomorphisms 70

4.2 Computation of Pseudomorphisms 72

4.2.1 Some Related Notions 72

4.2.2 Higher Order Function PDC 74

4.2.3 First Example – Postmorphism PDC Algorithm for Scan . . . 76

4.2.4 Illustrations 78

4.3 Case Studies 80

4.3.1 Broadcast 80

4.3.2 A Better Scan Algorithm 83

4.3.3 Polynomial Evaluation 84

4.3.4 Matrix Multiplication 86

4.3.5 Monotonic Sort – A Second Order PDC 89

5 Sequential Divide-and-Conquer **91**

5.1 Basic Notions 91

5.2 The Programming Construct SDC 92

5.2.1 SDC 92

5.2.2 An Example 94

5.3 Case Studies 95

5.3.1 Gaussian Elimination 95

5.3.2 Linear Lower Triangular System 98

5.4 Parallelism in SDC Algorithms 101

5.5 Balanced Division in SDCs 103

6 Complexity Analysis **107**

6.1 Communications 107

6.2 Non-Recursive Function Forms 111

6.2.1 The Time Complexities 111

6.2.2 Parallel and Sequential Forms 116

6.3 Divide and Combine Operations . . . 117
6.4 Recursive Function Forms . . . 118
6.4.1 PDC . . . 119
6.4.2 SDC . . . 121
6.5 Case Studies . . . 122
6.5.1 Scan . . . 123
6.5.2 Monotonic Sort . . . 127
6.5.3 Linear Lower Triangular System . . . 127
6.6 Processor Complexity . . . 128
6.6.1 Basic Notions . . . 128
6.6.2 Dynamic Division . . . 129
6.6.3 Base Functions . . . 131
6.6.4 DC Algorithms . . . 132
6.7 Epilogue . . . 133

7 Implementation on the CM **135**

7.1 Preliminaries . . . 137
7.1.1 Bit Operations . . . 137
7.1.2 Flat Vectors . . . 138
7.1.3 Representation of Recursive Arrays . . . 139
7.2 Left-Right Divide and Combine Operations . . . 141
7.3 Communication . . . 143
7.4 General Cases . . . 146
7.5 Complexity . . . 147
7.6 Optimization of PDC Algorithms . . . 148
7.7 Implementation and Performance on the CM . . . 149

8 Conclusion **153**

8.1 Ending Remarks 153

8.2 Limitations 155

8.3 Related Work 156

8.4 Future Work 160

A Benchmark Graphs **163**

List of Figures

1.1 Division of a linear triangular system 13

1.2 Effect of left-right division on binary indices 19

2.1 An example of a division tree 29

3.1 Some constructed divisions over matrices 49

3.2 Communications over a pair of vectors of the same size 58

4.1 Different types of pseudomorphisms (k=2) 71

4.2 A Divacon graph for scan 79

4.3 A simplified Divacon graph for Scan 80

4.4 Premorphism PDC algorithm for reverse 81

4.5 Premorphism PDC algorithm for broadcast 82

4.6 Postmorphism PDC algorithm for broadcast 83

4.7 Scan PDC algorithm without broadcast 84

4.8 Three matrix block multiplications 87

4.9 Monotonic sort – a second order PDC algorithm 90

5.1 Division of a linear triangular system 98

5.2 Unbalanced division of a linear triangular system 104

6.1 Locality of correspondent and mirror-image communications 110

6.2 Processor complexity of the matrix multiplication PDC algorithms . . 132

6.3 Time and processor complexities of DC algorithms 134

A.1 Broadcast by Divacon on the CM 164

A.2 Scan by Divacon on the CM 165

A.3 Polynomial evaluation by Divacon on the CM 166

A.4 Monotonic sort by Divacon on the CM 167

A.5 Linear triangular systems by Divacon on the CM 168

A.6 Recursive array operations on the CM 169

A.7 Communications over recursive arrays on the CM 170

Chapter 1

Introduction

1.1 Objective

Divide-and-conquer (DC) is a well-known strategy with a long history in the area of computation. More than two thousand years ago, the algorithm Archimedes used to find the value of π to lie between 3.1415924 and 3.1416016 is essentially a DC algorithm [21]. Many of the important algorithms added to the computing literature in this century are also based on divide-and-conquer, for example, Danielson and Lanczos's fast Fourier transform (FFT) algorithms (1942) and Strassen's matrix multiplication (1969). Aho *et al.*, in their book *The Design and Analysis of Algorithms* [3], identified divide-and-conquer as an effective programming paradigm and illustrated its power by giving DC algorithms for a wide range of problems including integer multiplication, sort, search, matrix algorithms, and Chinese remaindering.

Since the late 60s, computer scientists have been facing the challenge of exploiting the enormous computation power offered by parallel computers. Due to rich parallelism and the machine-independent nature implicit in the notion of divide-and-conquer, DC algorithms originally intended for sequential computers can be adopted on parallel computers, and yield high performance in most cases [43, 54, 17]. New

DC algorithms designed for parallel computation are also emerging. Some of the examples are Batcher's bitonic sort (1968) [6], Ladner and Fischer's prefix (1980) [31], and Wang's tridiagonal equations (1981) [56]. The observation has been made by many [43, 40, 20, 36] that divide-and-conquer is an effective paradigm not only for sequential but also for parallel computation.

Despite all this, divide-and-conquer has so far been an informal notion without a mathematical identity. In the literature, divide-and-conquer is thus described, illustrated with examples, but not formally defined. The informal status of DC has been observed, for example, by Nelson and Snyder in [40],[1] and by Mou and Hudak in [36]. This situation is unsatisfactory. By Dijkstra's argument [15], we may say that divide-and-conquer so far can be only regarded as a craft rather than a discipline of study, owing to its lack of formality. Consequently, the full potential of divide-and-conquer has not been brought out because there is no basis for us to understand, teach, reason about, and manipulate divide-and-conquer algorithms as mathematical objects.

The thesis of this dissertation is that divide-and-conquer is a good model for concurrency – conceptually, theoretically, and pragmatically. To support this, we will present:

- a formal model of DC which permits DC algorithms to be designed by studying the algebraic properties of the problems;
- a small set of primitives and functional forms by which DC algorithms can be specified hierarchically, concisely, and with enhanced modularity;
- a collection of applications of the model to a variety of problems, including scan, polynomial evaluation, matrix multiplication, triangular system solution, and sorting;

[1] Nelson and Snyder, in fact, expressed their doubt about the possibility of formalizing programming paradigms such as divide-and-conquer.

- a tool that can be applied to DC algorithms in a systematic way to derive the time and processor complexity of DC algorithms, and show that the applications presented in this dissertation are optimal or sub-optimal in some well-defined sense;

- a prototype implementation of the model on the Connection Machine (CM) [52] that suggests a class of parallel programming languages which are high-level, easy to use, and efficient.

It should be pointed out that the proposed DC model is powerful enough to subsume most known parallel programming constructs such as broadcast, reduction, scan, sort, inner product, and FFT.

1.2 Overview

1.2.1 Organization

The dissertation is divided into eight chapters:

- Chapter 1: Introduction.

- Chapter 2: This provides a general theoretical foundation for the rest of the dissertation. The concept of *space* is introduced, and the *divide* and *combine* operations over spaces are defined. We also point out how divide operations introduce algebra into a space domain.

- Chapter 3: The general theory of previous chapter is applied to a particular domain – arrays. In addition to divide and combine operations, we also study other operations over arrays, and functional forms which allow complex operations to be specified in terms of simple operations. The notation introduced in this chapter will be referred to as *Divacon* notation and used throughout this dissertation.

- Chapter 4: We derive the algebraic model for divide-and-conquer by generalizing the notion of morphism to *pseudomorphism.* The expressiveness of the model is demonstrated through detailed examples. DC algorithms based on this model are called *parallel divide-and-conquer* (PDC) because the recursive application of the computed function over the substructures can be computed in parallel.

- Chapter 5: The model for *sequential divide-and-conquer* (SDC) algorithms – those in which the recursive function application over the substructures must be computed sequentially – is introduced. We show that despite the apparent sequential nature, SDC algorithms can gain substantial speedup with excellent efficiency. The source of parallelism in SDC algorithms is investigated and compared with that of PDC algorithms.

- Chapter 6: Tools for time and processor complexity analysis of DC algorithms are developed. The effectiveness of the tools is illustrated with complete examples. The performance of DC algorithms appearing in the dissertation is given.

- Chapter 7: The key issues of the parallel implementation of the DC model are studied, and a prototype CM implementation is presented. Benchmarks of the primitive operations and several DC algorithms are also presented.

- Chapter 8: Conclusion. Limitations of the model and future work are discussed, and we comment on related work.

1.2.2 Summary

In the following, we give a detailed summary for each of the chapters of the dissertation excluding the introduction and conclusion. The purpose of doing so is three-fold: first, it gives readers an opportunity to grasp the most important ideas behind the work without having to go through the dissertation; second, it helps to motivate the formal notations, algorithms, and proofs in the chapters; finally, it helps readers to understand the contents of the dissertation since the exposition in the summary involves less formalism and more intuition than that in the chapters.

Chapter 2, Space Domain Algebras

A formal notion for divide-and-conquer calls for a formal notion for the divide operation, which in turn calls for a formal notion for the objects of the operation. In this dissertation, the general model for the objects is the *space.*

A space is no more than a set, called *universe*, with *relations* over the set. A graph, for example, is a space where the universe is the set of its vertices; there is only one binary relation over the universe, which is called the set of its edges. Viewing functions as special cases of relations, we can also say that all algebraic structures such as lattice, groups, and rings are spaces.

The notion of space allows us to define the *subspace* relation over spaces. One space A is a subspace of another B if A's universe is a subset of B's universe and each relation of A is a subrelation of the correspondent relation of B. By this definition, for example, a graph G_1 is a subspace of another G_2 if and only if G_1 is a subgraph of G_2 by graph theory. With the notion of subspace, the *divide* operation is defined. The *combine* operation can then be defined as the inverse of the divide operations.

Since we can talk about the sizes of the spaces, which are the cardinality of their universes, we can compare the size of a space with the sizes of its subspaces generated by a divide operation. Accordingly, divide operations can be classified as *balanced* or *unbalanced.* The recursive application of a divide operation to a space yields its division tree, which has logarithmic height if the divide function is balanced.

A *space domain* is a collection of spaces. In the end of Chapter 2 we make an important observation that the divide operation introduces an *algebra* into a space domain, because a domain with a combine operation is an algebra and a combine operation is determined given a divide operation.

Chapter 3, Arrays and Array Operations

After having laid out a general theoretical foundation, we focus our attention on a particular type of space - arrays. We will develop a formal notion for arrays under the framework of space, define a few divide functions over vectors, show how divisions over higher dimensional arrays and multiple arrays can be defined in terms of divisions over vectors, and present combine operations as the inverse of divisions.

Divide and combine functions take arrays apart and put them together without ever changing the indexed values of their entries. Apparently, for most applications, we would also need operations with the opposite nature. These operations manipulate the indexed values of the array entries but preserve the array structures. To emphasize the difference, we call divide and combine *relational operations*, while others are called *universal operations.*

The universal operations over arrays can be in turn divided into two categories with orthogonal nature. A *local* operation requires no communication between the array entries whereas a *communication* operation performs only inter-entry communication (without local computation). The separation between the two types allows universal operations over arrays of either type to be specified with a function, called a *generator*, defined over the array's entries: for local operations the generator can decide its new value by applying the generator to its old index value; for communication operations each entry can decide its communication partner by applying the generator to its own index.

With the above "primitive" operations defined, we next study functional forms which can be used to construct more complex operations from simpler ones. Most of the functional forms discussed, with possibly different names and unusual notions, are familiar to the readers, e.g., function composition, if-then-else, and map a function over a tuple (like the map in Lisp over a list). There are, however, a couple of unusual forms, including *filter* and what we call *sequential distribution over tuples.* The reason for introducing the latter will become clear in Chapter 5.

A few issues addressed in this chapter deserve separate elaboration here:

Polymorphism: A divide function over arrays is said to be *polymorphic* if it can be defined in terms of a partition over its index set. To put it another way, a polymorphic divide function is independent of the indexed values of the arrays. The *head-tail*, *left-right*, and *even-odd* divisions are all polymorphic while the division used in quicksort is not. The distinction is important not only because polymorphic and non-polymorphic divisions are completely different mathematical objects but also because the former can be computed on parallel computers efficiently whereas the latter cannot. This dissertation studies exclusively divide-and-conquer with polymorphic divisions.

Normalization: An array is normalized if its indices start from zero (or tuples of zeros) and are consecutive. Normalized arrays are preferable to non-normalized counterparts for the easy grasp of their index domains. However, we have to deal with non-normalized arrays because they are generated by divide operations even if the operand array is normalized. For example, the right subvector produced by the left-right division does not start with index zero; neither of the subvectors produced by an even-odd division is consecutive. Normalization is the bijection that maps a non-normalized array to its normalized counterpart.

Relative indexing scheme: It simply states that an operation over array(s) is to be performed (at least conceptually) by first normalizing the array(s), then applying the operation, and finally performing the inverse of the normalization. This scheme greatly simplifies the specification of communications in DC algorithms. For example, the frequently used communication pattern called *correspondent* is one during which each array entry communicates with the entry in the sibling subarray with the same relative index while the array is recursively divided. Without the relative indexing scheme, the communication generator would be a function that takes the level of a division as a parameter; with the relative indexing scheme, the generator is simply the identity function. Other common communication patterns, including *broadcast* and *mirror-image*, also can be specified with very simple generators under the relative indexing scheme.

Divacon notation: As Dijkstra pointed out in [15], a programming language is not merely a set of notations, it represents, and even forces us to follow, a particular way of thinking. The way of thinking implied by the notion of divide-and-conquer is not well reflected in any existing programming languages. This can be clearly seen by noting that arrays cannot even be divided in these languages. It is therefore necessary to introduce a new set of notations for DC algorithms.

While defining arrays and the auxiliary data types including tuples, index sets, and structures, primitive operations over the data types, and combining forms used to construct more complex operations, we also define a set of notations for the types, primitives, and functional forms. This set of notations is called the *Divacon notation*, which allows DC algorithms to be written in a way that is highly modular and concise.

Chapter 4, Parallel Divide-and-Conquer

In this chapter we present the algebraic model for divide-and-conquer, show how it can be computed, and demonstrate its expressive power with examples.

The embryo of our model is the *morphism* (also called *homomorphism*) [13, 16, 27]. A function f from one algebra (a set closed under an operation) $(X, +)$ to another $(Y, *)$ is a morphism if

$$f(x_1 + x_2) = (f\ x_1) * (f\ x_2)$$

Let ":" denote function composition, $!f(x_1,\ x_2) = (f\ x_1,\ f\ x_2)$. Then the above can be rewritten as

$$f : +(x_1, x_2) = * : !f(x_1, x_2)$$

which can be simplified to

$$f : + = * : !f$$

The morphism captures the notion of divide-and-conquer in that it reduces the function application over one compound argument to multiple applications over the sub-arguments. Many, if not most, mathematical operations with which we are familiar are morphisms with respect to certain algebras, for example, logarithm, differentiation, and Fourier transform. With parallel computation in mind, we are however more interested in morphisms with respect to array domain algebras induced by polymorphic divisions. An example is the *reduce* function as defined in APL, which takes a binary associative operator $\oplus$ and a vector as arguments, and returns the sum with respect to $\oplus$ of all the vector entries. The function $(reduce\ \oplus)$ is a morphism since the following can be easily verified:

$$(reduce\ \oplus) : c_{lr} = \oplus : !(reduce\ \oplus)$$

where c_{lr} denotes the left-right combine operation. For example,

$$\begin{aligned} & (reduce\ +) : c_{lr}\ ([1\ 2],\ [3\ 4]) \\ = \ & + : !(reduce\ +)\ ([1\ 2],\ [3\ 4]) \\ = \ & +\ ((reduce\ +)\ [1\ 2],\ (reduce\ +)\ [3\ 4]) \\ = \ & +\ (3, 7) = 10 \end{aligned}$$

Many functions are not morphisms, or at least they are not morphisms with respect to interesting algebras from the parallel computation point of view. The function *scan* as in APL, which takes a binary operation and a vector as arguments and returns a vector of which each entry is the partial sum of all the entries with smaller or equal indices, for instance, is not a morphism since

$$(scan\ \oplus) : c_{lr} \neq c_{lr} : !(scan\ \oplus)$$

However, if we introduce the following function h_{scan}, which, when applied to two vectors, will add the last entry of the first vector to each of the second, namely,

$$\begin{aligned}
&h_{scan} \oplus (v_1,\ v_2) = (v_1,\ v_2'),\\
&\quad \text{where } v_2'\ i = \oplus\ (v_2\ i,\ v_1\ (|v_1| - 1))\\
&\qquad (\text{where } |v_1| \text{ is the size of } v_1,\ |v_1| - 1 \text{ is the index of the last entry of } v_1)
\end{aligned}$$

then we will have

$$(scan\ \oplus) : c_{lr} = c_{lr} : h_{scan} : !(scan\ \oplus)$$

The presence of "!" above indicates that *scan* is subject to divide-and-conquer because one application of *scan* can be reduced to two over smaller arguments despite the fact that it is not a morphism. For example,

$$\begin{aligned}
& (scan\ +) : c_{lr}\ ([1\ 2],\ [3\ 4]) \\
= \ & c_{lr} : h_{scan} : !(scan\ +)\ ([1\ 2],\ [3\ 4]) \\
= \ & c_{lr} : h_{scan} : ((scan\ +)\ [1\ 2],\ (scan\ +)\ [3\ 4]) \\
= \ & c_{lr} : h_{scan}\ ([1\ 3], [3\ 7])) \\
= \ & c_{lr}\ ([1\ 3],\ [6\ 10]) = [1\ 3\ 6\ 10]
\end{aligned}$$

The function *scan* is an example of what we call a *postmorphism.* A function f from algebra (X, c_x) to algebra (Y, c_y) is a postmorphism if there exists a postadjust function $h : Y^2 \to Y^2$ such that

$$f : c_x = c_y : h : !f$$

Similarly, a function $f : X \to Y$ is a *premorphism* if there exists a preadjust function $g : X^2 \to X^2$ such that

$$f : c_x = c_y : !f : g$$

A simple example of premorphisms is the inverse operation over a vector, where g will simply exchange the two subvectors, assuming the combine operator is c_{lr}.

A *pseudomorphism* is a function f that has both a postadjust function h and a preadjust function g, such that

$$f : c_x = c_y : h : !f : g$$

Clearly, morphism, postmorphism, and premorphism are all special cases of pseudomorphism, where one or both of the adjust functions happen to be the identity function. At this point, it may be helpful to look at Figure 4.1 where the four different types of morphisms are depicted.

After the concept of pseudomorphism is studied in Section 4.1, we show in Section 4.2 how pseudomorphisms can be computed in terms of their constituents. Quite obviously, the recursive behavior of a pseudomorphism has been determined by the equation defining the pseudomorphism; the computation will be fully specified if we further specify the base case of the recursion. It follows that a pseudomorphism can be computed if we can determine its divide, combine, preadjust, postadjust, base predicate, and base functions. The higher order function *PDC*, standing for "*parallel divide-and-conquer*", is defined to take the above functional arguments in that order, and return a function that is equivalent to the pseudomorphism. For example, the function *scan* discussed above can now be defined as

$$scan \oplus = PDC\ (d_{lr},\ c_{lr},\ id,\ h_{scan} \oplus,\ atom?,\ id)$$

where *id* is the identity function, and *atom?* is the predicate that returns true for arrays of size one.

In Section 4.2, we also discuss how DC algorithms can be nested to yield higher order DC algorithms, and introduce some illustration schemes for DC algorithms so that we can "see" the computation of PDC with pictures.

In the last section, the concepts, notations, and illustrations previously developed are applied to a number of problems such as broadcast, polynomial evaluation, matrix multiplication, and monotonic sort. For each of the above problems, I give one or more divide-and-conquer algorithms in Divacon notation. All the algorithms are preceded by discussion in English; the intention is to show (informally) how the explicit knowledge of the pseudomorphism model can aid in the design of DC algorithms and how the model allows us to derive DC algorithms from the mathematical properties of the problems.

Chapter 5, Sequential Divide-and-Conquer

The essence of divide-and-conquer is the recursive reduction of a function application to two or more applications on the subarguments. Naturally, there may or may not be an order imposed over the recursive applications on the subarguments depending on the natures of the problems. A DC algorithm without the imposed order is said to be a *parallel divide-and-conquer* (PDC) because we can compute the recursive applications in parallel. A DC algorithm with the imposed order is said to be a *sequential divide-and-conquer* (SDC) because we must compute the multiple recursive applications one at a time.

To show that SDC algorithms are useful in applications, let us first look at two examples:

1. We have shown that *scan* is a postmorphism and therefore can be computed by PDC, in which *scan* is recursively applied to the left and right subvector in parallel. The PDC code is desirable for parallel computation, but would appear unnatural to an average Lisp programmer. If he is presented with the definition for scan, and asked to write a DC program for scan with left-right division, he is likely to write something corresponding to this:

$$scan\ v = c_{lr}\ (scan\ v_l,\ scan\ v_r')$$

where

$$v_r'\ i = (eq?\ 0\ i) \rightarrow (v_r\ i + v_l\ |v_l - 1|);\ v_r\ i$$

$$(v_l,\ v_r) = d_{lr}\ v$$

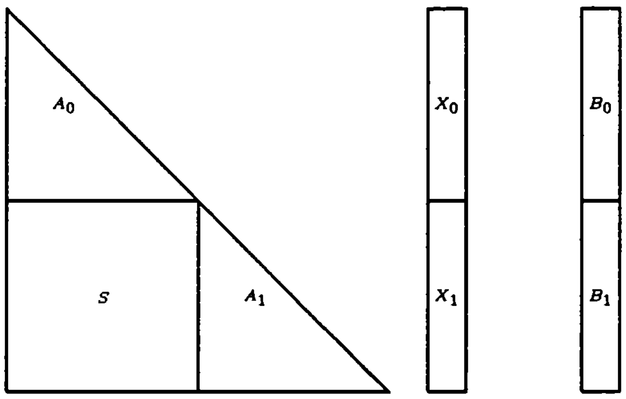


Figure 1.1: Division of a linear triangular system

where $p \longrightarrow e_1; e_2$ is the "if-then-else" construct.

From the sequential computation point of view, the above code is very efficient as only the first entry (compared to all in the case of PDC) of the right subvector is "adjusted" at each level of recursion. The key feature we should observe, however, is that the adjustment is made after the recursion over the left subvector and before the recursion over the right. It follows that these two recursions cannot be done in parallel.

2. Given a linear lower triangular system, we can solve it by dividing it into two half-sized lower triangular systems as shown in Figure 1.1.

   It is easy to show that the solution to the system is the catenation of the following systems

$$A_0X_0 = B_0$$
$$A_1X_1 = B_1 - SX_0$$

   Again, we should observe that the solution to the first subsystem (X_0) is used to adjust the second subsystem. And therefore, the first subsystem must be solved before the recursion over the second begins.

In this chapter, we introduce the notion of *crossmorphism* to model SDC algorithms, show how it can be computer by a higher order function, and demonstrate the usefulness of the model with some examples.

A number of interesting questions naturally arise:

- Given the linear order in SDC algorithms over the recursive applications, is it possible that more than one constituent can be computed in parallel at any moment of SDC computation?
- If no, is there any parallelism in SDC algorithms?
- If yes, what is the source of parallelism in SDC algorithms?
- Can the parallelism be effectively exploited, and how?
- Will the balance of division permit additional speedup in SDC algorithms?

Reflection on these questions reveals that a distinction in mathematics can be made between two types of parallelism: *multiplicative* and *additive*. Roughly speaking, the former corresponds to what others may call "expression level" or "control level" parallelism, the latter "data level parallelism" [23]. PDC algorithms can be shown to contain both types of parallelism; SDC algorithms only contain additive parallelism, which, however, still allows them to be computed on parallel computers with significant speedup in many situations.

Chapter 6, Complexity Analysis

In the previous two chapters, we have seen how divide-and-conquer algorithms can be developed under the formal DC models and expressed in Divacon notation hierarchically. The main purpose of this chapter is to provide tools of complexity analysis for such DC algorithms. By applying these general tools to specific algorithms, in the end of this chapter we will also be able to show that most DC algorithms in this dissertation are all optimal or nearly optimal in terms of their performance on parallel computers.

We start by studying the communication complexity. By doing so we can discuss complexity without involving the tedious specification of machine architecture and implementations. Although the communication time is affected by three factors, namely the size of message (size), the distance the message travels (locality), and the fan-in and fan-out of each communicating agent (pattern), we make an argument that the pattern is the most important factor to DC algorithms. The pattern of communication can be studied through its graph, which is decided by the set of communication agents and the send-receive binary relation over the set. Since communication in Divacon is given by functions, the fan-in of each node in a communication graph is at most one. We say a communication is *permutative* (*broadcasting*) if each node in the graph has $O(1)$ ($O(n)$) fan-out. The *correspondent* and *mirror-image* communications widely used in DC algorithms therefore are permutative. We will show how these two communications in fact have perfect locality on hypercube machines, therefore taking $O(1)$ time. Broadcasting communications can be computed, as we showed in Chapter 4, by divide-and-conquer with permutative communications in logarithmic steps, thus taking $O(\log n)$ time, where n is the fan-out of the communication.

It is easy to see that the complexity of a Divacon functional form can be given by the complexity of its constituents, which allows us to reduce the task of performing the analysis of a Divacon program to the analysis of its constituents. Formally, we can view the time analysis as a mapping from Divacon syntax to time (functions). Let us refer to this mapping as the *time complexity function.* The reducibility of the analysis to the constituents actually implies that the time complexity function is a morphism. The domain algebra of the morphism is the set of Divacon programs with the functional forms as operators, and the co-domain algebra is time (functions) with arithmetic operators.

For example, the time used by function composition is obviously the sum of the time used by the composed constituents. Formally, let $\mathcal{T}$ denote the time complexity function; then we have

$$\begin{aligned} \mathcal{T}[\![f : g\ x]\!] &= \mathcal{T}[\![f\ y]\!] + \mathcal{T}[\![g\ x]\!] \\ &\text{where } y = g\ x \end{aligned}$$

Observe that the above equation suggests that the function $\mathcal{T}$ is a morphism with respect to the operators ":" and "+". Also to be observed is that the above equation holds regardless whether the form is computed sequentially or in parallel.

For another example, consider the time complexity used by the array distribution $!f$, which applies f to each entry of the array. Assuming the array entries have constant size, and f takes constant time on constant sized arguments, then to compute $!f$ sequentially takes time linear to the size of the array, and to compute it in parallel takes only constant time. Let $\mathcal{T}_s$ and $\mathcal{T}_p$ denote the sequential and parallel time complexity functions respectively, and n the size of the array. The above discussion gives us:

$$\mathcal{T}_s[\![!f\ A]\!] = O(n)$$
$$\mathcal{T}_p[\![!f\ A]\!] = O(1)$$

We say that a functional form is a *parallel form* if substantial speedup ($Ts/Tp \neq O(1)$) can be expected when computed in parallel. Array distribution therefore is a parallel form while function composition is not. After giving the morphism equation for each functional form, we will see that the only other two non-recursive parallel forms besides array distribution are tuple distribution and communication generation.

The two DC forms in Divacon, namely PDC and SDC, are different from other forms due to their recursive nature. Take the PDC form, for example: by applying the complexity function to both sides of its definition we will obtain a recursive definition of complexity function for the PDC form. In other words, the recursive definition of the DC forms in the domain of Divacon programs results in the recursive definition of the complexity function of the DC forms in the domain of complexity.

The complexity of both PDC and SDC forms therefore can be analyzed by solving the recursive equations in the domain of complexity.

Our discussion about time complexity analysis is completed with a brief discussion on the time used by divide and combine functions, and a long section in which we illustrate with examples how we can apply the complexity function to DC algorithms and mechanically derive their time complexity. We will also be able to observe that polymorphic divide and combine operations in fact do not contribute to the time complexity of DC algorithms.

Processor complexity refers to the number of processors required by parallel algorithms under certain constraints. The constraint we impose is reflected by what we call the one-one norm, which enforces a one-one correspondence between the array entries and processors. It immediately follows that anti-polymorphic operations including communication and array distribution do not contribute to the processor complexity of DC algorithms because they do not alter the structure, nor the size, of arrays. Since all adjust functions in our DC algorithms are compositions of anti-polymorphic operations, we can focus our attention on divide, combine, and base functions for the purpose of processor complexity analysis.

In the end of this chapter, we list the time and processor complexities of all the DC algorithms in this dissertation and show that most DC algorithms are optimal or sub-optimal in the sense that their processor-time products are of the order of the best known sequential time, within a logarithmic factor.

Chapter 7, Implementation on the CM

In this chapter, we show the parallel realization of the proposed DC models by showing how the Divacon notation can be implemented with the parallel language *Lisp [53] on the CM [52].

From a data structure point of view, the essential differences between Divacon and *Lisp are:

- Parallel data structures in *Lisp are flat in the sense that they cannot be divided and combined, while Divacon arrays are recursive (non-flat);
- Communication in *Lisp must be specified in the flat global address space, while Divacon arrays allow communication to be specified by relative indexing into the recursively divided subarrays.

Divide-and-conquer is a form of recursion which calls for recursive data structures (with relative communication scheme). CM languages including *Lisp cannot express DC algorithms directly owing to the flatness of their parallel data structures. Because our DC algorithms can be decomposed into either recursive array operations or operations that are well supported in *Lisp, the key to a CM implementation of Divacon is the implementation of the recursive Divacon arrays.

Now let us first consider how to implement the left-right division over vectors. Since the division is polymorphic, it depends on and affects only the vector indices. Figure 1.2 illustrates with an example how the division maps each relative index to a new relative index while the vector is recursively divided. Observe that the new relative index is related to the old relative index for each vector entry in a simple way – the new one can be obtained from the old by masking off the most significant (leftmost) bit of their binary representations.

It is straightforward to see that the left-right combine operation involves an opposite mapping (unmasking) on the binary representations of the indices. It follows that assuming the vector entries are distributed over a set of processors each processor can decide the new relative index without communication with others under the left-right divide and combine operations. Therefore, these two operations can be computed in parallel in constant time by the manipulation of binary indices.

From the above discussion, we can see that the binary number of an index i at any stage of the divide and combine operations can be partitioned into three sections: Q, B, and R, where Q consists of the masked bits on the left, B consists of one bit,

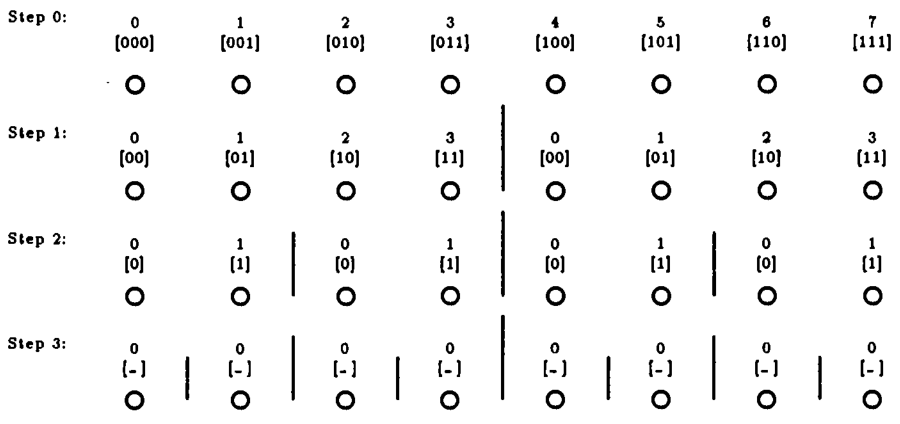


Figure 1.2: Effect of left-right division on binary indices

also called the dividing bit, which is just masked off by a divide operation or is about to be unmasked by a combine operation, and R consists of the bits on the right which represents the present relative index of the entry. We will write the above as $i = Q\hat{}B\hat{}R$. Now let V_l and V_r be two sibling subvectors, which were divided from, or will be combined into, one vector V, and let $i_l = Q_l\hat{}B_l\hat{}R_l$ and $i_r = Q_r\hat{}B_r\hat{}R_r$ be binary indices of two arbitrary entries from V_l and V_r respectively. Then we can prove the following two important equations: $Q_l = Q_r$ and $B_l = \neg B_r$, where "$\neg$" is the bit inversion operation.

Now let us consider how to support the relative indexing communication of Divacon. Since *Lisp provides communication by absolute indices, all we need to show is how relative indices can be translated to absolute indices at any stage of the divide and combine process. To be concrete, let us consider inter-vector communication with the left-right operations. Let X be an entry with the index whose binary representation is I, let Y be the communication partner of X with relative index whose binary representation is J. How can X figure out the absolute index of Y knowing only its relative index J? The two equations shown in the previous paragraph provide a simple solution: let $I = Q_x\hat{}B_x\hat{}R_x$, and J' be the absolute index of Y, then $J' = Q_x\hat{}\neg B_x\hat{}J$. Note that this translation uses only the information that the entry

X has in itself. It follows that the translation from relative indices to absolute indices for all vector entries can be computed in parallel locally in constant time.

In this chapter we also show how the above implementation strategy can be generalized to even-odd division and higher dimensional arrays, describe the representation of recursive arrays (by a number of flat parallel data structures), and give the algorithms which implement divide, combine, and index translation operations based on the ideas presented above.

Although the PDC construct is defined as a recursive process, we will show that it can be easily implemented, in a way transparent to programmers, with iteration, and therefore can be computed more efficiently than a naive implementation.

In the last section, we briefly describe the present implementation of Divacon on the CM which is accomplished by embedding Divacon in *Lisp. The benchmarks of divide, combine, correspondent communication, mirror-image communication, and broadcast communication together with the benchmarks for a number of DC algorithms are presented in the end. These benchmarks are good enough to sustain a claim that the basic recursive array operations can be computed in parallel in constant time and the application programs are asymptotically sub-optimal. In the next version of the implementation, the absolute time performance of the same algorithms is expected to improve by a factor of between four and ten.

1.3 Suggestions on Reading this Dissertation

Read this introduction! It is also important to read the summary for the chapters in the introduction carefully. To understand the significance, the limitations, and the position of the work, you may choose to read the conclusion before you start to tackle any other chapters. It may also be helpful to review the summary of a chapter before or while reading the chapter itself in order to keep a picture of the chapter in mind.

Chapters 2 and 3 can be thought of as preliminary for the contents in the chapters that follow. Roughly speaking, Chapter 2 introduces formal notions, and Chapter 3 introduces formal notations. You may first glance through the two chapters, and use them as a reference in reading the other chapters.

The chapter about sequential divide-and-conquer algorithms - Chapter 5 - is relatively independent of other chapters. It may be skipped in a self-contained reading.

Readers are assumed to be familiar with the basic notions used in functional programming such as *currying*, *higher order functions*, and *pattern-matching.* An introduction to functional programming can be found in [22, 1], while more recent development of the field can be found in [18, 41]. Familiarity with the notion *abstract interpretation* [2] is not assumed but will help in understanding the discussion on complexity analysis in Chapter 6.

Chapter 2

Space Domain Algebras

To understand divide-and-conquer, we must understand the operation *divide* first. This, in turn, cannot be achieved without a good understanding of the objects of the division.

In this chapter, we provide formal notions for objects, called *spaces*, and two operations over space domains, namely *divide* and *combine*. We also examine the relation between the two operations, and finally point out how the divide operations introduce algebras into space domains. Since our model for DC algorithms is based on the notion of *morphisms*, which are mappings from one algebra to another that preserve operations, the concepts introduced in this chapter will be crucial in later discussions.

2.1 Spaces and Space Domains

A *space* is a pair $S = (U, R)$, where U is a set called the *universe*, and R is a set of relations over U called the *relation*.[1] Since functions are special cases of relations,

[1] It should be pointed out that the concept of *space* here is essentially identical to that of *structure* used in predicate calculus [32], although they are studied from different angles and for different purposes.

they can also be members of space structures. It immediately follows that all common algebraic structures, such as posets, lattices, and groups, can be viewed as spaces with certain properties.

Abstract data structures and data structures found in programming languages can also be interpreted as spaces. Graphs in graph theory, for examples, can be viewed naturally as spaces. A graph G with vertex set V and edge set E is the space $G = (V, E)$, where E should be considered as a binary relation over V. Lists and trees in turn can be viewed as graphs with special properties. Vectors, matrices, and higher dimensional arrays can all be modeled by spaces; we will discuss this in the next chapter.

The *size* of a space $S = (U, R)$, denoted by $|s|$, is defined to be the cardinality of its universe $|U|$; similarly, the *arity* of the space is defined to be the cardinality of its relation. A space is *finite* if its size is a finite number; a space is *simple* if its arity is 1. If a space S is simple, then instead of writing $S = (U, \{r\})$ we may choose to write $S = (U, r)$ to improve the readability. In this dissertation, we will study exclusively finite and simple spaces.

Let $S = (U, r)$, $S' = (U', r')$ be two (simple) spaces. We say S is a *subspace* of S', written $S \sqsubseteq S'$, if and only if

$$U \subseteq U' \text{ and } r \subseteq r'$$

Since a k-ary relation r over a set U is no more than a subset of the Cartesian product U^k, one relation can be a subset of another. When a relation r is a subset of another r', we also say that r is a subrelation of r'. Thus, the above conditions mean that a subspace must have a subuniverse and a subrelation of the space of which it is a subspace.

By this definition, a subgroup of a group in algebra is a subspace of the group space; a subgraph of a graph is a subspace of the graph; a subtree of a tree is a subspace of the tree, and so on.

Let $S = (U, r)$ be a space, and $W \subset U$ its subuniverse. The set of subspaces of S with W as the universe, denoted by $S(W)$, is called *the subuniverse U' spanned subspace.* A partial ordering $\leq$ can be introduced to the set $S(W)$: for S' and S'' in $S(W)$,

$$S' \leq S'' \Leftrightarrow S' \sqsubseteq S''$$

Informally speaking, although all the subuniverse spanned spaces have the same size, the subspaces that are greater by this ordering have richer structures than the smaller subspaces.

Let r be a k-ary relation over a set S (therefore a subset of S^k), and S' a subset of S. The *restriction* of the relation r to the subset S', denoted by $r|'_S$, is [33, 16]

$$r|'_S = r \cap S'^k$$

Now it is easy to verify that the *maximum subspace* of $S(W)$, where $S = (U, r)$, is

$$S_{max} = (W,\ r|_W)$$

This maximum subspace S_{max} is also called the *subuniverse W induced subspace* of the space S.

Given two spaces $S = (U, r)$, $S' = (U', r')$, where r and r' are k-ary relations, we say S and S' are isomorphic to each other if there is a bijection $\phi : U_1 \rightarrow U_2$, such that

$$(u_0, ..., u_k) \in r \Leftrightarrow (\phi\ u_0, ..., \phi\ u_k) \in r'$$

For example, two lists are isomorphic if and only if they have the same length; two graph spaces are isomorphic to each other if and only if they are isomorphic graphs by graph theory [14].

A *space domain*, or simply *domain*, is a set of finite or infinite spaces with certain common properties. Some space domains we will encounter in latter discussions are

$\mathcal{L}$: the domain of lists.

$\mathcal{B}$: the domain of binary trees.

$\mathcal{A}$: the domain of arrays.

$\mathcal{V}$: the domain of vectors.

$\mathcal{M}$: the domain of matrices.

A space domain $\mathcal{S}$ is a subdomain of another $\mathcal{S}'$ if $\mathcal{S} \subset \mathcal{S}'$. For example, $\mathcal{L}$ and $\mathcal{B}$ in the list above are subdomains of the graph domain. $\mathcal{V}$ and $\mathcal{M}$ are subdomains of $\mathcal{A}$. Vector, matrix, and array space domains will be studied in detail in Chapter 3.

We define the special space *nil* to be $(\{\}, \{\})$. In other words, the space *nil* has empty universe and empty relation. It follows from the definition that *nil* is a subspace of any space. Moreover, we define it to be a member of any space domain.

2.2 Operations over Space Domains

2.2.1 Divide Operations

Basics

Let $\mathcal{S}$ be a space domain. A mapping $d : \mathcal{S} \rightarrow \mathcal{S}^k$ is a k-ary divide function over $\mathcal{S}$ if, whenever $d\ S = S_0, \ldots, S_{k-1}$, where $S, S_0, \ldots, S_{k-1} \in \mathcal{S}$, $S = (U, r)$, $S_i = (U_i, r_i)$, for $i = 0, \ldots (k-1)$, we have

1. $\{U_1, \ldots, U_k\}$ is a partition of U.
2. $S_i = S|_{U_i}$.

It follows from the definition that

Proposition 2.1 *Let d be a divide function on domain $\mathcal{S}$, $S \in \mathcal{S}$, $d\ S = S_0, \ldots S_{k-1}$, $S = (U, R)$, $S_i = (U_i, R_i)$ for $i = 1$ to k. Then*

1. $S_i \sqsubseteq S$ for $i = 0$ to $(k-1)$ (Subspace).

2. (a) $U_i \bigcap U_j = \emptyset$ if $i \neq j$ (Disjoint).

   (b) $U_0 \bigcup \cdots \bigcup U_{k-1} = U$ (Complete).

3. *Let* $S_i' = (U_i', R_i')$, where $U_i' = U_i$, $S_i \sqsubseteq S$. *Then* $S_i' \sqsubseteq S_i$ (Maximum).

Proof*:*

1. *Since* $s_i = s|_{U_i}$.

2. *Since* $\{U_1, \ldots, U_k\}$ *is a partition of* U.

3. *Since subuniverse induced subspaces are maximum.*

Q.E.D.

By considering *nil* as a special element of the domain S^k, we will allow it to be returned by divide functions. A space s is *dividable* by d if $d\ s \neq nil$, otherwise the space is *undividable.* A divide function d over $\mathcal{S}$ thus partitions $\mathcal{S}$ into two subdomains, the *dividable subdomain* $\mathcal{S}_d$ and the *undividable subdomain* $\overline{\mathcal{S}_d}$ defined by

$$\mathcal{S}_d = \{S \mid S \in \mathcal{S},\ d\ S \neq nil\}$$
$$\overline{\mathcal{S}_d} = \mathcal{S} - \mathcal{S}_d$$

As an example, let us consider the following function defined over lists

$$dl_{ht}\ () = nil$$
$$dl_{ht}\ (a) = nil$$
$$dl_{ht}\ (a_0, a_1, \ldots, a_{n-1}) = ((a_0),\ (a_1, \ldots, a_{n-1}))$$

The function dl_{ht} can be verified to be a divide function over the domain of $\mathcal{L}$. It divides a list L with size greater than one into two sublists; the first sublist consists of the first element of L, the second consists of the rest. Lists of size equal or smaller than one are not dividable, and are in the undividable subdomain $\overline{\mathcal{L}_{dl_{ht}}}$.

Note that the divide function dl_{ht} resembles, but is not the same as, the "car-cdr" or "head-tail" division over lists commonly found in functional programming languages [22, 41]. The essential difference is that the "head" returned by the latter is the first element, not the list consisting of the first element, and thus is not a divide function over $\mathcal{L}$ by our definition.

Division Tree

It should be clear that the subspace relation $\sqsubseteq$ is a partial ordering over a space domain $\mathcal{S}$. A divide function d over $\mathcal{S}$ introduces another partial ordering $\prec$ over $\mathcal{S}$, called the *descendant relation.*

Let d be a divide function over $\mathcal{S}$, S and $S' \in \mathcal{S}$. We say S' is a descendant of S, written $S' \prec_d S$ if and only if

1. $d\ S = (S_0, \ldots, S_{k-1})$ and $S' = S_i$ for some $0 \leq i < k$.

2. or, $S' \prec_d S_i$ for some $0 \leq i < k$.

It is easy to verify that $\prec$ indeed is a well-defined partial ordering of $\mathcal{S}$. Moreover, the relation $\prec$ is a stronger one than that of $\sqsubseteq$, in that the former implies the latter but not vice versa:

$$S' \prec_d S \Rightarrow S' \sqsubseteq S$$

Given a divide function d over $\mathcal{S}$, a space $S \in \mathcal{S}$, we can identify all the descendants of S in its *descendant set* defined by

$$S_{\prec_d} = \{S' \mid S' \in \mathcal{S},\ S' \prec_d S\}$$

A simple but important fact we can derive about the poset $(S_{\prec_d}, \prec_d)$ is

Proposition 2.2 *If d is a k-ary divide function over $\mathcal{S}$, $S \in \mathcal{S}$, and the descendant set $S_{\prec_d}$ is not empty, then the* Hasse diagram *[16] of the poset $(S_{\prec_d}, \prec_d)$ is a k-ary tree. Moreover, the root of the tree is S, and the leaves are undividable spaces of $\mathcal{S}$.*

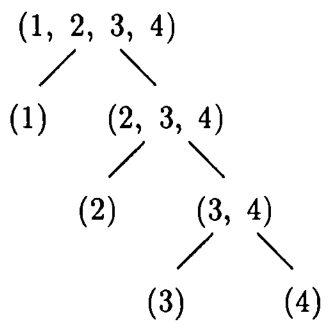


Figure 2.1: An example of a division tree

The tree in above is called the *division tree* of space S with respect to divide function d. The division tree of a space can be explicitly constructed by recursively dividing the space until all the leaves are undividable.

As an example, the division tree of the list (1, 2, 3, 4) with respect to the divide function dl_{ht} is given in Figure 2.1.

Balance

Let d be a divide function over $\mathcal{S}$. We say d is *balanced* if for any dividable space $S \in \mathcal{S}$, there exists a constant $m > 1$ such that

$$d\ S = (S_0, \ldots, S_{k-1}) \ \Rightarrow\ |S_i| \leq |S|/m,\ i = 0 \text{ to } (k-1)$$

The constant m above is called the *division factor* of the divide function d. It follows immediately from the definition that

Proposition 2.3 *Let d be a balanced division over $\mathcal{S}$ with division factor m, S a dividable space in $\mathcal{S}$, and H the height of the division tree of S with respect to d. Then $H \leq \log_m |S|$.*

A divide function which is not balanced is *unbalanced*. An unbalanced division d over $\mathcal{S}$ is *decremental* if for any dividable space $S \in \mathcal{S}$, there exists a constant integer $b > 1$ such that

$$d\ S = (S_0, \dots, S_{k-1}) \ \Rightarrow\ |S_i| \geq |S| - b, \quad \text{for some } 0 \leq i < k$$

The constant b above is called the ***division subtrahend*** of the divide function d. It follows immediately from the definition that

Proposition 2.4 *Let d be a balanced division over $\mathcal{S}$ with division subtrahend b, S a dividable space in $\mathcal{S}$, and H the height of the division tree of S with respect to d. Then $H \leq |S|/b$.*

The head-tail divide function dl_{ht} of the last section, for example, is decremental unbalanced division with subtrahend one. We will study some of the balanced divisions over arrays in Chapter 3; all of them have division factors of powers of two.

Both balanced and unbalanced division play important roles in parallel DC algorithms. However, balanced divisions generally yield better parallel time performance, as we will see in later chapters.

2.2.2 Combine Operations

Let $\mathcal{S}$ be a space domain. A mapping $c : \mathcal{S}^k \rightarrow \mathcal{S}$ is a k-ary combine function over $\mathcal{S}$ if whenever $c(s_0, \dots, s_{k-1}) = s$, where $S, S_i \in \mathcal{S},\ s_i = (U_i, R_i)$ for $i = 1$ to $k, S = (U, R),\ s \neq nil$, we have

1. $\{U_1, \dots, U_k\}$ is a partition of U.
2. $S_i = S|_{U_i}$, for $i = 1$ to k.

When *nil* is returned by a combined function, the k spaces $(S_0, \dots, S_{k-1})$ are said to be *incompatible*.

As an example, consider the following *append* operation over lists as defined in Lisp

$$
\begin{aligned}
&\mathit{append}\ (L_0, L_1) = L \\
&\quad \text{where} \\
&\qquad L = (a_0, \ldots, a_{m-1}, a_m, \ldots, a_{n-1}) \\
&\qquad (a_0, \ldots, a_{m-1}) = L_0 \\
&\qquad (a_m, \ldots, a_{n-1} = L_1
\end{aligned}
$$

It should be easy to verify that *append* is a binary combine function over the space domain of $\mathcal{L}$.

Observed that combine functions have functionality opposite to that of divide functions. It follows that a combine function can be the inverse of some divide function on the same domain. Formally, let d and c be respectively a divide and a combine function for the domain $\mathcal{S}$. The combine function c is a *left inverse* of the divide function d if and only if for all dividable spaces in $\mathcal{S}$, we have

$$c : d = id$$

where ":" denotes function composition, "*id*" denotes the identity function.

For example, the combine function *append* is a left inverse of the divide function dl_{ht} of Section 2.2.1, since

$$\mathit{append} : dl_{ht}\ L = L$$

for any L in the dividable subdomain $\mathcal{L}_{dl_{ht}}$.

It is well known that a function has a left inverse if and only if it a is one-one mapping [33, 16]. And since the divide functions we study in this dissertation are all one-one mappings, their left-inverses always exist. However, since these divide functions are not onto mappings, their left-inverses are generally not unique [33, 16].

The combine function *append*, for instance, is not the only left-inverse of the divide function dl_{ht}. Consider the following combine function over list domain $\mathcal{L}$, which will only append two lists when the first list has exactly one element, and the second has one or more elements

$$cl_{ht}\ ((a),(a_1,\ldots,a_{n-1}) = (a,a_1,\ldots,a_{n-1}), \quad \text{where } n \geq 1$$
$$cl_{ht}\ (L_0,L_1) = nil$$

Like the function *append*, cl_{ht} is also a legitimate left inverse of the divide function dl_{ht}. These two left inverses are nevertheless different. In fact, the function cl_{ht} can be easily shown to be a subfunction of the function *append*.

Let d be a divide function over $\mathcal{S}$, and C the set of all its left-inverses, namely

$$C = \{d \mid c : d = id\}$$

We define the *least left inverse of d*, denoted by d^{-1}, to be the function that has the smallest graph.[2] This function can be uniquely determined by the intersection of all the function graphs of the left inverses:

$$d^{-1} = \bigcap_{c \in C} c$$

For example, ${dl_{ht}}^{-1} = cl_{ht}$. The least left inverse of a divide function contains no "arbitrary information" unrelated to the divide function. In later discussions, we may simply say "left inverse" to mean "the least left inverse".

2.3 Operation Induced Domain Algebras

First, let us review the very basic concept of *algebra*. An *algebra* is a set $\mathcal{S}$ and a set of finitary operations under which $\mathcal{S}$ is closed [48, 16]. For the purpose of this thesis, however, it is sufficient to consider algebras with only one operation. An algebra thus will be considered, throughout this dissertation, as a pair $A = (\mathcal{S}, \oplus)$, where $\mathcal{S}$ is a set,

[2]The graph of a function $f : X \rightarrow Y$ is the set of pairs of the form $\{(x,y) \mid x \in X,\ y = fx,\ y \neq nil\}$. See [45, 16].

and $\oplus$ is an operation with respect to which S is closed. In other words, if $\oplus$ is a k-ary operation, and $S_0, \ldots S_{k-1} \in \mathcal{S}$ are k elements of $\mathcal{S}$, then $S = \oplus(S_0, \ldots, S_{k-1}) \in \mathcal{S}$.

The arity of the operation of an algebra is also called the *arity* of the algebra. In this dissertation, we will encounter both binary and higher arity algebras.

It is easy to verify that a combine function over a space domain defines a domain algebra. Moreover, a divide function can introduce an algebra to a space domain through its left inverse. Formally,

Proposition 2.5

1. *If c is a combine operation over space domain $\mathcal{S}$, then $(\mathcal{S}, c)$ is an algebra.*
2. *If d is a divide function over space domain $\mathcal{S}$, then $(\mathcal{S}, d^{-1})$ is an algebra.*

The algebra $(\mathcal{S}, d^{-1})$ is called the divide function d induced algebra over $\mathcal{S}$. In Chapter 4, we will see the significance of the division induced algebras in our abstract model for divide-and-conquer.

Chapter 3

Arrays and Array Operations

In this chapter, we will apply the notions introduced in Chapter 2 to a particular space domain, namely arrays. Besides the divide and combine operations, which only manipulate the structure of arrays, we will also study a number of operations with exactly opposite nature, namely operations that only change the values of array entries but always preserve the structures. As we will see in later chapters, DC algorithms over arrays can generally be decomposed into these two orthogonal types of operations.

A number of related data structures, such as tuples, index sets, and structures will also be discussed. The notations introduced in this chapter will be used throughout the dissertation.

3.1 Tuples

A k-tuple has the form

$$\vec{x} = (x_0, \ldots, x_{k-1})$$

where x_i, $0 \leq i < k$ is said to be the ith *component* of $\vec{x}$.

A component of a tuple can be accessed by the *projection* function:

$$\mathit{proj}\ i\ \vec{x} = x_i$$

The value of a component can be mapped to a different value by the *injection* function:

$$\mathit{inj}\ i\ v\ \vec{x} = (x_0, \ldots, x_{i-1}, v, x_{i+1}, \ldots, x_{k-1})$$

Let $\vec{b} = (b_0, \ldots, b_{k-1})$ be a k-tuple of binary bits with m non-zero bits, and $\vec{x}$ a k-tuple as in above. We can derive from $\vec{x}$ and $\vec{b}$ a m-tuple which consists of only the components of $\vec{x}$, to which the corresponding bits in $\vec{b}$ are *ones*. This is done by the *select* function:

$$\mathit{sel}\ \vec{b}\ \vec{x} = \vec{y},\ \text{where}\ \vec{y} = \{x_i \mid b_i = 1, i = 0 \text{ to } (k-1)\}$$

Given k sets $U_0, \ldots, U_{k-1}$, their *Cartesian product* or simply *product* is a set of k tuples [16]

$$\begin{aligned} W &= U_0 \times \cdots \times U_{k-1} \\ &= \textstyle\prod_{i=0}^{k-1} U_i \\ &= \{(u_0, \ldots, u_{k-1}) \mid u_i \in U_i,\ \text{for } i = 0 \text{ to } (k-1)\} \end{aligned}$$

If $U_0 = U_1 = \cdots = U_{k-1} = U$, the product of the k set is designated by $W = U^k$.

Similarly, given k partially ordered sets (*posets*),

$$(U_0, \leq_0), \ldots, (U_{k-1}, \leq_{k-1})$$

their *direct product* [16] is the poset

$$W = (U_0 \times \cdots \times U_{k-1}, \leq)$$

where for $w, w' \in W$, $w \leq w'$ iff

$$\mathit{proj}\ i\ w \leq_i \mathit{proj}\ i\ w',\ \text{for } i = 0 \text{ to } (k-1)$$

We will often write U for a poset $(U, \leq)$ when the ordering is clear from the context.

3.2 Index Sets

3.2.1 Basic Notions

An index set is a direct product of subsets of integers. Formally, let N denote the set of all non-negative integers (with the usual ordering), $I_0, \ldots, I_{k-1} \subset \mathcal{N}$. An index set is a direct product with the form

$$\begin{aligned} I &= I_0 \times \cdots \times I_{k-1} \\ &= \prod_{j=0}^{k-1} I_j \end{aligned}$$

The cardinality of I_j, $m_j = |I_j|$ is called I's *size along the jth dimension*; the k-tuple $(P_0, \ldots, P_{k-1})$ is the *shape* of the index set; the product $m = \prod_{j=0}^{k-1} m_j$ is the *size* of the index set.

A k dimensional index set $I = \prod_{j=0}^{k-1} I_j$ is a *normalized index set* if for each $0 \leq j < k$, the set I_j has the form

$$I_j = \{i \mid 0 \leq i < P_j\}$$

In other words, each set I_j consists of all the integers between zero and P_j (excluding P_j itself).

3.2.2 Normalization

Non-normalized index sets lack the simplicity of normalized index sets, and generally it is harder for us to grasp their distribution pattern and scope. Fortunately, it is easy to transform a non-normalized index set to a normalized one: we first sort for each I_j, $0 \leq j < k$, into a sequence

$$I_j = (i_0^j, \ldots, i_{P_j-1}^j)$$

so that

$$i^j_m < i^j_{m+1}, \text{ for } 0 \leq m < (P_j - 1)$$

We can then use a simple mapping Γ_j, defined below, to map each integer of each set I_j to a new integer according its position in the sorted sequence:

$$\Gamma_j \; i^j_m = m$$

We can then define the mapping

$$\Gamma = (\Gamma_0, \ldots, \Gamma_{k-1}), \text{ where } \Gamma \; (i_0, \ldots, i_{k-1}) = (\Gamma_0 \; i_0, \ldots, \Gamma_{k-1} \; i_{k-1})$$

It should be observed that the normalization transformation Γ is a bijection for a given index set. Therefore, the inverse transformation Γ^{-1} can be performed.

For example, the index set $\{(2),(6),(4),(0)\}$ can be normalized to $\{(0),(1),(2),(3)\}$ by the normalization transformation $\Gamma \; i = i/2$, the inverse of the transformation is $\Gamma \; i = 2 * i$.

3.3 Arrays

3.3.1 Basic Notions

An *array* in abstract is a mapping $A : I \rightarrow U$, where I is its *indexing set*, and U is its *indexed set*. The *dimension*, the *shape* and the *size* of an array are respectively the dimension, shape, and size of its index set. We will occasionally talk about the *type* of arrays, which means the type of their indexed set. An array is *normalized* if its index set is normalized.

Some very basic operations defined over arrays include

Shape:	$\$A \Rightarrow$ shape of A.
Size :	$\$\$A \Rightarrow$ size of A.
Indexing :	$\$A\ \vec{i} \Rightarrow$ application of A to the index $\vec{i}$.

The *graph* of a function $f : X \rightarrow Y$, denoted by $\mathcal{G}\ f$, is the set of argument-value pairs of the form [16, 45]

$$\{(x, y) \mid x \in X,\ y = f\ x\}$$

Since arrays are functions, we can talk about *graphs of arrays.* Given an array $A : I_{\preceq} \rightarrow V$, its graph $\mathcal{G}A$ is

$$\{(\vec{i}, u) \mid x \in I,\ u = A\ \vec{i}\}$$

The partial order $\preceq$ over the index set I induces a partial order $\leq$ over the elements of the graph $\mathcal{G}A$, which is

$$\vec{i} \prec \vec{i'} \ \Leftrightarrow\ (\vec{i}, u) \leq (\vec{i'}, u')$$

An array $A : I_{\leq} \rightarrow V$ therefore can be viewed as a space of the form

$$A = (\mathcal{G}A, \leq)$$

where $\leq$ is the partial order over $\mathcal{G}A$ induced by the order over the index set I.

The elements in the graph of an array with the form $(\vec{i}, v)$ are called the *entries* of the array, where $\vec{i}$ and v are respectively called the *indexing value* and *indexed value* of the entry. We will sometimes refer to the "indexed value" of an entry simply by the "value" of the entry when the ambiguity can be resolved from the context.

A *subarray* A' of an array A is just a subspace of A, which we write $A' \sqsubseteq A$. By definition, if $A' \sqsubseteq A$, then $\mathcal{G}A' \subseteq \mathcal{G}A$. It follows that if an entry is in both the array A and a subarray $A' \sqsubseteq A$, then the entry must have the same indexing and indexed values in A and A'.

By making the partial order over the graph of an array implicit, the array can be represented by the graph alone. We therefore can informally identify an array space with its graph. As we will see in Chapter 7, the graph of an array is closely related to the distribution of an array over multiprocessors.

The space domain of all arrays is a very large one. Often we only need to deal with restricted array domains such as vectors, matrices (Section 3.3.2), and other arrays with specific small dimensionalities. Array domains with fixed dimensionality can still further be refined into subdomains according to their types.

3.3.2 Vectors and Matrices

Following the convention, we call one- and two-dimensional arrays respectively *vectors* and *matrices.* We will also denote a vector space of n entries

$$V : I \rightarrow U = (\{(i,u) \mid i \in I\},\ \leq)$$

by

$$[(i_0,\ u_0), \ldots, (i_{n-1},\ u_{n-1})]$$

where $i_p < i_{p+1}$ for $0 \leq p < (n-1)$.

Similarly, a matrix of shape (n, m)

$$M : I \rightarrow U = (\{((i,j),u) \mid (i,j) \in I\},\ \leq)$$

can be displayed by an "array" of n rows and m columns

$$\begin{bmatrix} ((i_0, j_0),\ u_{00}) & \cdots & ((i_0, j_{m-1}),\ u_{0(m-1)}) \\ \vdots & & \vdots \\ ((i_{n-1},\ j_0), u_{(n-1)(m-1)}), & \cdots & ((i_{n-1},\ j_{m-1}), u_{(n-1)(m-1)} \end{bmatrix}$$

where $i_p \leq i_{p+1}$, for $0 \leq p < (n-1)$, $j_q \leq j_{q+1}$, for $0 \leq q < (m-1)$.

3.3.3 The Relative Indexing Scheme

A non-normalized array can be *normalized* by normalizing its index set. More precisely, let $A : I \rightarrow U$ be an array, and Γ the normalization transformation for the index set I. Then the normalization of A is another array $A' = \Gamma A = \|A\|$, defined by

$$A' : I' \rightarrow U,$$

$$A'i = A : \Gamma^{-1}\, i$$

where ":" is the function composition. It should be clear that for a normalized array, the Γ transformation is simply an *identity function*, which we will denote by "*id*".

Array indexing therefore can also be performed by first normalizing the arrays. This leads to the *relative indexing scheme.* We also find it is convenient to allow *backward (relative) indexing*, which also requires an array A to be normalized first, but when the argument index is $\vec{i}$, the actual relative index used is $\$A - \vec{i}$, where "$-$" denotes element-wise subtraction between tuples.

Normalize:	$\|A\| = \Gamma\ A = A'$.
Relative Indexing:	$A\ [\vec{i}] = A : \Gamma\ \vec{i}$.
Backward Indexing:	$A\ [-\vec{i}] = A : \Gamma\ (\$A - \vec{1} - \vec{i})$, where $\vec{1} = (1, \ldots, 1)$.

Observe that in the above, relative indexing is given a different syntax from that for *absolute indexing* by enclosing the index in a pair of square brackets. But for the sake of conciseness, from now on we will denote relative indexing without the brackets unless otherwise specified.

In the rest of the dissertation, we also adopt the following two conventions related to the relative scheme:

1. A vector or a matrix can be displayed without the indices of the entries whenever the normalization function is not the issue of the discussion.

2. If f is a function over arrays depending on the indices of the arrays, and A is an array, then the application $f\ A$ by default is meant

$$f\ A \Rightarrow \Gamma^{-1} : f : \Gamma\ A$$

For example, the vector V and the matrix M of the above section can now be written respectively as

$$[u_0, \ldots, u_{n-1}]$$

and

$$\begin{bmatrix} u_{00} & \cdots & u_{0(m-1)}) \\ \vdots & & \vdots \\ u_{(n-1)(m-1)}, & \cdots & u_{(n-1)(m-1)} \end{bmatrix}$$

The normalization functions for V and M above are respectively

$$\Gamma_v\ i_p = j$$
$$\Gamma_M\ (i_p, i_q) = (p, q)$$

which will be left unspecified whenever the point can be made without it.

Clearly, the relative indexing convention also enables us to define operations over arrays as though all arrays are normalized, and therefore greatly facilitates the discussions in the remaining sections of this chapter.

It should be pointed out that the relative indexing convention not only gives us notational convenience, but also fits nicely with our intuitive notion of recursive computation over arrays. Moreover, the communication over arrays found in many applications, can be described by very simple functions which otherwise would be rather complicated if defined over absolute indices.

3.4 Divide Operations

3.4.1 Basic Notions

Let d be a k-ary divide function over an array domain $\mathcal{A}$, $A \in \mathcal{A}$ dividable, and

$$d\ A = (A_0, \ldots, A_{k-1})$$

Then by definition, the subarrays $A_0, \ldots, A_{k-1}$ form a partition of A, where A_i is the ith equivalent class or block of the partition.

Since a partition can be given by its quotient function [16, 33], which maps each element to a block of the partition, a divide function over arrays can also be given in terms of its quotient function. Let d be a k-ary division over arrays, $d\ A = (A_0, \ldots, A_{k-1})$. Then the *quotient function* q_d *of the division* d is a mapping from the array entries to the integer set $\{0, \ldots (k-1)\}$, such that an entry is mapped by q_d to i if and only if the entry is mapped by d from A to A_i, where $0 \leq i < k$.

For example, the binary divide function d_{ht} that partitions a vector of length greater than 1 into its "head" and "tail" can be defined in terms of the quotient function

$$q_{ht}\ (i, u) = \begin{cases} 0, & \text{if } i = 0, \\ 1, & \text{otherwise} \end{cases}$$

The division d_{qs} used in *quicksort* can be modeled by a 3-ary division over vectors. Let V be a vector. Then the quotient function associated with the division is

$$q_{qs}(i, u) = \begin{cases} 0, & \text{if } u < V\ 0, \\ 1, & \text{if } u = V\ 0, \\ 2, & \text{otherwise.} \end{cases}$$

We call a divide function defined by a quotient function the *divide function induced by the quotient function.* When d is induced by q, we write

$$d = \mathcal{D}\ q$$

For example, from the above we have

$$\begin{aligned} d_{ht} &= \mathcal{D}\ q_{ht} \\ d_{qs} &= \mathcal{D}\ q_{qs} \end{aligned}$$

A fundamental difference between d_{ht} and d_{qs} should be observed: the behavior of divide function d_{ht} is independent of the indexed values of the arrays, while d_{qs} is not. We call divide functions that are independent of the indexed values *polymorphic* functions.[1]

Formally, a divide function d is polymorphic if and only if $d = \mathcal{D}\ q$, and there exists quotient function q' defined over index sets such that

$$q(\vec{i}, u) = j \Leftrightarrow q'\ (\vec{i}) = j$$

Polymorphic divide functions have some valuable properties:

- They can be applied to arrays of different types as long as the dimensions of the arrays are compatible. A polymorphic divide function generally can be shared by many divide-and-conquer algorithms.

- They are static in the sense that the result of the division can be predicted before the (indexed) values of the arrays are known.

- On parallel computers, polymorphic divide functions can be implemented without communication between the processors.

All the divide functions we will discuss in the rest of the dissertation are polymorphic unless otherwise stated.

[1]Not to be confused with the more general notion of *polymorphic function* used in functional programming circles [51, 41].

3.4.2 Balanced Divisions

We first show how the partitions defined over the index sets naturally induce divide functions over arrays with the index sets to which the partitions are applicable.

Like the restrictions of relations which we mentioned in Section 2.1, we can define the *restriction of a function to a subdomain.* Let $f : X \rightarrow Y$ be a function; therefore it is a subset of $(X \times Y)$, X' a subset of X, and $f\ X' = Y'$ (the range of X' is Y'). Then the *restriction* of f to X', denoted by $X|_{X'}$ is [33]

$$f' = f \cap (X' \times Y)$$

Now, let $\mathcal{A}$ be an array domain, $\mathcal{I}_{\mathcal{A}}$ be $\mathcal{A}$'s index domain (Section 3.3.1), and Π a partition function over $\mathcal{I}$ such that for $I \in \mathcal{I}$, either $\Pi\ I = nil$ (and in that case we say Π is not applicable to I), or

$$\Pi = \{I_0, \ldots, I_{k-1}\}$$

We can then define a function $d_\Pi : \mathcal{A} \rightarrow \mathcal{A}^k$ by

$$d_\Pi\ A = \{A|_{I_0}, \ldots, A|_{I_{k-1}}\}$$

Proposition 3.1 *The function d_Π is a valid polymorphic divide function over $\mathcal{A}$.*

Proof

To show that d_Π is a divide function, we put the arrays in the form of spaces, let $A = (\mathcal{G}A, \leq)$, $A|_{I_j} = (\mathcal{G}A_j, \leq)$ for $0 \leq j < (k-1)$. Clearly,

1. *$\{\mathcal{G}A_0, \ldots, \mathcal{G}A_{k-1}\}$ is a partition of $\mathcal{G}A$.*

2. *$(G_{A_j}, \leq) = (\mathcal{G}A, \leq)|_{\mathcal{G}_{A_j}}$,* for $0 \leq j < 0$.

The divide function d_Π is polymorphic since the quotient function of Π is obviously independent of the indexed values.

Q.E.D.

The divide function d_Π above is called the *divide function induced by the partition* Π.

Proposition 3.1 allows us to define polymorphic divide functions over array domains easily. In fact, the head-tail division over vectors of Section 3.4.1 can serve as our first example. Let Π_{ht} be the partition over index sets defined by the quotient function q_{ht}. Then the divide function $d_{ht} = \mathcal{D}\ q_{ht}$ clearly can be defined equivalently as

$$d_{ht} = d_{\Pi_{ht}}$$

Now consider a one-dimensional index set I of size 2^m for some integer $m \geq 1$. Any index $i \in I$ therefore can be represented by a binary number with m bits. Let Φ be the encoding function that maps an integer to its binary representation, namely

$$\Phi\ i = (i_0, \ldots, i_{m-1})$$

where i_0 is the most significant bit of the binary number, and i_{m-1} is the least significant.

We can then define two partitions Π_{lr} and Π_{eo} by defining their respective quotient functions q_{lr} and q_{eo}:

$$\begin{aligned} q_{lr}\ i &= i_0 \\ q_{eo}\ i &= i_{m-1} \\ &\text{where } (i_0, \ldots, i_{m-1}) = \Phi\ i \end{aligned}$$

The divide functions over one dimensional arrays induced by Π_{lr} and Π_{eo}

$$\begin{aligned} d_{lr} &= \mathcal{D}\ q_{lr} = d_{\Pi_{lr}} \\ d_{eo} &= \mathcal{D}\ q_{eo} = d_{\Pi_{eo}} \end{aligned}$$

are respectively called the *left-right* and *even-odd* divisions. The following proposition explains the reason for giving them such names.

Proposition 3.2 *Let V be a vector where $\$\$V = 2^m$ for $m \geq 1$, i.e.*

$$V = [v_0, \ldots, v_{2^m-1}]$$

then

$$d_{\mathrm{lr}}\ V = ([v_0, v_1, \ldots, v_{2^{m-1}-1}],\ [v_{2^{m-1}}, v_{2^{m-1}+1} \ldots, v_{2^m-1}])$$
$$d_{\mathrm{eo}}\ V = ([v_0, v_2, \ldots, v_{2^m}],\ [v_1, v_3, \ldots, v_{2^m-1}])$$

For example,

$$d_{lr}\ [a\ b\ c\ d\ e\ f\ g\ h] = ([a\ b\ c\ d],\ [e\ f\ g\ h])$$
$$d_{eo}\ [a\ b\ c\ d\ e\ f\ g\ h] = ([a\ c\ e\ g],\ [b\ d\ f\ h])$$

In contrast,

$$d_{ht}\ [a\ b\ c\ d\ e\ f\ g\ h] = ([a],\ [b\ c\ d\ e\ f\ g\ h])$$

It is obvious that the divide functions d_{lr} and d_{eo} are balanced, while the divide function d_{ht} is not.

3.4.3 Dividing Higher Dimensional Arrays

In this section, we shall show how divide functions over higher dimensional arrays can be constructed from those for one dimensional arrays.

Let Π_q be a partition over one dimensional index sets, where q is the quotient function of the partition Π_q. We can then define another partition $\Pi_{(q'\ i)}$ with the quotient function $(q\ i)$ defined over higher dimensional index sets by[2]

$$(q'\ i) = q : (proj\ i)$$

In other words, the quotient function $(q'\ i)$ acts on the ith component of a multiply dimensional index the same way as q does on a one dimensional index. This enables us to define

$$(d_{\Pi_q}\ i) = d_{\Pi_{(q'\ i)}}$$

where $q'\ i$ is defined as in above. Divide functions defined this way are said to be *constructed by dimensional projections.*

[2]See Section 3.1 for the definition of the function *proj*.

For example, $(d_{ht}\ 0)$ divides a matrix into two submatrices: the first row and the rest of the rows; $(d_{lr}\ 0)$ applied to a matrix divides it into two submatrices with equal number of rows; $(d_{eo}\ 1)$ applied to a matrix divides it into two submatrices consisting respectively of all the even and all the odd columns.

Divide functions on higher dimensional arrays can also be *constructed by intersection.* By set theory the intersection of two partitions $\Pi = \Pi_1 \cap \Pi_2$ has blocks which are the intersections of the blocks in Π_2 and Π_2 [16]. Similarly, let $d_0, \ldots, d_{k-1}$ be m divide operations induced by partitions $\Pi_0, \ldots, \Pi_{m-1}$ respectively. We can define their *intersection* by

$$(d_0\ i_0) \times \cdots \times (d_{m-1}\ i_{m-1}) = d_\Pi$$
$$\text{where } \Pi = (\Pi_0\ i_0) \times \cdots \times (\Pi_{k-1}\ i_{k-1})$$

Just as notation, when the dimension parameters of the divide functions are consecutive and well ordered, we can write

$$d_0 \times \cdots \times d_{m-1} = (d_0\ 0) \times \cdots \times (d_{m-1}\ (m-1))$$

The *power* of a divide function can then be naturally defined as

$$d^m = \underbrace{d \times \cdots \times d}_{m}$$

It is obvious that, given k_j as the arity of the jth divide function in the intersection for $j = 0$ to $(m-1)$, the intersection has arity of $\prod_{j=0}^{m-1} k_j$. This is to be contrasted with construction by projection, where the arity remains the same after the construction.

In Figure 3.1, we give illustrations of various divide functions by construction. The following are two concrete examples.

Examples:

$$\text{Let} \quad M = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{bmatrix}$$

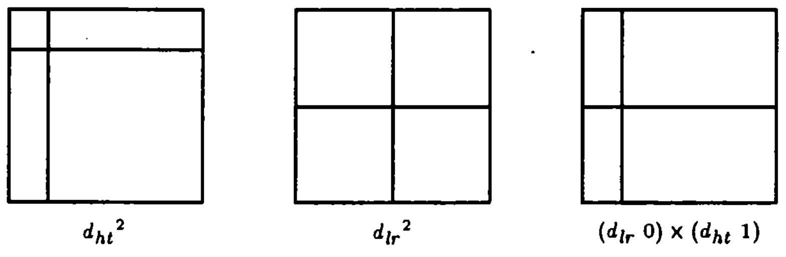


Figure 3.1: Some constructed divisions over matrices

then

1.

$$d_{lr}\ 0\ M = (M_0, M_1)$$

where

$$M_0 = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \end{bmatrix}, \quad M_1 = \begin{bmatrix} 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{bmatrix},$$

2.

$$d_{lr}^2\ M = (M_{nw}, M_{ne}, M_{sw}, M_{se})$$

where

$$NW = \begin{bmatrix} 0 & 1 \\ 4 & 5 \end{bmatrix}, \quad NE = \begin{bmatrix} 2 & 3 \\ 6 & 7 \end{bmatrix}, \quad SW = \begin{bmatrix} 8 & 9 \\ 12 & 13 \end{bmatrix}, \quad SE = \begin{bmatrix} 10 & 11 \\ 14 & 15 \end{bmatrix}$$

3.4.4 Dividing Multiple Arrays

So far all the divide operations are defined over single arrays. To compute functions of higher arity, we also need divide operations over multiple arrays, namely, tuples of arrays.

Let d_0, d_1 be two divide functions over array domains $\mathcal{A}_0$ and $\mathcal{A}_1$ respectively. Let k_0, k_1 be respectively their arities, $A_0 \in \mathcal{A}_0$, $A_1 \in \mathcal{A}_1$ two dividable arrays, and therefore

$$d_0\ A_0 = (A_{00}, \ldots, A_{0(k_0-1)})$$
$$d_1\ A_1 = (A_{10}, \ldots, A_{1(k_1-1)})$$

We define the *outer-product* of d_0 and d_1, written $d = d_0 \cdot d_1$, to be

$$d\ (A_0, A_1) = \begin{matrix} ((A_{00}, A_{10}), & \ldots, & (A_{00}, A_{1(k_1-1)}), \\ \vdots & \ddots & \vdots \\ (A_{0(k_0-1)}, A_{10}), & \ldots, & (A_{0(k_0-1)}, A_{1(k_1-1)})) \end{matrix}$$

Clearly, the arity of the full outer-product of two divisions is the product of the arities of the two divisions. The outer-product of the above thus divides a pair of arrays into $k = k_0 * k_1$ sub-pairs.

For example, let a divide function over a pair of vectors $d_{vv} = d_{lr} \cdot d_{lr}$. Then

$$d\ ([1\ 2\ 3\ 4], [a\ b\ c\ d]) = (([1\ 2], [a\ b]), ([1\ 2], [c\ d]), ([3\ 4], [a\ b]), ([3\ 4], [c\ d]))$$

Frequently, we do not really need all the sub-pairs produced by an outer-product division. This can be done by composing an outer-product with the tuple selection operation (see Section 3.1). For example, let

$$d_{vv6} = sel\ (1, 0, 1, 0) : d_{lr} \cdot d_{lr}$$

then

$$d_{vv6}\ ([1\ 2\ 3\ 4], [a\ b\ c\ d]) = (([1\ 2], [a\ b]),\ ([3\ 4], [a\ b]))$$

For conciseness of notation, we will allow the binary tuple to be given in the decimal form for tuple selection operation. For example, the divide function d_{vv6} of the above can be redefined by

$$d_{vv6} = sel\ 6 : d_{lr} \cdots d_{lr}$$

The above discussion is limited to outer-product of two divisions. The outer-product of more than two divisions can be similarly defined.

Observe that all primitive divide operations and the divide operations defined by their products have the following static property: the size of the array equals the sum of the sizes of the subarrays. In contrast, divide functions defined by outer-product may or may not expand in size. For example, the divide function d_{vv} increases the size of the array(s) by a factor of two, while d_{vv6} does not.

Let k and m be respectively the arity and division factor of a balanced division d. The *expanding factor* β of d is defined to be [36]

$$\beta = k/m$$

By definition, if d is a divide function with expanding factor β, C is the sum of the sizes of the argument arrays, and C' is the sum of the sizes of the resultant arrays, then we have

$$C' \leq \beta * C$$

A balanced division is said to be *static* if its expanding factor is one, and *dynamic* otherwise.

Aho *et al.* in [3] (Section 2.6) discussed the effect of expanding factors on the time complexity of DC algorithms computed on sequential computers. In Chapter 6, we will see the impact of the dynamic property in the parallel complexity of DC algorithms.

3.5 Combine Operations

Recall that in Section 2.2.2 we showed that, given a divide operation over a space domain, a combine operation over the same domain can be defined as the minimum left inverse of the divide operation. By applying this to the divide operations over arrays, which we have studied with some breadth and depth in previous sections, we automatically have many of the combine operations over array domains in hand.

However, we would like to take some steps to improve the syntactic appearance of our notations. We define:

1. $c_{lr} = d_{lr}^{-1}$.

2. $c_{eo} = d_{eo}^{-1}$.

3. let $c_i = d_i^{-1}$ for $i = 0$ to $(m-1)$
   in $(c_0\ i_0) \times \cdots \times (c_{m-1}\ i_{m-1}) = ((d_0\ i_0) \times \cdots \times (d_{m-1}\ i_{m-1}))^{-1}$.

4. $c_0 \times \cdots \times c_{m-1} = (c_0\ 0) \times \cdots \times (c_{m-1}\ (m-1))$.

5. $c^m = \underbrace{c \times \cdots \times c}_{m}$.

We also say that an array combine operation c is a *polymorphic combine operation* if $c = d^{-1}$ and d is polymorphic. The combine functions c_{lr}, c_{eo} and c_{ht} therefore are polymorphic combine operations. The combine function used in *mergesort* [3] is an example of a non-polymorphic combine operation.

Examples:

1. $c_{lr}\ ([a\ b], [c\ d]) = [a\ b\ c\ d]$.

2. $c_{eo}\ ([a\ b], [c\ d]) = [a\ c\ b\ d]$.

3. $c_{lr}\ ([a], [b\ c\ d]) = c_{eo}\ ([a], [b\ c\ d]) = nil$.

4. Let

$$M_0 = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \end{bmatrix}, \quad M_1 = \begin{bmatrix} 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{bmatrix}$$

then

$$c_{lr}\ 0\ (M_0, M_1) = M$$

where

$$M = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{bmatrix}$$

5. Let

$$NW = \begin{bmatrix} 0 & 1 \\ 4 & 5 \end{bmatrix},\quad NE = \begin{bmatrix} 2 & 3 \\ 6 & 7 \end{bmatrix},\quad SW = \begin{bmatrix} 8 & 9 \\ 12 & 13 \end{bmatrix},\quad SE = \begin{bmatrix} 10 & 11 \\ 15 & 16 \end{bmatrix}$$

then

$$c_{lr}{}^2\ (NW,\ NE,\ SW,\ SE) = M$$

where M is the same as in (4).

3.6 Universal Array Operations

So far we have studied the divide and combine operations over arrays. These operations only take arrays apart and put them together but never affect the indexed values of the array entries. We call operations of this nature *relational functions*, since from the space point of view divide and combine operations act on the relational values of the array spaces. Computations over arrays also need operations of an orthogonal nature, that is, operations that map array entries to new values. We call these operations *universal functions* since they do affect the array entry values.

By the above definition, universal functions over arrays still may or may not affect the index sets of the argument array(s). A universal function is said to be *anti-polymorphic* if it preserves the index sets of the argument(s). In other words, anti-polymorphic universal functions preserve the structures of their arguments. We will be studying only anti-polymorphic universal functions.

3.6.1 Distribution

Let $f : X \rightarrow Y$ be a function. We define $!f$ (read as "bang f") to be a function from arrays of type X to arrays of type Y by

$$!f\ A = A', \text{ where } A'\ \vec{i} = f : A\ \vec{i}$$

The function $!f$ is called a *distribution of f*, where f is the *distribution generator*. The operator "!" should be considered as a higher order function that takes a generator defined over the arrays' indexed values, and returns a function defined over the arrays. It is similar to the "map" operation in functional programming languages [22, 41, 18], but different since they operate on different data structures.

Besides arrays of primitive types, we will often encounter arrays of tuples – in particular, arrays of pairs.[3] Unary operators over pairs such as such as $+$, $*$, min, max, and the projection functions over tuples are the common distribution generators in our applications. When the tuples have higher arity, the generators are often defined with pattern matching [41].

Examples:

1. $!\mathrm{sq}\ [1\ 2\ 3\ 4] = [1\ 4\ 9\ 16]$.
2. $!+\ [(1,2)\ (3,4)\ (5,6)\ (7,8)] = [3\ 7\ 11\ 15]$.
3. $!\max\ [(3,6)\ (9,2)\ (4,1)\ (3,3)] = [6\ 9\ 4\ 3]$.
4. Let $A = [(a,b)\ (c,d)\ (e,f)]$. Then,

$$\begin{aligned} &!self\ \ A = [a\ c\ e], \\ &!other\ A = [b\ d\ f]. \\ &\quad \text{where } self\ (x,y) = x, \\ &\qquad\quad other(x,y) = y. \end{aligned}$$

[3]This is because the communication functions to be introduced in 3.6.3 generally map arrays of type T to arrays of two-tuples of type T.

5. $!f\ [(1,2,3)\ (4,5,6)] = [5,26]$, where $f(x,y,z) = x*y+z$

6. Strong local functions applied to matrix:

$$!*\begin{bmatrix} (1,2) & (3,4) \\ (5,6) & (7,8) \end{bmatrix} = \begin{bmatrix} 2 & 12 \\ 30 & 56 \end{bmatrix}$$

It should be observed that array distribution functions require no communication among the array entries. Moreover, the order of the applications of f over the entries are not specified. It follows that f can be applied to all entries in parallel. The distribution functions in this sense are *parallel operations.*

3.6.2 Construction

Given k functions f_0 and f_{k-1}, their *construction*, denoted by $(f_0, \ldots, f_{k-1})$, is a function over k-tuples defined by

$$(f_0, \ldots, f_{k-1})\ (A_0, \ldots, A_{k-1}) = (f_0\ A_0, \ldots f_{k-1}\ A_{k-1})$$

The functions f_i for $i = 0$ to $k-1$ are called the *constituents* of the construction.

For the case when all the k constituents are identical, we write

$$(f, \ldots, f) = !f$$

where "$!f$" can be thought of as a *distribution over tuples.*

Just as the generator of array distribution can be computed in parallel over array entries, the constituents of a construction can be computed in parallel over the tuple components. We call both of them *local functions* since they do not require communication among the array entries or the tuple components.

In our application, the constituents of a construction are often functions over arrays. A construction is said to be *strongly local* if all its components are array distributions. Thus a strong local function is defined over tuples of arrays and has the following form

$$(!f_0, \ldots, !f_k)$$

When all the constituents are identical in the above, we write

$$(!f, \ldots, !f) = !!f$$

where "!!" is read as "bangbang".

Examples:

1. $(!\min, !\max)$ $([(3,6)(5,4)],\ [(6,3),(4,5)]) = ([3\ 4],[6\ 5])$, (strong).
2. $!! +$ $([(1,2)(3,4)],\ [(5,6)(7,8)]) = ([3\ 7],[11\ 15])$, (strong).
3. $(id, !*)$ $([a\ b],\ [(c,d)\ (e,f)]) = ([a\ b],[c*d\ e*f])$, (strong).
4. (reduce, reduce) $([1\ 2],\ [3\ 4) = (3,7)$, where reduce $V = \sum_{i=0}^{(\$\$V-1)} V\ i$, (weak).
5. pair-wise-add $([1\ 2],\ [3\ 4]) = [4\ 6]$ is not a local function, where pair-wise-add $(V_0, V_1) = V$, where $V\ i = V_0\ i + V_1\ i$ for $i = 0$ to $(\$\$V_0 - 1)$.

3.6.3 Communication Functions

Let $f : \mathcal{I} \rightarrow \mathcal{I}$ be a function from an index domain to itself. We define the function $\#f$ (read as "sharp f") to be a function over arrays with index sets in $\mathcal{I}$ by

$$\#f\ A = A', \quad \text{where} A'\ \vec{i} = (A\ \vec{i},\quad A : f\ \vec{i})$$

The function $\#f$ is called a *communication function* generated by the *communication generator* f. The operator $\#$ thus is a higher order function that transforms a function over index sets to a function over arrays that performs communication. A communication function maps each array entry to a pair, where the first component is the entry's original value, and the second is the value of another entry whose index is determined by applying the generator to the entry's own index.

What the above defines is only the *intra-array* communication. Let $A : I \rightarrow X$, $B : J \rightarrow Y$ be two arrays, f a function from I to J. We define the *inter-array* communication from B to A to be:

$$\#f\ (A,\ B) = A', \text{ where } A'\ \vec{i} = (A\ \vec{i},\ B : f\ \vec{i})$$

This above communication function is *one-directional*, since only the array A is mapped to a new array A' using the values from B. A *bi-directional* communication can be defined with a pair of communication generators:

$$\#(f_a,\ f_b)(A,\ B) = (\#f_a\ (A,\ B),\ \#f_b\ (B,\ A))$$

When $f_a = f_b = f$, the above communication function can be rewritten as $\#!f$.

A wide range of parallel algorithms share communication patterns that can be expressed with very simple communication generators. Some of these generators are (see Figure 3.2):

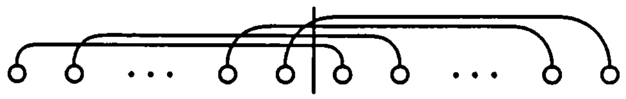
$corr\ \vec{i} = \vec{i}$	correspondent communication
$mirr\ \vec{i} = \$ - \vec{i}$	mirror-image communication
$br\ \vec{c}\ \vec{i} = \vec{c}$	broadcast from one entry to all
$last\ \vec{c}\ \vec{i} = \$ - \vec{c}$	broadcast from a entry with backwards index

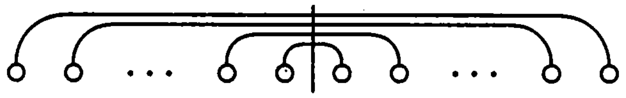
The special communication generator *null* is used to denote the null communication.

Examples:

1. $\#mirr\ [1\ 2\ 3\ 4] = [(1,4)\ (2,3)\ (3,2)\ (4,1)]$

2. $\#corr\ ([a\ b],\ [c\ d]) = [(a,c)\ (b,d)]$


(a) Correspondent communication


(b) Mirror-image communication

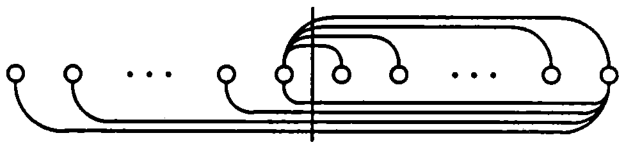

(c) (Last 0) broadcasting communication

Figure 3.2: Communications over a pair of vectors of the same size

3. $\#(null,\ last\ 0)$ ([1 2], [3 4]) = ([1 2], [(3,2), (4,2)]

4. $\#!corr$ ([a b], [c d]) = ([(a,c) (b,d)], [(c,a) (d,b)])

5. All the above generators are applicable to higher dimensional arrays, e.g.

$$\#(last(0,0)) \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} (1,4) & (2,4) \\ (3,4) & (4,4) \end{bmatrix}$$

6. Let $row\_br\ (i,j) = (i,0)$. Then $\#row\_wise\_br$ broadcasts the value of the first entry of each row to all entries of the row, e.g.

$$\#row_br \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} (1,1) & (2,1) \\ (3,3) & (4,3) \end{bmatrix}$$

Likewise, simple communication generators can be defined for column-wise broadcast, broadcast of diagonal elements along corresponding rows or columns for matrices, and broadcast vector elements to corresponding rows or columns of an array:

$col\_br\ (i,j) = (0,j)$	column wise broadcast from first row

$dia\_row\_br\ (i,j) = (i,i)$	row wise broadcast from the diagonal
$dia\_col\_br\ (i,j) = (j,j)$	column wise broadcast from the diagonal
$v\_m\_row\ (i,j) = i$	row wise broadcast from a vector
$v\_m\_col\ (i,j) = j$	column wise broadcast from a vector

Communication functions, by their definition, move pieces of data around but never perform any local computation. This is to be contrasted with local functions which have the opposite characteristics. Communication functions and local functions are in this sense orthogonal to each other. Many useful operations such as reversion of vectors, transposition of arrays, array pair-wise operations, and many others can be defined by the composition of communications followed by local functions:

Examples:

1. $(!\min,\ !\max)\ :\ \#!corr\ ([5\ 8\ 1\ 4],\ [3\ 2\ 7\ 0]) = ([3\ 2\ 1\ 0],\ [5\ 8\ 7\ 4])$

2. Let $reverse = !other\ :\ \#mirr$. Then $reverse\ [1\ 2\ 3\ 4] = [4\ 3\ 2\ 1]$.

3. Let $[+] = !+\ :\ \#corr$. Then $[+]\ ([1\ 2],\ [3\ 4]) = [4\ 6]$.

4. Let $[\oplus] = !\oplus\ :\ \#corr$. Then it performs entry-wise binary array operation with respect to a binary operation $\oplus$,

5. Let $transpose = !other\ :\ \#reverse$. Then the function $transpose$ transposes a matrix.

Finally, let us point out that a communication function fetches for each entry a value from another entry, but the order over the entries in which the fetching operation performed is unspecified. This means that whenever possible all the fetching operations can be performed in parallel. Communication functions in this sense are also parallel operations.

3.6.4 Functional Forms

John Backus used the term *functional forms* in [4] to refer to constructs used to construct new function from one or more other functions. In other words, a functional form is a higher order function which takes other function(s) as argument(s) and returns another function as a result. We will often refer to the arguments of a functional form as its *constituents*.

Functional forms are not new to us. The functions we studied in previous sections, in fact, are all defined by functional forms. The following is a summary of the functional forms we have encountered:

Function Composition denoted by ":".

Array Distribution denoted by "!".

Construction denoted by $(f_0, \ldots, f_{k-1})$.

Communication denoted by "#"

Tuple Distribution (also) denoted by "!".

Now let us define some other functional forms:

If-Then-Else denoted by the standard syntax of $p \rightarrow f; g$ with the usual meaning of

$$p \rightarrow f; g\ x \equiv \text{if } p\ x \text{ then } f\ x \text{ else } g\ x$$

If-Then denoted by $p \rightarrow f$, defined by

$$p \rightarrow f\ x = p \rightarrow f; id\ x$$

Filter denoted by $(p \Rightarrow f)$, defined by

$$(p \Rightarrow f)\ A = A' \text{where}\ \ A'\ \vec{i}\ \equiv\ \text{if } p\ \vec{i} \text{ then } f\ A\ \vec{i} \text{ else } A\ \vec{i}$$

The difference between if-then-else and filter should be observed: for example,

$$\begin{aligned} !(<3? \rightarrow \mathrm{sq})\ [4\ 3\ 2\ 1] &= [4\ 3\ 4\ 1] \\ (<3? \Rightarrow !sq)\ [4\ 3\ 2\ 1] &= [16\ 9\ 4\ 1] \end{aligned}$$

Sequential Tuple Distribution denoted by $!f :: \vec{\mu}$, where $\vec{\mu} = (\mu_0, \ldots, \mu_{k-2})$ is a tuple of functions. It is defined by

$$\begin{aligned} &!f :: \vec{\mu}\ (x_0, \ldots, x_{k-1}) = (y_0, y_1, \ldots, y_{k-1}) \\ &\quad \text{where} \\ &\qquad y_0 = f x_0 \\ &\qquad y_1 = f : \mu_0(x_1, y_0) \\ &\qquad \ldots \\ &\qquad y_{k-1} = f : \mu_{k-2}(x_{k-1}, y_{k-2}) \end{aligned}$$

Due to the dependency relation, the constituents of this form must be performed one by one in a linear order. This contrasts with the distribution over tuples by the operation "!" alone, which we may refer to as "parallel tuple distribution." In some applications, the function f is not applied to all the substructures; we define that if the ith component of $\vec{\mu}$ has the form $\%\mu_i$, then $!f :: \vec{\mu}$ is defined to be the same as above except that the ith equation is replaced by

$$y_i = \mu_i(x_i, x_{i-1})$$

The usefulness of this sequential distribution form will be shown in Chapter 5.

3.6.5 Zip and Unzip

A structure of m objects $a_0, \ldots, a_{m-1}$ can be defined by

$$a = struct\ (a_0, \ldots, a_{m-1})$$

where the objects a_i for $i = 0$ to $(m-1)$ are called *fields* of the structure a. Unlike that for tuples, a field of a structure is accessed by its name, not by its index. For example, for structure a of above, we have

$$a.a_i = a_i \text{ for } i = 0 \text{ to } (m-1)$$

Given m arrays of same shape $(A_0, \ldots, A_{m-1})$, their *Z-structure* is an array B of the same shape, where each entry is a *structure* of the entries from the m arrays with the same index. For example, with B as above, we have

$$B\ \vec{i} = struct\ (A_0\ \vec{i}, \ldots, A_{m-1}\ \vec{i})$$

The *zip* and *unzip* operations are used to, respectively, construct and disperse Z-structures. Thus, let $A, B_0, \ldots B_{m-1}$ be as above, we have

$$\begin{aligned} &zip\ (A_0, \ldots, A_{m-1}) = B \\ &unzip\ B = (A_0, \ldots, A_{m-1}) \end{aligned}$$

The arrays $A_0, \ldots, A_{m-1}$ are called *fields* of the Z-structure B. A field of the Z-structure is accessed also by naming it. For example, for the Z-structure B in above, we have

$$B.A_i \ = \ A_i$$

3.7 Divacon Notation

While studying array and array operations in this chapter, we have also studied some relevant data structures such as tuples and structures and introduced a set of notations for expressing operations defined over these data structures. This set of notation from now on will be referred to as the *Divacon notation*, which will be used to specify all the DC algorithms in the rest of this dissertation.

Although much of the functionality that we need could be captured in a modern functional language such as Haskell [18], it should be noted that Divacon notation has some unique features which are not found in most programming languages, but which are crucial for DC algorithms:

- Arrays can be recursively divided and combined.
- Communication operation over arrays.
- Relative indexing scheme.
- DC functional forms.

We therefore feel that it is more of a matter of necessity than a matter of taste to adopt the Divacon notation over some well established programming language in our discussion.

Chapter 4

Parallel Divide-and-Conquer

4.1 Pseudomorphism: the Basic Models

Divide-and-Conquer is a notion that has been illustrated by examples and understood only via intuition in the past. With the concepts developed in the last two chapters, we will show that this notion may be formalized. The formal model we propose for divide-and-conquer is called the *pseudomorphism*, which is a generalization of the morphism in algebra. To demonstrate the expressiveness of the model, we present in this chapter a total of twelve DC algorithms.

4.1.1 Morphisms

In their excellent textbook *The Design and Analysis of Computer Algorithms* [3], A. V. Aho, *et al.* describe divide-and-conquer this way:

> *A common approach to solving a problem is to partition the problem into smaller parts, find solutions for the parts, and then combine the solutions for the parts. This approach, especially when used recursively, often yields efficient solutions to problems in which the subproblems are smaller versions of the original problem.*

If we interpret the "partition" as divide operations, and the "smaller version of the original problem" as recursive application of the function over smaller arguments as one would do in functional notation, then the above informal notion can be expressed in the notation we developed as

$$f = c : !f : d$$

or back in English: to compute a function f applied to an argument, we apply a divide function d, then apply f recursively to each of the subarguments from the division, and finally apply a combine function c to get the result.

Functions that are subject to this approach are called *morphisms* in algebra [16]. Formally, a function f from one algebra (X, c_x) to another (Y, c_y) is a morphism (see Figure 4.1 (a)) if the equality

$$f : c_x(x_0, \ldots, x_k) = c_y : (f\ x_0, \ldots, f\ x_k)$$

holds whenever $c_x(x_0, \ldots, x_k) \neq nil$. With our Divacon notation, the above can be rewritten more concisely as

$$f : c_x = c_y : !f$$

To see how this equation is related to what we derived from the informal notion of divide-and-conquer, let d be a divide function such that c_x is a left inverse, apply d to both sides of the above equation from the right, and note that $c_x : d = id$. Then we have

$$\begin{array}{rll} & f : c_x : d & = c_y : !f : d \\ \Rightarrow & f & = c_y : !f : d \end{array}$$

As an example, let us consider the function *reduce* (as defined in APL), which takes an associative binary operator $\oplus$ and a vector v as arguments, and returns a scalar which is the sum of all the vector entries with respect to $\oplus$, i.e.

$$reduce\ \oplus\ v = \bigoplus_{i=0}^{|v|-1} v(i)$$

Let V be the domain of regular vectors of type U. Then the function (*reduce* $\oplus$) is a morphism from (V, c_{lr}) to $(U, \oplus)$, since we have (as can be proven easily)

$$(reduce\ \oplus) : c_{lr} = \oplus :!(reduce\ \oplus)$$

For example,

$$\begin{aligned} &(reduce\ +) : c_{lr}\ ([1\ 2],\ [3\ 4]) \\ &= + : !(reduce\ +)\ ([1\ 2],\ [3\ 4]) \\ &= + : (reduce\ + [1\ 2],\ reduce\ + [3\ 4]) \\ &= +(3\ 7) \\ &= 10 \end{aligned}$$

which is the reduction of vector [1 2 3 4] with respect to the operator $+$.

Morphisms constitute a fairly broad class of functions encountered in mathematics and engineering. Differentiation, integration, convolution, and Fourier transformation are classic examples. Morphisms capture the notion of divide-and-conquer since the application of a morphism to an argument can be reduced to more than one application to the sub-arguments by definition.

4.1.2 Postmorphisms

Given a binary associative operator $\oplus$, the function (*scan* $\oplus$) when applied to a vector returns a new vector which is a vector of partial sums with respect to $\oplus$. More precisely,

$$scan \oplus v = v',\ \text{where} v'\ i = \bigoplus_{j=0}^{i} v\ j$$

e.g. *scan* $+ [1\ 2\ 3\ 4] = [1\ 3\ 6\ 10]$.

The function *scan* is related to *reduce* in that the the last entry of *scan* gives the result of reduce given the same binary operator and same input. *Reduce* has been successfully shown to be a morphism, one might think that *scan* also is. Now let us examine whether this is true. In order to do so, we first need to identify the associated algebras. The only natural choice in the case of *scan* is to use (V, cat) both as the domain and co-domain algebra. Were *scan* a morphism, the following equation would hold:

$$(scan\ \oplus) : cat = cat : !(scan\ \oplus))$$

This is unfortunately not the case, as can be shown by the following counter-example

$$\begin{aligned} & (scan\ +) : cat([1\ 2],\ [3\ 4]) \\ = {} & cat : !(scan\ +))([1\ 2],\ [3\ 4]) \\ = {} & cat : (scan\ \ +[1\ 2],\ scan\ \ +[3\ 4]) \\ = {} & cat\ ([1\ 3],\ [3\ 7]) \\ = {} & [1\ 3\ 3\ 7] \qquad (\neq \text{(the correct result of) } [1\ 3\ 6\ 10]) \end{aligned}$$

However, the incorrect result [1 3 3 7] contains much of the information about the proper result [1 3 6 10]. First of all, the left half [1 3] completely agrees with the correct result. Secondly, the right half can be made correct by adding the last entry on the left to each entry on the right.

This motivates us to define the function

$$h_{scan}\ \oplus = (id,\ !\oplus) : \#(nil,\ last\ 0)$$

which, when applied to two vectors, will leave the left vector untouched, but will add the last entry on the left to each entry on the right. For example,

$$\begin{aligned} & (h_{scan}\ +)\ ([1\ 3],\ [3\ 7]) \\ = {} & (id,\ +) :\ \#(nil,\ last\ 0)\ ([1\ 3],\ [3\ 7]) \\ = {} & (id,\ +)\ \ ([1\ 3],\ [(3,\ 3)\ (7,\ 3)]) \\ = {} & (id\ [1\ 3],\ \ !+[(3,3)\ (7,\ 3)] \\ = {} & ([1\ 3],\ [6\ 10]) \end{aligned}$$

Although scan is not a morphism, once the above function is defined, we can show that scan satisfies an equation similar to that for morphisms, which is

$$(scan\ \oplus) : cat = cat : (h_{scan}\ \oplus) : !(scan\ \oplus)$$

This equation tells us that despite the fact that scan is not a morphism, its application to an vector can also be reduced to more than one application over the subvectors, and therefore can be computed in a divide-and-conquer fashion. To describe functions of this nature, we introduce the notion of *postmorphism.*

A function $f : (X, c_x) \rightarrow (Y, c_y)$ is a postmorphism if there exists a *postadjust function* (see Figure 4.1 (c)) $h : Y^k \rightarrow Y^k$, where k is the arity of the algebra (Y, c_y), such that

$$f : c_x = c_y : h : !f$$

The name "postadjust function" reflects the fact that postadjust function is to be applied only after the function f has been recursively applied to the subargument. Morphisms obviously can be regarded as special cases of postmorphisms, where the postadjust functions happen to be the identity function.

4.1.3 Premorphisms

The function *reverse* reverses the order of the entries in vectors; for example,

$$\mathit{reverse}\ [1\ 2\ 3\ 4] = [4\ 3\ 2\ 1]$$

The function *reverse* is not a morphism from the algebra (V, c_{lr}) to itself; since if it were, we would have

$$\begin{aligned} &\mathit{reverse} : c_{lr}\ ([1\ 2],\ [3\ 4]) \\ = &\ c_{lr} : !\mathit{reverse}\ ([1\ 2],\ [34]) \\ = &\ c_{lr}\ ([2\ 1],\ [4\ 3]) \\ = &\ [2\ 1\ 4\ 3] \end{aligned}$$

which is not correct.

Let us define a function [1]

$$g_{reverse} = !\mathit{other} : \ \#!\mathit{corr}$$

[1]Recall that in Chapter 3 we defined $\mathit{other}(a,\ b) = b$, $\mathit{corr} = \mathit{id}$.

which, when applied to two vectors of the same size, will exchange the entries in the correspondent position, for example:

$$g_{reverse}\ ([1\ 2],\ [3\ 4]) = ([3\ 4],\ [1\ 2])$$

Now we can show that the following equation holds for the function *reverse*:

$$reverse := c_{lr} : !reverse : g_{reverse}$$

For example,

$$\begin{aligned} & reverse : cat([1\ 2],\ [3\ 4]) \\ = {} & c_{lr} : !reverse : g_{reverse}\ ([1\ 2],\ [3\ 4]) \\ = {} & c_{lr} : !reverse\ ([3\ 4],\ [1\ 2]) \\ = {} & c_{lr}\ ([4\ 3],\ [2\ 1]) \\ = {} & [4\ 3\ 2\ 1] \end{aligned}$$

The presence of $!reverse$ in the above equation means that *reverse* can be computed in a divide-and-conquer fashion. To describe functions with this nature, we introduce the notion of *premorphism.*

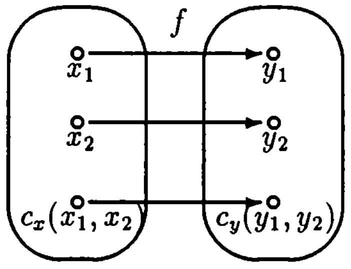
A function $f : (X, c_x) \rightarrow (Y, c_y)$ is a premorphism (see Figure 4.1 (b)) if there exists a *preadjust* function $g : X^k \rightarrow X^k$, where k is the arity of the algebra (X, c_x), such that

$$f : c_x = c_y : !f : g$$

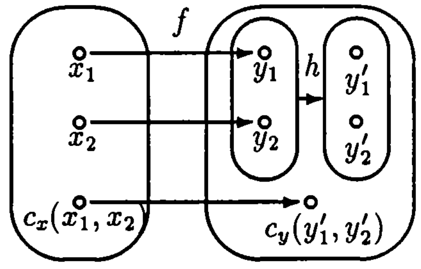
In premorphisms, the preadjust functions are always applied before the function f is recursively applied to the subarguments: hence the name "preadjust" function. Morphisms are obviously special cases of postmorphisms, where the preadjust functions happen to be the identity function.

4.1.4 Pseudomorphisms

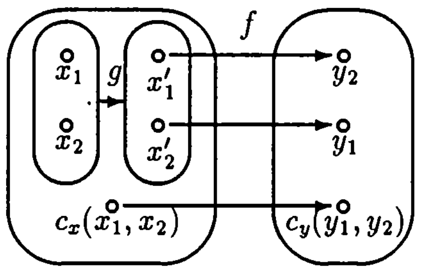
We have shown how the class of functions called morphisms in algebra is subsumed by either the premorphism model or the postmorphism model. A natural question that



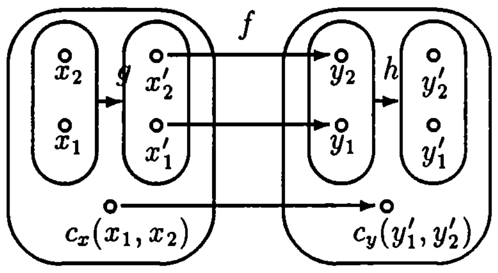
(a) Morphisms



(c) Postmorphisms



(b) Premorphisms



(d) Pseudomorphisms

Figure 4.1: Different types of pseudomorphisms (k=2)

can be raised is whether these two generalized models can be unified. The answer is quite obvious: we can define a model which has both pre- and post- adjust functions, and then premorphism and postmorphism become special cases of the new model.

Formally, we say a function $f : (X, c_x) \rightarrow (Y, c_y)$ is a *pseudomorphism* (see Figure 4.1 (d)) if there exists a preadjust function $g : X^{k_x} \rightarrow X^{k_x}$, and a postadjust function $h : Y^{k_y} \rightarrow Y^{k_y}$, where k_x and k_y are respectively the arity of the algebras (X, c_x) and (Y, c_y), such that

$$f : c_x = c : h : !f : g$$

Obviously, pure morphisms, premorphisms, and postmorphisms are all special cases of pseudomorphisms, where one or both of the adjust functions happen to be the identity function.

It should be pointed out that although pseudomorphisms have fewer algebraic properties than morphisms due to the generality of the model, the key property we are interested in in morphisms is not lost: pseudomorphisms by definition can be computed recursively over the subarguments just like morphisms (since $f = \ldots !f \ldots$). This is why we say the pseudomorphism model captures the notion of divide-and-conquer. Moreover, there is no order imposed by the definition on the recursive applications over the subarguments, which means that the recursions can be carried out **in parallel**. Pseudomorphism is therefore a model for what we call *parallel divide-and-conquer* (PDC) algorithms. In Chapter 5, we will see a different class of algorithms called *sequential divide-and-conquer* (SDC).

We have developed our discussion on general grounds. We have not assumed the arities of the algebras are always two, nor have we assumed the equality of the arities of the domain and co-domain algebras. Our experience has shown that the generality is indeed demanded in applications of the models to real problems – in particular, when dealing with higher dimensional arrays. However, in all the examples of this section, the algebra arities are two and equal. This will also be the case with most but not all the problems that we will study in this dissertation.

4.2 Computation of Pseudomorphisms

4.2.1 Some Related Notions

A predicate p over a space domain S is a mapping from S to the boolean set *Boolean*= *true*, *false*.[2] It follows that a predicate p over a domain S partitions it into exactly two subdomains,

$$\begin{aligned} S_p &= \{s \mid s \in S \text{ and } \ p\ s\} \\ S_{\neg p} &= \{s \mid s \in S\text{and } \neg p\ s\} \end{aligned}$$

We call the spaces in S_p *base spaces*, and the predicate will sometimes be called the *base predicate*.

[2]Note that the set *Boolean* does not contain *nil* as an element.

One base predicate we frequently use for the array space domain is *atom?*, which returns *true* if and only if an array has size one.

Let d be a divide function over S, and S' a subdomain of S. We say a space $s \in S$ is *grounded* on S' by d if either

1. $s \in S'$, or

2. $d\ s \neq nil$, and if $d\ s = (s_0, \ldots, s_{k-1})$ then s_i is grounded on S' for $i = 0$ to $k-1$.

Intuitively, a space is grounded on S' by d if it is either a member of S' or it is dividable by d and all the leaves in the recursive division tree are the spaces in S'.

Given a divide function d and a base predicate p over S, we define the *restricted subdomain* by d and p, written $S_{d,p}$, to be

$$S_{d,p} = \{s \mid s \in S,\ s \text{ is based on } S_p \text{ by } d\}$$

Finally, let f be a function over S, and p a base predicate over S. Then a function f_b is called a *base function* of f with respect to p if $f|_{S_p} = f_b|_{S_p}$. In other words, a base function must (and only needs to) agree with f for all the base spaces.

Examples:

1. Let V be the domain of vectors. Then $V_{atom?}$ consists of all vectors of size one. A vector v is grounded on $V_{atom?}$ by d_{lr} or d_{eo} if and only if $|v| = 2^m$ for $m \geq 0$. The restricted subdomain by *atom?* and d_{lr} or d_{eo} contains all vectors whose sizes are powers of two.

2. For the function *reduce* of Section 4.1.1, and the predicate *atom?*, one of the identity functions is a base function.

4.2.2 Higher Order Function PDC

Let us first define the following higher-order function PDC which takes five functions as argument and returns a function as the result:

$$PDC(d, c, g, h, p, f_b) = f_{dc}$$
$$\text{where } f_{dc} = p \rightarrow f_b; c : h : !f_{dc} : g : d$$

We can then show

Proposition 4.1 *Let $f : (X, d^{-1}) \rightarrow (Y, c)$ be a pseudomorphism with preadjust function g and postadjust function h; let p be a base predicate, f_b a base function of f for p, X' the restricted subdomain by p and d, namely $X' = X_{p,d}$. Then*

$$f|_{X'} = \text{PDC}(d, c, g, h, p, f_b)$$

Proof: All we need to show is that $f_{dc}\ x = f\ x$ for any $x \in X'$. Since $x \in X'$, x is either a base space or a dividable grounded space.

Case 1 x is a base space. Obviously we have

$$f_{dc}\ x = f_p\ x = f\ x.$$

Case 2 x is a dividable grounded space. Then by the definition of pseudomorphism, we have

$$\begin{aligned} & f\ x \\ = {} & f : d^{-1} : dx && \text{(since x is dividable))} \\ = {} & f : d^{-1}(x_0, \ldots, x_{k-1}) \\ = {} & c : h : !f : g(x_0, \ldots, x_{k-1} && \text{(by the definition of pseudomorphisms)} \\ = {} & c : h : !f : g : dx \end{aligned}$$

On the other hand, by the definition of PDC,

$$f_{dc}\ x = c : h : !f_{dc} : g : dx$$

since the above two recursive equations differ only in the names for the recursive function. It follows that the recursive functions are equal as long as they agree on the base spaces, which we know to be the case.

Q.E.D.

The significance of Proposition 4.1 is that a pseudomorphism can be computed by PDC given its constituents, provided the arguments are from the restricted domain. This, in turn, means that once identified to be a pseudomorphism, a function can be computed by PDC in the spirit of divide-and-conquer.

PDC is an acronym for *parallel divide-and-conquer.* We will also (loosely) call the function defined by PDC a "parallel divide-and-conquer".[3] but use the lower case abbreviation *pdc* to distinguish it when necessary. The argument functions of PDC are called the *constituents* of the defined *pdc.* The functionalities of the pseudomorphism and its constituents were all given previously, and can be summarized here as

pseudomorphism $f : X \rightarrow Y$

divide $d : X \rightarrow X^k$

combine $c : Y^k \rightarrow Y$

preadjust $g : X^k \rightarrow X^k$

postadjust $h : Y^k \rightarrow Y^k$

base predicate $p_b : X \rightarrow Boolean$

base function $f : X' \rightarrow Y'$, where$X' \subseteq X,\ Y' \subseteq Y$

It should be noted that Proposition 4.1 says nothing about what we can do if we happen to apply the pseudomorphism to a space which is not in the restricted

[3]Strictly speaking, the *pdc* defined by PDC is an algorithm for a function rather than the function itself; the function itself is the given pseudomorphism.

domain. Since the restricted subdomain can in some cases be an infinitely smaller portion of the domain itself, could it mean that it is not applicable most of the time?

To make discussion simple, let us first examine the effect of the restriction on one dimensional arrays. It should be easy to see that if the unbalanced division $d_{bin}\ m$ is used with the base predicate (*size* m), then the restricted subdomain coincides with the original. However (as shown by the example of last section), if we use balanced divisions d_{lr} or d_{eo}, a vast majority of the vectors in the domain will be lost in the restricted subdomain, which consists of only the vectors of sizes of power of two.

Fortunately, this problem is not a serious one. Although the restricted subdomain is small, it spans the entire domain in the sense that for every vector v in the original domain but not in the restricted domain, there is a unique vector v' in the subdomain such that $|v'|$ is the smallest power of two, equal to or greater than $|v|$. It is therefore always possible to embed a "lost" vector into a vector in the subdomain by padding it with some "dummy" entries. To compute a pseudomorphism on v, we can then compute f on v' by PDC by embedding v into v'; the result for $f\ v$ should be an easily separable portion of $f\ v'$.

The above discussion can be easily generalized to higher arity divisions on vectors and divisions on higher dimensional arrays when the balanced division(s) is (are) applied to one or more dimensions of arrays. Some details can be found, for example, in [3, 57]. This justifies the convention we will use from now on: instead of making the restricted subdomain an issue, we will simply say that a PDC computes a pseudomorphism.

4.2.3 First Example – Postmorphism PDC Algorithm for Scan

Scan (also called *prefix*) was first identified as a useful programming construct in APL back in 1962 [26]. Since then it has been introduced in a number of programming languages such as FP [4], and *Lisp of the CM [53]. While *scan* appears to be inherently sequential, many efficient parallel algorithms have been proposed on a number of data structures [31, 23, 39].

In Section 4.1.2, we gave a definition for *scan* on vectors, and showed that *scan* is a postmorphism. In this section, we will give the PDC program.[4]

We learned from Section 4.1.2 that ($scan \oplus$) is a postmorphism from the algebra $(V, {d_{lr}}^{-1})$ to itself, where the postadjust function is

$$h_{scan} \oplus = (id,\ !\oplus) : \#(nil,\ last\ 0)$$

Therefore, we know all the core constituents of the *pdc*: the divide function is d_{lr}, the combine function is c_{lr}, and the preadjust function is *id*, the postadjust function is given above. All we need to do now is to identify the base predicate and base function.

Although we have many (in fact an infinite number of) choices about the base predicate and base function, the simplest and most intuitive ones are *atom?* and *id* respectively. It is easy to verify that these two functions satisfy the constraints imposed by the definition of base predicate and base functions. This leads us to define the following PDC, which computes *scan* correctly by Proposition 4.1:

Algorithm 4.1 Postmorphism PDC algorithm for scan.

$$scan \oplus = PDC(d_{lr},\ c_{lr},\ id,\ (h_{scan} \oplus),\ atom?,\ id)$$
$$\text{where } h_{scan} \oplus = (id,\ !\oplus) : \#(nil,\ last\ 0)$$

The above algorithm says that, to compute *scan* with respect to a binary associative operator $\oplus$ over a vector v, we first test whether vector v is atomic, and if it is we return the vector itself; otherwise, we apply the divide function d_{lr}, apply

[4]The PDC algorithm for scan given here is a naive one. A better PDC algorithm will be given in Section 4.3.2.

scan recursively over the subarguments, postadjust the results, and then apply the combine function c_{lr}. The key component of the algorithm is the postadjust function h_{scan}, which when applied to two subvectors will fetch the last entry of the left for each entry of the right, and add ($\oplus$) the fetched value to its own value. The adjust function h_{scan} thus will only affect the value of the right subvector.

Example 4.1 Computation of scan by PDC

$$
\begin{aligned}
& scan\ +\ [1\ 2\ 3\ 4\ 5\ 6\ 7\ 8] \\
&= c_{lr} : (h_{scan}\ +) : !(scan\ +) : id : d_{lr}\ \ [1\ 2\ 3\ 4\ 5\ 6\ 7\ 8] \\
&= c_{lr} : (h_{scan}\ +) : !(scan\ +) : id\ ([1\ 2\ 3\ 4], [5\ 6\ 7\ 8]) \\
&= c_{lr} : (h_{scan}\ +) : !(scan\ +)\ ([1\ 2\ 3\ 4],\ [5\ 6\ 7\ 8]) \\
&= c_{lr} : (h_{scan}\ +)\ ([1\ 3\ 6\ 10],\ [5\ 11\ 18\ 26]) \\
&= c_{lr} : (h_{scan}\ +)\ ([1\ 3\ 6\ 10],\ [5\ 11\ 18\ 26]) \\
&= c_{lr} : (id,\ !\oplus) : \#(nil,\ last\ 0)\ ([1\ 3\ 6\ 10], [5\ 11\ 18\ 26]) \\
&= c_{lr} : (id,\ !\oplus)\ ([1\ 3\ 6\ 10],\ [(5,10)\ (11,10)\ (18,10)\ (26,10)]) \\
&= c_{lr}\ ([1\ 3\ 6\ 10],\ [15\ 21\ 28\ 36]) \\
&= [1\ 3\ 6\ 10\ 15\ 21\ 36]
\end{aligned}
$$

4.2.4 Illustrations

The computation process of a PDC on a given argument can be depicted by a *divacon graph* [36]. A divacon graph consists of two trees connected back-to-back by crossing edges, where these components are:

1. Divide Tree (on the top): Corresponds to the repeated application of the divide function and preadjust function, until the base spaces are reached at leaf level(s).

2. Combine Tree (at the bottom): Corresponds to the repeated application of the postadjust function and combine function until the "root" is reached again.

3. Crossing Edge: Corresponds to the application of the base function.

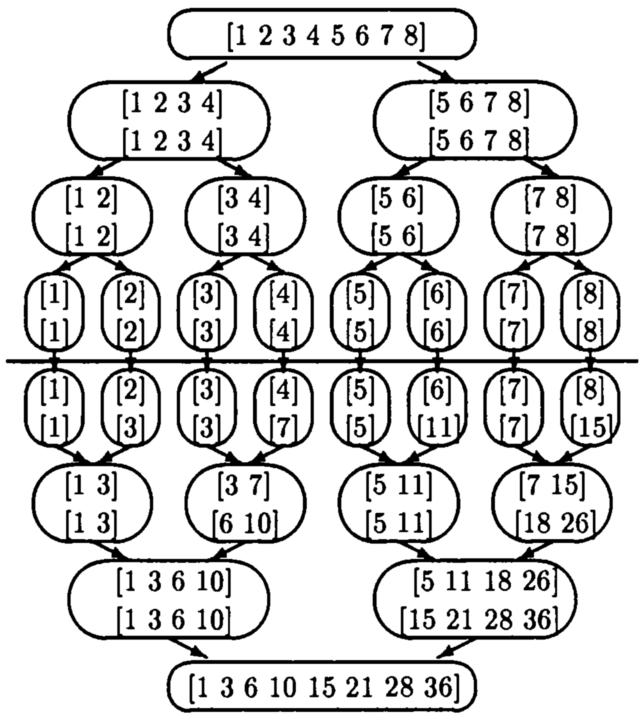


Figure 4.2: A Divacon graph for scan

Note also that the input and output of the PDC are represented by the roots of the divide and combine trees respectively.

In Figure 4.2, we give the divacon graph for (*scan* +) [1 2 3 4 5 6 7 8].

Divacon graphs give us snapshots of the entire computation process, showing the result of each application of each constituent function. Although there are occasions when this much detail can be appreciated, we often do not need so much detail to understand the process. A *simplified divacon graph* is different from a divacon graph in the following aspects

- If the *pdc* is a postmorphism (premorphism), only the combine (divide) tree is shown because the adjust function shown in the divide tree is always identity.

- There are no directed edges in the graph because the subspace relation indicated by the edges can often be easily inferred.

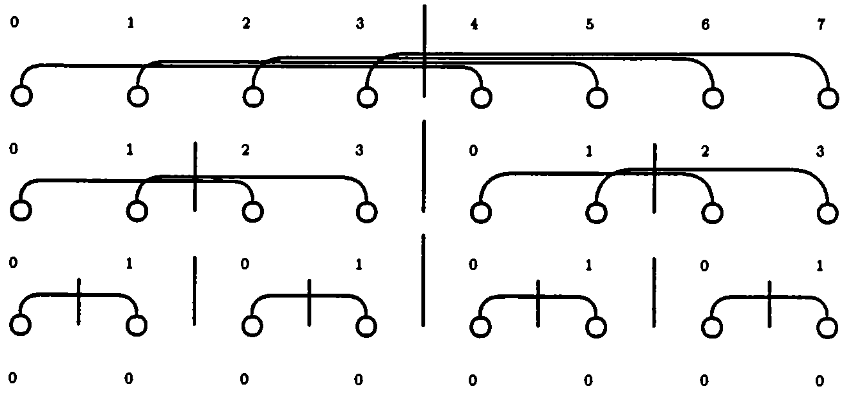


Figure 4.5: Premorphism PDC algorithm for broadcast

preadjust function:[7]

$$g_{broad} = (id, !other) : (nil, \#corr)$$

The above adjust function can be used to compute *broad* with any balanced division on vectors, including d_{lr} or d_{eo}, in a premorphism PDC algorithm.

Algorithm 4.3 Premorphism PDC algorithm for *broad*. See Figure 4.5.
Note: the divide and combine functions can be replaced by respectively d_{eo} and c_{eo}.

$$broad = PDC(d_{lr},\ c_{lr},\ g_{broad},\ id,\ atom?,\ id)$$

The function *broad* also has the interesting property that we can compute it by postmorphism with the same adjust function and the same divide function.

Algorithm 4.4 Postmorphism PDC algorithm for *broad*. See also Figure 4.6.
Note: the divide and combine functions can be replaced by respectively d_{eo} and c_{eo}.

$$broad = PDC(d_{lr},\ c_{lr},\ id,\ g_{broad},\ atom?,\ id)$$

[7]This adjust function assumes balanced division. In fact, it brings all entries from the first subvector to the second, and thus does more than absolutely needed. On the other hand, it is a correct adjust function for *broad*. Moreover, we can show that it will not cost more time on hypercube machines.

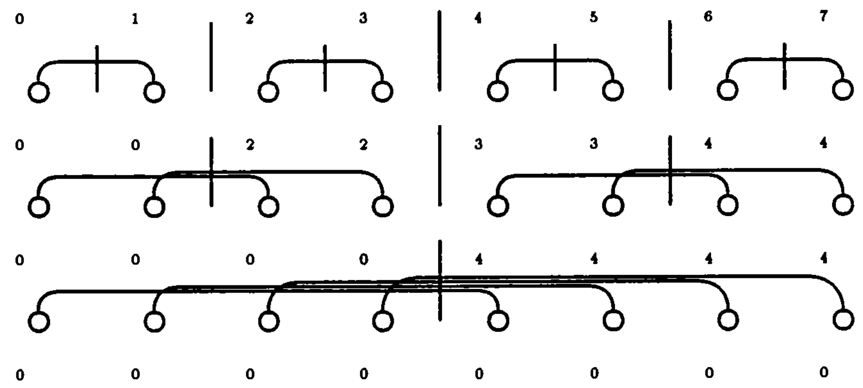


Figure 4.6: Postmorphism PDC algorithm for broadcast

4.3.2 A Better Scan Algorithm

The (*last* 0) communication in Algorithm 4.1 implies $O(n)$ fan-out, which, even if we implement it with the broadcast PDC algorithm will cost $O(\log n)$ (parallel) time. The complexity of Algorithm 4.1 therefore is $O(\log^2 n)$, which is not very good since there are known parallel logarithmic algorithms.

This, however, is not difficult to overcome by replacing the broadcast communication by *correspondent* communication. It is obvious that we could do so if and only if we can make the value of the last entry of each subvector available at all entries. Initially, when the subvectors are atomic, every one indeed has the value of the last entry, which happen to be identical to its own value. Inductively, suppose each entry in two subvectors has the last entry value in addition to its own, each entry can then be updated by correspondent communication followed by some local computation to reflect the last entry value of the combined vector.

The above discussion leads to Algorithm 4.5, which assumes the input vector has been transformed into vectors of pairs of identical elements. The output is also a vector of pairs, where the second elements reflect the scanned values of the vector entries.

Algorithm 4.5 Scan PDC Algorithm without Broadcast (Figure 4.7).

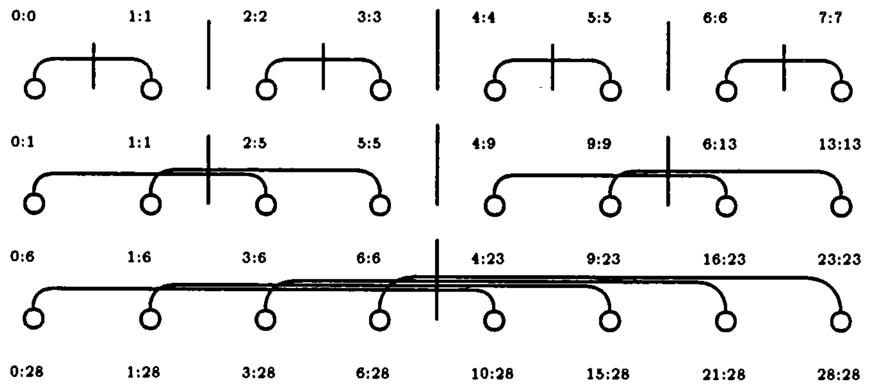


Figure 4.7: Scan PDC algorithm without broadcast

$$scan \oplus = PDC(d_{lr},\ c_{lr},\ id,\ (h_{scan} \oplus),\ atom?,\ id)$$

$$\text{where } f_b\ x = (x, x)$$

$$h_{scan} = (!loc1 \oplus, !loc2 \oplus) : \#!corr$$

$$loc1\ \oplus\ ((x1, sum1), (x2, sum2) = (x1, sum1 \oplus sum2)$$

$$loc2\ \oplus ((x1, sum1), (x2, sum2) = (x1 \oplus sum2, sum1 \oplus sum2)$$

As will be shown in Chapter 6, the complexity of Algorithm 4.5 indeed improves by a logarithmic factor in comparison to Algorithm 4.1 on hypercube machines. We call the technique used in Algorithm 4.5 *broadcast dissolving*, which in fact can be applied to many other problems.

4.3.3 Polynomial Evaluation

Assume that a polynomial of order $(n-1)$

$$P^n = a_0 + a_1 x^1 + \ldots + a_{n-1} x^{n-1}$$

is represented by two vectors A and X, where

$$A = [a_0 \ldots a_{n-1}]$$

$$X = [1\ x \ldots x]$$

Let us define the following function *poly*

$$poly\ P^n = VP, \text{ where } VP\ i = \Sigma_0^i\ a_i * x^i$$

Clearly, the function poly subsumes the ordinary polynomial evaluation since the last entry of its result gives the value of the entire polynomial.

Quite obviously, the function *poly* can be computed by the following program:

Algorithm 4.6 Polynomial Evaluation by PDC Composition.

$$poly(A, X) = (scan\ +)\ :\ [\times]A : (scan\ \times)\ X)$$

where the $(scan\ \times)$ computes the powers of x, and the $(scan\ +)$ computes the partial sums of the terms. The above program is asymptotically optimal since it has $O(\log n)$ time complexity on hypercube machines. However, it goes through two phases of divide-and-conquer which in fact can be merged.

Given a polynomial P^n, let us define

$$P_l^{n/2} = a_0 + a_1 * x + \ldots + a_{n/2-1} * x^{n/2-1}$$

$$P_r^{n/2} = a_{n/2} + a_{n/2} * x + \ldots + a_{n-1} * x^{n-1}$$

Then evidently we have

$$P^n = P_l^{n/2} + x^{n/2} * P_r^{n/2}$$

which immediately leads to the following one-phase DC algorithm:

Algorithm 4.7 One Phase PDC Algorithm for Polynomial Evaluation[8].

```
$poly = unzip.0 : poly_{dc} : zip$
  where
    $poly_{dc} = PDC(d_{lr},\ c_{lr},\ id,\ (id, !loc) :\ \#(nil,\ (last\ 0),\ atom?,\ poly_b)$
```

[8]The *zip* and *unzip* operations used in this and the following algorithm are defined in 3.6.5.

$$poly_b(a,\ x) = (v = a * x,\ pwr = x)$$
$$loc((v1, pwr1),\ (v2, pwr2)) = (v1 + v2, pwr^2)$$

The (*last* 0) communication used above implies broadcast. It can, however, be replaced by correspondent communication with the broadcast-dissolving technique that we used for *scan* by introducing a new variable and making the broadcasted value available at all entries:

Algorithm 4.8 Polynomial PDC without Broadcast Communication

```
$poly = unzip.0\ :\ poly_{dc} : zip$
  where
  $poly_{dc} = PDC(d_{lr},\ c_{lr},\ id,\ loc : \#!corr,\ atom?,\ poly_b) : zip$
  $poly_b\ (a, x) = (me = a * x, last = a * x, pwr = x)$
  $loc = (!left,\ !right)$
  $left((me_1, last_1, pwr_1),\ (me_2, last_1, pwr_2)) = (me1, (last_1 + pwr_2 * last_2), pwr_1^2)$
  $right((me_1, last_1, pwr_2),\ (me_2, last_1, pwr_2)) = (last_1 + me_1 * pwr_2, (last_1 + pwr_2 * last_2), pwr_2^2)$
```

4.3.4 Matrix Multiplication

Given two arrays, $A(m \times n)$ and $B(n \times k)$, the product $C = AB$ is defined to be

$$C(i,j) = \sum_{k=0}^{n} a(i,k) * b(k,j)$$

There are clearly two levels of parallelism implied by the definition: firstly, all entries can be computed in parallel; secondly, each entry is an inner product between two vectors, which can be efficiently computed in parallel, for example, by divide-and-conquer.

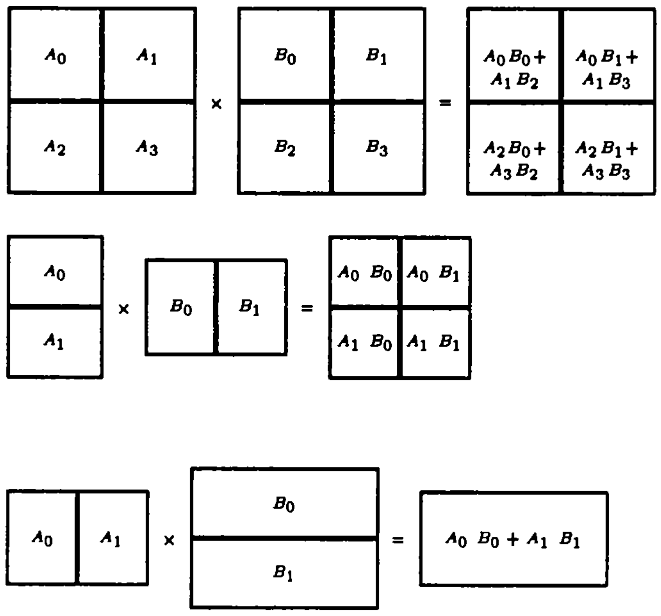


Figure 4.8: Three matrix block multiplications

Our interest is however in direct divide-and-conquer algorithms for matrix multiplication. A basic fact from linear algebra is that: the definition of matrix multiplication applies not only to matrix entries, but also to matrix blocks provided the matrices are properly divided (partitioned). Figure 4.8 gives three equations about matrix block multiplication, each of which suggests a divide-and-conquer algorithm for matrix multiplication.

Corresponding to the first equation, Algorithm 4.9 employs an 8-ary division which is defined by the selected outer-product (Section 3.4.4) of block matrix divisions. The postadjust function performs pair-wise matrix additions on eight pairs of submatrices, and thus reduces the number of pairs by half. The base function is simply the identify function.

Algorithm 4.9 PDC Algorithm for Matrix Multiplication (8-ary Division).

```
$mm = (d,\ c,\ id, h_{mm}, atom?,\ id)$
    where
        $d = sel\ (1,1,0,0,0,0,1,1,1,1,0,0,0,0,1,1) : {d_{lr}}^2 \cdot {d_{lr}}^2$
        $c = {c_{lr}}^2$
        $h_{mm}(p_0,p_1,p_2,p_3,p_4,p_5,p_6,p_7) = (p_0[+]p_2,\ p_1[+]p_3,\ p_4[+]p_6,\ p_5[+]p_7)$
        .
```

Corresponding to the second equation, Algorithm 4.10 employs a 4-ary division which is defined by the (unselected) outer-product of block matrix divisions. Since both the adjust functions are identity functions, this is a pure-morphism PDC algorithm. The base function is the dot-product of two vectors, which can be computed by a pair-wise multiplication followed by a reduction with respect to addition.

Algorithm 4.10 PDC Algorithm for Matrix Multiplication (4-ary Division).

```
$mm = (d,\ c,\ id, id, atom?,\ mm_b)$
    where
        $d = d_{lr} \cdot d_{lr}$
        $c = {c_{lr}}^2$
        $mm_b = reduce : [*]$
```

Corresponding to the last equation, Algorithm 4.11 employs the binary division defined by the selected outer-product of block matrix divisions. The postadjust function performs pair-wise matrix additions on two matrices and reduces them into one. The base function is the outer-product of two vectors, denoted by $\times$, and maps two vector of size n into a matrix of size n by n.

Algorithm 4.11 PDC Algorithm for Matrix Multiplication (Binary Division).

$$mm = (d,\ id, id,\ [+],\ mm_b)$$

where

$$d = sel\ (1,0,0,1) : (d_{lr}\ 1) \cdot (d_{lr}\ 0)$$

$$h_{mm} = [+]$$

$$mm_b\ (A, B) = A \times B$$

Observe that the divide functions in both Algorithm 4.9 and Algorithm 4.10 are dynamic divide functions with expanding factor two (Section 3.4.4), while the divide function in Algorithm 4.11 is static. On the other hand, the base functions in the three algorithms respectively preserve, shrink, and increase the sizes of the arguments. These characteristics will be shown to have great impacts on the time and process complexities of these algorithms when computed on parallel computers (Chapter 6).

4.3.5 Monotonic Sort – A Second Order PDC

PDCs can not only be composed together as shown in Section 4.3.3, but can also be nested into other PDCs to compute more complicated problems. A *higher order* PDC is a PDC that uses other PDC in its constituent(s). The number of nesting levels is called the *order* of the PDC. In this section, we will show a second order sorting PDC algorithm.

Different from the well-known bitonic sort algorithm, the following PDC program always arranges entries in a monotonic ascending order. It should be observed that the program is a second-order pseudomorphism, where a premorphism is nested inside a postmorphism.

Algorithm 4.12 A Second Order PDC Sorting Algorithm.

$$sort = PDC(d_{lr},\ c_{lr},\ id,\ loc : \#!mirr,\ atom?,\ id)$$

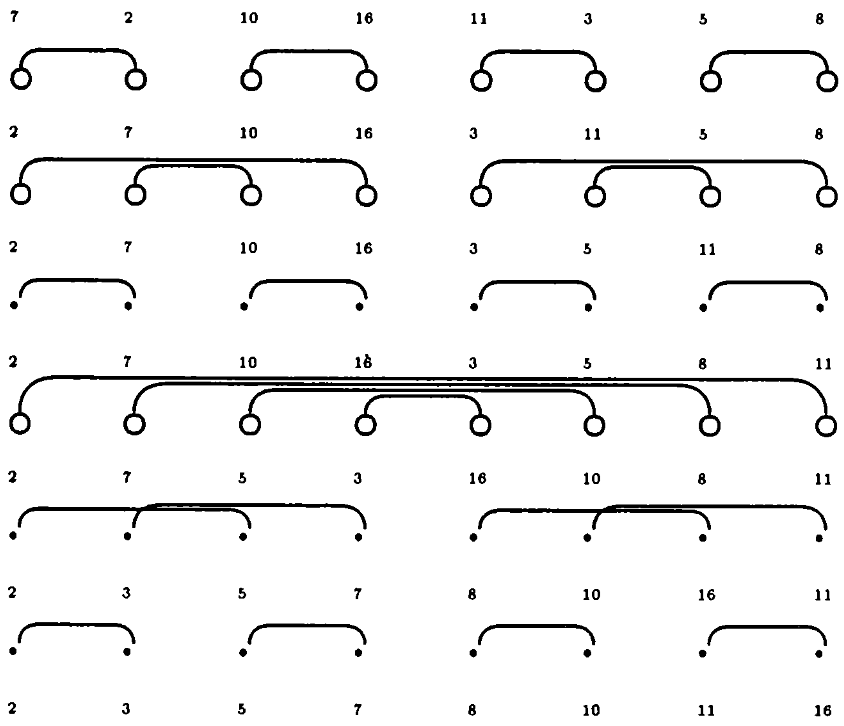


Figure 4.9: Monotonic sort – a second order PDC algorithm

where $loc = !nest \; : \; (!\min, !\max)$

$$nest = PDC(d_{lr},\; c_{lr},\; (!\min, !\max) : \#!corr,\; id,\; atom?,\; id)$$

The algorithm is illustrated in Figure 4.9, in which computation corresponding to the top level is indicated by the circles, and computation corresponding to the second level is indicated by the solid dots.

Chapter 5

Sequential Divide-and-Conquer

In all divide-and-conquer algorithms, we divide a problem into several subproblems and then recursively solve the subproblems. The pseudomorphism model introduced in the last chapter corresponds to the class of DC algorithms in which the order of the recursive computations over the subproblems is left unspecified. The subproblems therefore can be computed in parallel by the PDC programming construct.

There are also problems which can be solved by recursively reducing them to subproblems whose computations have a particular order imposed upon them. Problems of this nature are the subject of this chapter.

5.1 Basic Notions

Let (X, c_x) and (Y, c_y) be two k-ary algebras. We say a function $f : X \rightarrow Y$ is a *crossmorphism* if there are $(k-1)$ *crossadjust functions* $\vec{\mu} = (\mu_0, \ldots, \mu_{k-2})$, where $\mu_i : X \times Y \rightarrow X$ for $i = 0$ to $(k-2)$ such that[1]

$$f : c_x = c_y : f :: \vec{\mu}$$

[1] Recall that "::" is the *sequential tuple distribution* (between a function and a vector of functions). See 3.6.4.

By definition, a pure morphism is a special case of the crossmorphism, where each of the cross adjust functions happens to be the projection function that returns the first element of pairs.

The scan function over vectors with respect to a binary associative operator, which we studied in the last chapter in fact is (also) a crossmorphism. Let us define

$$\mu_{scan} \oplus = (eq?\ 0) \Rightarrow (!\oplus : \#(last\ 0))$$

Then it is easy to verify that the following equation holds for the function $(scan \oplus)$:

$$scan \oplus : c_{lr} = c_{lr} : (scan \oplus) :: \mu_{scan}$$

The function $(scan \oplus)$ has been shown in Chapter 4 to be a pseudomorphism. The above example therefore shows that crossmorphisms and pseudomorphisms are not mutually exclusive. From now on, we will extend the meaning of pseudomorphism so that it includes crossmorphisms.

5.2 The Programming Construct SDC

5.2.1 SDC

Let us define the following higher order function SDC, which takes functions and tuples of functions, and returns another function as the result:

$$SDC(d, c, \vec{\mu}, p, f_b) = f_{dc}$$
$$\text{where } f_{dc} = p \rightarrow f_b; c : f_{dc} :: \vec{\mu} : d$$

We can then show

Proposition 5.1 *Let $f : (X, d^{-1}) \rightarrow (Y, c)$ be a crossmorphism with crossadjust functions $\vec{\mu}$, let p be a base predicate, f_b a base function of f for p, X' the restricted subdomain by p and d, namely $X' = X_{p,d}$, then*

$$f|_{X'} = \mathrm{SDC}(d, c, \vec{\mu}, p, f_b)$$

Proof: Similar to that of Theorem 4.1.

Q.E.D.

In other words, let

$$f = SDC(d, c, \vec{\mu}, p, f_b)$$

and let $x \in X_{p,d}$, then we can compute $f\ x$ by

1. if $p\ x$ then return $f_b\ x$;

2. else, return $c_y\ (y_0, y_1, \ldots, y_{k-1})$ where

$$\begin{aligned}
&y_0 = f x_0 \\
&y_1 = f : \mu_0\ (x_1, y_0) \\
&\ldots \\
&y_{k-1} = f : \mu_{k-2}\ (x_{k-1}, y_{k-2}) \\
&(x_0, \ldots, x_{k-1}) = d\ x
\end{aligned}$$

Due to the imposed order, SDC must apply the recursive applications of the defined function in a sequential order. This is to be contrasted with PDC where the recursions can be (but do not have to be) performed in parallel. This is the reason we call the construct SDC, standing for *sequential divide-and-conquer*. The function arguments to a SDC are called the *constituents* of the defined crossmorphism. The significance of Proposition 5.1 is that once the constituents of a crossmorphism are identified, the crossmorphism can then be computed by SDC. Similar to what we did in Chapter 4, we will loosely speaking of a crossmorphism as a "sequential divide-and-conquer", or a *sdc*.

We summarize the functionalities of a crossmorphism and its constituents

crossmorphism $f : X \rightarrow Y$

divide $d : X \rightarrow X^k$

combine $c : Y^k \rightarrow Y$

crossadjust $g : X^k \rightarrow X^k$

base predicate $p_b : X \rightarrow Boolean$

base function $f : X' \rightarrow Y'$, where $X' \subseteq X,\ Y' \subseteq Y$

5.2.2 An Example

From Section 5.1, we learned that *scan* is a crossmorphism from the algebra (V, c_{lr}) to (V, c_{lr}) with the crossadjust function

$$\mu_{scan} \oplus = (eq?0) \Rightarrow (!\oplus : \#(last\ 0))$$

By Proposition 5.1, the function $(scan\ \oplus)$ can then be computed by the following SDC algorithm

Algorithm 5.1 Crossmorphism PDC algorithm for scan.

$$scan\ \oplus = SDC(d_{lr},\ c_{lr}, (\mu_{scan}\ \oplus),\ atom?,\ id)$$
$$\text{where } \mu_{scan} \oplus = (eq?\ 0) \Rightarrow (!\oplus : \#(last\ 0))$$

Algorithm 5.1 means that to compute *scan* with respect to a binary associative operator $\oplus$ over a vector v, we first test if vector v is atomic, if it is, the result is the vector itself; otherwise, we apply the divide function d_{lr}, apply *scan* recursively **first** over the left subvector, the crossadjust function μ_{scan} is then applied to the right subvector and the result of the left subvector, we **then** apply *scan* recursively over the adjusted right subvector, and finally apply the combine function c_{lr}. The key component of the algorithm is the crossadjust function μ_{scan}, which when applied to two vectors will fetch the last entry of the left subvector and add it $(\oplus)$ it to the first entry of the right subvector. The crossadjust function μ_{scan} thus will only affect the value of the first entry of the right subvector.

Example 5.1 Computation of (*scan* +) by SDC

$$\begin{aligned}
& scan\ +\ [1\ 2\ 3\ 4\ 5\ 6\ 7\ 8] \\
&= c_{lr} : (scan\ +) :: (\mu_{scan}\ +) : d_{lr}\ [1\ 2\ 3\ 4\ 5\ 6\ 7\ 8] \\
&= c_{lr} : (scan\ +) :: \mu_{scan}\ +)\ ([1\ 2\ 3\ 4],\ [5\ 6\ 7\ 8]) \\
&= c_{lr}\ (scan[1\ 2\ 3\ 4],\ scan : \mu_{scan}\ ([5\ 6\ 7\ 8],\ scan[1\ 2\ 3\ 4])), \\
&= c_{lr} : ([1\ 3\ 6\ 10],\ (scan\ +) : \mu_{scan}\ ([5\ 6\ 7\ 8],\ scan[1\ 3\ 6\ 10])), \\
&= c_{lr} : ([1\ 3\ 6\ 10],\ (scan\ +)\ [15\ 6\ 7\ 8]) \\
&= c_{lr}\ ([1\ 3\ 6\ 10],\ [15\ 21\ 28\ 36]) \\
&= [1\ 3\ 6\ 10\ 15\ 21\ 36]
\end{aligned}$$

5.3 Case Studies

5.3.1 Gaussian Elimination

Let A be an n by n matrix, X a column vector of n unknowns, B a vector of size n, a linear system with n unknowns then has the form

$$AX = B$$

Gaussian elimination is a reduction of the above to an equivalent system [12]

$$A'X = B'$$

where A' is a lower triangular matrix.

A common approach to Gaussian elimination is to do it in n steps; at the ith step the ith diagonal element is used as the pivot element, and the entries in the ith column below the diagonal element are reduced to zeros [19]. In this section, we show that Gaussian elimination is a crossmorphism, and give an SDC algorithm by identifying the constituents.

We will represent a linear system by by a matrix M of shape $(n, n+1)$, where M's first n columns correspond to the matrix A, and the last column corresponds to the vector B. For a matrix of shape (n, m), where $n < m$, there is always a square submatrix of shape (n, n) consisting of the columns with indices from $(m - n - 1)$ to $(m - 1)$. This square matrix in the following discussion will be referred to as *main square submatrix of M*.

Instead of Gaussian elimination, we define the function ge which takes a matrix P of shape (n, m) where $n < m$ as argument, and returns a matrix P' of the same shape but from which all the entries under the main diagonal of the main square submatrix of P are eliminated. Hence, the function ge subsumes Gaussian elimination since when given a matrix of size $(n, m + 1)$ representing a linear system, the function ge will return the result of Gaussian elimination.

Follows the definition, it is easy to see that

$$ge : (c_{lr}\ 0) \neq (c_{lr}\ 0) : !ge$$

In other words, for the algebra induced by the unbalanced division that divides a matrix into two submatrices R and Q, where R consists of the first row, Q the rest, the function ge is not a pure morphism. The reason is simple, in comparison with the left-hand side, the right-hand side has an extra non-zero column.

Now let R be a row matrix of shape (1, m), where the first k entries are zeros,

$$R = \begin{bmatrix} 0 & \cdots & 0 & R_k & R_{k+1} & \cdots & R_{m-1} \end{bmatrix}$$

and let Q a matrix of shape (k, m), where all the entries in the first k columns are zeros, namely

$$Q = \begin{bmatrix} 0 & \cdots & 0 & B_{(0,k)} & B_{(0,(k+1)} \cdots & B_{0,(m-1)} \\ 0 & \cdots & 0 & B_{(1,k)} & B_{(1,(k+1)} \cdots & B_{1,(m-1)} \\ & & & \cdots & & \\ 0 & \cdots & 0 & B_{(k,k)} & B_{(k,(k+1)} \cdots & B_{k,(m-1)} \end{bmatrix}$$

We define the following function μ, which will eliminate entries $B(i, k)$ for $i = 0$ to

$(n-1)$ using R_k as the pivot.

$$\begin{aligned}
&\mu(Q,R) = loc : \#(rowbr\ k) : \#(colbr\ 0)(Q,R) \\
&\quad \text{where} \\
&\qquad loc\ ((x,y),\ (u,v)) = x * v/u - y \\
&\qquad k = m - n \\
&\qquad (m,n) = \$Q
\end{aligned}$$

Observe that when the communication function $\#(colbr\ 0)$ is applied to (Q,R), a matrix, say Q', is returned, where

$$Q'(i,j) = (Q(i,j), R(0,j))$$

the intra-array communication $(rowbr\ k)$ is then applied to Q' another matrix, say Q'', is applied where

$$\begin{aligned}
&Q''(i,j) \\
&= (Q'(i,j), Q'(i,k)) \\
&= (Q(i,j), R(0,j), (Q(i,k), R(0,k))
\end{aligned}$$

The function *loc* defined by pattern matching then performs local computation within each entry, and maps Q'' to Q''', where

$$Q'''(i,j) = Q(i,j) * R(0,k)/Q(i,k) - R(0,j)$$

For entries on the kth column, since $j = k$, the above becomes

$$Q'''(i,k) = Q(i,k) * R(0,k)/Q(i,k) - R(0,j) = 0$$

Therefore, given Q and R, the function μ eliminates the kth column of Q using R as the pivot row.

With the function μ defined, it is easy to verify that

$$ge : c_{lr} = c_{lr} : ge :: \mu$$

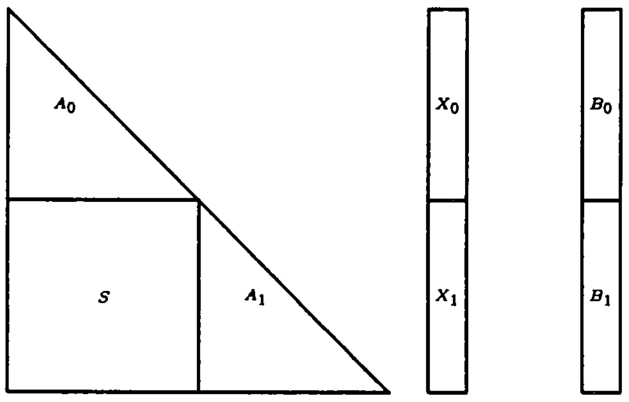


Figure 5.1: Division of a linear triangular system

in other words, the function *ge* is a crossmorphism with the crossadjust function μ. It is also a simple matter to verify that (*atom*? 0) and *id* can serve respectively as the base predicate and base function for the function *ge*, we therefore have the following SDC algorithm for Gaussian elimination.

Algorithm 5.2 Gaussian Elimination by SDC

$$ge = SDC((d_{ht}\ 0),\ (c_{ht}\ 0),\ \mu_{ge},\ (atom?\ 0),\ id)$$
where μ *as defined above.*

5.3.2 Linear Lower Triangular System

A linear lower triangular system (LLTS) has the form

$$AX = B$$

where A is a lower triangular matrix of size n, X is a column vector of n unknowns to be solved, B is a column vector of size n.

In Figure 5.1, we show how the lower triangular matrix can be divided into two smaller triangular matrices A_0 and A_1, and a square matrix S. Also in the same figure, we show how the vector X can be divided into X_0 and X_1, the vector B into B_0 and B_1. Now we have two smaller triangular systems derived from the original

$$A_0X_0 = B_0$$
$$A_1X_1 = B_1$$

Can we solve the original system by solving the two smaller systems recursively and catenating the solution vectors? The answer is no. The reason for the answer is simple: the solution of the original system depends on the entire triangular matrix A, including the entries in the square submatrix S in Figure 5.1; the solutions to the two smaller triangular systems on the other hand are independent of S.

It can be observed, however, that the solution to the first subsystem $A_0X_0 = B_0$ indeed coincides with values of the first half unknowns in the system $AX = B$. The correct second half of the solution can be found by solving

$$A_1X_1 = B', \text{where } B' = B_1 - X_0S$$

The above discussion points to a SDC algorithm for linear lower triangular system solver: we first divide the system as shown in Figure 5.1; then solve the top sub-LLTS; then use the solution and the square submatrix to adjust the right hand side of the bottom LLTS; then solve the adjusted bottom LLTS; finally, catenate the two sub-solutions.

Since triangular matrix is a special case of matrix, the divide function over triangular systems can be defined in terms of divide functions over matrices. We thus define the divide function for LLTS by

$$d_{LLTS}(A, B) = ((A_0, B_0), S, (A_1, B_1))$$
$$\text{where}$$
$$(A_0, Zeros, S, A_1) = d_{lr}^2\ A$$
$$(B_0, B_1) = (d_{lr}0)B$$

If we choose to divide the LLTS recursively all the way to LLTS of size one, the base predicate can be defined as

$$atom_{LLTS}(A, B) = atom?\ A$$

The base function for LLTS then obviously can be defined as[2]

$$LLTS_b(A, B) = B(0,0)/A(0,0)$$

Note that although the divide function has arity of three, the final vector is only the concatenation of two sub-solution vectors. We therefore must also define a special combine function for LLTS

$$c_{LLTS}(V_0, X, V_1) = c_{lr}(V_0, V_1)$$

From the above discussion, clearly we need two operations for adjustment. The first one is to perform multiplication between a square matrix and a column vector, which can be computed by the PDC *mm* of the Section 4.3.4. The second one is to perform the pair-wise subtraction, which can be computed by a correspondent communication followed by an array distribution (Section 3.6.3).

We therefore have the following SDC algorithm for LLTS:

Algorithm 5.3 Linear Lower Triangular System (LLTS) by SDC

$$LLTS\ = SDC(d_{LLTS},\ c_{LLTS},\ (\%\mu_0, \mu_1)\ atom_{LLTS}?,\ LLTS_b)$$

where

$$\mu_0(S, X) = mm(S, X)$$

$$\mu_1(X, V) = X[-]V$$

[2]In this discussion, we have treated column vectors B and X as one column matrices, this is why B in the following is indexed by pairs.

Note that the "%" in front of μ_0 is to prevent recursive application of LLTS to the result of μ_0 (which would be an error) (see Section 3.6.4). The "[-]" in the above program is the entry-wise subtraction between two vectors (and higher dimensional arrays), which, as discussed in Section 3.6.3, can be decomposed into a *correspondent* communication and an array distribution in Divacon notation.

Algorithm 5.3 is a second order divide-and-conquer since one of its adjust functions calls for another (P)DC algorithm (*mm*). It is also an example showing how SDC and PDC algorithms can interact with each other.

5.4 Parallelism in SDC Algorithms

Parallelism in the PDC algorithms we studied in Chapter 4 is apparent: the recursive computation over the substructures can be done in parallel because there is no a dependency imposed over the recursions. This is no longer true with the case of SDC algorithms we study in this chapter.[3] One may ask if the crossmorphism model and SDC construct have any significance for the purpose of parallel computation.

To answer the question, let us first consider the time used by DC algorithms on sequential computers. Suppose f is a function computed by a DC algorithm where binary balanced division is used, $T_f(n)$ is the time used to compute f on data of size n, then $T_f(n)$ is given by the following recurrence regardless of whether f is a PDC or SDC

$$T_f(n) = 2 * T_f(n/2) + T_{dc}(n) + T_{adjust}(n) + T_p(n)$$
$$T_f(C) = T_0$$

where

T_{dc}: time used by divide and combine operations.

[3]In fact, it can be proven formally that there is a total order over the SDC constituents due to the data dependency relation over the constituent functions. See [37].

T_{adjust}: time used by (pre, post, cross) adjust function(s)

T_p : time used by the base predicate.

C, T_0 : some constants representing the size of base space, and time used to compute the base function respectively.

From mathematical point of view, the solution of the above recurrence can be reduced in two different ways:

- Eliminate the factor of two in the recursive term $2 * T_f$.
- Reduce the values of other non-recursive terms: T_{dc}, T_{adjust}, and T_p.

Now let us consider computing f on parallel computers.

In the case of PDC, the recursive applications of f can be computed in parallel. It follows that the factor of two disappears in the recurrence equation. A speedup therefore can be expected when f is computed by parallel computer.

In the case of SDC, the factor of two cannot be eliminated because the recursive applications of f must be computed sequentially even give a parallel computer. A speedup however can still be expected assuming that we can reduce the time to compute other constituents by computing the constituents in parallel.

Let us consider Algorithm 5.3 as an example. Since the adjust function μ_0 has the dominating cost among the constituents, the recurrence for the time complexity can be written as

$$T_{LLTS}(n) = 2 * T_{LLTS}(n/2) + O(T_{\mu_0}(n))$$
$$T_f(1) = T_0$$

Since we know that μ_0 is the multiplication between a square matrix and a vector, it can be computed sequentially in $O(n^2)$ times. On the other hand, we will see that the same multiplication can be computed in parallel in $O(\log(n))$ time. This gives

$$T_{LLTS}(n) = \begin{cases} 2 * T_{LLTS}(n/2) + O(n^2) = O(n^2 * \log(n)), & \text{sequential case} \\ 2 * T_{LLTS}(n/2) + O(\log(n)) = O(n), & \text{parallel case} \end{cases}$$

Thus, the SDC Algorithm 5.3 can gain a $O(n * \log(n))$ speedup when computed on parallel computers despite its sequential nature.

The speedup of SDC algorithms on parallel computers showed us that there can be parallelism in SDC algorithms. Clearly, the parallelism can only be explained by the parallelism inside of the constituent functions. We call this type of parallelism *additive parallelism* since it reduces the complexity of a DC algorithm by reducing the additive terms in the recurrence. In contrast, we call the parallelism obtained by computing two or more functions in parallel *multiplicative parallelism.*

It should be pointed out PDC algorithms can contain both types, not just multiplicative type, of parallelism. The SDC algorithms, on the other hand, only contains additive parallelism.

5.5 Balanced Division in SDCs

In a PDC algorithm, a constituent function can be applied to all the subspaces at a given level of the divacon graph. It follows that smaller depth of the divacon graph takes shorter time to compute. Balanced division therefore is crucial to the performance of PDCs computed on parallel computers. In a SDC algorithm, however, a constituent function can never be applied to more than one subspace simultaneously. In other words, the nodes of a divacon graph must be traversed by a SDC process one by one in a linear order. It is therefore not clear if the balanced divisions, which will only reduce the depth, not the size, of the divacon graph, can contribute to a better speedup for parallel SDC algorithms. In this section, we will make an investigation of the question by studying some examples.

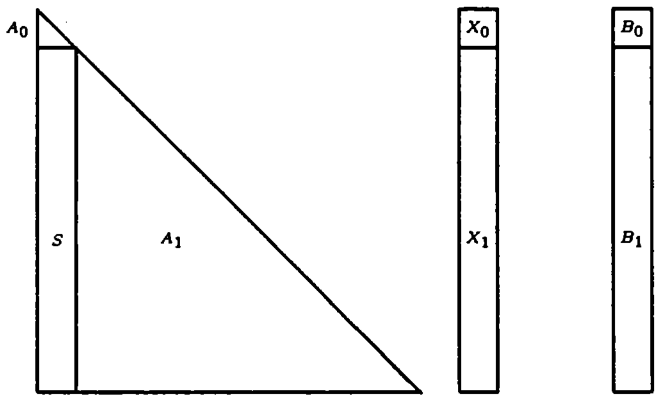


Figure 5.2: Unbalanced division of a linear triangular system

Algorithm 5.3 is a SDC algorithm using balanced division. It in fact can be smoothly converted to unbalanced versions. In an extreme case, we can divide the triangular system of size n to two smaller triangular systems with size one and $(n-1)$ respectively. The division is illustrated in Figure 5.2, and can be defined as

$$
\begin{aligned}
& d_{LLTS_u}(A, B) = ((A_0, B_0), S, (A_1, B_1)) \\
& \quad \text{where} \\
& \qquad (A_0, Zeros, S, A_1) = d_{ht}{}^2\ A \\
& \qquad (B_0, B_1) = (d_{ht}\ 0) B
\end{aligned}
$$

Corresponding to the above unbalanced division, we will use the following unbalanced combine operation

$$c_{LLTS_u}(V_0, X, V_1) = c_{ht}(V_0, V_1)$$

Now, we can simply substitute the above unbalanced divide and combine operations into Algorithm 5.3 to have an unbalanced SDC algorithm for linear triangular systems.

Algorithm 5.4 Linear Lower Triangular System (LLTS) by SDC (Unbalanced)

$$LLTS \;=\; SDC(d_{LLTS_u},\; c_{LLTS_u},\; (\%\mu_0, \mu_1)\; atom_{LLTS}?,\; LLTS_b)$$

where

$$\mu_0(S, X) = mm(S, X)$$
$$\mu_1(X, V) = X[-]V$$

It should be pointed out that the adjust function μ_0 is now degenerated to the matrix multiplication between a column vector and a matrix of size one. If one prefers, it can be rewritten as a broadcasting (singleton matrix entry) followed by local multiplication (each entry of the column multiplies itself to the broadcasted value).

The time complexity for the above algorithm clearly can be given by the following recurrence:

$$T_{LLTS_u}(n) = T_{LLTS}(n-1) + O(T_{\mu_0}(n))$$
$$T_f(1) = T_0$$

On parallel computers, the time $T_{\mu_0} = \Omega(\log(n))$[4]because of the broadcast required by μ_0. By a simple induction, we can show that $T_{LLTS_u} = O(n * \log(n))$. This is to be contrasted to the $O(n)$ complexity of the balanced Algorithm 5.3. The balanced division therefore has brought us an additional $O(\log(n))$ speedup.

The above example tells us that balanced division can (but not always) contribute to speeding up parallel SDC algorithms. For a given SDC problem, whether we can benefit from balancing can be best decided by comparing the solutions of the two recurrences, one for the unbalanced, another the balanced.

[4]Here Ω means "at least of the order of".

Chapter 6

Complexity Analysis

The central issue in the last two chapters can be said to be the specification of DC algorithms. Our interest in this chapter is however different. What we will study now is how to measure the resources consumed by DC algorithms with given specifications.

Our approach is to show how the complexity of the functional forms can be given in terms of their constituents. The complexity of DC algorithms specified hierarchically by functional forms thus can be analyzed systematically given the complexity of the primitives. The effectiveness of the methodology is illustrated with a number of full examples.

6.1 Communications

As first step, we will give an abstract characterization of communication.

Consider a set $V = \{v_0, \ldots, v_{n-1}\}$ of n elements. A *communication* over V is to send the value of each element v_i to a designated set of elements $V_i \subseteq V$ associated to v_i for $i = 0$ to $(n-1)$. Equivalently, a communication can be described by a binary relation R over V defined by $v_i R v_j$ if and only if $v_j \in V_i$. It follows that a communication over a set V can be characterized by a directed *communication graph* $D = (V, R)$, where R is the relation defined by the communication.

Some notions about communications can then be defined:

fan-in of a vertex the indegree of the node in the communication graph.

fan-out of a vertex the outdegree of the node in the communication graph.

fan-in of the communication the fan-in of the the vertex with maximum fan-in among all vertices.

fan-out of the communication the fan-out of the vertex with maximum fan-out among all vertices.

partial permutation communication a communication whose fan-in and fan-out equal or smaller than one.

permutation communication a communication in which every vertex has fan-in and fan-out exactly equal to one.

broadcast communication a communication in which one vertex has O(n) fan-out, all the others have O(1) fan-in in each connected component of the size n.

Suppose that a set V is embedded into a set P of multiprocessors, the time taken by a communication over V depends on not only the communication graph, but also the topology of the interconnections between the processors in P and the mapping that embeds the V into P. If we restrict our attention to the family of hypercube parallel computer and those that are equivalent in communication capacities, such as butterfly and cube connected-cycles, then the communication graph alone provides information about the communication complexity that is independent to the mapping from V to P.

For instance, by the work of Valiant [55], Ho and Johnsson [29], we know that on a hypercube machine

- a permutation communication takes $O(\log n)$ time.
- a broadcast communication takes $O(\log n)$.

where n is the size of the set V.

Nevertheless, the mapping that embeds V to P does have an impact on the locality of the communication, namely, the distance that a message has to travel from the source vertex to the destination. The *ideal locality* of a communication is such that every pair of communication partners are direct neighbors in the topology of the host parallel computer. While ideal locality cannot generally be realized, it can be attained for certain permutation communications.

Let $V = v_0, \ldots v_{n-1}, v_n, \ldots, v_{2n-1}$, where $n = 2^m$. A *correspondent communication over* V has the graph $D_{corr} = (V, R_{corr})$, where $v_i R_{corr} v_j$ if and only if $i = j(modn)$. Similarly, a mirror-image communication over V has the graph $D_{mirr} = (V, R_{mirr})$, where $v_i R_{mirr} v_j$ if and only if $v_i + v_j = 2n - 1$.

Proposition 6.1 *The correspondent communication over V in above has ideal locality by mapping the indices of elements of V with binary coding to processors of a binary hypercube. Similarly, the mirror-image communication over V of the above can realize ideal locality by mapping the indices of elements of V with Gray coding* [11] *to processors.*

Proof

- Correspondent Communication: By definition of R_{corr}, $v_i R v_j$, if and only if i and j differ only in their most significant bit in their binary representation of $(m + 1)$ bits. The vertices v_i and v_j therefore are the $(m + 1)$th dimensional neighbors of each other on binary hypercube if binary coding is used.
- Mirror-Image Communication: By definition of R_{mirr}, $v_i R_{mirr} v_j$ if and only if i and j differ in their most significant bit in their Gray code representation of

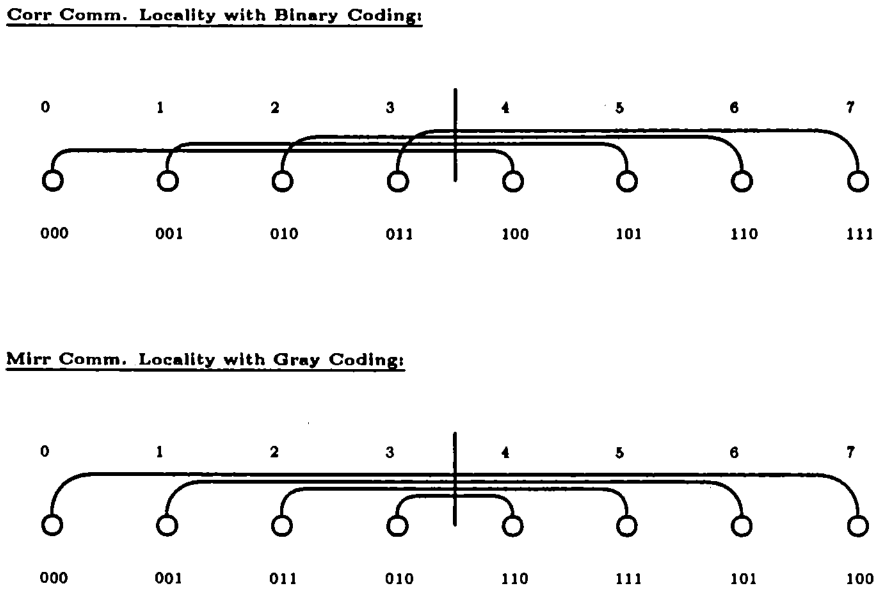

Corr Comm. Locality with Binary Coding:
0
1
2
3
4
5
6
7
000
001
010
011
100
101
110
111
Mirr Comm. Locality with Gray Coding:
0
1
2
3
4
5
6
7
000
001
011
010
110
111
101
100

Figure 6.1: Locality of correspondent and mirror-image communications

$(m+1)$ bits. The vertices v_i and v_j therefore are also the $(m+1)$th dimensional neighbors of each other on binary hypercube if Gray coding is used.

Q.E.D.

The above proposition is illustrated for the case that the set V contains eight elements in Figure 6.1.

The above discussion can be applied to the communications defined with our Divacon notations. Since a communication over array(s) is always defined by a communication function, not a relation, the fan-in of array entries is always one at most. It is easy to see that broadcast, correspondent, and mirror-image communications over arrays can be respectively modeled by the broadcast, correspondent, and mirror-image communication we discussed in above over a set. We therefore can establish the bounds of the time used by communication functions most frequently used in Divacon programs. Let n be the size of the (sub)arrays, all the communication generators as defined in Section 3.6.3, we assume in this chapter that

- The *correspondent* communication over arrays generated by the generator *corr* takes $O(1)$ time provided binary code mapping.

- The *mirror-image* communication over arrays generated by the generator *mirr*$O(1)$ time provided Gray code mapping.

- All broadcast communications takes $O(\log n)$ time, including communications generated by $(br\ \vec{i})$. *col-br, dia-to-row-br, dia-to-col-br, v-to-m-row, v-to-m-col.*

As we will see in the next Chapter that the above claims are sustained by our implementation of arrays on the Connection Machine.

6.2 Non-Recursive Function Forms

6.2.1 The Time Complexities

Function forms construct new functions with given functions. Consequently, the time complexity of functions defined by function forms can be given in terms of the complexities of their constituent functions.

In the following, we use the following notations.

1. $\mathcal{T}_s[\![f x]\!]$: time used to compute function (form) f on argument x on sequential computers.

2. $\mathcal{T}_p[\![f x]\!]$: time used to compute function (form) f on argument x on (SIMD) parallel computers.

3. $\mathcal{T}[\![f x]\!]$: time used to computer function (form) f on argument x on both sequential and parallel computers.

Array Distribution

Strong local function $!f$ defined over arrays applied the function f to every entry of the array. We will focus our attention to the usual cases where the array entries have sizes of $O(1)$.

On sequential computers, the function f must be applied to each entry of an array one after another (although the order does not matter). Since the entries have bound size, each application takes O(1) time. Consequently,

$$\mathcal{T}_s[\![!f\ A]\!] = |A| \tag{6.1}$$

On parallel computers, the applications of f can be applied to all entries at the same time. We therefore have

$$\mathcal{T}_p[\![!f\ A]\!] = O(1) \tag{6.2}$$

The complexity difference between the sequential and parallel case in above reveals one of the means of exploiting the additive parallelism found in both PDC and SDC algorithms (Section 5.4). Additive parallelism is also exploited in parallel computation by the parallel execution of communication functions.

Function Composition

Function composition is a sequential operation. The time used to compute f composed with g is the sum of the time used by f and g. Or formally,

$$\mathcal{T}[\![f : g\ x]\!] = \mathcal{T}\ [\![g\ x]\!] + \mathcal{T}[\![f(g\ x)]\!] \tag{6.3}$$

Since function composition is associative, the above implies that

$$\mathcal{T}\ (f_0 : \cdots : f_{k-1}\ x) = \textstyle\sum_{i=0}^{k-1} \mathcal{T}\ f_i\ x_i \text{ where } x_i \ = \begin{cases} x, & \text{if } i = 0 \\ f_i : \cdots, f_0\ x, & \text{Otherwise} \end{cases}$$

Observe that the above holds for both sequential and parallel computations.

Function Construction

Function construction applies a tuple of functions to a tuple argument in such a way that the kth function is applied to the kth component of the argument. Generally, on both sequential computers and SIMD parallel computers, the time to compute a construction is the sum of the time used by each function application. This gives,

$$\mathcal{T}[\![(f_0, \ldots, f_{k-1})\ (x_0, \ldots, x_{k-1})]\!] = \mathcal{T}[\![f\ x_0]\!] + \cdots + \mathcal{T}[\![f_{k-1}\ x_{k-1}]\!] \tag{6.4}$$

If-Then-Else

The time used to compute a if-then-else construct clearly depends on the results of the predicate on particular data. In other words,

$$\begin{aligned} \mathcal{T}[\![p \rightarrow f; g\ x]\!] &= \mathcal{T}[\![px]\!] + \mathcal{T}[\![hx]\!] \\ &\text{where } h = \text{if } p\ x \text{ then } f \text{ else } g \end{aligned}$$

It is also obvious that the time used by an if-then-else form is in the order of the most complex one of the three constituents.

$$\mathcal{T}[\![p \rightarrow f; gx]\!] = O(\max(\mathcal{T}[\![f\ x]\!], \mathcal{T}[\![g\ x]\!], \mathcal{T}[\![p\ x]\!])$$

Filter

Unlike the if-then-else form, the relation between a filter and its constituents depends on the particular given constituents. However, an upper bound of the time used by a filter can be easily given by

$$\mathcal{T}[\![p \Rightarrow fx]\!] = O(\mathcal{T}[\![!p\ x]\!] + \mathcal{T}[\![f\ x]\!]) \tag{6.5}$$

where $(!px)$ is really meant to apply p to each index of (the array) x. On parallel computers, the predicate over the index set can be computed in parallel, and takes no more than $O(1)$ time assuming the indices have $O(1)$ sizes. Therefore,

$$\mathcal{T}_p[\![p \Rightarrow fx]\!] = O(\mathcal{T}_p[\![f\ x]\!]) \tag{6.6}$$

Parallel Distribution Over Tuples (PDT)

PDT is a special form of function construction where all the k functions in the function tuple are identical. On sequential computers, a PDT form still has to be computed one function application a time, therefore the time used equals to the sum of the time used by all the applications

$$\mathcal{T}_s[\![!f(x_0,\ldots,x_{k-1})]\!] = \textstyle\sum_{i=0}^{k-1} \mathcal{T}_s[\![f\ x_i]\!]$$

On parallel computers (both SIMD or MIMD), all the k function applications can be computed in parallel. Therefore,

$$\mathcal{T}_p[\![!f(x_0,\ldots,x_{k-1}]\!] = \max(\mathcal{T}_p[\![f\ x_0]\!],\ldots,\mathcal{T}_p[\![f\ x_{k-1}]\!])$$

Obviously, if the arguments have the same size, then the above can be further simplified to

$$\mathcal{T}_p[\![!f(x_0,\ldots,x_{k-1}]\!] = O(\mathcal{T}_p[\![f\ x]\!])$$
$$\text{where } x = x_i, \text{ for any } 0 \leq i < k$$

The difference between the sequential and parallel cases reveals the fundamental means of exploiting multiplicative parallelism found in PDC algorithms (Section 5.4). From the above we can also see why balanced divide function is crucial to PDC algorithms: when the k arguments $(x_0,\ldots,x_{k-1})$ are the results of unbalanced division, the difference between the parallel complexity and sequential complexity will virtually vanish.

Sequential Distribution Over Tuples (SDT)

Clearly, the constituents of a sequential function mapping function form must be computed in a linear order, regardless of whether the mapping is executed on sequential or parallel computers.

$$\mathcal{T}[\![!f :: (g_0, \ldots, g_{k-2})\ (x_0, \ldots, x_{k-1})]\!] = \sum_{i=0}^{k-1} \mathcal{T}[\![f\ x'_i]\!] + \sum_{i=0}^{k-2} \mathcal{T}[\![g_i\ (x_{i+1},\ y_i)]\!]$$

where

$$\begin{aligned} y_0 &= f\ x_0 \\ y_i &= f\ x'_i, && \text{for} i = 1 \text{ to } (k-1) \\ x'_i &= g(x_i, y_{i-1}), && \text{for } i = 1 \text{ to } (k-1) \end{aligned}$$

We should add that if the adjust function g_i is prefixed with the escape symbol "%", the function f will not be applied to x'_i, and the corresponding term in the summation should be deleted.

In the case that all the variables have sizes of $O(n)$, the above can be simplified to

$$\mathcal{T}[\![!f :: (g_0, \ldots, g_{k-2})\ (x_0, \ldots, x_{k-1})]\!] = k * (\mathcal{T}[\![f]\!]\ n) + \sum_{i=0}^{k-2} \mathcal{T}[\![g_i]\!]\ n$$

Again, if $b < (k-1)$ adjust functions are prefixed by the escape character "%", then the coefficient k in the above should be subtracted by b.

Unlike the PDT form, the relation between the complexities of a SFM form and its constituents remains unchanged when we move from sequential computer to parallel computer. We should be reminded, however, SFM can still be speeded up on parallel computers provided that the individual constituents can be speeded up.

Communication

For the sake of completeness, we also repeat here the result of Section 6.1 with the new notations.

Let the arguments of the communication functions be of size $O(n)$. On sequential computers,

$$\left.\begin{array}{l} \mathcal{T}_s[\![\#fx]\!] \\ \mathcal{T}_s[\![\#f(x_a, x_b)]\!] \\ \mathcal{T}_s[\![\#(f_a, f_b)(x_a, x_b)]\!] \end{array}\right\} = O(n) \qquad (6.7)$$

Intra-array and one-directional communication on parallel computers have the complexity of

$$\left.\begin{array}{l} \mathcal{T}_p[\![\#f x]\!] \\ \mathcal{T}_p[\![\#f(x_a, x_b)]\!] \end{array}\right\} = \left\{\begin{array}{ll} O(1), & \text{if } f = corr \text{ or } mirr, \\ O(\log n), & \text{otherwise} \end{array}\right. \tag{6.8}$$

The bi-directional communication, on the other hand, can be reduced to the above

$$\begin{array}{lll} \mathcal{T}_p[\![\#(f_a, f_b)(x_a, x_b)]\!] & = \mathcal{T}_p[\![f]\!]O(n), & \text{if } f_a = f_b = f \\ & = \mathcal{T}_p[\![f_a(x_a, x_b)]\!] + \mathcal{T}_p[\![f_b(x_b, x_a)]\!] & \text{otherwise} \end{array} \tag{6.9}$$

6.2.2 Parallel and Sequential Forms

We saw in the previous section that the time complexity of each functional form can be expressed in terms of the complexities of its constituents. However, some functional forms have the same dependency on both sequential computers and parallel computers, others do not.

To characterize the difference, we first define the *speedup* of a functional form to be the ratio between its sequential and parallel complexities:

$$\mathcal{R}[\![f(x_0, \ldots, x_{k-1}]\!] = \frac{\mathcal{T}_s[\![f(x_0, \ldots, x_{k-1}]\!]}{\mathcal{T}_p[\![f(x_0, \ldots, x_{k-1}]\!]}$$

For example, the speedup for array distribution is

$$\mathcal{R}[\![!f\ A]\!] = \frac{\mathcal{T}_s[\![!f\ A]\!]}{\mathcal{T}_p[\![!f\ A]\!]} = O(|A|)$$

and the speedup for function composition is

$$\mathcal{R}[\![f : g\ x]\!] = \frac{\mathcal{T}_s[\![f : gx]\!]}{\mathcal{T}_p[\![f : gx]\!]} = O(1)$$

Clearly the speedup of a functional form tells us how much it can be computed in parallel. This naturally leads to the the following definition:

Let f be a functional form with arity k, $(x_0, \ldots, x_{k-1})$ be the arguments to f, and $|x_i| = O(n)$ for $i = 0$ to $k-1$, we say f is a *parallel functional form* if and only if

$$\mathcal{R}[\![f\ (x_0, \ldots, x_{k-1})]\!] \neq O(1)$$

By this definition, array distribution, parallel distribution over tuples, and all communications are parallel functional forms. The rest of the functional forms are sequential functional forms.

6.3 Divide and Combine Operations

Recall that we pointed out the difference between the polymorphic and non-polymorphic divide operations over arrays in Chapter 3. Since polymorphic divide operations can be expressed in terms of quotient functions over the array's index sets, an array entry can decide which subarray it belongs to and its new relative index in the subarray without communicating with others. This means on parallel computers, polymorphic divide operations can be computed in parallel for all entries. In contrast, the same operations on sequential computer will take the time proportional to the array sizes.

Therefore, let A be an array, $\$\$A = O(n)$, d a polymorphic divide function, then

$$\mathcal{T}_s[\![d\ A]\!] = O(n)$$
$$\mathcal{T}_p[\![d\ A]\!] = O(1)$$

Division over higher dimensional arrays by projection (Section 3.4.3) of the form $(d\ i)$ is simply applying the division d along the mth dimension of the array, and therefore has the same complexity as the above:

$$\mathcal{T}_s[\![(d\ i)\ A]\!] = O(n)$$
$$\mathcal{T}_p[\![(d\ i)\ A]\!] = O(1)$$

Division by intersection (Section 3.4.3), however, has slightly different behavior:

$$\mathcal{T}[\![(d_0\ i_0) \times \cdots \times (d_{k-1}\ i_{k-1})A]\!] = \textstyle\sum_{i=0}^{k-1}\ \mathcal{T}[\![(d_i\ i)\ A]\!]$$

However, assuming the arity of the intersection k is $O(1)$, the above again reduces to

$$\mathcal{T}_s[\![(d_0\ i_0) \times \cdots \times (d_{k-1}\ i_{k-1})A]\!] = O(n)$$
$$\mathcal{T}_p[\![(d_0\ i_0) \times \cdots \times (d_{k-1}\ i_{k-1})A]\!] = O(1)$$

Polymorphic combine functions are defined as the left inverses of the corresponding divide functions, and can be computed with the same number of steps as the divide functions (see Chapter 7). Therefore, let $c = d^{-1}$, $A_0, \ldots, A_{k-1}$ arrays of $O(n)$ sizes.

$$\mathcal{T}_s[\![c\ (A_0, \ldots, A_{k-1}]\!] = O(n)$$
$$\mathcal{T}_p[\![c\ (A_0, \ldots, A_{k-1}]\!] = O(1)$$

We conclude that polymorphic divide and combine operation are highly parallel operations because they both have $O(n)$ speedup.

6.4 Recursive Function Forms

Like other function forms, the time complexity of PDC and SDC forms are also decided by their constituent functions. Unlike other function forms, PDC and SDC are recursive. The complexities of PDC and SDC therefore are related to those of the constituents by recurrences. The solutions to the recurrence equations, namely their least fixed points, give us the complexities of these recursive forms.

We will restrict our discussion to DC algorithms with polymorphic divide and combine functions, and antipolymorphic adjust functions. We therefore can assume that divide and combine functions can be compute in at most $O(1)$ parallel and $O(n)$ sequential time; and the adjust functions will never alter the sizes of their arguments.

To simplify the discussion, we also assume the base predicate can be computed in $O(1)$ time, and the base spaces have $O(1)$ sizes, hence, the base function can always be computed in $O(1)$ time.

In the following discussion, we use n to denote the size of argument and intermediate variable, k the arity of the divide functions, m the division factor of balanced division, b the subtrahend of the unbalanced division (see Section 2.2.1).

6.4.1 PDC

Let us consider a function f defined by a PDC

$$f = PDC(d,\ c,\ g,\ h,\ p,\ f_b)$$

By unfolding the definition of *PDC*, we have

$$f\ x = (p \rightarrow f_b; c : h : !f : g : d)\ x$$

Balanced PDC

By applying the complexity function $\mathcal{T}_p$ to both the left-hand side and the right-hand side, we get the following recurrence for balanced PDC's sequential and parallel time complexity:

$$\mathcal{T}_p[\![f\ x]\!] = \begin{cases} O(1), & \text{if } p\ x \\ \mathcal{T}_p[\![f]\!]n/m + \mathcal{T}_p[\![d]\!]n + \mathcal{T}_p[\![c]\!]n + \mathcal{T}_p[\![g]\!]n + \mathcal{T}_p[\![h]\!]n \end{cases}$$

Similarly, we can get PDC's sequential time complexity by applying $\mathcal{T}_s$ to both side of its definition:

$$\mathcal{T}_s[\![f\ x]\!] = \begin{cases} O(1), & \text{if } p\ x \\ k * \mathcal{T}_s[\![f]\!]\ n/m + \mathcal{T}_s[\![d]\!]\ n + \mathcal{T}_s[\![c]\!]\ n + \mathcal{T}_s[\![g]\!]\ n + \mathcal{T}_s[\![h]\!]\ n \end{cases}$$

Observe that one of the adjust functions in a non-puremorphism PDC takes at least $O(1)$ parallel and $O(n)$ sequential time, which are respectively the parallel and sequential time taken at most by polymorphic divide/combine operations. Therefore, for non-puremorphism PDC algorithms, we can derive from the above

$$\mathcal{T}_p[\![f]\!]x = \begin{cases} O(1), & \text{if } p\ x \\ \mathcal{T}_p[\![f]\!]n/k + \mathcal{T}_p[\![g]\!]n + \mathcal{T}_p[\![h]\!]n \end{cases} \quad (6.10)$$

$$\mathcal{T}_s[\![f]\!]n = \begin{cases} O(1), & \text{if } p\ x \\ m * \mathcal{T}_s[\![f]\!]n/k + \mathcal{T}_s[\![g]\!]n + \mathcal{T}_s[\![h]\!]n \end{cases} \quad (6.11)$$

It is easy to show that the solution of the above recurrence for parallel time can be approximated by

$$\mathcal{T}_p[\![f]\!]n = O(\log n * (\mathcal{T}_p[\![g]\!]n + \mathcal{T}_p[\![h]\!]n)) \tag{6.12}$$

Obviously, the approximation of the parallel time complexity of postmorphism or premorphism PDC algorithms can be further simplified to

$$\mathcal{T}_p[\![f]\!]n = \begin{cases} O(\log n * \mathcal{T}_p[\![g]\!]n), & \text{if } f \text{ is a premorphism} \\ O(\log n * \mathcal{T}_p[\![h]\!]n), & \text{if } f \text{ is a postmorphism} \end{cases} \tag{6.13}$$

The approximation for the sequential case depends on the expanding factor of the division $\beta = k/m$. However, when the division is static, namely $\beta = 1$ (see Section 2.2.1), we have (for a general pseudomorphism)

$$\mathcal{T}_p[\![f]\!]x = O(\log n * (\mathcal{T}_p[\![g]\!]n) + \mathcal{T}_p[\![h]\!]n)$$

Unbalanced PDC

Following a similar procedure as the above, we can obtain the recurrences of unbalanced PDC's parallel and sequential time complexity, which are respectively

$$\mathcal{T}_p[\![f]\!]n = \begin{cases} O(1), & \text{if } p\ x \\ \mathcal{T}_p[\![f]\!](n-b) + \mathcal{T}_p[\![g]\!]n + \mathcal{T}_p[\![h]\!]n \end{cases}$$
$$\mathcal{T}_s[\![f]\!]n = \begin{cases} O(1), & \text{if } p\ x \\ \mathcal{T}_s[\![f]\!](n-b) + \mathcal{T}_s[\![g]\!]n + \mathcal{T}_s[\![h]\!]n \end{cases}$$

The approximations to the above are respectively

$$\mathcal{T}_p[\![f]\!]n = O(n * (\mathcal{T}_p[\![g]\!]n + \mathcal{T}_p[\![h]\!]))$$
$$\mathcal{T}_s[\![f]\!]n = O(n * (\mathcal{T}_s[\![g]\!]n + \mathcal{T}_s[\![h]\!]))$$

Observe that unlike the balanced case, the factors on the right-hand sides of the above have become equal (to n). It means unbalanced PDC cannot effectively exploit the multiplicative parallelism even if computed on parallel computers. However, it does not mean unbalanced PDC cannot have speedup on parallel computers at all. Just like SDC algorithms, the additive parallelism can still be exploited by computing the adjust functions (and the divide/combine functions) in parallel.

6.4.2 SDC

Let us consider a function f defined by a SDC

$$f = SDC(d,\ c,\ \vec{\mu},\ p,\ f_b)$$

where $\mu = (\mu_0, \ldots, \mu_{k-2})$

By unfolding the definition of *PDC*, we have

$$f\ x = (p \rightarrow f_b; c : h : f :: \vec{\mu} : g : d)\ x$$

Again, in the following discussion,we denote the arity of the divide functions by k, the division factor of balanced division by m, the subtrahend of the unbalanced division by p.

Balanced SDC

By applying the complexity functions $\mathcal{T}_p$ and $\mathcal{T}_s$ respectively to both side of the definition of SDC, we can have

$$\mathcal{T}_p[\![f\ x]\!] = \begin{cases} O(1), & \text{if } p\ x \\ m * \mathcal{T}_p[\![f]\!]n/k + \mathcal{T}_p[\![\mu_0]\!]n + \cdots + \mathcal{T}_p[\![\mu_{k-2}]\!]n \end{cases}$$
$$\mathcal{T}_s[\![f\ x]\!] = \begin{cases} O(1), & \text{if } p\ x \\ m * \mathcal{T}_s[\![f]\!]n/k + \mathcal{T}_s[\![\mu_0]\!]n + \cdots + \mathcal{T}_s[\![\mu_{k-2}]\!]n \end{cases}$$

We should add that if the adjust function μ_i is prefixed with the escape symbol "%", the ith term $\mathcal{T}[\![\mu_i]\!]$ should then be deleted, the factor m should also be subtracted by one for each escaped preadjust function. In the following discussion, this adjustment for escaped adjust functions will be assumed and won't be repeated each time.

Observe that in the above, the parallel and sequential complexities have the same form. This is the reflection of the fact that SDC algorithms have no multiplicative parallelism.

In the case of static division, therefore $k = m$, the solution of the above recurrences can both be approximated by

$$\mathcal{T}[\![f\ x]\!] = O(n * (\mathcal{T}[\![\mu_0]\!]n + \cdots + \mathcal{T}[\![\mu_{k-2}]\!]n)) \tag{6.14}$$

Like approximations we have been using, the bound given in above may not be tight. An example is the parallel balance SDC algorithms for lower triangular linear system, which by solving the recurrence we will have $\mathcal{T}_p[\![f]\!]n = O(n)$ whereas by approximation we get $\mathcal{T}_p[\![f]\!]n = O(n * \log n)$ (see Section 6.5).

Unbalanced SDC

We will directly give the recurrence for unbalanced SDC given in the form shared by both the parallel case and sequential cases

$$\mathcal{T}[\![f\ x]\!] = \begin{cases} O(1), & \text{if } p\ x \\ \mathcal{T}_p[\![f]\!](n-p) + \mathcal{T}_p[\![\mu_0]\!]n + \cdots + \mathcal{T}_p[\![\mu_{k-2}]\!]n & \end{cases}$$

and the approximations

$$\mathcal{T}[\![f\ x]\!] = O(n * (\mathcal{T}[\![\mu_0]\!]n + \cdots + \mathcal{T}[\![\mu_{k-2}]\!]n))$$

6.5 Case Studies

The studies made in previous sections provided us a basis to make time complexity analysis for DC algorithms in a pretty much mechanical manner. In this section, we will demonstrate the procedure with some concrete DC examples.

6.5.1 Scan

We have so far encountered altogether three DC algorithms for *scan*: Algorithm 4.1 is a PDC algorithm using broadcast communication; Algorithm 4.5 is a variation of Algorithm 4.1 in which the broadcast communication is eliminated; Algorithm 5.1 is a SDC algorithm. In this section, we will first analyze the parallel complexity of each of them, and then make a brief comparison with their sequential complexities.

PDC with Broadcast

Recall that Algorithm 4.1 for scan was

$$\begin{aligned} &scan\ \oplus = PDC(d_{lr},\ c_{lr},\ id,\ (h_{scan}\ \oplus),\ atom?,\ id)\\ &\quad \text{where} h_{scan}\ \oplus = (id,\ !\oplus) : \#(nil,\ last\ 0)\end{aligned}$$

To make the complexity analysis, we can apply equation 6.10 to the above and have

$$\mathcal{T}_p[\![scan]\!]n = \begin{cases} O(1), & \text{if } n \le C \\ \mathcal{T}_p[\![f]\!]n/2 + \mathcal{T}_p[\![h]\!]n, & \text{otherwise} \end{cases} \tag{6.15}$$

We will first study the term $\mathcal{T}_p[\![h]\!]$. Since it is a function composition, we apply Equation 6.3

$$\mathcal{T}_p[\![(id,\ !\oplus) : \#(nil,\ last\ 0)]\!]n = \mathcal{T}_p[\![(id, !\oplus)]\!]n + \mathcal{T}_p\#(nil,\ last\ 0)n$$

The first term in the above is function construction of array distributions, we therefore can successively apply Equations 6.4 and 6.2 to get

$$\begin{aligned} \mathcal{T}_p[\![(id, !\oplus)]\!]n &= \mathcal{T}_p[\![id]\!]n + \mathcal{T}_p[\![!\oplus]\!]n \\ &= O(1) + O(1) \\ &= O(1) \end{aligned}$$

The second term is a broadcast communication, and by Equation 6.8,

$$\mathcal{T}_p\#(nil,\ last\ 0)n \quad = \quad O(\log n)$$

By substituting these results into Equation 6.15, we have

$$\mathcal{T}_p[\![scan]\!]n = \begin{cases} O(1), & \text{if } n \leq C \\ \mathcal{T}_p[\![f]\!]n/2 + O(\log n), & \text{otherwise} \end{cases}$$

Finally, by applying Equation 6.13 to the above, we conclude

$$\mathcal{T}_p[\![scan]\!]n = O(\log^2 n) \tag{6.16}$$

PDC Algorithm without Broadcast

Recall that (Section 4.3.2) Algorithm 4.5 was

$$\begin{aligned} &scan \oplus = PDC(d_{lr},\ c_{lr},\ id,\ (h_{scan} \oplus),\ atom?,\ id) \\ &\quad \text{where } f_b\ x = (x, x) \\ &\qquad h_{scan} \oplus = (!loc1 \oplus,\ !loc2 \oplus) : \#!corr \\ &\qquad loc1\ ((x1, sum1),\ (x2, sum2) = (x1,\ sum1 \oplus sum2) \\ &\qquad loc2\ ((x1, sum1),\ (x2, sum2) = (x1 \oplus x2,\ sum1 \oplus sum2) \end{aligned}$$

This time, we will first concentrate on the complexity of its postadjust function $(h_{scan} \oplus)$.

$$\begin{aligned} & \mathcal{T}_p[\![(!loc1 \oplus, !loc2 \oplus) : \#!corr]\!] \\ = \; & \mathcal{T}_p[\![(!loc1 \oplus, !loc2 \oplus)]\!](n, n) + \mathcal{T}_p[\![\#!corr(n, n)]\!] \quad \text{(Eq. 6.3)} \\ = \; & \mathcal{T}_p[\![!loc1 \oplus]\!]n + \mathcal{T}_p[\![!loc2 \oplus]\!]n + \mathcal{T}_p[\![\#!corr]\!](n, n) \quad \text{(Eq. 6.4)} \\ = \; & O(1) + O(1) + \mathcal{T}_p[\![\#!corr]\!](n, n) \quad \text{(Eq. 6.2)} \\ = \; & O(1) + O(1) + \mathcal{T}_p[\![\#corr]\!](n, n) + \mathcal{T}_p[\![\#corr]\!](n, n) \quad \text{(Eq. 6.9)} \\ = \; & O(1) + O(1) + O(1) + O(1) \quad \text{(Eq. 6.8)} \\ = \; & O(1) \end{aligned}$$

By applying Equation 6.13, we have

$$\mathcal{T}_p[\![scan]\!]n = O(\log n * O(1)) = O(\log n)$$

Hence, Algorithm 4.5 is asymptotically faster by a logarithmic factor than Algorithm 4.1 due to the elimination of broadcast communication.

The SDC Algorithm

The SDC Algorithm 5.1 in Section 5.2.2 is

$$scan \oplus = SDC(d_{lr},\ c_{lr}, (\mu_{scan} \oplus),\ atom?,\ id)$$

$$\text{where}\mu_{scan} \oplus = (eq?\ 0) \Rightarrow (!\oplus : \#(last\ 0))$$

Again let us first determine the complexity of the adjust function $(\mu_{scan}\oplus)$

$$\begin{aligned}
& \mathcal{T}_p[\![(eq?\ 0) \Rightarrow (!\oplus : \#(last\ 0))]\!] \\
= \ & \mathcal{T}_p[\![!\oplus : \#(last\ 0)]\!] \quad \text{(Eq. 6.6)} \\
= \ & \mathcal{T}_p[\![!\oplus]\!]n + \mathcal{T}_p[\![\#(last\ 0)]\!](n,n) \quad \text{(Eq. 6.3)} \\
= \ & O(1) + \mathcal{T}_p[\![\#!last\ 0]\!](n,n) \quad \text{(Eq. 6.2)} \\
= \ & O(1) + O(\log n) \quad \text{(Eq. 6.8)} \\
= \ & O(\log n)
\end{aligned}$$

By substituting the above into Equation 6.14, we have

$$\mathcal{T}_p[\![scan]\!]n = O(n * O(\log n)) = O(n * \log n)\ .$$

The above shows that Algorithm 5.1 has very poor parallel time performance in comparison with the others.[1]

[1] We would like to pointed out that the above actually is an over-estimation of the complexity as a result of applying the approximation equation 6.6. Since there is always only one vector entry that satisfies the predicate of the filter, the (*last* 0) communication reduces to a partial permutation communication. The communication function therefore in fact takes $O(1)$ as opposed to $O(\log n)$ time. The algorithm thus can be computed in $O(n)$ time

Sequential Complexities

The sequential time complexities of the three *scan* DC algorithms can be obtained by applying the sequential complexity function $\mathcal{T}_s$ to their definitions with the procedures similar to that of the above. We will omit the detail and simply give the results.

Sequential Time of Scan DC Algorithms:

$$\begin{aligned} \text{Algorithm 4.1 (with broadcast)} &\Rightarrow O(n * \log n) \\ \text{Algorithm 4.5 (without broadcast)} &\Rightarrow O(n * \log n) \\ \text{Algorithm 5.1} &\Rightarrow O(n) \end{aligned}$$

Although Algorithm 4.1 and Algorithm 4.5 PDC algorithms have the same asymptotic complexity, the former is obviously faster on sequential computers due to the less complicated adjust function. We therefore have the following order of the three algorithms by their absolute sequential time

$$\mathcal{T}_s[\![\text{Algorithm 5.1}]\!] < \mathcal{T}_s[\![\text{Algorithm 4.1}]\!] < \mathcal{T}_s[\![\text{Algorithm 4.5}]\!]$$

Interestingly, the order in the parallel world is exactly reversed as we have shown in the previous subsection. Moreover, the order is not only true in terms of absolute time but also true by their asymptotic behaviors. In other words, if we use $\ll$ to denote the *little oh* relation [44] over functions, we can write

$$\mathcal{T}_p[\![\text{Algorithm 4.5}]\!] \ll \mathcal{T}_p[\![\text{Algorithm 4.1}]\!]\mathcal{T}_p[\![\text{Algorithm 4.1}]\!] \ll \mathcal{T}_p[\![\text{Algorithm 4.5}]\!]$$

The reversal of the relation on sequential complexities and parallel complexities can often be observed in other DC algorithms (e.g. polynomial evaluation). It indicates to us the fundamental difference between the sequential world and parallel world.

6.5.2 Monotonic Sort

In this section, we will study the parallel time complexity of the second-order PDC algorithm for monotonic sort given in Section 4.3.5 (Algorithm 4.12).

$$sort = PDC(d_{lr},\ c_{lr},\ id,\ loc : \#!mirr,\ atom?,\ id)$$
$$\text{where} loc = !nest\ :\ (!\min, !\max)$$
$$nest = PDC(d_{lr},\ c_{lr},\ (!\min, !\max) :\ \#!corr,\ id,\ atom?,\ id)$$

We first look into the preadjust function of the nested PDC

$$(!\min, !\max) :\ \#!corr$$

The communication function takes $O(1)$ time since it is *correspondent* communication. The function construction takes also $O(1)$ time since both its constituents are array distributions. Therefore, the total time of the adjust function is $O(1)$, the total time of this nested PDC is $O(\log n)$.

We next look into the postadjust function of the top-level PDC

$$loc : \#!mirr = !nest\ :\ (!\min, !\max) : \#mirr$$

We have learned the complexity of the left-most constituent of the above function composition, which is $O(\log n)$; the other two constituents also have $O(1)$ time complexity since they are respectively permutation communication and array distribution. The postadjust function itself therefore has complexity of $O(\log n)$.

By equation 6.13, the parallel time complexity of Algorithms 4.12 is $O(\log^2 n)$

6.5.3 Linear Lower Triangular System

In Chapter 5, we presented two SDC algorithms for the problem of linear lower triangular systems: Algorithm 5.3 using balanced division and Algorithms 5.4 using unbalanced division. To make a point on parallelism in SDC algorithms, we analyzed the parallel time complexities of the two algorithms, which were respectively $O(n)$ and $O(n * \log n)$, with a method less formal than what developed in this chapter. By applying the parallel complexity functions recursively over the definitions of the algorithms, these results from Chapter 5 can be easily confirmed. Similarly, by applying the sequential time complexity functions recursively, we can also show that the two algorithms take respectively $O(n^2 * \log n)$ and $O(n^2)$ sequential time. Observe that the performance reversal phenomenon pointed out in the end of Section 6.5.1 repeated here: the better sequential algorithm has worse parallel performance; conversely, the faster parallel algorithm is slower on sequential computers.

6.6 Processor Complexity

6.6.1 Basic Notions

Besides time, an algorithm executed on parallel computers also consumes another precious resource – processors. To measure the consumption of processors, *processor complexity* (PC) is defined to be the number of processors a parallel algorithm uses, which is a function of the size of the problem. When a parallel algorithm is executed on a parallel computer, it may use different number of processors at different time steps. Therefore, processor complexity of a given problem of a given size is actually a function over time steps. The *maximum processor complexity* (MPC) is defined to be the maximum value of this function.

However, the number of processors taken by a parallel algorithm can vary greatly if no other constraint is given. Indeed, every parallel algorithm can be computed by one processor if we choose. We therefore need to first establish a norm to talk about processor complexities. A natural choice for us is to count the number of processors with the constraint that one array entry is mapped to one processor at all times. This is what we call *one-one norm* of processor complexity.

As a matter of fact, the one-one norm was implicit in the discussion of previous sections. In Section 6.2 array distribution and permutation communication were said to have $O(1)$ parallel time complexity, which is only true if we use the number of processors equal to the processor complexity under the one-one norm (or within a constant factor).

With the notion of one-one norm, processor complexity can be studied by calculating the total size of the active subarrays, namely, subarrays to which the constituent functions of a DC algorithm are being applied.

6.6.2 Dynamic Division

Processor complexity of PDC algorithms is most obviously influenced by the dynamic nature of their divide functions. Each time a divide function is applied, the total size of the subarrays may increase by the expanding factor of the division. The processor complexity of a PDC algorithm that employs a dynamic division therefore should be thought of as a non-decreasing function over the division steps during the stage of division.

The following proposition gives a bound on processor complexity of PDC algorithm with dynamic divisions by giving the bound of the total size of the subarrays at the end of division stage.

Proposition 6.2 *Let d be a k-ary division of division factor m, then for an input array of size N, the total size of the subarrays at the leaf level of the division tree is* $O(n^{\log_m k})$.

Proof

It is clear that at stage i the total size of the subarrays is

$$P' = N * \beta^i$$

By Proposition 2.3, the height of the division tree is $O(\log_m N)$. The number of division steps is therefore also $O(\log_m N)$. Let P be the total size of the subarrays in the end of division stage, we then have

$$\begin{aligned} P &= O(N * \beta^{\log_m N}) \\ &= O(N * (k^{\log_m N}/m^{\log_m N})) \\ &= O(k^{\log_m N}) \\ &= O(k^{\log_k N/\log_k m}) \\ &\doteq O(N^{\log_m k}) \end{aligned}$$

Q.E.D.

Static divisions in fact can be considered as special case where the expanding factor $\beta = 1$. Proposition 6.2 therefore can be applied to static divisions as well. By setting $m = k$, the proposition confirms our intuition that the total number of processors used by a PDC algorithm employing static division under one-one norm equals to the size of the original input array N. Most of the PDC algorithms we have studied fall into this category.

In Section 4.3.4, we studied three PDC algorithms for matrix multiplication. The first one (Algorithm 4.9) uses a 8-ary division with division factor $m = 4$. By Proposition 6.2, the processor complexity at the end of dividing stage is (note the input size is $O(N^2)$)

$$P = O((N^2)^{\log_4 8}) = O(N^3)$$

The second algorithm uses a 4-ary division (Algorithm 4.10) with division factor $m = 2$. Proposition 6.2 thus gives the processor complexity of

$$P = O((N^2)^{\log_2 4} = O(N^4)$$

The above is a correct upper bound but an overestimation of the processor complexity. By studying the conditions of Proposition 6.2, we can realize that it assumes that the leaf array spaces in the division tree are of sizes $O(1)$. This condition is definitely being violated by the division of Algorithm 4.10, since the division process stops when the size of subarray has size $O(1)$. In fact, Algorithm 4.10 has exactly the same processor complexity as Algorithm 4.9 because it has the same expanding factor and goes through the same number of division steps.

The third algorithm with binary division (Algorithm 4.11) is static, and therefore has processor complexity of $O(N^2)$ during the division stage.

6.6.3 Base Functions

In many DC algorithms, the base spaces have sizes of $O(1)$, and in that case, the base functions generally do not alter the order of the total size of subarrays, and therefore have no impact on the processor complexity of DC algorithms.

However, when the base spaces have non-constant bounded sizes, the base functions may have a major effect on the total size of the subarrays. Unlike the divide function which always have non-decreasing effect on processor complexity, base functions can both increase or decrease the order of the total size of subarrays.

The three matrix multiplication PDC algorithms 4.3.4 again are good examples in this matter. We can see that the size of the base spaces in the three algorithms are respectively $O(1)$, $O(n)$, and $O(n)$, which are mapped by the base functions respectively to $O(1)$, $O(1)$ and $O(n^2)$. We summarize this by

$$\begin{array}{llcl}
\text{Algorithm 4.9} & O(1) & \xrightarrow{f_b} & O(1) \\
\text{Algorithm 4.10} & O(n) & \xrightarrow{f_b} & O(1) \\
\text{Algorithm 4.11} & O(n) & \xrightarrow{f_b} & O(n^2)
\end{array}$$

The three algorithms thus respectively maintain, decrease, and increase the total size of subarrays.

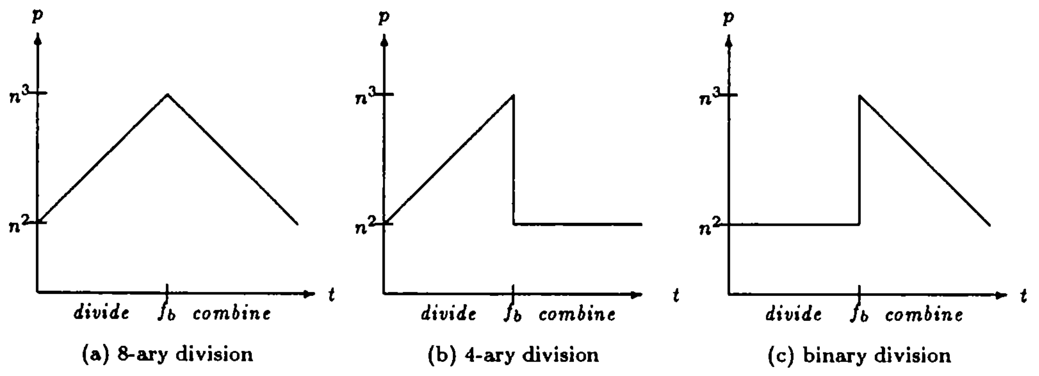


Figure 6.2: Processor complexity of the matrix multiplication PDC algorithms

6.6.4 DC Algorithms

Just as divide functions never decrease the processor complexity during the division phase of a PDC algorithm, combine function never increase it during the combine phase. The adjust functions, since they are antipolymorphic (see Section 3.6), have no effect on the order of the total size of subarrays. We thus can conclude that the maximum processor complexity of a PDC algorithm is entirely decided by the behavior of divide functions and base functions. The processor complexity analysis of PDC algorithm thus is the fairly simple matter of studying the dynamic natures of these two constituents, which we have studied in above.

It is interesting to portray the processor complexity as the function of time steps for PDCs with dynamic nature. In Figure 6.2, we did so for the three matrix multiplication algorithms.

SDC algorithms have the characteristic that only one constituent function can be computed at any given time. The total sizes of the subarrays which is the argument of the computed constituent therefore should be the measurement of the processor complexity at the moment. The processor complexity of SDC algorithms as a function of computational steps therefore has quite complex curves. However, the maximum processor complexity is easy to decide and equal to the total size of the largest active subarrays, which occur in the beginning and ending steps of SDC algorithms. By this measurement, all the SDC algorithms in Chapter 5 can be shown to have linear processor complexity.

The above analysis however could have overestimated the processor complexities of SDC algorithms with unbalanced division. For example, by computing one column at a time, Algorithm 5.4 obviously can use $O(n)$ processors rather than $O(n^2)$ processors without loss of time performance despite that the first division is done over arrays of $O(n^2)$ sizes. Essentially, optimization as the above is achieved by delaying the divide operation on certain array entries. For polymorphic divide and combine operations the delayed evaluation can be easily arranged. We therefore can generally measure the processor complexity of SDC algorithms by the total size of the subarrays computed by the crossadjust functions at a given time.

6.7 Epilogue

In Figure 6.3, we give the time and processor complexities of all the DC algorithms in this dissertation.

The *PT product* in Figure 6.3 is simply the product of the time and processor complexities of a given DC algorithm. The PT product is often regarded as a measurement of the total resources consumed by a parallel algorithm. Clearly, the practical lower bound of the PT product equals to the time used by fastest known sequential algorithms. The ratio

$$E = PT\ Product/T, \text{ where T : time of the fastest known sequential algorithm}$$

is called the *efficiency* of the parallel algorithm, which in the ideal case is $O(1)$.

Problem	Algorithm	Type	Time	Processor	PT Product
scan	4.1	PDC	$O(\log^2 n)$	$O(n)$	$O(n * \log^2 n)$
scan	4.5	PDC	$O(n \log n)$	$O(n)$	$O(n * \log n)$
scan	5.1	SDC	$O(n)$	$O(1)$	$O(n)$
broadcast	4.3	PDC	$O(\log n)$	$O(n)$	$O(n * \log n)$
broadcast	4.3	PDC	$O(\log n)$	$O(n)$	$O(n * \log n)$
polynomial	4.6	PDC	$O(\log n)$	$O(n)$	$(n * \log n)$
polynomial	4.7	PDC	$O(\log^2 n)$	$O(n)$	$(n * \log^2 n)$
polynomial	4.8	PDC	$O(\log n)$	$O(n)$	$(n * \log n)$
matr. mult.	4.9	PDC	$O(\log n)$	$O(n^3)$	$(n^3 * \log n)$
matr. mult.	4.10	PDC	$O(\log n)$	$O(n^3)$	$(n^3 * \log n)$
matr. mult.	4.11	PDC	$O(\log n)$	$O(n^3)$	$(n^3 * \log n)$
mono. sort	4.12	PDC	$O(\log^2 n)$	$O(n)$	$(n * \log^2 n)$
Gau. elim.	5.2	SDC	$O(n * \log n)$	$O(n)$	$(n^2 * \log^2 n)$
tri. sys.	5.4	SDC	$O(n * \log n)$	$O(n)$	$(n^2 * \log^2 n)$
tri. sys.	5.3	SDC	$O(n)$	$O(n^2)$	(n^3)

Figure 6.3: Time and processor complexities of DC algorithms

More often than not, parallel algorithms cannot attain the optimal $O(1)$ efficiency. A parallel algorithm is said to be *sub-optimal* if its efficiency is $O(1/(\log^c n))$ where c is some small constant.

With the above notions, the following claims can be verified about the DC algorithms listed in Figure 6.3

- All the PDC algorithms are sub-optimal.
- All the SDC algorithms with the exception of Algorithm 5.3 are sub-optimal.

Another possible measurement of the total resource consumed by a parallel algorithm is the integration of processors over time, which is called *PT integration.* It can be shown [34] that the SDC algorithm 5.3 is sub-optimal if the efficiency is calculated with the PT integration.

Chapter 7

Implementation on the CM

The recursive programming style has long been exploited in sequential computing because it generally yields high-level and efficient programs. In parallel computing, recursion is still more desirable due to the additional complexity of programming parallel computers.

Recursion, however, calls for data structures that can be recursively divided and combined, which we will refer to as *recursive data structures.* The list structure in Lisp and other functional programming languages, for example, is recursive; without the recursive nature of lists and similar data structures, recursion would not play such an important role in these programming languages [49, 41, 18].

But the development of recursive parallel data structures has not received the attention it deserves. For example, parallel data structures in all programming languages on the CM are non-recursive, or, we might say, *flat.* In particular, the data structure *parallel variable* (PVAR) in *Lisp [53] is flat since it can be neither divided nor combined. Several drawbacks result from the flatness of parallel data structures. First, recursion cannot be expressed directly in a parallel context. Second, multiplicative parallelism, which is exploited by applying functions over multiple sub-structures simultaneously (Chapters 5 and 6), is conceptually excluded because the very notion of "sub-flat-structures" is a contradiction in terms. Finally, communications over the structures must be specified by global addresses at all times.

To see why communication by global address may cause extra burdens, let us recall that the communications discussed in Chapter 3 and used in later chapters can all be expressed essentially by identity and constant functions. That, however, can no longer be the case if the relative indexing scheme is removed. The same communications would have to be specified by much more complex communication generators under a global indexing scheme.

With the divide and combine operations, arrays in Divacon notation are clearly recursive. Indeed, the conciseness and readability of our DC algorithms are, to a great extent, the result of the availability of recursive arrays. It should not then be in question that we do have need of high level interface provided by *recursive arrays* (RA). The question is how, and how efficiently, can we implement recursive arrays on parallel computers where only flat arrays are provided.

This chapter is intended to offer some preliminary evidence that recursive arrays can in fact be efficiently implemented on parallel computers. Since DC algorithms in Divacon notation are defined in terms of operations over recursive arrays, this means that DC algorithms in Divacon can be executed on parallel computers. We will discuss the representation of recursive arrays by flat arrays, present implementation algorithms for RA operations, give an introduction to a version of Divacon running on the CM, make an analysis of the operation complexities, and finally give benchmarks of the RA operations and benchmarks of some applications DC algorithms.

7.1 Preliminaries

7.1.1 Bit Operations

Our implementation algorithms make intensive use of bit operations over integers. These operations can all be expressed by sequences of logical operations over binary representations of integers such as bit-wise AND, bit-wise LOGXOR, and rotation of the binary numbers [49]. But the code would be quite difficult to understand if we presented the implementation with those low level operations. We therefore prefer to write the code using high-level bit operations such that the thread of thinking behind the design can be seen more easily. This section describes the high level bit operations we will use later.

First let us define a coding function Ψ similar to the the coding function Φ defined in Section 3.4.2. It takes two integers w and b as arguments, and returns a tuple of w binary bits corresponding to the binary representation of b. We also define the function Ψ^{-1}, which is the inverse of Ψ.

$$\Psi\ w\ b = (b_0, \ldots, b_{k-1})$$
$$\Psi^{-1}\ (b_0, \ldots, b_{k-1}) = b$$

In the following, we will be using the select operations over the binary tuples (Section 3.1) produced by the above coding function. These select operations can be understood easier if we introduce the following mask functions first. Let i and m be non-negative integers. We define

$$mask\_l\ i\ w = (\underbrace{0, \ldots, 0}_{w-i}, \underbrace{1, \ldots, 1}_{i})$$
$$mask\_r\ i\ m = (\underbrace{0, \ldots, 0}_{i}, \underbrace{1, \ldots, 1}_{w-i})$$

The two *selection operation over integers* we will be using are *select-right* and *select-left*, defined respectively by

$$\begin{aligned} sel\_r\ i\ w\ b &= \Psi^{-1} : sel\ (mask\_l\ i\ w) : \Psi\ b \\ sel\_l\ i\ w\ b &= \Psi^{-1} : sel\ (mask\_r\ i\ w) : \Psi\ b \end{aligned}$$

where i, m, and b are non-negative integers. For example,

$$\begin{aligned} sel\_r\ 2\ 4\ 15 &= 3 \\ sel\_l\ 3\ 4\ 15 &= 7 \end{aligned}$$

Given two tuples, $\vec{a} = (a_0, \ldots, a_m)$ and $\vec{b} = (b_0, \ldots, a_m)$, their *catenation* is

$$cat(\vec{a}, \vec{b}) = (a_0, \ldots, a_m, b_0, \ldots, a_m)$$

Since the operation is associative, it can be generalized to take more than two arguments.

We can now define the *integer catenation* operation. Let a and b be two integers. We define

$$icat((w_a, a), (w_b, b)) = \mathcal{D}^{-1} : cat(\mathcal{D}w_a b, \mathcal{D}w_b b)$$

Like the catenation of tuples, integer catenation can be applied to more than two arguments. As an example,

$$\begin{aligned} & icat\ ((2, 1),\ (3, 7)) \\ = \ & \mathcal{D}^{-1} : cat\ ((0, 1),\ (1, 1, 1)) \\ = \ & \mathcal{D}^{-1}(0, 1, 1, 1, 1) \\ = \ & 15 \end{aligned}$$

7.1.2 Flat Vectors

We implement recursive arrays with flat vectors. This means we will represent a RA in terms of some flat vectors, and we will define RA operations in terms of operations on flat vectors. It is therefore helpful to first specify precisely what a flat vector is and what its operations are.

A flat vector can be viewed as a normalized vector (but one never to be divided). A flat vector of size n can be created by

$$mkv\ n\ f = V, \text{ where } V\ i = f\ i, \text{ for } i = 0 \text{ to } n$$

where f is the *initialization function* of V.

Array distribution works for flat arrays the same way as for recursive arrays. Let V be a flat array. Then its distribution operation is

$$!!f\ V = V', \text{ where } V'\ i = f : V\ i$$

Flat arrays are only subject to intra-array communication with absolute indexing. Let $V_{source}, V_{partner}, andV_{destination}$ be three flat arrays. We define [1]

$$comm\ V_{source}\ V_{partner} = V_{destination} \text{ where } V_{destination}\ i = V_{source}\ V_{partner}\ i$$

Thus, the communicated values are from V_{source}, the indices of the entries that send the values are given by $V_{partner}$, the values communicated will be held in $V_{destination}$.

7.1.3 Representation of Recursive Arrays

We represent a recursive vector of size n by the Z-structure (Section 3.6.5) of the following seven flat vectors

A : absolute indices, constant integers, $A\ i = i$

R : relative indices, variable integers, $R\ i \leq i$, initially $R\ i = i$

W : width of binary number of indices, constant integers, $W\ i = \log n = m$

[1] People who are familiar with *Lisp on the Connection Machine will immediately recognize that this communication operation *comm* corresponds to the *pref* operation on *parallel variables*.

J : level of division, variable integers, $J\ i \leq m$, initially $J\ i = 0$

S : side of array entries, $S = 0$ means left, $S = 1$ right, variable, initially undefined

U : indexed value of the recursive vector entries

C : used to hold values obtain from communication

A RV is created by the function *mkrv* with arguments n and f, where n is the size, f is the initialization function

$$mkrv\ n\ f = zip(A,\ R,\ J,\ W,\ S,\ U)$$

where

$$A = mkv\ n\ id$$

$$R = A$$

$$J = mkv\ n\ \lambda.x\ 0$$

$$W = mkv\ n\ \lambda.x\ \log n$$

$$S = mkv\ n\ \lambda.x\ nil$$

$$U = mkv\ n\ f \qquad C = mkv\ n\ \lambda.x\ nil$$

An entry of the Z-structure V thus is a tuple of the form

$$V\ i = (a,\ r,\ w,\ j,\ s,\ u),\ \text{for } i = 0 \text{ to } (n-1)$$

where a, r, w, j, s, and u are respectively the (indexed value of the) ith entry of A, R, W, J, S, and U.

For example, let $V = mkrv\ 8\ (\lambda.x\ 3 * x)$, then

$$\begin{array}{ll}
V.A = & [\ 0\ \ 1\ \ 2\ \ 3\ \ 4\ \ 5\ \ 6\ \ 7\ ] \\
V.R = & [\ 0\ \ 1\ \ 2\ \ 3\ \ 4\ \ 5\ \ 6\ \ 7\ ] \\
V.J = & [\ 0\ \ 0\ \ 0\ \ 0\ \ 0\ \ 0\ \ 0\ \ 0\ ] \\
V.W = & [\ 3\ \ 3\ \ 3\ \ 3\ \ 3\ \ 3\ \ 3\ \ 3\ ] \\
V.S = & [\ nil\ \ nil\ \ nil\ \ nil\ \ nil\ \ nil\ \ nil\ \ nil\ ] \\
V.U = & [\ 0\ \ 3\ \ 6\ \ 9\ \ 12\ \ 15\ \ 18\ \ 21\ ] \\
V.C = & [\ nil\ \ nil\ \ nil\ \ nil\ \ nil\ \ nil\ \ nil\ \ nil\ ]
\end{array}$$

7.2 Left-Right Divide and Combine Operations

Recall that the left-right division is associated with a partition over index set, whose quotient function projects the most significant bit of the binary representations of the indices (Section 3.4.2). It follows that the most significant bit of the present index of an entry tells us to which subvector the entry belongs, and the remaining bits give us the new relative index of the entry in the subvector. Equivalently, at the jth stage of division, the jth most significant bit gives us the side of the entry, and the least significant $(w - j)$ bits of the absolute index gives us the its new relative index, where w is the width of the absolute index. This leads to the following simple implementation of the left-right division.

$$
\begin{aligned}
& d_{lr} = !D_{lr} \\
& D_{lr} v = v' \\
& \quad \text{where} \\
& \quad\quad v'.a = u.a \\
& \quad\quad v'.r = sel\_r\ u'.j\ w\ u.a \\
& \quad\quad v'.j = j + 1 \\
& \quad\quad v'.w = w \\
& \quad\quad v'.s = ith\_bit\ (w - u.j)\ u.a \\
& \quad\quad v'.u = u
\end{aligned}
$$

For example, let V be as above, and $V' = d_{lr}\ V$. Then

$$
\begin{array}{rcccccccccc}
V'.A = & [& 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 &] \\
V'.R = & [& 0 & 1 & 2 & 3 & 0 & 1 & 2 & 3 &] \\
V'.J = & [& 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 &] \\
V'.W = & [& 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 &] \\
V'.S = & [& 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 &] \\
V.U = & [& 0 & 3 & 6 & 9 & 12 & 15 & 18 & 21 &] \\
V'.C = & [& nil & nil & nil & nil & nil & nil & nil & nil &]
\end{array}
$$

Note that although the RV V is not physically divided, it can be viewed logically representing two subvectors: the left one consisting of the first four entries, the right one consisting of the rest. Each entry's relative index in the subvector to which it belongs is given by its R field, whereas the subvector it belongs to is indicated by its S field.

Now let us observe the effect of applying d_{lr} again to V'. Let $V'' = d_{lr}\ V'$. Then we will have

$$\begin{array}{rcccccccccc}
V''.A = & [& 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 &] \\
V''.R = & [& 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 &] \\
V''.J = & [& 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 &] \\
V''.W = & [& 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 &] \\
V''.S = & [& 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 &] \\
V''.U = & [& 0 & 3 & 6 & 9 & 12 & 15 & 18 & 21 &] \\
V''.C = & [& nil & nil & nil & nil & nil & nil & nil & nil &]
\end{array}$$

Just as V can be interpreted as a representation of two subvectors of size four, V'' can be viewed as a representation of four subvectors of size two. This can be generalized: if a RV V is a representation of p subvectors, then the application of d_{lr} returns a RV that is a representation of $2 * p$ subvectors.

The left-right combine operation is implemented by a generator called C_{lr}, which essentially is the inverse of D_{lr}

$$
\begin{aligned}
& d_{lr} = !D_{lr} \\
& D_{lr} v = v' \\
& \quad \text{where} \\
& \qquad v'.a = v.a \\
& \qquad v'.r = sel\_r\ u'.j\ w\ u.a \\
& \qquad v'.j = j \dot{-} 1 \\
& \qquad v'.w = v.w \\
& \qquad v'.s = ith\_bit\ (w - u.j)\ u.a \\
& \qquad v'.u = v.u \\
& \qquad v'.c = v.c
\end{aligned}
$$

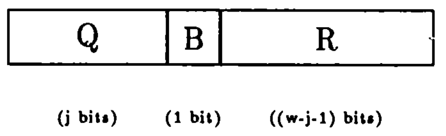
It is easy to verify that $c_{lr} : d_{lr} = id$ holds for all RVs for which d_{lr} is well-defined. For example, let V, V', and V'' be as above. Then

$$
\begin{aligned}
c_{lr}\ V'' &= V' \\
c_{lr}\ V' &= V
\end{aligned}
$$

7.3 Communication

Since communications over arrays in DC algorithms are always specified with generators defined in terms of relative indices (Chapter 3), whereas the target parallel computer only supports communication given in terms of absolute indices, the central issue of implementing communication is the translation of the indexing scheme, rather than the communication per se.

Since relative indices are caused by divide functions, let us examine the effect of division on absolute indices. Clearly, at level j of the d_{lr} division, the binary number of the absolute index is partitioned into three fields. From left to right, the three fields have width j, one, and $(w - j - 1)$ as shown below:



Let us name the three fields by Q, B, and R respectively. Then the field R represents the relative index of the entry, B represents the side which the entry belongs to, and Q contains the bits that are masked at the stage.

The translation algorithm we propose is based on the following observation:

Proposition 7.1 *Let V_l and V_r be respectively the left and right subvectors of an array V, v_l and v_r any two entries belonging respectively to V_l and V_r, Q_l, B_l respectively the Q, B fields of v_l's absolute index, and Q_r, B_r respectively the Q, B fields of V_r's absolute index. Then*

$$Q_l = Q_r$$
$$B_1 = \neg B_r$$

Proof By definition of d_{lr}.

Q.E.D.

The above proposition suggests that if we know the relative index of a communication partner, which is given by applying the communication generator to the relative index, we can then translate it into absolute index by complementing the B field and concatenating it with the Q field of its own absolute index. This leads to the following implementation of the inter-array communication function $\#(f_l, f_r)$, where f_l and f_r are arbitrary communication generators.

```
$tran\ f_l\ f_r = !TRAN\ f_l\ f_r$
$TRAN\ v = partner$
  where
    $r = v.s = 0 \rightarrow\ f_l\ v.r;\ f_r\ v.r$
    $neg\_b = \neg v.s$
    $q = sel\_l\ i\ w(mask\_r\ i\ w)$
    $partner = icat(q, neg\_b, r)$
```

The function *tran* therefore returns a flat vector which specifies the absolute indices of the communication partners for all entries in a RV; the communication is therefore ready to be submitted to the system.

$$\begin{aligned}
&\#(f_l,\ f_r)V = V' \\
&\quad \text{where} \\
&\qquad V'.(A,R,J,W,S,U) = V.(A,R,J,W,S,U) \\
&\qquad V'.C = comm\ (tran\ f_l\ f_r\ V)\ V.U
\end{aligned}$$

Note that in the above $V.(A,R,J,W,S,U)$ denotes respectively the V's A, R, J, W, S, U fields. This implementation of communication therefore has no effect on any fields of an RV except for the field C, which will hold the values obtained from the communication.

For example, let V' be as in Section 7.2, and $V^{(3)} = \#(corr,\ corr)\ V'$. Then

$$\begin{array}{rl}
V^{(3)}.A = & [\ 0\ \ 1\ \ 2\ \ 3\ \ 4\ \ 5\ \ 6\ \ 7\ ] \\
V^{(3)}.R = & [\ 0\ \ 1\ \ 2\ \ 3\ \ 0\ \ 1\ \ 2\ \ 3\ ] \\
V^{(3)}.J = & [\ 1\ \ 1\ \ 1\ \ 1\ \ 1\ \ 1\ \ 1\ \ 1\ ] \\
V^{(3)}.W = & [\ 3\ \ 3\ \ 3\ \ 3\ \ 3\ \ 3\ \ 3\ \ 3\ ] \\
V^{(3)}.S = & [\ 0\ \ 0\ \ 0\ \ 0\ \ 1\ \ 1\ \ 1\ \ 1\ ] \\
V^{(3)}.U = & [\ 0\ \ 3\ \ 6\ \ 9\ \ 12\ \ 15\ \ 18\ \ 21\ ] \\
V^{(3)}.C = & [\ 12\ \ 15\ \ 18\ \ 21\ \ 0\ \ 3\ \ 6\ \ 9\ ]
\end{array}$$

The function $\#(f_l, f_r)$ actually applies to RVs that represent any number of subvectors. For example, let V'' be as in Section 7.2, and $V^{(4)} = \#(corr, corr)\ V''$.

Then .

$$
\begin{array}{rl}
V^{(4)}.A = & [\ 0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ ] \\
V^{(4)}.R = & [\ 0\ 1\ 0\ 1\ 0\ 1\ 0\ 1\ ] \\
V^{(4)}.J = & [\ 2\ 2\ 2\ 2\ 2\ 2\ 2\ 2\ ] \\
V^{(4)}.W = & [\ 3\ 3\ 3\ 3\ 3\ 3\ 3\ 3\ ] \\
V^{(4)}.S = & [\ 0\ 0\ 1\ 1\ 0\ 0\ 1\ 1\ ] \\
V^{(4)}.U = & [\ 0\ 3\ 6\ 9\ 12\ 15\ 18\ 21\ ] \\
V^{(4)}.C = & [\ 3\ 0\ 9\ 6\ 18\ 21\ 12\ 15\ ]
\end{array}
$$

7.4 General Cases

So far we have only showed the implementation of one dimensional recursive arrays, left-right divisions over such arrays, and communications assuming left-right divisions. We thus still need to show how the above implementations can be generalized to other divisions and higher dimensional arrays.

First, let us consider the even-odd division. In Section 3.4.2, we showed that even-odd division is defined by a quotient function symmetric to that of left-right. Instead of projecting the most significant bit, the quotient function for even-odd projects the least significant. This means the even-odd division and combine operations can be implemented by easy modifications of the code we have for left-right divisions. We can also prove a proposition similar to Proposition 7.1 about the three fields produced by even-odd divisions: only the roles of the Q and R fields are reversed. The communications over recursive vectors recursively divided by even-odd divisions thus can be implemented pretty much the same way as we showed for left-right divisions.

The unbalanced head-tail divide and combine operations are relatively easy to implement in comparison with the balanced operations. The mapping from a relative index to a new relative index defined by unbalanced operations is essentially just addition and subtraction.

Higher dimensional recursive arrays are implemented with flat vectors by maintaining multiple fields of binary numbers, each representing indices along one dimension. The divide and combine operations along a dimension are performed by applying the same algorithms for recursive vectors over the field representing the dimension. The product of the division and combine operations is performed by applying each factor operation to the corresponding dimensions. The translation of indices required by our communication scheme is achieved by applying the translation over each dimension. In other words, it requires virtually no new algorithms to implement higher dimensional recursive arrays, given the implementation for one dimensional recursive arrays.

Array distribution is another primitive parallel functional form which we have not shown how to implement. However, this is hardly necessary because array distribution behaves the same on both flat and recursive arrays.

Finally, let us also point out that the definitions of other functional forms can be considered as the code for their parallel implementations because they are given hierarchically in terms of operations discussed above.

7.5 Complexity

Let us assume that flat vectors are distributed on parallel computers, one entry per processor. It is easy to see that the creation, divide, combine, and index translation operations of recursive arrays can all be computed with no inter-processor communications by the above implementation. Moreover, all of them are computed in a constant number of steps independent of the size of the recursive arrays. We therefore have

Proposition 7.2 *The creation, polymorphic divide and combine, and index translation operations of recursive arrays all take $O(1)$ parallel time.*

This proposition justifies the assumption we made in Section 6.3 on the complexities of polymorphic divide and combine operations. It also means there is no penalty associated with the adoption of our relative indexing scheme as far as the asymptotic complexity of communications is concerned.

7.6 Optimization of PDC Algorithms

There are a few quite obvious optimizations which apply to most PDC parallel algorithms.

First of all, we can convert the recursive definition to an equivalent iterative one. Take PDC on one dimensional arrays as an example, let the sizes of the array and the base array be respectively 2^n and 2^m, $k = n - m$. Then a PDC

$$f = PDC(d, c, g, h, p, f_b)$$

can be implemented by

$$(c : h)^k : f_b : (g : d)^k$$

where $(c : h)^n$ is meant to apply the composition $(c : h)$ over all subarray pairs simultaneously for k times (similarly for $(g : d)^k$). Besides saving the overhead associated with recursion, the above implementation also has avoided the cost of applying the base predicate p at each level of division.

Secondly, the $2*k$ iterations implied in the above implementation can be reduced to k iterations for pre or post morphisms where the adjust function h or g is the identity function. This is possible because when the postadjust and preadjust function are identity, we have respectively

$$(c : h)^k = c^k$$
$$(g : d)^k = d^k$$

and c^k and d^k can be computed by functions written specially for them which do not actually go through the k steps.[2]

The above optimizations have been built into the present version Divacon implementation. Other optimization techniques and their theoretical foundations are discussed in [35].

7.7 Implementation and Performance on the CM

By embedding it into the language *Lisp, an initial version of Divacon is now available on the Connection Machine. The author and a small group of other users have used this version of Divacon to develop application programs on the CM for a reasonably wide range of problems. This includes most algorithms presented in this dissertations and many others such as Fibonacci sequence, difference equations, banded linear triangular systems [38], tridiagonal linear systems, and LU decomposition of full and banded matrices.

Several applications developed with the Divacon package have been benchmarked on the Connection Machine Model CM-2 [52] at Yale, which is a 4K processor machine. In the Appendix, we present a number of benchmark graphs, including

Figure A.1 Benchmarks for the PDC broadcast algorithms. There are altogether five curves appearing in the graph. Three of the five curves plot the timings of Algorithm 4.3. They are however benchmarked under different terms. The first one, labelled by "pseudo-morphism", reflects the time performance of the program just as it is; the second one, labelled by "pre-morphism", is benchmarked with the optimization of avoiding the logarithmic steps of combine operations (see last section); the last one, labelled by "pre-morphism(no-coll)" is benchmarked by using the option argument *no-collision* when the underlying *Lisp

[2]The similarity between the optimization of *tail-recursion* [1] and the optimization for premorphism PDC should be observed.

communication function *!!pref* [53] is called. We can do so because the communication used in Algorithm 4.3 is a permutation. The curve labelled by "post-morphism" is the benchmark graph for Algorithm 4.4, with the optimization of skipping the logarithmic steps of division. As one would expect, this curve almost overlaps with the one labelled by "pre-morphism". The last curve is labelled by by the function $(18 * \log n)$, which is intended to be the asymptote of the curve labelled by "pre-morphism(no-coll)". Note that the horizontal coordinate has logarithmic scale, a logarithmic function therefore should be a straight line in the graph. Obviously, despite their difference in real time, each of the benchmark curves indicates $O(\log n)$ time performance of the corresponding program.

Figure A.2 Benchmarks of Algorithm 4.5 for *scan*. There are altogether four curves in the graph, which are labelled and to be interpreted in a way similar to the above. Again, the time performance is $O(\log n)$.

Figure A.3 Benchmark for the polynomial evaluation program, Program 4.6.

Figure A.4 Benchmark graph for Algorithm 4.12, which is a second order PDC algorithm for the monotonic sort. Recall that the top level PDC is a postmorphism with mirror-image communication, whereas the nested level PDC is a premorphism with correspondent communication (Section 4.3.5). When this program is benchmarked, optimization was applied such that the combine phase of the premorphism and the division phase of the postmorphism are all avoided. We did choose to turn on the no-collision option for two, one, and none of the two types of communications. This is why we have multiple curves in the graph; each of them, however, seems to confirm the predicated $O(\log^2 n)$ performance (Chapter 6). Time performance can be seen to be about $O(\log^2 n)$.

Figure A.5 Benchmark of Algorithm 5.3 – the SDC linear lower triangular system

using balanced division. Unlike others, this graph is drawn on linearly scaled horizontal coordinate. The curve in the graph indicates $O(n)$ time performances of the algorithm on the CM, which confirms what is predicated in both Chapter 5 and Chapter 6.

The primitive operations over recursive arrays were also benchmarked. Figure A.6 shows the time used by the creation (*mkrv*), left-right division d_{lr}, and left-right division d_{lr}. Figure A.7 shows the time used by various communications. What should be observed is that Proposition 7.2 is quite convincingly confirmed in practice.

Our experience shows that programming in Divacon notation often can improve the programming productivity drastically. The time to write and debug a Divacon program for the applications mentioned above, for example, ranged from 5 to 30 minutes.

Chapter 8

Conclusion

8.1 Ending Remarks

The central subject of this dissertation is the algebraic model for divide-and-conquer and its parallel realization. There are, however, cross-links to other areas of theoretical and systems computer science.

First of all, this work suggests a new class of parallel programming languages based on divide-and-conquer. Languages of this class are fundamentally different from other parallel languages and possess some unique features: (1) recursive parallel data types, (2) parallel primitives and functional forms with mutually orthogonal functionalities, and (3) powerful divide-and-conquer constructs. Such languages are freed from what Backus called *the von Neumann style* for a simple reason – their design reflects the studies of the mathematical properties of the problems rather than the architecture of the machines. A prototype of the class is embodied in the Divacon notation.

Second, it puts forward two new programming constructs, PDC and SDC, which demonstrate at least three significant advantages. First, they are general enough to subsume many other common constructs such as broadcast, reduction, scan, transpose, sort and FFT. In so doing, secondly, they reveal the inter-relation among these other constructs, i.e., that they all share a common set of divide, combine and communications functions. Finally, this clearly causes a reduction in the cost of development because each of the above constructs can be "assembled" with the DC constructs and the shared constituents rather than developed independently.

Third, this dissertation introduces the notion of recursive arrays in the design of parallel languages. Conventional arrays are designed implicitly with the von Neumann machine in mind. Its primary operation – indexing – assumes the computation is to be performed by stepping through the array one entry at a time. It follows that iteration is the dominant style of programming in array processing. Conventional arrays are also flat in the sense that they cannot be explicitly taken apart and put together. By contrast, recursive arrays allow array computation specified collectively over the array entries and support recursive programming style by providing divide, combine, communication, and distribution operations.

Fourth, a taxonomy of parallel operations is provided. It is highly desirable to identify a small set of primitive parallel operations through which general parallel computations can be expressed. This is possible only if the set is orthogonal in the sense that the primitive functionalities do not intersect. By this principle, operations acting on array indices and array values are separated, and operations blending local computation and communication are not primitive. The study of the constituents of DC algorithms has lead to the identification of such a set, and concurrently provides enhanced program modularity as demonstrated by the Divacon notation.

Fifth and last, this work helps to identify to the sources of parallelism in parallel programs. Depending on the impact on the execution time, parallelism in DC algorithms is divided into two types: multiplicative parallelism and additive parallelism. These two types of parallelism roughly correspond to *control parallelism* and *data parallelism*, discussed by Hillis and Steele in [23]. An underlying assumption made in [23] is that the two types of parallelism are antithetical – the premise of exploiting one is the sacrifice of the other. The CM programming languages, in which only data parallelism is explicitly exploited, have clearly been designed on this premise. In contrast, we have seen that both types of parallelism can be exploited simultaneously and harmoniously under the DC models with the Divacon notation.

To summarize, the work has not only led to the formation of an algebraic model for the programming paradigm of divide-and-conquer, but also has shed light from a unique angle on the following general issues: the design of parallel programming languages, the selection of parallel programming constructs, the nature of parallel arrays, the taxonomy of parallel primitives, and the sources of parallelism.

8.2 Limitations

Although the DC model presented here is very broad, we have studied only a subset of DC algorithms under the model. This subset might be termed *polymorphic DC algorithms*, categorized by the divisions they employ.

There are good reasons for studying polymorphic divide and combine operations first - they are conceptually simple, rich in parallelism, shared by many algorithms, and easy to implement on parallel computers. However, if we restrict ourselves to polymorphic operations we may fail to find DC solutions in other domains. Many graph problems such as minimum spanning tree, connected component, and graph coloring, for example, have not been known to have efficient direct polymorphic DC algorithms.[1]

Non-polymorphic divide and combine operations, therefore, should not be disregarded. Some classic examples of non-polymorphic operations include the division in

[1] Here, *direct* polymorphic DC algorithms for graph problems refer to those in which the graph itself is recursively divided and the algorithm is recursively applied. Note that it is well known that many graph problems can be solved by using a number of programming constructs, for example, broadcast and reduction. Since these constructs can be computed with polymorphic DC algorithms, many graph problems can indeed be solved *indirectly* by polymorphic DC algorithms.

quicksort [3], the combine operation in *mergesort* [3], the *separators* in VLSI layout algorithms [54], *Euler partitioning* of graphs [20], used for example in graph coloring problems, and domain decomposition [7, 17] in numerical analysis problems. Divide-and-conquer algorithms based on these non-polymorphic operations have been shown to be parallelizable in many cases. Whether the Divacon programming system can be extended to gracefully handle non-polymorphic DC algorithms is an open question at present.

The DC model is defined in terms of algebraic properties of the problems and not in terms of machine operations; hence, most of the discussion has been independent of the machine architectures. However, the claim that DC algorithms generally yield optimal or sub-optimal asymptotic performance is true only for the implementation on hypercube machines or their isomorphs including butterfly, cube-connected cycles, and perfect shuffle, all characterized by small diameter (logarithmic distance between any two processors) and rich inter-processor connections. Whether the DC models can be realized efficiently on parallel machine architectures with different topology is yet to be investigated.

8.3 Related Work

An in-depth study of divide-and-conquer as programming paradigm can be found in [3]. Aho *et al.* showed therein how divide-and-conquer can be applied to a broad class of problems, including finding the maximum and minimum element of a set, integer multiplication, permutation, finding the kth smallest element of a set, multiplication between a Toeplitz matrix and a column vector, matrix multiplication, LU decomposition, FFT, and Chinese remaindering. More important, they studied the general structure and complexity analysis of DC algorithms. Although the algorithms were presented for sequential computing, most of them can be easily parallelized.

Preparata and Vuillemin's work presented in [43] is, in our opinion, a milestone in the studies on DC algorithms. Preparata and Vuillemin not only point out the significance of divide-and-conquer to parallel computation, but also set forth two operational models for divide-and-conquer – *descend* and *ascend*. These models were applied to many problems including FFT and sort, and it was shown how they can be implemented efficiently on cube-connected-cycle parallel computers. It should be pointed out that *descend* and *ascend* are instances of our premorphism and postmorphism DC models where division is *left-right* and communication *correspondent*. Therefore, *ascend* and *descend* should be considered as special cases of the DC models in this dissertation. Nevertheless, these two restricted models were found to be so general that they applied to "all the interesting algorithms for parallel processing known to the authors." Preparata and Vuillemin noted how DC algorithms can be nested to form what they call "composite (DC) algorithms" or what I call "higher order DC algorithms". They are also the only authors, to the best of my knowledge, who realized the potential significance of "developing a high level, general purpose language for parallel programming (based on DC models)".

Preparata and Vuillemin also pointed out in [43] the duality between the two DC models, which allows a DC algorithm under one model to be transformed to another by a *bit reversal permutation* on the input. This duality corresponds to an equivalent relation over DC algorithms that can be proven under the DC models in this dissertation – a premorphism with left-right division is equivalent to a postmorphism with even-odd division providing that the communication is *correspondent*, and vice versa [35]. This equivalent relation allows many DC algorithms with left-right division to be converted automatically into a DC algorithm with even-odd division, or conversely, and also establishes a link between the DC models and the so-called *odd-even cyclic reduction* used, for example, by Johnsson in his tridiagonal system algorithm [28].

Smith in [46, 47] explored the possibility of deriving divide-and-conquer algorithms automatically from the formal specifications of the problems, and successfully applied his method to several problems such as quicksort, mergesort, and maximum sum over the matrix regions. The discussion he made therein on divide-and-conquer actually implied a morphism model for DC. He also used a notation similar to FP and Divacon to specify DC algorithms.

Stone's work on perfect shuffle in [50] should be mentioned. The parallel algorithms appearing in [50] including FFT, polynomial, bitonic sort, and matrix transposition, are all actually, although not so stated, canonical DC algorithms. Stone showed how these algorithms can be cleverly computed by perfect shuffle. His work also implied the separation between the array operations that handle indices and the array operations that handle the indexed values; this separation is made explicit in this dissertation by introducing the concept of relational functions and universal functions. Finally, Stone also observed how the operations over the indices can be easily performed on the binary representations of the indices, which can in fact be translated into our implementation of balanced divisions over recursive arrays.

Iverson showed us with the language APL how computation can be specified in terms of a collection of programming constructs including reduction, scan, transpose, and inner-product [26, 42]. APL thus opened a new avenue to computing which deviates from the one-word-at-a-time von Neumann style. The success of APL suggests that parallel computation may be harnessed by providing efficient parallel implementations of certain programming constructs. And this idea has been intensively pursued. Ladner and Fischer in [31] showed how prefix (scan) can be computed efficiently by boolean circuit and how to simulate a finite state transducer with their solution; Huang in [25] showed us the parallel solution to some graph problems, such as minimum spanning tree and connected components, by implementing broadcast and reduction on mess-of-trees. The hypercube implementation of broadcast, which can be viewed as a special case of scan, is well studied by Johnsson and Ho [29, 30]. Mu and Chen studied how broadcast, reduction, and scan can be performed on dynamically linked data structures on iPSC [39]. Hillis and Steele illustrated the application of scan to problems including parsing of regular language and region labelling [23].

Blelloch in [8, 9] stressed the generality of scan and discussed its application to problems from the areas including graph theory, numerical analysis, and computational geometry. The PDC and SDC higher order functions proposed in this dissertation can be considered as new parallel programming constructs. The viability of the DC constructs is rooted in their generality – they subsume all the abovementioned constructs, and is also supported by the fact that DC constructs, like other parallel programming constructs, can be implemented efficiently on hypercube machines despite the generality.

Related to the above is a general programming technique called *recursive doubling* (also called *pointer jumping*) [25, 8, 20]. Recursive doubling can be shown to compute exactly the function scan and therefore has the same power and weakness (one directional dependency) that scan has in terms of functionality. It follows from the previous discussion that recursive doubling can also be computed by the DC constructs assuming random accessible data structure like arrays. (Recursive doubling might still be a better choice than DC to perform scan over linked data structures like lists.)

Hillis and Steele's model of data parallel computation [23, 24] is related to this work in several ways. First of all, both the data parallelism model and the DC model encourage the perception that a piece of data is a processing unit. Secondly, our implementation of the DC model is erected on top of data parallelism programming systems. Finally, the CM languages based on the notion of data parallelism, in particular *Lisp, provide an excellent environment for the CM implementation of the DC models.

So much work has been done in the development of sequential or parallel divide-and-conquer algorithms that it is impossible to enumerate. I would however like to mention Danielson and Lanczos's FFT [19], Ladner and Fischer's prefix [31], Batcher's bitonic sort [6], and Strassan's matrix multiplication algorithms [3]. Without the insight gained in the study of their DC algorithms, the formation of the models presented in this paper would be impossible.

Backus' work on FP [4, 5] has a sound impact on the design of Divacon notation. Backus's notion of functional forms has been adopted; his penetrating comments on von Neumann style programming provided crucial guidance in selecting the constituent parallel operations of the DC models.

8.4 Future Work

A complete programming system, according to Hoare [10], should include methods by which one can reason about programs and by which they can be transformed from one form to another to achieve higher efficiency. On the one hand, this work on DC models does not constitute such a complete system owing to the absence of these methods. On the other hand, the formality associated with the DC models and Divacon notation has provided a solid basis for their development. In fact, effort and progress have been made in this respect since the first draft of this dissertation was written. The preliminary results of this effort already enable us to automatically eliminate broadcast communication, transform premorphism to postmorphism algorithms and vice versa, and transform the composition of a premorphism and postmorphism into one pseudomorphism under certain conditions [35]. New and more general equivalent relations between DC algorithms, however, are still to be discovered.

As a programming language, the Divacon notation is yet to be given its formal syntax and semantics. A compiler for Divacon is being developed at Brandeis University. The performance of Divacon applications is expected to improve significantly once the compiler is developed. The equivalent relations between DC algorithms will be used for code optimization during the compilation. A project aimed at performing automatic complexity analysis of Divacon programs is also under way.

Applications of the DC models to more and broader range of problems are expected. These applications should yield asymptotically optimal or sub-optimal parallel time performance. What remains to be seen is whether they can compete with applications developed under other models in terms of megaflops delivered on given parallel computers.

Appendix A

Benchmark Graphs

This appendix includes the following benchmark graphs showing the time performance of several Divacon programs presented in the document and the time performance of primitive recursive array operations:

Figure A.1 Broadcast. Algorithm 4.3 and Algorithm 4.4.

Figure A.2 Scan. Algorithm 4.5.

Figure A.3 Polynomial evaluation. Algorithm 4.6.

Figure A.4 Monotonic sort. Algorithm 4.12.

Figure A.5 Triangular systems. Algorithm 5.3.

Figure A.6 Primitive recursive array operations.

Figure A.7 Communications over recursive arrays.

These benchmarks were collected by the author on the 4096 processor Connection Machine Model CM-2 at Yale between 12th April and 28th April 1989.

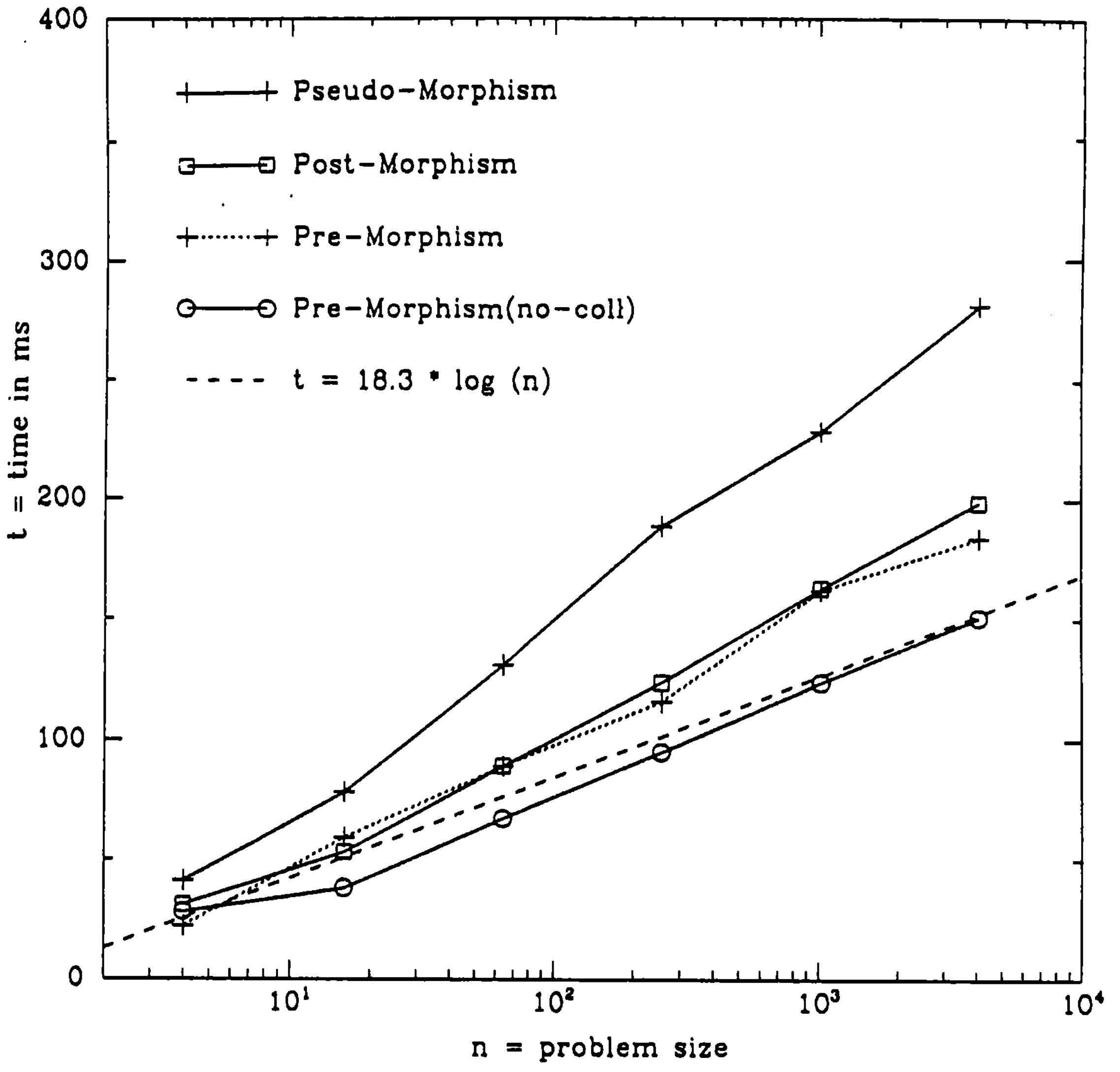


Figure A.1: Broadcast by Divacon on the CM

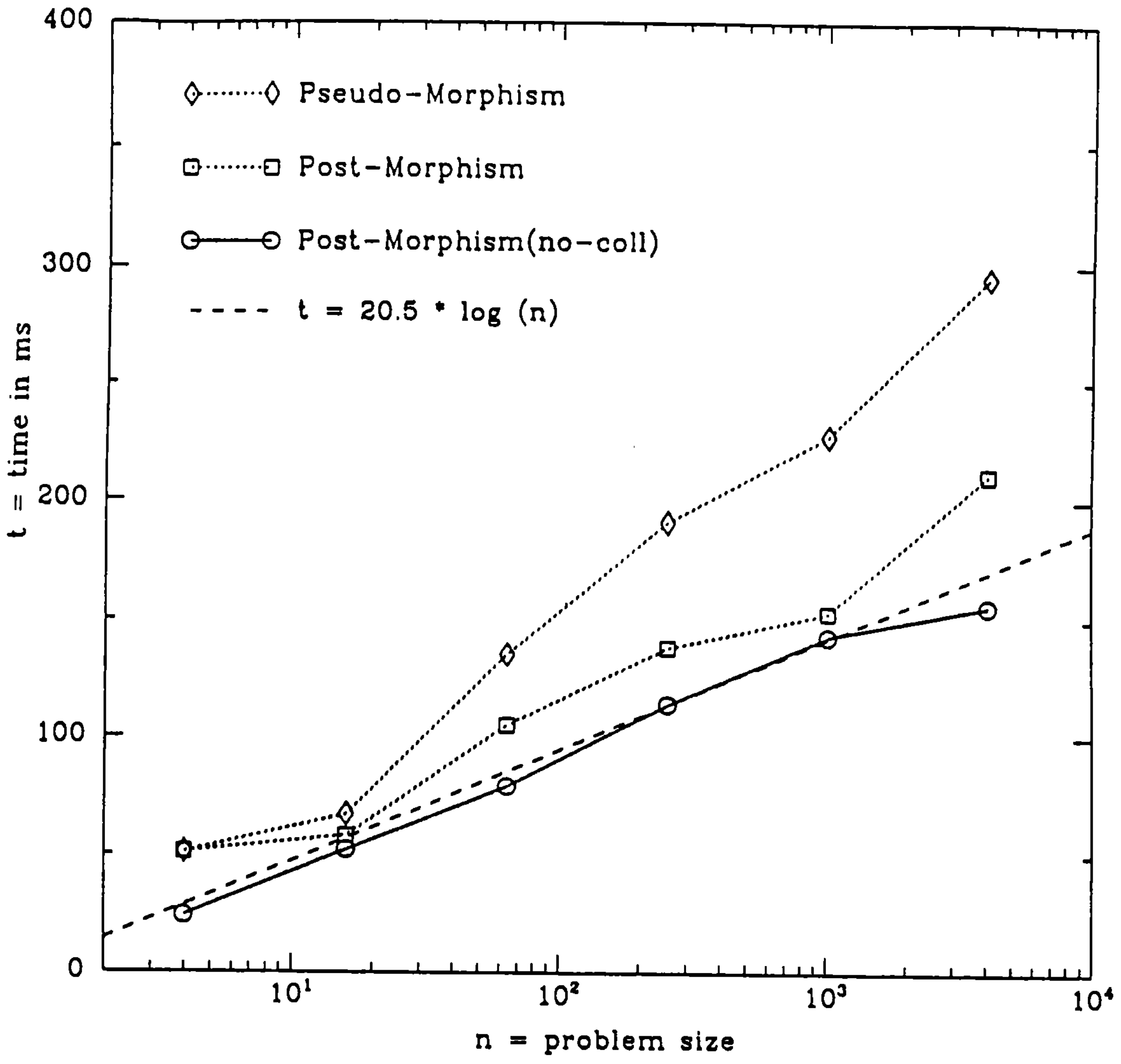


Figure A.2: Scan by Divacon on the CM

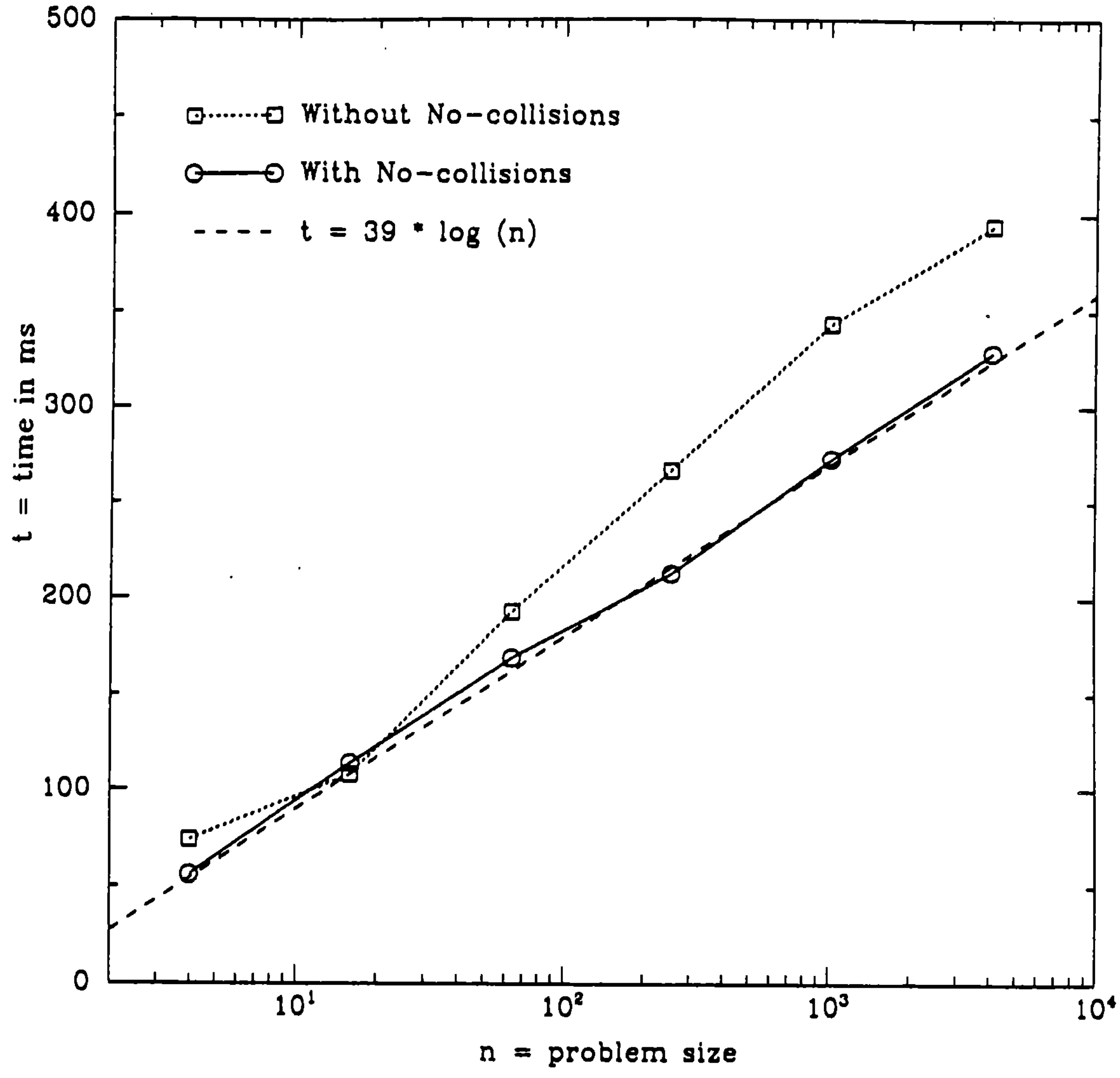


Figure A.3: Polynomial evaluation by Divacon on the CM

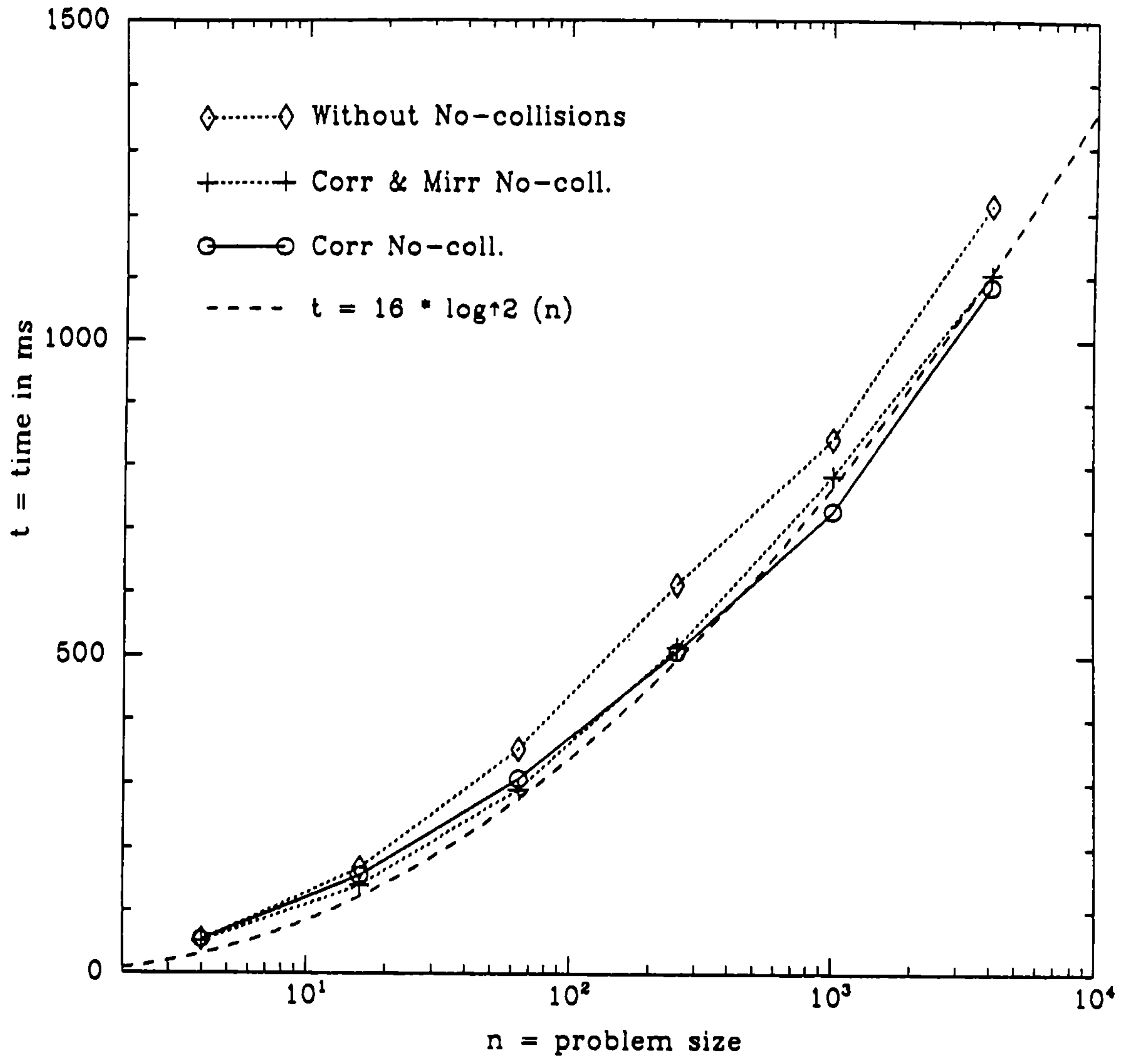


Figure A.4: Monotonic sort by Divacon on the CM

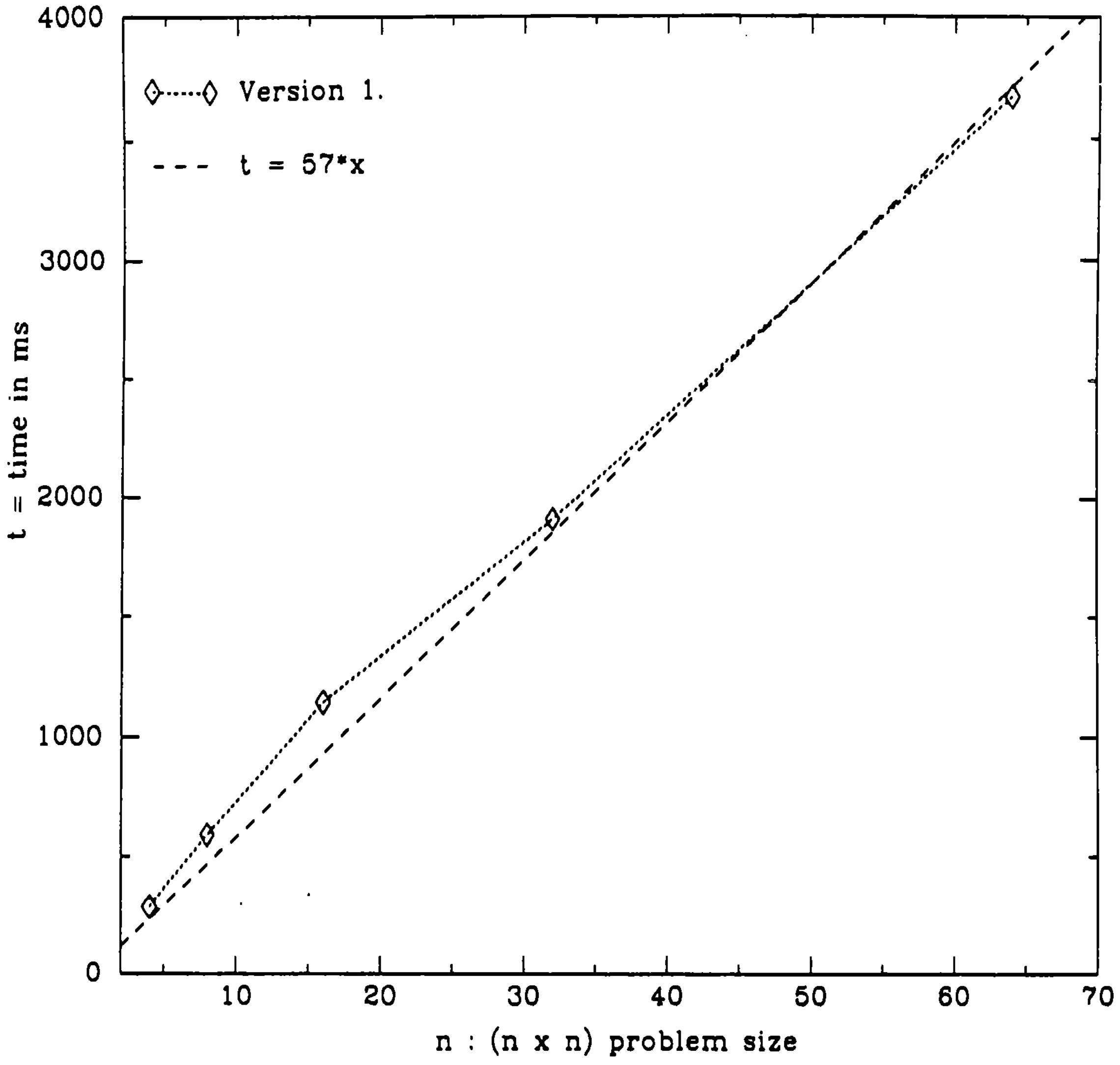


Figure A.5: Linear triangular systems by Divacon on the CM

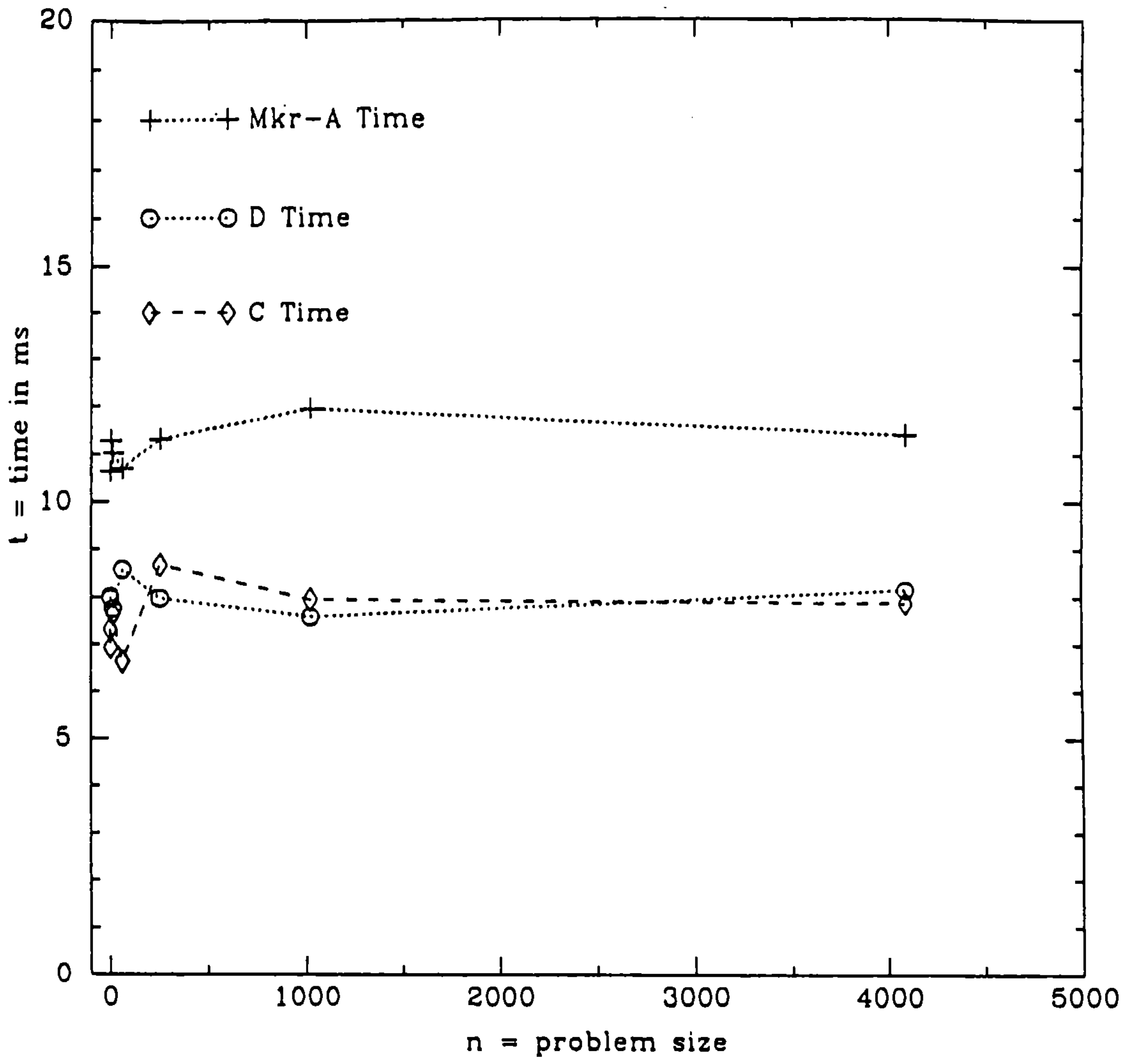


Figure A.6: Recursive array operations on the CM

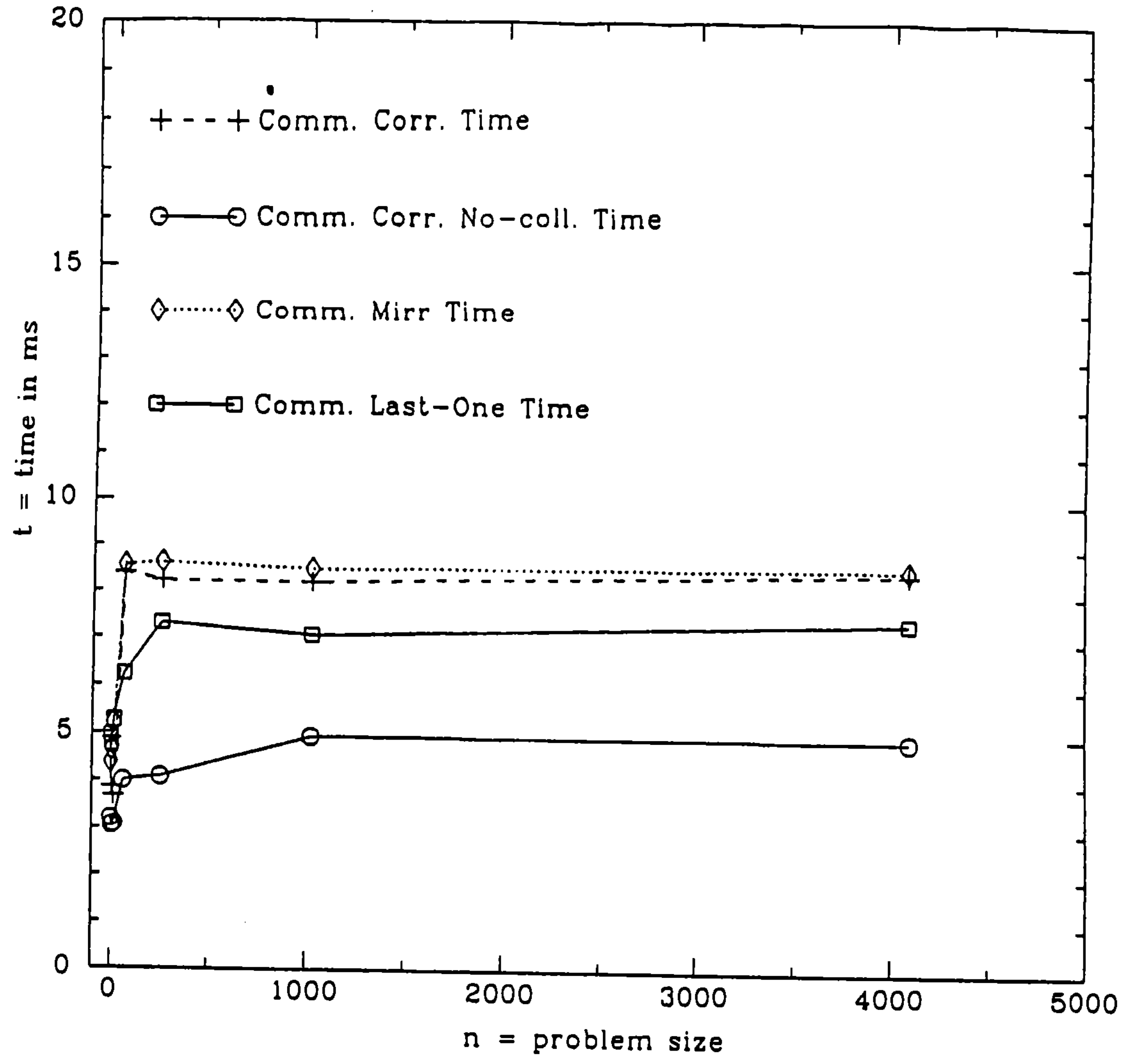


Figure A.7: Communications over recursive arrays on the CM

Bibliography

[1] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs.* The MIT Press, 1985.

[2] S. Abramsky and K. Hankin, editors. *Abstract Interpretation of Declarative Languages.* Ellis Horwood Limited, Chichester, 1986.

[3] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addision-Wesley, 1974.

[4] J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978.

[5] J. W. Backus. The algebra of functional programs: Function level reasoning, linear equations and extended definitions. In *Lecture Notes in Computer Science*, volume 107, pages 1–43. Springer-Verlag, 1981. Formalization of Programming Concepts.

[6] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the AFIPS Spring Joint Computer Conference*, volume 32, pages 307–314, 1968.

[7] J. M. Berger and H. S. Bokhari. A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Transactions on Computers*, C-29(5):570–580, May 1987.

[8] Guy Blelloch. Applications and algorithms on the Connection Machine. Technical Report TR87-1, MIT AI Laboratory, February 1987.

[9] Guy Blelloch. Scans as primitive parallel operations. In *International Conference on Parallel Processing*, 1987.

[10] K. M. Chandy and J. Misra. *Parallel Program Design - A Foundation*, page vi. Addison-Wesley, 1989. Foreword by Hoare.

[11] G. C. Clark, Jr. and J. B. Cain. *Error-Correction Coding for Digital Communications.* Plenum Press, 1981.

[12] S. D. Conte and Carl de Boor. *Elementary Numerical Analysis - An Algorithmic Approach.* McGraw-Hill Book Company, 1972.

[13] D. Van Dalen, H. C. Doets, and H. de Swart. *Sets: Naive, Axiomatic and Applied.* Pergamon Press, 1978.

[14] Narsingh Deo. *Graph Theory with Applications to Engineering and Computer Science.* Prentice-Hall, 1974.

[15] W. Edsger Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[16] L. L. Dornhoff and F. E. Hohn. *Applied Modern Algebra.* Macmillan Publishing Co., Inc, 1978.

[17] G. Fox *et al.* *Solving Problems on Concurrent Processors*, volume 1. Prentice-Hall, 1988.

[18] P. Hudak *et al.* Report on the functional programming language Haskell. Technical Report YALEU/DCS/RR666, Yale University, Department of Computer Science, December 1988.

[19] W. H. Press *et al.* *Numerical Recipes - The Art of Scientific Computing.* Cambridge University Press, 1986.

[20] A. Gibbons and W. Rytter. *Efficient Parallel Algorithms.* Cambridge University Press, 1988.

[21] Thomas L. Heath. *A History of Greek Mathematics*, volume Vol. I. Oxford, 1921.

[22] Peter Henderson. *Functional Programming – Application and Implementation.* Prentice-Hall International, Englewood Cliffs, NJ, 1980.

[23] W. D. Hillis and G. L. Steele Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, December 1986.

[24] W. Daniel Hillis. *The Connection Machine.* The MIT Press, 1985.

[25] M.-D. Huang. Solving graph problems with optimal speed up on mesh-of-tree networks. In *Proc. Twenty-sixth Annual Symposium on Foundations of Computer Science*, pages 232–240. IEEE Computer Society Press, 1985.

[26] K. A. Iverson. *A Programming Language.* Wiley, New York, 1962.

[27] Nathan Jacobson. *Basic Algebra* I. W. H. Freeman and Company, San Francisco, 1974.

[28] S. Lennart Johnsson. Solving tridiagonal systems on ensemble architectures. *SIAM J. Sci. Stat. Comput.*, 8(3):354–391, 1987.

[29] S. Lennart Johnsson and Ching-Tien Ho. Spanning balanced trees in boolean cubes. Technical Report YALEU/DCS/TR508, Yale University, January 1987.

[30] S. Lennart Johnsson and Ching-Tien Ho. Spanning graphs for optimum broadcasting and personalized communication in hypercubes. *IEEE Trans. Computers*, 38(9):1249–1268, September 1989.

[31] R. E. Ladner and M. J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27(4):831–838, 1980.

[32] H. R. Lewis and C. H. Papadimitriou. *Elements of The Theory of Computation.* Pretice-Hall, Inc, 1981.

[33] George McCarty. *Topology - An Introduction with Application to Topological Groups.* Dover Publications, Inc., 1967.

[34] Z. G. Mou, S. Anderson, and P. Hudak. Parallelism in sequential divide-and-conquer. Technical Report YALEU/DCS/TR683, Yale University, Department of Computer Science, February 1989.

[35] Z. G. Mou and B. Carpentieri. *On the Transformation and Composition Properties of Divide-and-Conquer Algorithms.* In preparation.

[36] Z. G. Mou and P. Hudak. An algebraic model for divide-and-conquer algorithms and its parallelism. *The Journal of Supercomputing*, 2(3):257–278, November 1988.

[37] Z. G. Mou and P. Hudak. Parallel programming in DIVACON. Technical Report YALEU/DCS/TR675, Yale University, Department of Computer Science, 1990. (to appear).

[38] Z. G. Mou, D. E. Keyes, and W. D. Gropp. *Balanced Divide-and-Conquer Algorithms for the Fine-Grained Parallel Direct Solution of Dense and Banded Triangular Linear Systems and their Connection Machine Implementation. Fourth SIAM Conference on Parallel Processing for Scientific Computing*, December 1989.

[39] Z. Mu and Marina Chen. Communication-efficient distributed data structure. In *Hypercube Multiprocessors*, pages 67–78, 1987.

[40] P. A. Nelson and L. Snyder. Programming paradigms for nonshared memory parallel computers. In L. H. Jamieson, D. B. Gannon, and R. J. Douglass,

editors, *The Characteristics of Parallel Algorithms*, pages 3–20. The MIT Press, 1987.

[41] Simon Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall International, Englewood Cliffs, NJ, 1987.

[42] R. P. Plivaka and S. Pakin. *APL: The Language and Its Usage.* Prentice-Hall, 1962.

[43] F. P. Preparata and J. Vuillemin. The cube-connected cycles: A versatile network for parallel computation. *Communications of the ACM*, 8(5):300–309, May 1981.

[44] E. M. Reingold, J. Nievergelt, and N. Deo. *Combinatorial Algorithms: Theory and Practice.* Prentice-Hall, 1977.

[45] David A. Schmidt. *Denotational Semantics – A methodology for Language Development.* Allyn and Bacon, Inc., 1986.

[46] D. R. Smith. Top-down synthesis of divide-and-conquer algorithms. *Artificial Intelligence*, pages 43–96, 1985.

[47] D. R. Smith. Applications of a strategy for designing divide-and-conquer algorithms. *Science of Computer Programming*, 8:213–229, 1987.

[48] D. F. Stanat and D. F. McAllister. *Discrete Mathematics in Computer Science.* Prentice-Hall, Englewood Cliffs, NJ, 1977.

[49] Guy L. Steele Jr. *Common Lisp : The Language.* Digital Press, 1984.

[50] H. S. Stone. Parallel processing with the perfect shuffle. *IEEE Transactions on Computers*, C-20(2):153–160, February 1971.

[51] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* The MIT Press, 1977.

[52] Thinking Machines Corporation, Cambridge, Massachusetts. *Connection Machine Model CM-2 Technical Summary*, version 5.1 edition, May 1988.

[53] Thinking Machines Corporation, Cambridge, Massachusetts. **Lisp Reference Manual*, version 5.0 edition, September 1988.

[54] J. D. Ullman. *Computational Aspect of VLSI.* Computer Science Press, 1984.

[55] L. G. Valiant. A scheme for fast parallel communications. *SIAM J. Computing*, 11(2), 1982.

[56] H. H. Wang. A parallel method for tridiagonal equations. *ACM Transactions on Mathematical Software*, 7(2):170–183, June 1981.

[57] David S. Wise. Matrix algebra and applicative programming. In *Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science*, pages 134–153. Springer, 1987.