The Journal of Supercomputing, 2, 257–278 (1988)

An Algebraic Model for Divide-and-Conquer and Its Parallelism*

ZHIJING G. MOU
PAUL HUDAK
Yale University, Department of Computer Science, New Haven, CT 06520

Abstract. A formal algebraic model for divide-and-conquer algorithms is presented. The model reveals the internal structure of divide-and-conquer functions, leads to high-level and functional-styled algorithms specification, and simplifies complexity analysis. Algorithms developed under the model contain vast amounts of parallelism and can be mapped fairly easily to parallel computers.

1. Introduction

Divide-and-conquer is a well-known strategy for designing parallel algorithms: A problem is recursively subdivided into relatively independent components, which are in turn operated on in parallel [Aho et al. 1974, Jamieson et al. 1987, Preparata and Vuillemin 1981, Ullman 1984]. The method is both simple—even the most novice programmers find it easy to grasp—and effective—it is the basis for many of the best known parallel algorithms.

However, despite the widespread use of divide-and-conquer, it has received very little formal treatment in the literature. In this paper we develop a formal *algebraic* model of divide-and-conquer. Our motivation stems from the desire to answer the following questions:

- What is the class of problems that can be attacked by divide-and-conquer?
- What are the structural and domain properties of "divide" and "combine" functions?
- Are there other inherent constituents of divide-and-conquer algorithms aside from divide and combine functions?

We begin in the next section by noticing that *morphisms* in basic algebra [Dornhoff and Hohn 1978] resemble the fundamental structure of divide-and-conquer. However, many problems that we would like to solve by divide-and-conquer are almost, but not quite, morphisms. Therefore, we introduce the notion of *adjust functions* which allow

*This research was supported in part by DOE grant DOE FG02-86ER25012.

us to generalize the concept of morphisms to *pseudomorphisms* and thus complete the foundation for our model.

The model is further refined in Section 3, where the notions of *space, divide* function, and *combine* function are introduced. In Section 4, we study the nature of adjust functions and point out the relation between adjust functions and *interspace communication.*

In Section 5 we show how divide-and-conquer algorithms are specified in terms of the constituent functions. We follow in Section 6 with ten examples demonstrating the applicability of our model. In Section 7 we discuss in principle the parallel implementation of divide-and-conquer under our model, and complexity and trade-off issues are addressed in Section 8.

Aside from answering theoretical questions, the advantages of our approach in practice include the following:

- *It aids in the design of divide-and-conquer algorithms.* The algorithm design is reduced to the problem of identifying the constituent functions.
- *It improves the clarity and modularity of programming.* The model's structure suggests simple constructs or higher order functions to capture divide-and-conquer behavior in a clean way. Furthermore, different divide-and-conquer algorithms often share the same constituent functions, thus encouraging reuse of parts.
- *It facilitates the parallel implementation of divide-and-conquer algorithms.* Once we

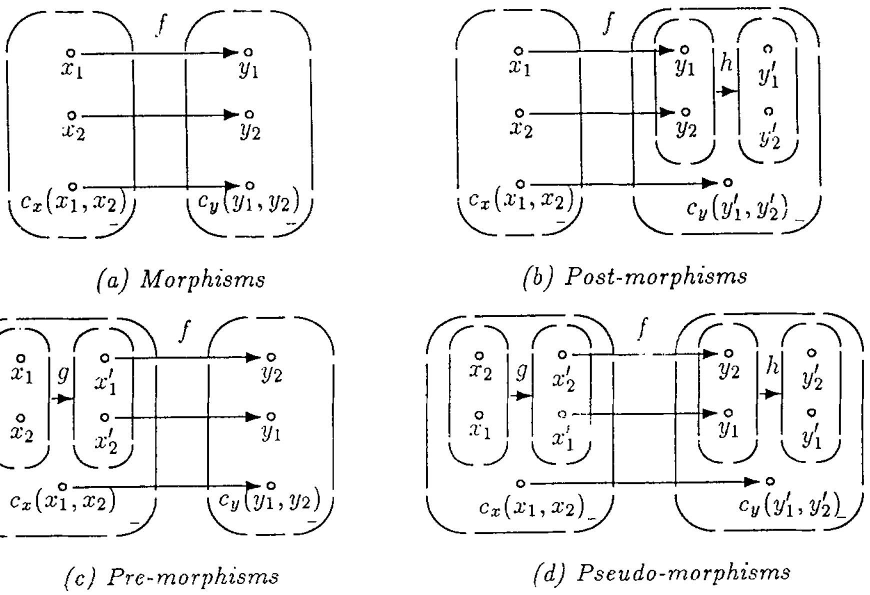


Figure 1. Different types of pseudo-morphisms ($k = 2$).

understand how to map spaces to and implement the constituent functions on parallel computers, we can implement any divide-and-conquer algorithm.

- *It simplifies complexity analysis.* The complexity of divide-and-conquer algorithms can be easily derived from the complexity of the constituent functions.

Notation. We generally write function application in *curried* form, as in $f\,x\,y$, which is equivalent to $(f\,x)\,y$. Function composition is denoted $f \circ g \circ h$, and associates to the right. Tuples are denoted by $\langle x_1, x_2, \ldots, x_k \rangle$ and vectors by $[x_1\, x_2 \ldots x_k]$. The latter notation is overloaded in that we also use $[X, c_x]$ to denote an algebra—the distinction is always clear from context. A vector v is indexed by writing $v(i)$, and its length is $|v|$.

2. Pseudo-morphisms as a Basis for Divide-and-Conquer

We first review some basic concepts in algebra. A set S together with a k-ary operator $c_s : S^k \rightarrow S$ is an *algebra* $[S, c_s]$ if S is closed under c_s (i.e., for any k elements $s_1, \ldots, s_k \in S$, we have $c_s\, \langle s_1, \ldots, s_k \rangle \in S$). Let function f be a mapping from set X to set Y. It is a *morphism* [Dornhoff and Hohn 1978] (see Figure 1(a) if there are algebras $[X, c_x]$ and $[Y, c_y]$, such that

$$f\,(c_x\, \langle x_1, \ldots, x_k \rangle) = c_y\, \langle f\,x_1, \ldots, f\,x_k \rangle$$

which we prefer to write as

$$(f \circ c_x)\, \langle x_1, \ldots, x_k \rangle = (c_y \circ (map\,f))\, \langle x_1, \ldots, x_k \rangle$$

$$\text{where } map\,f\, \langle x_1, \ldots, x_k \rangle = \langle f\,x_1, \ldots, f\,x_k \rangle$$

or just

$$f \circ c_x = c_y \circ (map\,f)$$

It is the presence of *map*'s that is the key to divide-and-conquer and that permits the parallel evaluation of the function on disjoint arguments.

As an example, consider the function *reduce* defined over vectors V of elements from set U:

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

where $\oplus$ is an associative binary operator defined over U. We can see that the function $(reduce\, \oplus)$ is a morphism from $[V, cat]$ to $[U, \oplus]$, where *cat* is the vector concatenation operator. For example,

$$((\textit{reduce}\ +) \circ \textit{cat})\ \langle [1\ 2], [3\ 4]\rangle = (+ \circ (\textit{map}\ (\textit{reduce}\ +)))\ \langle [1\ 2], [3\ 4]\rangle = 10$$

Morphisms constitute a fairly broad class of functions, and include all the *linear* functions encountered in mathematics and engineering such as differentiation, integration, convolution, Fourier transformation, and many more basic functions in arithmetic and linear algebra.

2.1 Postmorphisms

The linearity of morphisms has long been explored in sequential computations, and has obvious significance for parallel processing. Unfortunately, the class of morphisms, wide as it is, does not include many functions that intuitively can also be decomposed in a similar way as morphisms.

For example, consider the function $(\textit{scan}\ \oplus): V \to V$, where V is again the set of vectors with elements from set U. $\oplus$ is an associative binary operator over U, and

$$\textit{scan}\ \oplus\ v = v', \text{ where } v'(i) = \bigoplus_{k=1}^{i} v(k), \text{ for } i = 1 \text{ to } |v|$$

For algebra $[V, \textit{cat}]$, the function $(\textit{scan})\ \oplus)$ is not a morphism since $((\textit{scan}\ \oplus) \circ \textit{cat})\ \langle v_1, v_2\rangle$ is not equal to $\textit{cat}\ \langle \textit{scan}\ \oplus\ v_1, \textit{scan}\ \oplus\ v_2\rangle$. For example, $\textit{scan} + [1\ 2\ 3\ 4] = [1\ 3\ 6\ 10]$ while $\textit{cat}\ \langle \textit{scan} + [1\ 2], \textit{scan} + [3\ 4]\rangle = [1\ 3\ 3\ 7]$. However, upon closer inspection we find that the result is "almost" correct, and can be easily adjusted to the correct result. To see how, define the following function h to do the adjustment:

$$h\ \langle v_1\ \ v_2\rangle = \langle v_1, v'_2\rangle, \text{ where } V'_2(i) = v'_2(i) \circ v_1(|v_1|)$$

Then we can see that

$$((\textit{scan}\ \oplus) \circ \textit{cat})\ \langle v_1, v_2\rangle = (\textit{cat} \circ h \circ (\textit{map}\ (\textit{scan}\ \oplus)))\ \langle v_1, v_2\rangle$$

Formally, a function $f: [X, c_x] \to [Y, c_y]$ is called a *postmorphism* (see Figure 1b) if there exists a *postadjust function* $h: Y^k \to Y^k$ such that

$$(f \circ c_x)\ \langle x_1, \ldots, x_k\rangle = (c_y \circ h \circ (\textit{map}\ f))\ \langle x_1, \ldots, x_2\rangle$$

2.2. Premorphisms

Consider the function *shuffle* defined over vectors (of even length):

$$\textit{shuffle}\ (v) = v'$$
where $v'\ (i) = v\ ((\textit{decode} \circ \textit{left-shift} \circ \textit{encode})(i - 1))$, for $i = 1$ to $|V|$

where the function *encode* takes a number and encodes it in binary, *decode* does the opposite, and *right-shift* and *left-shift* shift the digits of a binary number, respectively, to the right and the left in wrapped-around fashion. Shuffle is not a morphism (nor is it a postmorphism) for algebra $[V, cat]$ because, for example,

$$\begin{aligned}&(shuffle \circ cat)\ \langle[1\ 2], [3\ 4]\rangle = [1\ 3\ 2\ 4]\\ &\neq (cat \circ (map\ shuffle))\ \langle[1\ 2], [3\ 4]\rangle = [1\ 2\ 3\ 4]\end{aligned}$$

However, if we introduce a function g: $V^2 \rightarrow V^2$ to adjust the subarguments before shuffle is applied to each of them,

$$\begin{aligned}&g\ \langle v_1, v_2\rangle = \langle v'_1, v'_2\rangle\\ &\quad \text{where } v'_1\ (i) = v_1\ (i), && \text{if } i \leqslant (|v_1|/2)\\ &\qquad\qquad\quad = v_2(|v_2| - i), && \text{otherwise}\\ &\qquad\quad\ \ v'_2\ (i) = v_2\ (i), && \text{if } i > (|v_2|/2)\\ &\qquad\qquad\quad = v_1\ (|v_1| - i), && \text{otherwise}\end{aligned}$$

Then the function shuffle can also "behave" like a morphism in the sense that it reduces to two applications on adjusted sub-arguments:

$$(shuffle \circ cat)\ \langle v_1, v_2\rangle = (cat \circ (map\ shuffle) \circ g)\ \langle v_1, v_2\rangle$$

Formally, a function f: $[X, c_x] = \rightarrow [Y, c_y]$ is called a *premorphism* (see Figure 1c) if there exists a *preadjust function* $g{:}X^k \rightarrow X^k$ such that

$$(f \circ c_x)\ \langle x_1, \ldots, x_k\rangle = (c_y \circ (map\ f) \circ g)\ \langle x_1, \ldots, x_k\rangle$$

2.3. *Pseudomorphisms*

More generally, we say a function $f{:}[X, c_x] \rightarrow [Y, c_y]$ is a *pseudomorphism* (see Figure 1d) if there exists a preadjust function $g{:}X^k \rightarrow X^k$, and a postadjust function $h{:}\,Y^k \rightarrow Y^k$, such that

$$f \circ c_x = c_y \circ h \circ (map\ f) \circ g$$

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 I.

3. Divide and Combine Functions on Space Domains

Pseudomorphisms capture the notion of divide-and-conquer in that they reduce the application of a function to several applications of the same function. However, this only makes sense if the new applications are somehow made to smaller "pieces" of the input. In fact, pseudomorphisms do not constitute a complete model for divide-and-conquer before the following questions are answered:

- Where is the notion of divide?
- What is the relationship between divide and combine?
- How do the algebras in pseudomorphisms come into existence?

These questions are answered in this section.

3.1. Space Domains

Generally functions are mappings defined over structured data, which we model as a structured set called a *space*. Formally, a space s is a pair (U,R), where U is a *universe* of elements and R is a set (often singleton) of relations over U called its *structure*. The *size* of a space $s = (U,R)$ is denoted by $|s|$, and is defined as the cardinality of the universe U.

For example, a graph (V,E) is modeled as a space with $U = V$ and $R = \{E\}$. Similarly, a vector is modeled as a space with U as the set of its entries and the only element of R being a total ordering over the entries reflecting the vector structure.

A *space domain* S is a (usually infinite) set of spaces with certain common properties. For example, all graphs constitute the graph space domain, and all vectors constitute the vector domain.

Given two spaces $s_1 = (U_1, R_1)$ and $s_2 = (U_2, R_2)$ in the same domain S, we say s_1 is a *subspace* of s_2, denoted by $s_1 \subset s_2$, if $U_1 \subset U_2$ and for each relation $r_1 \in R_1$, there exists a corresponding relation $r_2 \in R_2$, such that $r_1 \subset r_2$.

3.2. Divide Functions

Note that for a pseudomorphism $f:[X,c_x] \rightarrow [Y,c_y]$, the operator c_x often captures the notion of "combining" elements of X into larger ones. For example, we showed that (*reduce* $\oplus$) is a morphism from $[V, cat]$ to $[U, \oplus]$, where *cat* takes two subvectors and creates their concatenation. However, in divide-and-conquer algorithms we are foremost interested in the *divide* function that essentially performs the inverse of functions like *cat*. More specifically, for an algebra $[X,c_x]$ we are interested in a divide function d_x such that

$$c_x \circ d_x = I$$

where I is the identity function.

Let S be a space domain, and $d{:}S \to S^k$ a total function. We say that d is a k-ary *divide function* over S if it is total and whenever for a space $s = (U,R)$,

$$d\ s = \langle s_1, \ldots, s_k \rangle, \text{ where } s_i = (U_i,\ R_i) \text{ for } i = 1 \text{ to } k$$

then

(*1*) $s_i \subset s$ (*subspace*)

(*2*) $\cup_{i=1}^{k} U_i = U$ (*complete*)

(*3*) $U_i \cap U_j = \theta$, for $i \neq j$ (*mutually exclusive*)

For example, the following d_{lr} is a divide function for the vector space domain that will divide a vector into two approximately equal subvectors:

$$\begin{aligned} d_{lr}\ v &= nil, && \text{if } (|v| = 1) \\ &= \langle v_1, v_2 \rangle, && \text{otherwise} \end{aligned}$$

$$\begin{aligned} \text{where } v_1\ (i) &= v(i), && \text{for } i = 1 \text{ to } |v|/2 \\ v_2(i) &= v(i - |v|/2), && \text{for } i = (|v|/2 + 1) \text{ to } |v| \end{aligned}$$

We define *nil* to be a special element in S^k and hence the mapping mapping d_{l_r} is indeed total.

3.3. Combine Functions

A k-ary *combine function* for a domain S is a total mapping $c{:}S^k \to S$. For any k spaces $(s_1, \ldots, s_k) \in S^k$, we have

$$c\ \langle s_1, \ldots, s_k \rangle = s,\ s_i \subset s, \text{ for } i = 1 \text{ to } k$$

The vector catenation operator *cat*, for example, is a combine function over the vector space domain:

$$\begin{aligned} &cat\ \langle v_1, v_2 \rangle = v \\ &\quad \text{where } v(i) = v_1(i), && \text{if } (i \leqslant |v_1|) \\ &\quad\quad = v_2\ (i - |v_1|), && \text{if } (|v_1| < i \leqslant (|v_1| + |v_2|)) \end{aligned}$$

Note that *cat* is a left inverse of the divide function d_{lr}. However, the left inverse of a divide function is generally not unique. For example, another left inverse of d_{lr} is

$$\begin{aligned} \textit{picky-cat}\ \langle v_1, v_2\rangle &= \textit{nil}, && \text{if } (|v_1| - |v_2|) > 1 \\ &= \textit{cat}\ \langle v_1, v_2\rangle, && \text{otherwise} \end{aligned}$$

Picky-cat is a subfunction of *cat* since it is consistent with *cat* for all the values which are not mapped to *nil.*

The subfunction relation in fact defines a partial order over the set C of all the left inverses of a divide function d. We define the *inverse* of divide function d to be $d^{-1} = min(C)$. It can be shown that $d_{lr}^{-1} = \textit{picky-cat}$.

3.4. Division Induced Algebras

Obviously if c is a combine function over space domain S then $[S,c]$ is an algebra, and we say that the algebra is *induced* by the divide function d if $c = d^{-1}$. Since the divide function is usually the starting point for a divide-and-conquer algorithm, it is convenient to describe such an algorithm as a pseudomorphism from $[X,d^{-1}]$ to $[Y,c_y]$, where the algebra $[X,d^{-1}]$ is induced by the divide function d.

4. Adjust Functions

Adjust functions reflect the degree of "interspace communication," and have important ramifications on the complexity of divide-and-conquer. In this section, we will further explore the nature of adjust functions. In particular, we will show how adjust functions can be decomposed further into two types of functions, one reflecting interspace communication, the other not.

Consider an adjust function $a{:}S^k \rightarrow S^k$, where

$$a\ \langle s_1, \ldots, s_k\rangle = \langle s'_1, \ldots, s'_k\rangle, \quad \text{where } s,\ s_i,\ s'_i \in S$$

Although there is a correspondence between the space s'_i and the space s_i, the value of space s_i in general depends on not only space s_i but also on all s_j for $j \neq i$. Therefore we cannot in general decompose a into k subfunctions that do the individual mappings independently.

On the other hand, we could consider an adjust function to be the composition of two functions *ref* and *loc*: The former is what we call the *reference function* that fetches for each space the values needed from other spaces, and the latter is what we call the *local function* which performs a mapping on each space based on its own value and the values fetched from other spaces. In other words, $a = loc \circ ref$.

Aside from increasing our understanding of the communications complexity, we will see that decomposing the adjust function in this way has the additional benefit of uncovering some repeating communications patterns in many algorithms, which we can use in a modular way.

4.1. Reference Functions

In general the reference function can be represented as

$$ref\,\langle s_1,\ldots,s_k\rangle = \langle \vec{s}_1,\ldots,\vec{s}_k\rangle$$

$$\text{where } \vec{s}_i = \langle \vec{s}_i.1,\ldots,\vec{s}_i.k\rangle,\quad s_i,\ \vec{s}_i.j \in S \text{ for } i,j = 1 \text{ to } k$$

That is, a reference function maps each space s_i to a compound space $\vec{s}_i$ consisting of k component spaces $\langle \vec{s}_i.1,\ldots,\vec{s}_i.k\rangle$. Each component space $\vec{s}_i.j$ stores the values of space s_j referenced by space s_i. Note that component space $\vec{s}_i.i$ is the space s_i itself, and therefore can be omitted from the specification.

For example, the following are three reference functions that are commonly used by functions defined over vectors (see Section 6). They are also illustrated in Figure 2. We will see in the next subsection that the reference functions contained in the adjust functions for scan and shuffle are (last-m 1) and mirr, respectively.

$$\begin{aligned} &corr\,\langle v_1, v_2\rangle = \langle \vec{v}_1\,, \vec{v}_2\,\rangle \\ &\quad \text{where } \vec{v}_1\,.2(i) = v_2(i), \quad \text{for } i = 1 \text{ to } |v_1| \\ &\qquad\quad v_2\,.1\,(i) = v_1\,(i), \quad \text{for } i = 1 \text{ to } |\vec{v}_2| \end{aligned}$$

$$\begin{aligned} &mirr\,\langle v_1, v_2\rangle = \langle \vec{v}_1\,, \vec{v}_2\,\rangle \\ &\quad \text{where } \vec{v}_1\,.2(i) = v_2(|v_2| - i), \quad \text{for } i = 1 \text{ to } |v_1| \\ &\qquad\quad v_2\,.1(i) = v_1\,(|v_1| - i), \quad \text{for } i = 1 \text{ to } |\vec{v}_2| \end{aligned}$$

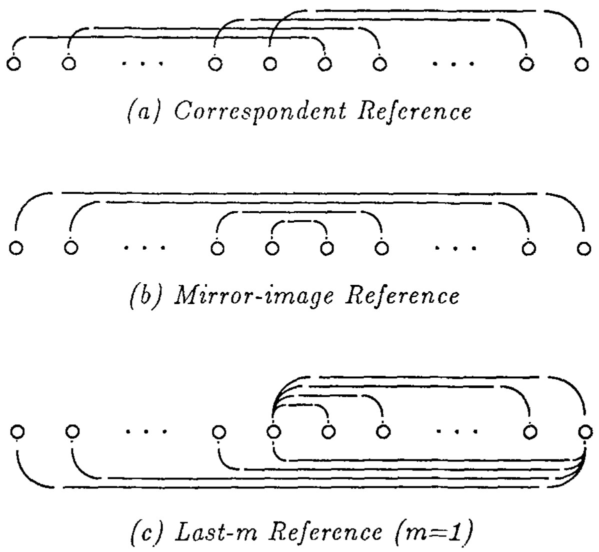

(a) Correspondent Reference
(b) Mirror-image Reference
(c) Last-m Reference (m=1)

Figure 2. Patterns of the commonly shared reference functions on vectors.

$$\begin{aligned}
&\textit{last-m } m\ \langle v_1, v_2\rangle = \langle \vec{v}_1, \vec{v}_2\rangle\\
&\quad\text{where } \vec{v}_1.2(i) = \textit{m-array}\\
&\qquad\qquad\text{where } \textit{m-array}\ (i) = v_2(|v_2| - i), \quad \text{for } i = 1 \text{ to } m\\
&\qquad\quad \vec{v}_2.1(i) = \textit{m-array}\\
&\qquad\quad\text{where } \textit{m-array}\ (i) = v_1(|v_1| - i), \quad \text{for } i = 1 \text{ to } m
\end{aligned}$$

4.2. Local Functions

A local function takes k "compound" spaces produced by a reference function and maps them to k spaces. The general form is thus

$$\textit{loc}\ \langle \vec{s}_1,\ldots,\vec{s}_k\rangle = \langle s'_1,\ldots,s'_k\rangle$$

Since the reference function has already performed the interspace references, the local function in fact consists of k subfunctions $\langle \textit{loc}_1,\ldots,\textit{loc}_k\rangle$, where $\textit{loc}_1$ $\vec{s}_i = s'_i$. Thus, instead of the above we will sometimes use the notation:

$$\textit{loc}\ \langle \vec{s}_1,\ldots,\vec{s}_k\rangle = \langle \textit{loc}_1,\ldots,\textit{loc}_k\rangle\ \langle \vec{s}_1,\ldots,\vec{s}_k\rangle = \langle \textit{loc}_i\ \vec{s}_1,\ldots,\textit{loc}_k\ \vec{s}_k\rangle$$

4.3. Examples

To ease the readibility of the following examples we define some auxiliary functions, beginning with *self*:

$$\textit{self}\ \vec{s}_i = \vec{s}_i.i$$

and for the case of $k = 2$, we also define a function *other*:

$$\begin{aligned}
\textit{other}\ \vec{s}_i &= \vec{s}_i.2, \quad \text{if } i = 1\\
&= \vec{s}_i.1, \quad \text{if } i = 2
\end{aligned}$$

Finally, we define the functional *entry-wise* which takes a binary operator defined over the entries of two vectors and returns a function which will perform the entry-wise operation to two vectors of equal length:

$$(\textit{entry-wise}\ \oplus)\ \langle v_1, v_2\rangle = v_3, \quad \text{where } v_3(i) = (i) \oplus v_2\ (i)$$

Now we can define the postadjust function in scan as

$$\begin{aligned}(h_{scan}\ \oplus) &= \langle loc_1, loc_2 \rangle \circ \textit{last-m}\\ \text{where } loc_1 &= self\\ loc_2 &= \text{entry-wise } \oplus\end{aligned}$$

and the preadjust function in shuffle as (observe that $(map\ f) = \langle f, \dots, f \rangle$):

$$\begin{aligned}&g_{shuffle} = (\text{map part-exch}) \circ mirr\\ &\quad \text{where } \textit{part-exch}\ \vec{v}_i = v'_i\\ &\qquad \text{where } v'_i(j) = (self\ \vec{v}_i)\ (j), \quad \text{if } (i = 1) \text{ and } (j \leqslant |v_i|/2)\\ &\qquad\qquad = (other\ \vec{v}_i)\ (j), \quad \text{if } (i = 1) \text{ and } (j > |v_i|/2)\\ &\qquad\qquad = (other\ \vec{v}_i)\ (j), \quad \text{if } (i = 2) \text{ and } (j \leqslant |v_i|/2)\\ &\qquad\qquad = (self\ \vec{v}_i)\ (j), \quad \text{otherwise}\end{aligned}$$

4.4. *Orthogonality of Divide/Combine and Adjust Functions*

Implicit in the definition of divide and combine functions is the fact that they do not alter the values of the elements in the space. We further define a *valid* divide or combine function as one that does not *depend* on the values either. Thus a valid divide or combine function depends only on and affects only the *structure* of a space. An example of nonvalid divide function is to partition a vector into two parts with an equal number of nonzero elements. All of the divide and combine functions considered in this paper are valid.

The concept of adjust function naturally suggests that it preserves the structures of the spaces. Now let a be an adjust function over space domain S, and

$$a\ \langle s_1, \dots, s_k \rangle = \langle s'_1, \dots, s'_k \rangle \quad \text{where } s_i = (U_i, R_i),\ s'_i = (U'_i, R_i)$$

We say that a is *valid* if the relations in R_i are always isomorphic to the relations R'_i for $i =$ to k. In other words, a valid adjust function depends on and affects only the *universe* of the spaces, and does not depend on or affect the structure of the space.

The orthogonal nature of valid divide and combine functions on one hand, and valid adjust functions on the other, should now be clear.

We should point out that the so-called divide and combine functions in some well-known algorithms are not valid by our definitions. For example, in quicksort [Aho et al. 1974]. The routine that partitions the input vector into two smaller vectors is not a valid divide function, and the function that merges two vectors in merge-sort is not a valid combine function. Interestingly, in Section 6 we will see that the merge routine in bitonic-sort, in fact, is itself a divide-and-conquer algorithm.

5. Recursive Computation of Divide-and-Conquer

Recall the relation defined in Section 2 for a pseudomorphism f:

$$f \circ c_x = c_y \circ h \circ (map\ f) \circ g$$

from which we can derive

$$f \circ c_x \circ d = c_y \circ h \circ (map\ f) \circ g \circ d$$

but since $c_x \circ d = \mathrm{I}$, we have

$$f = c_y \circ h \circ (map\ f) \circ g \circ d$$

This equation literally dictates the form of the functional, but there is one last detail needing discussion: Most divide-and-conquer algorithms reach a "bottom" level in the division process, which in our model occurs when a divide function returns *nil*. For example, d_{lr} returns *nil* for unit-length vectors. This behavior, in fact, defines the termination property of a divide-and-conquer algorithm.

At this bottom level in the division process a divide-and-conquer algorithm typically invokes a *basis function* f_b on the atomic elements. Thus we include f_b in our list of constituent functions, leading us to the following definition of the functional *DC*:

$$\begin{aligned} &DC(d, c, g, h, f_b) = f_{dc} \\ &\quad \text{where } f_{dc}\ x = f_b\ x, && \text{if } (d\ x = nil) \\ &\qquad\qquad = (c \circ h \circ (map\ d_{dc}) \circ g \circ d)\ x, && \text{otherwise} \end{aligned}$$

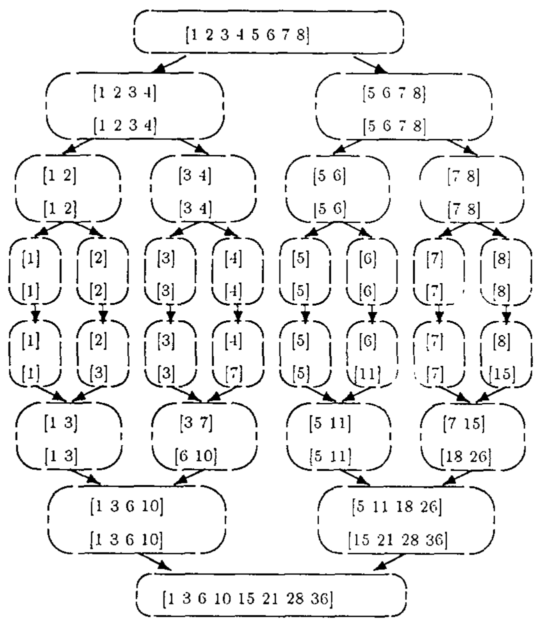
Let f: $[X,d^{-1}] \rightarrow [Y,c]$ be a pseudomorphism based on the algebras induced by the divide function d, with g as preadjust function and h as the postadjust functions, and let f_b be the base function of f with respect to d. Then we have $f = DC(d,c,g,h,f_b) = f_{dc}$. The function f_{dc} returned by the higher order function *DC* is called a *divide-and-conquer function*, or *divacon* for short. The functions,d,c,g,h and f_b, are called the constituents of the divacon.

The recursive computation of a divacon f_{dc} applied to a space x can be depicted by the *divacon graph DG* (f_{dc}, x), which consists of two phases.

- The *divide phase* corresponds to the repeated application of the divide function d and preadjust function g, until the atomic spaces are generated at the "leaves."

- The *combine phase* corresponds to the repeated application of the postadjust function h and combine function c until the "root" is reached again.

The base function f_b is what ties the two phases together. Graphically one can draw this as two trees connected at their leaves, as shown in Figure 3 for *DG* ((*scan* +), [1 2 3 4 5 6 7 8]), where the names of the component functions applied at each level are included for clarity. The *height* of a divacon graph is the height of one of these trees, which are symmetric.



level	applied function	contained reference
0'	$d = d_{lr}$	–
1	$g = I$	–
1'	$d = d_{lr}$	–
2	$g = I$	–
2'	$d = d_{lr}$	–
3	$g = I$	–
3'	$f_b = I$	–
-3'	$h = h_{scan}$	*last_m* 1
-3	$c = d_{lr}^{-1}$	–
-2'	$h = h_{scan}$	*last_m* 1
-2	$c = d_{lr}^{-1}$	–
-1'	$h = h_{scan}$	*last_m* 1
-1	$c = d_{lr}^{-1}$	–
-0'	–	–

Figure 3. The example divacon graph *DG* ((*scan* +), [1 2 3 4 5 6 7 8]).

6. Application Examples

To illustrate the expressive power, the concise form, the functional style, and the sharing of constituent functions of divacons, we present ten examples in this section. The efficiency of the algorithms was considered but was not the main criterion in selecting the examples.

The commonly used divide, combine, reference, and local functions such as d_{lr}, d_{lr}^{-1}, *corr, mirr, last-m, self, other*, and *entry-wise* are assumed to be global. The symbol $\oplus$ always stands for an associative binary operator, and the symbol I stands for the identity function. The length of a vector is conveniently assumed to be a power of two.

Example 1. *Reduce* function of a vector under an operator $\oplus$. It is a pure morphism without adjust functions.

reduce $\oplus$ = *DC* (d_{lr}, $\oplus$, I, I, *vector-to value*)
  where *vector-to-value* $[x] = x$

Example 2. *Reduce* computed by a nonbalanced divide function. The divide function used here corresponds to the built-in *car* and *cdr* functions offered in Lisp-like languages. It is easy to transform other example functions defined over vectors to pseudomorphisms with the unbalanced divide function d_{dt}.

```
$reduce \oplus = DC\ (d_{ht}, \oplus, I, I, vector\text{-}to\text{-}value)$
    where $d_{ht}\ v = head, tail$
        where $head\ (i) = v(1)$, for $i = 1$
              $tail\ (i) = v(i+1)$, for $i = 1$ to $(|v| - 1)$
```

Example 3. *Scan* over a vector under operator $\oplus$. This is a premorphism with identity function I as the base function.

$$scan \oplus = DC\ (d_{lr}, d_{lr}^{-1}, I, \langle self, (entry\text{-}wise \oplus)\rangle \circ (last\text{-}m\ 1), I)$$

Example 4. *Shuffle* of a vector is a premorphism with the base function $f_b = I$.

```
$shuffle = DC\ (d_{lr}, d_{li}^{-1}, (map\ part\text{-}exch) \circ mirr, I, I)$
    where $part\text{-}exch$ is as defined in Section 4.2
```

Example 5. *Broadcast* the value in v (1) to all entries of a vector v (br). It is presented below as a postmorphism; however, it can be defined as a premorphism as well.

$$br = DC\ (d_{lr}, d_{lr}^{-1}, I, \langle \text{self, other}\rangle \oplus \text{corr}, I)$$

Example 6. *Fibonacci* sequence (fib). Let $v(i) = i$ for $i = 1$ to N, then the function *fib* $v = v'$ where $v'(i) = fib(i)$. We compute this function by transforming the natural number sequence to a vector of pairs, performing divide-and-conquer over the pair vector, and finally transforming the pair vector back into numbers. The pair (1,1) corresponds to the coefficients in the definition $f(n) = f(n-1) + f(n-2)$, and the pair (2,1) corresponds to the coefficients in the equation $f(n) = 2\ (f-2) + f(n-3)$. The method used in this example actually can be easily generalized to solve any linear difference equations.

```
$fib = pair\text{-}to\text{-}number \circ fib_{dc} \circ number\text{-}to\text{-}pair$
    where $number\text{-}to\text{-}pair\ v = v_{pair}$
            where $v_{pair}\ (i) = (0,1)$,   if $(i \leqslant 2)$
                              $= (1,1)$,   if ($i$ is $even$)
                              $= (d,1)$,   if ($i$ is $odd$)
        $fib_{dc} = DC\ (d_{lr}, d_{lr}^{-1}, I, h_{fib}, I)$
          where $h_{fib} = \langle self, mult\rangle \circ (last\text{-}m\ 2)$
            where $mult\ \vec{v} = v$
              where $v(i) = ((c_1 * a_1 + c_2 * b_1), (c_1 * a_2 + c_2 * b_2))$
                where $(c_1, c_2) = (self\ \vec{v})\ (i)$
                      $(a_1, a_2) = ((other\ \vec{v})\ (i))\ (1)$
            $(b_1, b_2) = ((other\ \vec{v})\ (i))\ (2)$
    $pair\text{-}to\text{-}number\ v_{pair} = v$
      where $v(i) = c_{i2}$
        where $(c_{il}, c_{i2}) = v_{pair}\ (i)$
```

Example 7. *Bitonic sort.* Bitonic sort is a postmorphism with a premorphism as part of its postadjust function. Also, observe that the two nested morphisms share the same local function.

```
$bitonic\text{-}sort = DC\ (d_{l_r}, d_{lr}^{-1}, I, (loc \circ mirr), I)$
  where $loc = merge \circ (map\ comp\text{-}and\text{-}exch)$
    where $merge =\ DC\ (d_v, d_v^{-1}, (loc_{merge} \circ corr), I, I)$
      where $loc_{merge} = (map\ comp\text{-}and\text{-}exch)$

$where\ comp\text{-}and\text{-}exch\ (\vec{v}) = v_i$
  where $v_i(j) = (self\ \vec{v}_i)\ (j)$,      if $(i = 1)$ and $(self\ \vec{v}_i)\ (j) \leqslant (other\ v_i)\ (j)$
               $= (other\ \vec{v}_i)\ (j)$,    if $(i = 1)$ and $(self\ \vec{v}_i)\ (j) > (other\ \vec{v})\ (j)$
               $= (other\ \vec{v}_i)\ (j)$,    if $(i = 2)$ and $(self\ \vec{v}_i)\ (j) \leqslant (other\ \vec{v}_i)\ (j)$,
               $= (self\ \vec{v}_i)\ (j)$,     otherwise
```

Example 8. *Polynomial evaluation* (poly). $f(x) = a_0 * x^o + \ldots + a_n * x^n$. Let vector $X = [x \ldots x]$ with length $(n + 1)$. $A = [a_0 \ldots a_n]$, then poly can be defined in terms of divacons *reduce* and scan.

$$poly\ (A.X) = ((reduce +) \circ (entry\text{-}wise *))\ \langle A,\ scan * X\rangle$$

Example 9. *Integer multiplication* (im). Let x and y be two integers and let the function *nums-to-bins* (x,y) return two binary vectors X and Y representing x and y, respectively, then

```
$im = im_{dc} \circ nums\text{-}to\text{-}bins$
  where $im_{dc} = DC\ (d_{pv}, ++, I, h_{im}, im_b)$
    where $d_{pv}\ (X, Y) = $ nil,                    $if\ (|X| = 1\ and\ |Y| = 1)$
                   $= (X_1, Y_1), (X_1, Y_2), (X_2, Y_1), (X_2, Y_2)$,
                   otherwise
          where $(X_1, X_2) = d_{lr}(X)$
                $(Y_1, Y_2) = d_{lr}(Y)$
      $++\ (a, b, c, d) = a + b + c + d$
      $h_{im} = \langle loc_a, loc_b, loc_c, I\rangle \circ ref_{im}$
          where $ref_{im}\ (a, b, c, d) = (a, (b,c)\ c, d)$
                $loc_a\ a = a * 2^{|a|}$
                $loc_b\ (b, c) = (b + c) * 2^{|b|/2}$
                $loc_c\ c = 0$
      $im_b\ ([x], [y]) = x * y)$
```

Observe that multiplying an integer by a power of two can be achieved easily by the binary number that represents the integer.

Example 10. *Expression tree evaluation* (eval). Let t be a binary tree, where each internal node is a binary operator, and each leaf node contains some ground value. Then the evaluation is a pure-morphism. The divide function used in the morphism is not balanced (unless t is), so there is no obvious efficient parallel implementation. The problem of how to compute graph problems by balanced morphisms is beyond the scope of this paper. We present this example to show that morphisms can express problems in domains other than arrays.

```
eval = DC (b_t, apply, I, I, node-to-value)
    where b_t t = nil,                 if t is a tree of one node
                = (root, left, right),  otherwise
            where root is the root of t
                  left is the left subtree of t
                  right is the right subtree of t
        apply (op, v_1, v_2) = op (v_1, v_2)
        node-to-value node =  value of the node
```

7. Divide-and-Conquer on Parallel Computers

In this section we identify the grain of parallelism implied by divacons, and provide guidelines but not details of how divacons may be mapped to parallel machines.

For a divacon f_{dc}, an obvious approach to parallelism is to map the nodes in the divacon graph DG (f_{dc}, x) to processors in a parallel computer. The disadvantage of this approach, however, is that the computation of spaces at upper levels of the divacon graph become a bottleneck since upper level spaces have larger sizes.

For a better alternative, let us introduce the concept of *distributed space*. First recall that an m-ary *relation* r over a set U is a set of m-tuples of elements from U; a *subrelation* of r is a subset of r; a subrelation of r induced by an element $u \in U$, denoted by r (u), is the subset of r consisting of all tuples in r in which u is an element [Dornhoff and Hohn 1978]. With these notions, we can use an unstructured set s' to represent a space $s = (U, R)$:

$$s' = \{(u, \{r(u) \mid r \in R\}) \mid u \in U\}$$

We call the set s' the *distributed form* of the space s. The elements in a distributed space are called its *points*. A point has the form $(u, \{r(u)\})$, where u is an element of U and $\{r(u)\}$ consists of subrelations of the relations in R.

The alternative mapping strategy that we propose is to treat the points of a space x as the grains of parallelism. To show how f_{dc} x can be computed in parallel under this scheme, we only need to show how the application of each constituent function can be computed in parallel since the computation of a divacon reduces to the recursive applications of the constituent functions on disjoint spaces. We discuss this issue for each of the constituent functions below.

Recall that a valid divide function never alters the values of the points, and the mapping that it does is independent of the values of other points. Therefore the (valid) divide function can be computed in parallel by all the points (processors) without communication. By a similar argument, (valid) combine functions can be computed by all the points in the spaces of the parallel.

An adjust function can be analyzed in terms of its own constituent functions. The reference function obviously maps directly to the communication between processors, and will be further discussed in the next section. We know that no local functions contain interspace communication. With the notion of distributed space, we further classify them into *weakly local* and *strongly local* functions according to whether they contain interpoint communication or not. For example, the local functions in the bitonic sort divacon is weakly local, and the local functions in other examples are strongly local. The strongly local function can be computed locally by each point (processor) for all points in the space in parallel without communication, and the weakly local functions should be computed in turn by a divacon.

The base function f_b is a subfunction of f_{dc} but only defined trivially over the atomic spaces. The recursive application of divide and preadjust functions will eventually map each point to an atomic space (assuming atomic spaces have size one), and so f_b can be computed in parallel.

Observe that a node in a divacon graph never connects to a node other than its direct son or father. This characteristic of divacon graphs is equivalent to the *normal property* of parallel algorithms in Ullman [1984]. It implies that as long as the appropriate information is passed from level to level, when a computation enters a new level in the divacon graph we can reuse the processors from the previous level. Therefore, the number of processors used by a divacon at a particular level is exactly equal to the number of points in all the spaces at that level.

We have identified points in the spaces as the grain of parallelism in divacons. But we have purposely left the mapping between the set of points to the set of computers unspecified. The reason is that the performance of divacons on parallel computers is largely independent of the particular topology of a space and the particular topology of the parallel machines, as will be explained in the next section.

8. Time and Processor Complexities of Divacons

We show in this section how the time and processor complexity of divacons can be derived easily from those of its constituent functions. The concept of balance in divide-and-conquer is formally defined, and the impact of balance on time complexity is discussed.

8.1. Communication on Parallel Computers

The time complexity of parallel algorithms depends greatly on the cost of communication among the processors. The time used by a particular phase of communication is in turn attributed to two major factors: the locality of the communication and the pattern of the communication, where locality refers to whether or not the communicating processors have direct physical channels and the pattern refers to how evenly the messages are coming in and out over different processors in the parallel machine.

We observe that on small diameter machines, such as the hypercube and butterfly [Ullman 1984], the communication pattern is a much more important factor than the communication locality. With the same communication pattern, the communication time used by a phase of communication differs, at most, a logarithmic factor with and without locality [Valiant 1981, Ullman 1984]. With the same locality, different communication patterns may take time as little as constant or as much as linear (to the number of processors) with different communication pattern. We are therefore motivated to a parallel computer model where the communication pattern is the only factor affecting the communication time. The dissimilarity between our model and real machines is intentional, since we want to concentrate on more decisive aspects of complexity analysis.

We consider a *communication* among the processors of a parallel computer to be a binary sender-receiver over the set of processors; therefore it is a directed graph, called the *communication graph*. The *fan-in* and *fan-out* of a processor during a communication are, respectively, the indegree and outdegree of the node corresponding to the processor in the communication graph. The fan-in and fan-out of the communication are, respectively, the maximum indegree and outdegree of the communication graph.

Let T_{comm} denote the time required by a communication over the parallel computer, $\textit{fan-in}_{comm}$ and $\textit{fan-out}_{comm}$ denote the fan-in and fan-out of the communication; we assume

$$T_{comm} = \mathrm{O} \ (\textit{fan-in}_{comm} + \textit{fan-out}_{comm})$$

8.2. Time Complexity

The time used to compute a divacon $f = DC\,(d, c, g, h, f_b)$ is obviously the sum of the time used at all levels of the divacon graph $DG\,(f, x)$. Let $T\,(\textit{function}, n)$ denote the time used to compute a function on the space of size n, H denote the height of the divacon graph, n_i denote the size of the spaces at level i, and A denote the size of atomic spaces; then

$$T\,(\mathrm{f}, n) \leqslant \left(\sum_{i=0}^{H} T\,(d, n_i) + T\,(c, n_i) + T\,(g, n_i) + T\,(h, n_i)\right) + T\,(f_b, A)$$

Assuming the atomic spaces have bounded size, then the last term takes O(1) time; also since $n_i \leqslant n$ for all i, we have

$$T(f, n) = O(H * (T(d, n) + T(c, n) + T(g, n) + T(h, n)))$$

The above equation tells us that the complexity of a divacon can be easily derived in terms of the complexity of its component functions and the height of the divacon graph.

The two terms $T(d, n)$ and $T(c, n)$ reflect the time used by divide and combine functions. Assuming the functions d and c are valid, then $T(d, n) = T(c, n) = O(1)$ since they can be computed locally.

The other two terms $T(g, n)$ and $T(h, n)$ reflect the time used by preadjust and postadjust functions, respectively. Since an adjust function is the composition of local and reference functions, the time it uses is equal to the sum of the time used by its local and reference function; therefore we have

$$T(g, n) = T(ref_g, n) + T(loc_g, n)$$

$$T(h, n) = T(ref_h, n) + T(loc_h, n)$$

The term $T(ref, n)$ reflects the communication time used by the adjust function. It can be decided by the fan-in and fan-out of the communication corresponding to the reference function. The *corr* and *mirr* reference functions correspond to the communications with constant fan-in and fan-out and therefore $T(corr, n) = T(mirr, n) = O(1)$. The reference function *last-m* corresponds to communications with $O(n)$ fan-out, but it can be implemented by the divacon *br* in Section 6 with $O(log(n))$ time.

The term $T(loc, n)$ reflects the time taken by the local function after the interspace reference. Clearly, for strongly local functions, $T(loc, n) = O(1)$, since there is not even interpoint communication. Weakly local functions should be computed in turn by divacon, and analyzed recursively.

The height H of a divacon graph is the multiplying factor of the divacon complexity as shown above, and it depends on only the divide function. To classify the divide functions, we say that a divide function over a domain S is *balanced* if there exists a constant *division factor* $M > 1$ such that for any space $s \in S$, and $d(s) = s_1, \ldots, s_k$ the following relation holds:

$$(|s|/max(|s_1|, \ldots, |s_k|)) \geqslant M$$

By this definition, we can see that the divide function d_{lr} is balanced while d_{ht} is not. The tree divide function used in Example 10 is not balanced either, unless the tree itself is balanced.

A divacon with balanced divide function is called a *balanced divacon*. Balanced

f	scan	shuffle	br	reduce	sort	im	fib	poly
$T(f,n)$	$O(log^2 n)$	$O(logn)$	$O(logn)$	$O(logn)$	$O(log^2 n)$	$O(logn)$	$O(log^2 n)$	$O(log^2)$
$P(f,n)$	n	n	n	n	n	$O(n^2)$	n	$O(n)$

Figure 4. Complexities of example applications.

divacons can be shown to have *balanced divacon graphs* in the sense that both the two parts of the divacon graph are balanced, and therefore have $O(log(n))$ heights, where n is the size of the space. When the balanced divacons are mapped to parallel computers, most processors will be able to participate in the computation at all levels of the divacon graph, which leads to better efficiency and speedup. Indeed, for a balanced divacon $f = \text{DC}\ (d, c, g, h, f_b)$, the time complexity becomes

$$T(f, n) = O(\log(n) * (T(d, n) + T(c, n) + T(g, n) + T(h, n)))$$

It should be evident that nested balanced divacons yield polylogarithmic time performance; therefore there is a relation between the class of problems that can be computed by nested balanced divacons and the class *NC* [Cook 1985].

With the above discussion, we can very easily derive the time complexity of all example divacons in Section 6. In Figure 4, we have listed some of the results.

Although we have disregarded the locality as a factor of the communication cost, our approach in fact has offered a convenient handle to pursue the locality. This is because the communication inside divacons is reflected largely by the reference functions, and we can achieve the locality of communication by mapping from points of spaces to processors of machines that will make the reference local. For example, on hypercube machines, the mappings decided by a binary coding and gray coding of vectors will make the *corr* and *mirr* references local, respectively.

8.3. Processor Complexity of Divacons and Time-Processor Trade-off

Let $P(f, n)$ denote the number of processors required to compute a divacon $f = (d, c, g, h, f_b)$ on a space of size n. From the discussion in the last section, we know that $P(f_{dc}, n)$ depends on only the number of processors used at the level of a divacon graph that has the maximum number of total points. And we can derive it by induction easily.

Observe that for a k-ary divide function d with division factor m, if $d(s) = (s_1, \ldots, s_k)$ then the following relation holds:

$$|s_i| \leqslant \frac{|s|}{m}, \quad \text{for } i = 1 \text{ to } k$$

Under our mapping scheme, this leads to the following recurrence:

$$P(f, n) = k * p(f, n/m)$$

Obviously the base case for the recurrence is $P(f, 1) = 1$. The solution to the recurrence can be shown to be $O(n^{\log_m k})$.

The above tells us that the processor complexity of a divacon depends solely on the divide function of the divacon. And if the arity and the division factor of the divide function are known then the processor complexity is totally decided.

For a divide function d with arity k and division factor m, we define the *expanding exponential* α and the *expanding factor* β to be, respectively,

$$\alpha_d = log_m k$$

$$\beta_d = k/m$$

The constant α is called the expanding exponential since we have

$$P(f_{d_c}, n) = n^\alpha$$

The constant β is called the expanding factor since it reflects the ratio of processors used at two adjacent levels of the divacon graph.

When the α and β of a divide function are equal to one, we say the divide function is *space conservative*. Divacons with conservative divide functions are said to be space conservative divacons. Clearly, $P(f, n) = O(n)$ if f is space conservative.

We can see that the integer multiplication divacon is not space conservative, and it uses a divide function with $\alpha = 2$, and therefore $P(im, n) = n^2$; all the other divacon examples are space conservative, and therefore $P(f, n) = n$. The nonconservative divide functions often appear at the divacons defined over Cartesian product domains. The block division in matrix multiplication [Aho et al. 1974] is another example.

When the size of a space is large, there may not be enough processors in a real machine. For space conservative divacons, we can statically map more than one point to one processor. For nonconservative divacons, the static control is not very effective. However, we can dynamically force one processor to represent γ times more points than it did at the previous level, where γ is a constant called the *compression factor*. It is easy to see that when γ is equal to the expanding factor, the number of processors used by the divacon will become stable at all levels. It can also be shown that in terms of the efficiency of processors, neither the static nor the dynamic scheme will decrease the performance of the divacon.

9. Concluding Remarks

We have presented an algebraic model of divide-and-conquer algorithms, and showed how such algorithms can be specified, implemented, and analyzed in terms of the constituent functions.

The computational power of divacons is Turing-equivalent, although its structure more naturally fits certain classes of algorithms than others. The expressive power of divacons can, however, be further enhanced by relaxing some of the restrictions. For example, if we allow points to be shared by a number of spaces, then the very operationally oriented *pointer-jumping* algorithms [Huang 1985] can be naturally defined as divacons. Also, if the arity of the algebras is allowed to be variable, then some parallel graph algorithms, such as the *connected component* algorithms [Ullman 1984], can be modeled by divacons.

The significance of divide-and-conquer, even for sequential computation, has been pointed out by Aho et al. [1974]. They also pointed out the importance of balanced division, as did Berger and Bokhari [1987] and others. Several other researchers have emphasized its use in parallel computing. Preparata and Viullemin [1981] have informally described the divide-and-conquer paradigm and its implementation on cube-connected cycles. They also described two classes of divide-and-conquer: "descend" and "ascend," which are what we call pre- and postmorphisms. Smith [1987] has concentrated on the practical side of developing divide-and-conquer algorithms.

Most of the divacons in this paper are the functional abstraction of the algorithms that can be found in the literature, for example, in Aho et al. [1974], Stone [1981], Ullman [1984], and Ladner and Fischer [1980]. Many other divide-and-conquer algorithms based on balanced algebra, such as FFT and block matrix multiplication [Aho et al. 1974, Ullman 1984], can also be treated naturally by our approach.

We are presently developing a parallel programming language where divacons are the only form of recursion, and which we plan to implement as outlined in this paper.

Acknowledgments

We thank Steve Anderson for many helpful discussions, and Alan Perlis for his comments on the earlier versions of this paper. We are also grateful to Chris Hatchell for his help in preparing this manuscript.

References

Aho, A. V., Hopcroft, J. E., and Ullman, J. D. 1974. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Mass.

Berger, M. J., and Bokhari, S. H. 1987. A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Transactions on Computers*, C-36(5):570–580.

Cook, S. A. 1985. A taxonomy of problems with fast parallel algorithms. *Information and Control*, 64:2–22.

Dornhoff, L. L., and Hohn, F. E. 1978. *Applied Mordern Algebra.* Macmillan, New York.

Huang, M. -D. 1985. Solving graph problems with optimal speedup on mesh-of-tree networks. In *Proc. Twenty-sixth Annual Symposium on Foundations of Computer Science*, IEEE Computer Society Press, pp. 232–240.

Jamieson, L. H., Gannon, D. B., and Douglass, R. J. 1987. *The Characteristics of Parallel Algorithms.* The MIT Press, Cambridge, Mass.

Ladner, R. E., and Fischer, M. J. 1980. Parallel prefix computation. *Journal of the ACM*, 27(4):831–838.

Preparata, F. P., and Vuillemin, J. 1981. The cube-connected cycles: A versatile network for parallel computation. *Communications of ACM*, 8(5):300–309.

Smith, D. R. 1987. Applications of a strategy for designing divide-and-conquer algorithms. *Science of Computer Programming*, (8):213–229.

Stone, H. S. 1971. Parallel processing with the perfect shuffle. *IEEE Transactions on Computers*, C-20(2):153–160.

Ullman, J. D. 1984. *Computational Aspect of VLSI.* Computer Science Press.

Valiant, L. G. 1981. Universal schemes for parallel communication. In *Proc. Thirteenth Annual ACM Symposium on the Theory of Computing*, pp. 263–277.