9-12-1994

The Nondeterministic Divide

Arthur Charlesworth
University of Richmond, acharles@richmond.edu

Follow this and additional works at: http://scholarship.richmond.edu/mathcs-reports
Part of the Programming Languages and Compilers Commons

Recommended Citation
The Nondeterministic Divide

Arthur Charlesworth
Department of Mathematics and Computer Science
University of Richmond
Richmond, Virginia 23173
email: charlesl@mathcs.urich.edu

September 12, 1994

TR-94-07
The Nondeterministic Divide

ARTHUR CHARLESWORTH
University of Richmond

The nondeterministic divide partitions a vector into two non-empty slices by allowing the point of division to be chosen nondeterministically; i.e., by the underlying implementation. Support for high-level divide-and-conquer programming provided by the nondeterministic divide is investigated. A diva algorithm is a recursive divide-and-conquer sequential algorithm on one or more vectors of the same range, whose division point for a new pair of recursive calls is chosen nondeterministically before any computation is performed and whose recursive calls are made immediately after the choice of division point; also, access to vector parameters is only permitted during activations in which these parameters have unit length. The notion of a diva algorithm is formulated as a diva call, a restricted call on a sequential procedure. Numerous applications of diva calls are given. Diva calls are shown to support both reasoning based on strong induction and hierarchical reasoning based on binding free variables. Asymptotically optimal strategies are described for translating a diva call into code for a variety of parallel computers. Diva calls separate logical correctness concerns from implementation concerns. Numerous advantages of diva calls over the use of general reduction operators are identified.

Categories and Subject Descriptors: D.1.3 [Programming Techniques]: Concurrent Programming — parallel programming; D.3.2 [Programming Languages]: Language Classifications — concurrent, distributed, and parallel languages; D.3.3 [Programming Languages]: Language Constructs and Features — recursion

General Terms: Languages

Additional Key Words and Phrases: Nondeterminism, divide-and-conquer, associativity, reduction, ring, torus, hypercube, interconnection topology

This research was supported by National Aeronautics and Space Administration grant NAG-1-774, by Jet Propulsion Laboratory grant 95722, and by funding from the University of Richmond Faculty Research Committee. Portions of this research were conducted while the author was on sabbatical at the Institute for Parallel Computation of the University of Virginia during the 1989-90 academic year. Author's current address: Department of Mathematics and Computer Science, University of Richmond, Richmond, VA, 23173.
1 INTRODUCTION

This article is motivated by the search for high-level abstractions that support efficiently implementable parallel programming. In particular, the article investigates the feasibility of divide-and-conquer algorithms when the implementation is allowed to choose the point of division. Such a nondeterministic divide is shown to have several major advantages:

- **Portability across target computers.** Programs written using the nondeterministic divide can be ported without change to different target parallel computers that, for reasons of efficiency, should use different strategies for choosing division points.

- **Flexibility among target computer processors.** Different processors of the same parallel computer can use different strategies for choosing division points in carrying out the collective stage of a single divide-and-conquer computation. This is desirable even for highly regular, homogeneous parallel computers, such as a ring of identical processors.

- **Flexibility within a single target computer processor.** The grain size of a computation can be enlarged by a proper choice of division points, so that an appropriate stage of local computation precedes the stage of collective computation.

- **Support for reasoning based on strong induction.** Informal proofs of correctness based on strong induction are simplified when the details of how division points will be chosen are abstracted away.

- **Support for hierarchical reasoning.** In mathematics, understanding the effect of using $\sum_{j=1}^{n} x_j$ in a computation naturally precedes understanding the effect of using $\sum_{i=1}^{n} \sum_{j=1}^{n} x_{ij}$. Using the terminology of logic, $a$ in the first expression can be viewed as a free variable that is replaced by a bound variable in the second expression. The nondeterministic divide facilitates the same kind of hierarchical reasoning based on binding free variables.

When programmed using the nondeterministic divide, vector applications that would take linear time on unprocessors can be performed in logarithmic time on hypercubes and in square-root time on toruses.

The hierarchical reasoning technique is applicable to a general class of computations, such as computing for each component in a vector the first component to the right having a larger value than the given component; due to the flexibility of the nondeterministic divide, the quadratic number of calculations can be performed in linear time on a ring, torus, or hypercube having a linear number of processors, which is asymptotically optimal.
The article investigates the general concept of programming parallel computers using the nondeterministic divide, rather than proposing a construct to support the resultant style of programming in a particular language. Pseudocode notation is used that can be understood without elaborate syntactic and semantic definition. The design of nondeterministic divide constructs for an existing programming language depends on the characteristics of the particular language and is beyond the scope of the article. For instance, our most recent research in this direction involves extending C++ and is naturally leading to different results from earlier research we conducted on extending Ada [Charlesworth 1993].

The article is organized as follows. Background is provided in Section 2, applications are surveyed in Section 3, support for reasoning is described in Section 4, implementation results appear in Section 5, comparisons with general reduction operators are made in Section 6, and conclusions are made in Section 7.

The term slice means contiguous subvector (or contiguous subsequence) and, for a vector v, v[i:j] denotes the slice from v[i] through v[j]. The term node is used for a vertex in a tree diagram, rather than a unit of a parallel computer. Associativity of addition and multiplication is assumed, so the article assumes all integer and real numbers are accurately represented within the target computer; the usual arithmetic operations on such numbers are assumed to require constant time. (The standard assumptions in the preceding sentence provide a useful level of abstraction, although they cannot be satisfied on a physical computer.)

2 BACKGROUND

Effective computing requires adequate support from both hardware and software. The architecture of computers, once restricted to having a single processor for the execution of instructions, is evolving in diverse directions. Recent shared memory multiple instruction stream, multiple data stream (MIMD) computers have up to a few hundred processors. Current distributed memory MIMD computers have from the low hundreds to a few thousand processors, while the number of fine-grained processors in current single instruction stream, multiple data stream (SIMD) computers is in the tens of thousands. The hardware support for high-speed computing is expected to be increasingly in the form of heterogeneous computing systems [NSF 1992], systems that provide access to a variety of supercomputers and to workstation clusters.

A paramount question is how such diverse computers, and computers resulting from additional evolution, should be programmed. Indeed, the lack of adequate software support for parallel computers has been termed the "parallel software crisis" [Pancake 1991]. Creating a useful parallel programming concept is a balancing act among four primary goals:

- providing a level of abstraction with simple rules,
- supporting portability across parallel computers,
- providing sufficient expressibility for useful applications, and
- restricting the programmer to expressing high-level algorithms that permit generation of efficient code.

Approaches to parallel programming that are specific to particular computers tend to trade off portability for efficiency. On the other hand, other notable approaches, such as Unity [Chandy and Misra 1988] and GAMMA [Banatre and Le Metayer 1993], tend to trade off a degree of assurance of efficiency for expressibility. We are seeking a compromise, as indeed are many of the developers of these two different approaches.

Among the most efficiently implementable operations involving the participation of multiple processes are the associative operations, since partial computations of associative operations can be combined using a variety of efficient techniques, such as read/modify/write on MIMD shared memory computers, processor trees on MIMD hypercubes, and shifts or pointer doubling on SIMD computers. For this reason, support for computing reductions of vectors using standard associative functions, such as +, *, and, or, max, and min, is commonly provided within languages for parallel computing (and support for reductions was provided much earlier by APL [Iverson 1962]). Support for using less trivial programmer-defined associative functions in computing reductions of vectors has also been provided, in the form of general reduction operators. Such operators became available even in some early languages for parallel computing, such as iPSC/2 Fortran and C [Intel 1989] and the innovative and less conventional language Paralation Lisp [Sabot 1988]. But general reduction operators have a number of abstraction deficiencies, identified later in the article.

A leading strategy for programming both sequential and parallel computers is divide-and-conquer, which can generally reduce the conceptual complexity of a problem. On parallel computers there can be a significant performance advantage to the divide-and-conquer approach, when different parts of the solution can be computed on different processors.

In this article a restricted form of divide-and-conquer sequential algorithm is shown to have the major implementation advantages of general reduction operators. Yet, with suitable syntactic support, such an algorithm is shown to be free of the undesirable features of general reduction operators and thereby to enhance program correctness and abstraction.

Briefly, here is historical background for the article. The development of the nondeterministic divide followed directly from research on the multiway rendezvous [Charlesworth 1987]. The multiway rendezvous is a generalization of the two-way rendezvous in which two or more processes synchronize and communicate values, a block of code is executed, and the processes resume their independent executions. To facilitate efficient (such as log time) execution on a
parallel computer, the activities of the block of code should be specified by not-
tation at a high-level of abstraction. (The need for such high-level notation was
illustrated by using pseudocode, rather than a loop, for finding a sum in Example 3.3 of [Charlesworth 1987].) The nondeterministic divide was developed to
support the high-level notation of the diva procedure [Charlesworth 1989; 1990;
1993], which can be used to describe activities during a multiway rendezvous
block. The present article treats diva procedures apart from their support for
the multiway rendezvous. This is feasible because a call on a diva procedure
from within sequential code is similar to an implicit multiway rendezvous of
processes used to implement the call.

Independent of and contemporaneous with the development of the nonde-
terministic divide has been the development of high-level language constructs
in which programmers specify an algorithm for selecting the division point for
divide-and-conquer; results with the nondeterministic divide show what can be
accomplished without specifying such an algorithm at the programmer level and
are thus complementary to this research. Contemporaneous with all of this re-
search has been the development of strategies that support distributed shared
memory, DSM.

The goal of DSM is to free programmers from the responsibility of specifying
algorithms for distributing data across memories. DSM research is analogous
to the research reported here in the sense that both seek to shift details from
the programmer level to the implementation level. This article is orthogonal to
DSM research: when the article discusses implementation possibilities for the
nondeterministic divide, data is assumed to be distributed conveniently across
memories.

3 APPLICATIONS

Turning to the foreigner who wanted to be his new aide, the king
presented the challenge: “Devise a strategy that my workers can use
to find the weight of this worn-out wooden scepter, since I plan to
replace it with a silver one having exactly the same heft. Built into
niches of this castle are a variety of sensitive balancing scales. This
scepter greatly exceeds the capacity of any single scale.”

Immediately the foreigner responded: “I was given a similar task
in my previous job and the strategy I developed there will port here
without modification: Break the scepter into two pieces and keep
breaking wood into two pieces until each piece can be weighed on a
scale. Then add all the weights obtained. There’s no need for me
to specify where each piece of wood must be broken; such decisions
can be made by your workers based on the various capacities of the
scales. Clearly this weighing strategy is correct regardless of how
such decisions are made.”
diva procedure P (A: in Vector1; B: out Vector2;
   First, Last: in Integer;
   Data: in Type1;
   Ans: out Type2) is
   L, R: Type2;  -- Results for left and right slices
   Middle: Integer;
begin
   if First = Last then
      P1 (A[First], B[First], First, Last, Data, Ans);
   else
      Middle := Divide (First, Last);
      P (A, B, First, Middle, Data, L);
      P (A, B, Middle + 1, Last, Data, R);
      P2 (L, R, Ans);
   end if;
end P;

Figure 1: Pseudocode for a Diva Procedure

The nondeterministic divide partitions a vector into two nonempty slices by allowing the point of division to be chosen nondeterministically; that is, the point of division is not specified by the programmer but is chosen by the underlying implementation. A diva algorithm is a recursive divide-and-conquer sequential algorithm on one or more vectors of the same range, whose division point for a new pair of recursive calls is chosen nondeterministically before any computation is performed and whose recursive calls are made immediately after the choice of division point; also, access to vector parameters is only permitted during activations in which these parameters have unit length.

3.1 Pseudocode

The article uses the following pseudocode. Transmission by constant value and transmission by result, as both are defined in [Pratt 1984], are denoted in and out, respectively, and each parameter has exactly one of these two modes. Diva procedures are written using pseudocode as illustrated in Figure 1. One or more vector parameters appear at the beginning of the parameter list, followed by the integer parameters First and Last of mode in, followed by zero or more scalar parameters. Except perhaps for First and Last, each name for a scalar parameter of mode in appears in the same position in the two recursive calls as it does in the parameter list for P. None of the procedures P, P1, and P2 access nonlocal variables. Each formal parameter of P1 has the same mode as the parameter in the corresponding position of P's parameter list. The three formal parameters of P2 have mode in, in, and out, respectively. Calls on P1 and P2 can be written implicitly, as inline expansions. Locals other than L, R,
and \textit{Middle} may be declared. When there is no scalar parameter of mode \textit{out},
the declarations of \text{L} and \text{R} and the call on \text{P2} are omitted. The function \textit{Divide}
is defined by the underlying implementation so that

\[
\text{Middle} := \text{Divide} (\text{First}, \text{Last});
\]

guarantees $\text{First} \leq \text{Middle} < \text{Last}$.

The pseudocode for a nonrecursive call on \text{P} has the form

\[
P (X, Y, 1, N, B, C);
\]

and all vectors used in the call have subscripts ranging from 1 to \text{N}, permitting
the same division point for all vectors during any given recursive activation of
\text{P}. In addition, the pseudocode for a \text{diva} call can contain hyphen notation to
indicate multiple unhyphened calls. The pseudocode

\[
P (X, Y, 1, N, D[-], E[-]);
\]

has the same effect as performing \text{N} \textit{independent} unhyphened calls on \text{P}, where \text{N}
is the number of components of \text{X}, \text{Y}, \text{D}, and \text{E}. The final two parameters in the
first of the \text{N} unhyphened calls are \text{D}[1], and \text{E}[1], the final two parameters in
the second call are \text{D}[2], and \text{E}[2], etc. Hyphen notation is not used in a \text{diva}
call if the net effect depends on the order of execution of the \text{N} unhyphened calls.
If all \text{E}'s receive the same value, a scalar can be used as this actual parameter.
The pseudocode also uses hyphen notation to represent a subscript value: the
final two parameters in the first of the \text{N} calls given by

\[
P (X, Y, 1, N, [-], E[-]);
\]

are 1 and \text{E}[1], the final two parameters in the second call are 2 and \text{E}[2], etc.

\textbf{3.2 Examples}

\textit{Example 3.1: Compute the sum of the components of a vector.}
\textit{Solution:} Make the call

\[
\text{FindSum} (X, 1, N, \text{Total});
\]

where \text{X} is the vector and \text{FindSum} is declared in Figure 2.

\textit{Example 3.2: Find a component value of one vector corresponding to the maxi­
mum of another vector.}
\textit{Solution:} Make the call

\[
\text{FindMaxCorr} (X, Y, 1, N, \text{Answer});
\]

where \text{X} is a floating point valued vector whose maximum is sought, \text{Y} is a cor­
responding integer valued vector, and \text{FindMaxCorr} is defined in Figure 14 in
the Appendix, The desired answer is the value of \text{Answer}.\text{Corr}.

\textit{Example 3.3: Find the number of ascents in a sequence, where an ascent [Gra­
ham et al. 1989] occurs whenever one term has a smaller value than the next.}
\textit{Solution:} Make the call
diva procedure FindSum (A: in Vector; First, Last: in Integer;
Sum: out Integer) is
  -- Sum is assigned the sum of the component values of A[First:Last].
L, R: Integer; -- Results for left and right slices
Middle: Integer;
begin
  if First = Last then
    Sum := A[First];
  else
    Middle := Divide (First, Last);
    FindSum (A, First, Middle, L);
    FindSum (A, Middle + 1, Last, R);
    Sum := L + R;
  end if;
end FindSum;

Figure 2: A Diva Procedure to Find a Sum

FindNumber (S, 1, N, Answer);

where S is the sequence, FindNumber is defined in Figure 15 in the Appendix,
and Answer is a record consisting of integer fields Count, First, and Last. The
final two fields in the type of Answer are used by the diva procedure to preserve
information about the overlap between left and right slices.

Some additional applications are listed in Figure 3. A key to programming
most of these examples is to preserve needed information about the overlap be­
tween left and right slices. The particular overlap information that is needed
usually becomes clear to the programmer as the false branch of the if is being
written. Applications listed in Figure 3 that don’t mention finding a location
within a vector can be extended to finding such a location. For instance, in
addition to computing the maximum appreciation possible from the sequence
of values of a commercial stock, compute the locations of the first such pair of
values. The next example illustrates both the use of nontrivial overlap informa­

Example 3.4: An experiment has been performed, with each of two sensors col­
lecting a sequence of readings at regular time intervals. Find the location of the
longest winning streak of the first sensor’s readings over the second’s, and the
first such location in case of a tie.
Solution: Make the call

FindMaxWinningStreak (Sensor1, Sensor2, 1, N, Answer);
- Computing the maximum appreciation possible from the sequence of values of a commercial stock (i.e., the greatest increase from one term of a sequence to a later term of the sequence).

- Computing which term of a sequence has a value nearest a given value and the first such term in case of a tie.

- Computing the length of the longest plateau in a nondecreasing sequence, where a "plateau" is a slice whose components have equal values [Gries 1981].

- Computing the maximum sum among the nonempty slices of a sequence of positive and negative integers [Bentley 1986].

- Computing the length of the longest identical corresponding slice of two sequences.

- Computing the length of the longest ascending slice of a sequence, where an "ascending slice" is a slice whose component values are increasing.

- Computing the length of the longest ascending slice of a sequence such that the corresponding terms of another sequence are also ascending.

- Assigning a particular constant to each component of a vector.

- Assigning to a vector a linear combination of other vectors.

- Computing the number of runs in a sequence, where a "run" is an ascending slice not contained in a larger ascending slice [Knuth 1981].

- Computing the number of peaks in a sequence, where a "peak" is a term that is greater than both its predecessor and successor.

- Computing the position of the first term of a sequence of positive integers that is greater than a given value (or returning -1 if there is no such term).

- Computing the largest increase among the pairs of adjacent terms of a sequence.

- Computing the first record, in a sequence of records, whose fields have the most number of matches with the corresponding fields of a key.

Figure 3: Some Additional Applications of Diva Procedures
diva procedure FindSumPowers (A: in Vector; First, Last: in Integer; Power: in Real; Sum: out Real) is
-- Sum is assigned the sum of the powers
L, R: Real; -- Results for left and right slices
Middle: Integer;
begin
  if First = Last then
    Sum := A[First]-Power; -- Exponentiation is denoted `-`
  else
    Middle := Divide (First, Last);
    FindSumPowers (A, First, Middle, Power, L);
    FindSumPowers (A, Middle + 1, Last, Power, R);
    Sum := L + R;
  end if;
end FindSumPowers;

Figure 4: A Diva Procedure to Find the Sum of Powers

where the readings from the two sensors are in vectors Sensor1 and Sensor2
and FindMaxWinningStreak is defined in Figure 16 in the Appendix. The de­sired location starts at position Answer.Pos and has length Answer.Max.

The final example in this section prepares us for the next section.

Example 3.5: Assign \( \sum_{j=1}^{n} x_j^i \) to Answer.
Solution: Make the call

```
FindSumPowers (X, 1, N, A, Answer);
```

where FindSumPowers is given in Figure 4.

3.3 Specifying Cooperative Actions

Example 3.6: Assign \( \sum_{i=1}^{n} \sum_{j=1}^{n} x_j^i \) to Answer.
Solution: Make the calls

```
FindSumPowers (X, 1, N, Y[-], Z[-]);
FindSum (Z, 1, N, Answer);
```

where FindSumPowers is given in Figure 4. The first call calculates the \( n^2 \)
powers and \( n \) sums \( \sum_{j=1}^{n} x_j^i \), where \( 1 \leq i \leq n \), and assigns these values to the components of Z. The second call calculates the final sum and assigns it to Answer.
This approach to specifying cooperative actions can be applied to a variety
of problems. Section 5 describes how a translator can ensure the \( n^2 \) results of a
cooperative action are computed on a ring, torus, or hypercube in linear time.

**Example 3.7:** For each component in a vector, find the location of the first
component to the right that has a greater value, if there is such a location [Nicol
et al. 1991].

*Solution:* Make the call

\[
\text{FindRight} (X, 1, N, [-], X[-], \text{Answer}[-]);
\]

where \( X \) is the given vector and \text{FindRight} is defined in Figure 17 in the Ap­
pendix. For each \( I \), \text{Answer}[I].Loc is the desired location, if the Boolean
\text{Answer}[I].Found holds.

Several of the applications in Figure 3 can be extended to cooperative actions
in a natural way. For instance, a diva procedure for computing the maximum
appreciation possible from a sequence of values of a commercial stock can be
used to determine for each sequence value the maximum appreciation possible if
the stock is bought at the time represented by the sequence value. This section
looks at additional examples.

**Example 3.8:** Sort a vector.

*Solution:* Make the calls

\[
\text{FindCount} (X, 1, N, [-], X[-], \text{Position}[-]);
\text{Assign} (Y, 1, N, \text{Position}[-], X[-]);
\]

where the resulting vector \( Y \) is the sorted version of \( X \) and \text{FindCount} and \text{Assign}
are defined in Figures 18 and 19 in the Appendix. Note that the diva call on
\text{Assign} is capable of assigning an arbitrary permutation of \( X \) to \( Y \).

Quicksort [Hoare 1962] is a divide-and-conquer strategy that does not have a
straightforward treatment using diva procedures, since the division of a vector
by quicksort occurs only after some initial processing of the vector.

A simple approach to programming the \( n \) body problem is to use a coopera­
tive action to calculate the acceleration on each of \( n \) astronomical bodies due to
the masses and locations of the other bodies. Cooperative actions can also solve
problems in statistics. A single point can badly distort a linear regression; as
a result it is sometimes desirable to calculate the effect of omitting each point
(independently) from the computation of the linear regression. Similarly, it is
sometimes desirable to calculate which single piece of data, if omitted from a
set of data, would cause the resulting variance to become the smallest [Bowen
1968].

In the analysis of sorting algorithms, an inversion [Knuth 1973] occurs when­
ever a pair of terms is out of the desired order, regardless of how far apart the
Use strong induction on the length $n$ of the slice $A[\text{First:Last}]$.

**case one:** $n = 1$. Then First = Last and, since the sum of the components of $A$ is the single component of $A$, the result follows.

**case two:** Otherwise. Since the length of both slices $A[\text{First:Middle}]$ and $A[\text{Middle+1:Last}]$ is less than that of $A[\text{First:Last}]$, the induction hypothesis implies that, after the two recursive calls, $L$ is assigned the sum of the components of $A[\text{First:Middle}]$ and $R$ is assigned the sum of the components of $A[\text{Middle+1:Last}]$. Thus Sum is assigned the sum of the components of $A$.

Figure 5: Informal proof of correctness of FindSum

terms of the pair occur in the sequence. The number of inversions can be calculated using one diva call (containing hyphen notation) to find, for each term of the sequence, the number of inversion pairs the term is the second member of, and a second diva call to sum the resulting vector of counts. A similar approach can be used to find the "inversion table" [Knuth 1973], when the $n$ sequence values are a permutation of 1 to $n$; the $j^{th}$ component of the inversion table (a vector) is the number of inversion pairs whose second member is $j$.

4 SUPPORT FOR REASONING

4.1 Use of Strong Induction

The natural way to verify a diva procedure is to use strong induction on the length of the vector slices. This approach is illustrated in Figure 5 by the informal proof of correctness of the diva procedure in Example 3.1.

Reasoning about the correctness of most other diva procedures is more complex than Figure 5. But this increased complexity is solely due to increased complexity in parameters and the true branch and false branch of the if, rather than to additional complexity of the diva concept itself. This is illustrated in Figure 20 in the Appendix, which shows part of an informal proof of the diva procedure FindMaxWinningStreak.

Since the nondeterministic divide is fully compatible with such use of strong induction, the inability of assuming anything about the choice of division points does not make these proofs more complicated. By suppressing unnecessary details concerning the choice of division points, the nondeterministic divide simplifies such proofs.
4.2 Use of Hierarchical Reasoning

Mathematics gains much of its power from the hierarchical proof technique that allows us to bind an unbound variable. For instance, when we bind the free variable $a$ in $\sum_{i=1}^{n} x_{i}$ to obtain $\sum_{i=1}^{n} x_{j}$, facts proven about $\sum_{j=1}^{n} x_{j}$ carry over. The technique is similar to (and related to) the $\forall$-introduction rule of formal logic [Kleene 1971], which is used to deduce $\forall x A(x)$ from $A(x)$.

The same powerful hierarchical proof technique can be used with diva calls to greatly simplify both reasoning about and developing applications such as those in Section 3.3. Suppose we have proven that the call

$$P(X, 1, N, B, C);$$

assigns to $C$ the intended result corresponding to the first four actual parameters, regardless of the values of the first four actual parameters. Since the multiple calls denoted by

$$P(X, 1, N, D[-], E[-]);$$

are independent, the intended result corresponding to the parameters $X, 1, N,$ and $D[1]$ is assigned to $E[1]$, the intended result corresponding to the parameters $X, 1, N,$ and $D[2]$ is assigned to $E[2]$, etc. Moreover this holds when each component value of $D$ is equal to its subscript; i.e. when the notation $[-]$ is used instead of $D[-]$ in the call.

The proof technique can be illustrated using Example 3.7. We want to conclude that the call

$$\text{FindRight}(X, 1, N, [-], X[-], \text{Answer}[-]); \quad (1)$$

finds, for each component of $X$, the location of the first component to the right that has a greater value (if there is such a location). Strong induction is used first, to prove that the simpler call

$$\text{FindRight}(X, 1, N, L, B, C); \quad (2)$$

guarantees: $C.\text{Found}$ is true if and only if $X$ has a component to the right of $X[L]$ greater than $B$ and (if $C.\text{Found}$ is true then) $C.\text{Loc}$ is the location of the first such component. We can then conclude that call (1) guarantees for each $I$ from $1$ to $N$: $\text{Answer}[I].\text{Found}$ is true if and only if $X$ has a component to the right of $X[I]$ greater than $X[I]$ and (if $\text{Answer}[I].\text{Found}$ is true) $\text{Answer}[I].\text{Loc}$ is the location of the first such component.

The hierarchical technique simplifies the development of solutions to problems (i.e., writing diva procedures) in addition to supporting simple correctness arguments. When the diva procedure in question was being written to solve the full problem, the much simpler call (2) was the only call the programmer needed to consider. The difference between diva calls (1) and (2) is that certain free variables in (2) are bound in (1). Variable $L$ in (2) is a free variable in the sense that its value is not constrained to lie within any range. In moving from (2) to
(1), the programmer binds the free variable L to lie in the range from 1 to N and the free variables B and C to be the X and Answer components corresponding to the fourth actual parameter in one of the N unhyphenated calls represented by (1).

In their appropriate domain of problems, diva procedures simplify problem solving for software engineers like definite integrals simplify problem solving in a problem domain confronted by physical engineers. Consider how a physical engineer computes the volume of the solid between $x = 0$ and $x = 1$ formed by rotating the positive function $y = f(x)$ around the x axis. First an expression for the volume of a much simpler solid, a representative perpendicular disk, is found: $\pi (f(x))^2 dx$. Then the engineer binds the range of the free variable $x$ to yield the definite integral $\int_{x=0}^{x=1} \pi (f(x))^2 dx$. A definite integral is also like a diva call in the sense that it is at a higher-level of abstraction than implementation details, such as whether the integral will be calculated using the fundamental theorem of calculus, or a sequential numerical algorithm such as Simpson's rule, or a parallel numerical algorithm such as Monte Carlo.

5 IMPLEMENTATION

5.1 Uniprocessor

Implementing diva calls on uniprocessors is considered in this section; the next section considers their implementation on parallel computers. Since each processor in a parallel computer can perform part of the computation of a diva call independently, the observations of this section are used in the next.

Consider an arbitrary call on a diva procedure P, where the initial length of each of the vectors is n. Note that the number of executions of the true branch of the if of P is independent of the choice of division points during the call, and the same holds for the false branch: for the number of executions of the true branch is clearly $n$ (one execution for each element of the vector range), and a straightforward strong induction argument shows that the number of executions of the false branch is $n - 1$. To see the latter, assume the result holds when the length of the vectors is less than $n$ and that the division point for the original vectors results in initial and final slices of lengths a and b, respectively. Then the number of executions of the false branch is

$$(a - 1) + (b - 1) + 1 = (a + b) - 1 = n - 1.$$  

One option for implementing a diva call is to choose the division point of a vector slice at the middle of the slice (or, if the slice has even length, one component before the middle) and to make the recursive calls specified in the diva procedure; we call this the Middle Option. Another option is to implement

\footnote{This fact is intuitively obvious, since breaking a horizontal wooden scepter into $n$ pieces requires $n - 1$ vertical division points, regardless of the location of the division points.}
a diva call using a loop, by choosing the division point of a vector slice just before the beginning (or end) of the slice; we shall refer to this option as the Begin Option (or End Option, respectively). The loop implementation of the Begin (respectively, End) option accesses vector components from right to left (respectively, left to right) to simulate successive returns from recursive calls. The Begin and End options are different from tail recursion, although all can be implemented using a loop, since tail recursion requires that a recursive call occur last in a subprogram. Since the Begin Option is very similar to the End Option, this section considers just the Middle and End options. Both Begin and End options play a role in Section 5.2.2.

Since the Middle Option requires overhead for recursion, or the simulation of recursion, and the End Option requires only the overhead of a loop, there are many diva procedures P for which the time required for the End Option, averaged over all possible input values, is less than that required for the Middle Option.

In fact, there are many diva procedures for which the End Option is more time efficient than the Middle Option regardless of the input values to the procedure. The following theorem helps to illustrate this fact. In the statement of the theorem, the term conquer block refers to the body of P2 in Figure 1, when the call on P2 is expanded inline, as in most examples of pseudocode in the article.

THEOREM 1. Let P be a diva procedure satisfying the following condition: the maximum time for executing a logical path in the conquer block of P is less than the sum of the minimum time for executing a logical path in the conquer block and the minimum overhead required for two recursive calls on P (beyond that required for a loop). Then choosing division points for the vectors of P based on the End Option and using a loop to implement this option produces more time efficient code than any option requiring recursion.

PROOF. See Figure 21 in the Appendix.

With one exception, all diva procedures whose pseudocode appears in the article clearly satisfy the condition of the theorem. This is because the difference between the maximum and minimum time for executing a logical path in the conquer block of the diva procedure presented is no more than the time for executing a single addition, except for Example 3.4. For many implementations of recursion, the condition of the theorem is likely satisfied by the diva procedure FindMaxWinningStreak of Example 3.4 as well. The difference between the maximum and minimum time for executing a logical path in the conquer block of this diva procedure is is no more than the time for executing eight addition and/or Boolean and/or arithmetic comparison operations, plus the time for branching due to the execution of one if then else.

We now describe a diva procedure that, under reasonable assumptions, does
procedure P (A: in Vector1; B: out Vector2; First, Last: in Integer;
    Data: in Type1; Ans: out Type2) is
    L, R: Type2;
    Middle: Integer;
    Len: Integer;
begin
    Len := Last - First + 1;
    if Len <= NumIterations then
        P1 (A[First], B[First], First, First, Data, Ans);
        for I in First+ 1 .. Last loop
            P1 (A[I], B[I], I, I, Data, R);
            P2 (Ans, R, Ans);
        end loop;
    else
        Middle := (First + Last) / 2;
        P (A, B, First, Middle, Data, L);
        P (A, B, Middle + 1, Last, Data, R);
        P2 (L, R, Ans);
    end if;
end P;

Figure 6: Translation of Diva Procedure for Uniprocessor

not satisfy the condition of Theorem 1. The diva procedure performs a merge
sort of a vector A, and its first three formal parameters are A, First, and Last.
The fourth and final formal parameter is a record Ans for holding the answer.
Being a record, Ans is technically a scalar parameter; however a field of this
record is a vector B and other fields store information about the range of the
vector. The call P2(L, R, Ans) merges the sorted vectors L.B and R.B into
the sorted vector Ans.B so the maximum time for executing the conquer block
can be much greater than the minimum time. Assuming a reasonably efficient
implementation of parameter passing and recursion, for this diva procedure the
maximum time for executing a logical path in the conquer block is greater than
the minimum time plus the overhead for two recursive calls, so the condition in
the theorem is not satisfied.

One strategy for a sequential implementation of a diva call on P is to rewrite
P as illustrated by the pseudocode in Figure 6, where P1 and P2 are as in Figure
1. The compiler determines a value of NumIterations, which is 1 if the
Middle Option is chosen and the length, N, of the vectors in the nonrecursive
call on P if the End Option is chosen. (Of course, code can be generated requiring
no test on NumIterations for these two extremes.)

THEOREM 2. Let P be a diva procedure whose procedures P1 and P2 require
time bounded by a constant. An unhyphenated call on P can be executed on a uniprocessor in linear time in the length of the vectors in the call.

**PROOF.** Let \textbf{NumIterations} in Figure 6 be the length of the vectors in the call. QED

It follows that all applications mentioned in Section 3.2 (including those in Figure 3) can be performed in linear time on a uniprocessor, except perhaps for Example 3.5. (As mentioned at the end of the Introduction, the article is assuming integer and real numbers are accurately represented within the target computer and that the usual arithmetic operations take constant time. There is no guarantee that the exponentiation used in Example 3.5 can be performed in constant time.)

A diva call containing hyphen notation on a diva procedure satisfying the hypothesis of Theorem 2 can be executed on a uniprocessor in time that is quadratic in the length of the vectors in the call, since the call can be implemented as a loop of unhyphenated calls.

Theorem 1 is conservative not only by using maximums and minimums but also by ignoring compiler optimizations. When the End Option is used, the compiler can determine that the true branch of the \textbf{if} is the code that will be executed to compute the value of R. This additional information can be very useful.

As an illustration, consider Figure 7(a), which contains the true branch of the \textbf{if} of Figure 6 when the diva procedure \textbf{P} is \textbf{FindNumber} of Example 3.3, with \textbf{P1} and \textbf{P2} given inline expansions. Effective data flow analysis permits the elimination of variables whose values are not used, the use of a variable in an addition when the variable clearly has value zero, the replacement of \textbf{R.First} and \textbf{R.Last} by \textbf{A[I]}, and the elimination of code such as

```plaintext
else
  Ans.Count := Ans.Count;
```

Such transformations replace the sequential loop by

```plaintext
Ans.Count := 0;
Ans.First := A[First];
Ans.Last := A[Last];
for I in First + 1 .. Last loop
  if Ans.Last < A[I] then
    Ans.Count := Ans.Count + 1;
  end if;
  Ans.Last := A[I];
end loop;
```

and effective data flow analysis across loop iterations then yields Figure 7(b). A comparison of times required for the Middle and End options should ideally be made after such optimizations have been performed.
Ans.Count := 0;
Ans.First := A[First];
Ans.Last := A[Last];
for I in First + 1 .. Last loop
  R.Count := 0;
  R.First := A[I];
  R.Last := A[I];
  if Ans.Last < R.First then
    Ans.Count := Ans.Count + R.Count + 1;
  else
    Ans.Count := Ans.Count + R.Count;
  end if;
  Ans.First := Ans.First;
  Ans.Last := R.Last;
end loop;

Figure 7(a) Before Data Flow Analysis

Ans.Count := 0;
Ans.First := A[First];
for I in First + 1 .. Last loop
  if A[I - 1] < A[I] then
    Ans.Count := Ans.Count + 1;
  end if;
end loop;
Ans.Last := A[Last];

Figure 7(b) After Data Flow Analysis
A diva procedure for which the End Option is more efficient for shorter vectors and the Middle Option is more efficient for longer vectors is the merge sort described earlier in the section, whose conquer block is a merge and whose End Option is equivalent to the straight insertion sort. It would seem difficult for a compiler to determine analytically for a diva procedure such as merge sort an optimum value of `NumIterations` strictly between 1 and `N`. A compiler might, however, use a heuristic, possibly obtained as the result of an analytical study, based on characteristics of the conquer block.

We do not claim that data flow analysis for optimizing away inefficiencies in the implementation of diva calls for uniprocessors is always straightforward. In fact, in some cases, interaction between the translation system and the programmer may be desirable to achieve good efficiency. A recommended strategy for a compile-time decision among such options is beyond the scope of the article. For the purpose of the article, it is sufficient to observe that the nondeterministic divide provides flexibility to the compiler in generating efficient code and, as pointed out above, there are natural diva procedures for which a straightforward compiler could eliminate unnecessary overhead due to recursion.

5.2 Parallel Computers

This section illustrates some of the flexibility provided by the nondeterministic divide in the context of parallel computers.

The parallel computers considered are assumed to have distributed memory, since low-level distributive memory primitives can be simulated on shared memory systems. When possible the implementation algorithms are designed to use values that can feasibly be local to a processor. Numerous research projects are developing strategies to provide the illusion of shared memory in a distributed memory system. This article is orthogonal to those efforts. Thus, we assume vector components are placed on the processors of a parallel computer as we wish, without specifying what communication if any is necessary to achieve such a distribution.

The focus in the section is on the work of processors in support of a diva call. Two restrictions are made temporarily. First: each processor is responsible for just a single component of a vector or vectors. Second: the length of the vectors is even, or the square of an even number, or a power of two. Both restrictions can be removed by scaling the number of processors in the parallel computer, as explained in Section 5.3.

The kinds of parallel computers considered are a unidirectional ring, a bidirectional ring, a torus, a hypercube, and a SIMD computer having a hypercube interconnection network. There is no attempt at an exhaustive treatment. The key point to keep in mind is that the programmer who writes diva procedures, in a suitable programming language, doesn't need to think about the implementation details described here.

This section uses slightly different notation than the rest of the article. It
is customary to view processor numbers in a parallel computer as starting at 0 since this leads to simpler implementation descriptions for hypercubes and certain other parallel computers. On the other hand, the vectors used in a diva call have ranges that begin at 1, since this leads to a simpler level of abstraction: both simpler calling statements and simpler proofs of correctness. Thus when the section says a vector \( v \) is distributed so \( v[i] \) is located at processor \( p_i \), the vector \( v \) is related to a vector \( V \) in a diva call by the correspondence: \( v[0] = V[1] \), \( v[1] = V[2] \), etc. The rest of the section mentions just \( v \), rather than \( V \), so there should be no confusion. Whenever the section specifies that a processor makes a call on procedure \( P_i \), the call should use the single component of the vector or vectors the processor is responsible for, the single subscript the processor is responsible for (corresponding to both First and Last in the call), in addition to any other necessary in and out scalar parameters needed by the diva procedure.

5.2.1 Unidirectional Ring

As in the implementation for a uniprocessor, each vector component must be used in a call on \( P_1 \) to obtain a value that can be used in calls on \( P_2 \). For a variety of parallel computers, well-known and straightforward techniques are available for calculating the reduction of a commutative, associative function. But diva calls need not be commutative: parameters must be passed to \( P_2 \) in the proper order so the information from the left side of the vector precedes the information from the right side.

This point is illustrated by a unidirectional ring \( R \) of \( n \) processors, \( p_0, \ldots, p_{n-1} \), where the direction of communication permits information to be passed from \( p_i \) to \( p_{i-1} \) (and from \( p_0 \) to \( p_{n-1} \)). Let \( v \) be a vector distributed so \( v[i] \) is located at processor \( p_i \). Since addition is commutative, the sum of the vector components can be obtained by \( p_i \) by initializing a local sum to \( v[i] \) and generating a single round of \( n - 1 \) communications, with each processor adding each new number received into its local sum. This strategy fails for most diva procedures. For instance, consider applying the strategy to Example 3.3, to compute the number of ascents in a sequence. The strategy would cause processor \( p_{n-2} \) to find the number of ascents in the sequence \( v[n-2], v[n-1], v[0], \ldots, v[n-3] \), which is not the correct sequence.

Here is a correct strategy for the unidirectional ring \( R \): For \( 0 < i < n - 1 \), processor \( p_i \) views \( v \) as being divided into left and right slices so the left slice is \( v[0:i-1] \) and the right slice is \( v[i:n-1] \). (For \( i = 0 \) the left slice is \( v[0:0] \) and the right slice is \( v[1:n-1] \).) Assuming proper calls on \( P_1 \), each processor uses a loop (and 0 or more calls on \( P_2 \)) to calculate the result for the right slice before using another loop (and 0 or more calls on \( P_2 \)) to calculate the result for the left slice. This is feasible since vector values for the right slice will arrive, in sequence, prior to vector values for the left slice. The results for the left and right slices are combined with a final call on \( P_2 \). The strategy uses \( n - 1 \) stages of communication and computation for each processor; this is optimal, since
$n - 1$ is the minimum number of communications that ensure a given processor receives a value from each other processor.

This strategy is strongly supported by the flexibility of the nondeterministic divide. Only two processors ($p_0$ and $p_1$) make the same choice for the initial division point, used in the final call on $P2$ made by the processor.

Decomposition trees illustrate the strategy. The decomposition trees for a 16-component vector for processors $p_4$ and $p_5$ are in Figure 8. The tree for processor $p_4$ illustrates that the division point used for the original vector produces the slices $v[0:3]$ and $v[4:15]$, the division point used for the vector $v[0:3]$ produces the slices $v[0:2]$ and $v[3:3]$, etc. The order that the processor learns about the partial result for slices of length greater than one is indicated by the numbers to the right of nodes in the decomposition tree.

The implementation of a diva call containing hyphen notation on $R$ follows easily from the implementation described at the end of the next section.

### 5.2.2 Bidirectional Ring

The notation $v[i]$ and $p_i$ in the preceding section is used in this section as well. But here $R$ is a bidirectional ring, with an even number $n$ of processors, for which simultaneous two-way communications between adjacent processors of $R$ are more efficient than two separate one-way communications between the processors (such as might hold if the communication permitted was via remote
procedure calls). The strategy presented here implements a diva call on R so that each processor participates in \( n/2 \) stages of communication and computation. This is optimal, since \( n/2 \) is the minimum number of two-way communications that ensure a given processor receives a value from each other processor. The same number of stages is shown to suffice for the cooperative action of a diva call containing hyphen notation.

For a vector of length \( k \), let \( B(k) \) and \( E(k) \) denote the decomposition trees for the Begin and End options, respectively, as defined in Section 5.1. Number the leaves of a decomposition tree from left to right, starting with 0. The decomposition tree \( T_i \) for processor \( p_i \) of \( R \) is defined using \( B \)'s and \( E \)'s as follows, where \( m = n/2 \) and \( j = (i + m) mod(n) \). Make leaf \( i \) of \( T_i \) be the first leaf of \( E(m) \) and leaf \( j \) of \( T_i \) be the first leaf of \( B(m) \).

But, as necessary, let \( E(m) \) and \( B(m) \) wrap around from right to left, as now explained. For \( B(m) \) the wrap around rule means that whenever the above definition causes the first leaf of \( B(m) \) to be placed after leaf \( m \) of \( T_i \) (so there is not room for the rest of \( B(m) \) to fit as a subtree of \( T_i \)), instead of using the first leaf of \( B(m) \) use the first leaf of \( B(n - j) \), and make the first leaf of \( T_i \) be the first leaf of \( B(m - n + j) \). For \( E(m) \) the wrap around rule means that whenever the above definition causes the first leaf of \( E(m) \) to be placed after leaf \( m \) of \( T_i \), instead of using the first leaf of \( E(m) \) use the first leaf of \( E(n - i) \), and make the first leaf of \( T_i \) be the first leaf of \( E(m - n + i) \). [The phrase “wrap around” is appropriate, since the sum of \( (n - j) \) and \( (m - n + j) \) is \( m \) and the sum of \( (n - i) \) and \( (m - n + i) \) is \( m \).]

Finally, when the above definition does not make clear that a \( B \) or \( E \) subtree extends to the main right branch of \( T_i \), attach the subtree to the main left branch of \( T_i \). (This is arbitrary: attaching to main right branch is just as effective.) This completes the definition of \( T_i \).

The definition of \( T_i \) is illustrated in Figure 9. Ignoring the numbering of nodes for the time being, the figure shows the decomposition trees for a 16-component vector for processors \( p_4 \) and \( p_5 \). Here \( m = 8 \), since \( n = 16 \). For the case when \( i = 5 \), we have \( j = 13 \), so leaf 5 of the decomposition tree \( T_5 \) for processor \( p_5 \) is the first leaf of \( E(8) \) (which connects to the main left branch of \( T_i \)) and leaf 13 of \( T_5 \) is the first leaf of \( B(8) \), which wraps around from right to left. This means that leaf 13 of \( T_5 \) is the first leaf of \( B(3) \) and the first leaf of \( T_5 \) is the first leaf of \( B(5) \).

How does each processor in the ring perform its work using \( n/2 \) stages of communication and computation? Each processor begins by calling \( P_1 \) to yield the result for a slice of \( v \) of length 1. Each processor performs \( n/2 \) stages of communication and computation, alternately communicating with the processor to one side and then with the processor to the other side. In the first such stage even numbered processors swap a component value with their right neighbor and odd numbered processors swap a component value with their left neighbor, with \( p_0 \) considered to be to the immediate right of \( p_{n-1} \). Upon receiving a component value, each processor calls \( P_1 \) to obtain the partial answer for this component.
value, and then (whenever possible) P2 to combine this partial answer with a prior partial answer held by the processor, to obtain the partial answer for the appropriate slice of v specified in the decomposition tree for the processor. Each processor must maintain at most three such partial answers. The “wrap around” feature of the decomposition tree ensures that slices of v are handled correctly when, as the stages iterate, communications from the left neighbor (respectively, right neighbor) represent information about a slice of v to the right (respectively, left) of the processor’s position.

The algorithm just given implies the numbering of tree nodes in Figure 9. As illustrated in this figure, the first five component values that processor p6 receives from the left are components of v to the left but the remaining three component values the same processor receives from the same direction are components of v to the right. Although the final component value p6 receives that is to the right of v[5] is obtained via a communication from the left, the final component value p4 receives that is to the right of v[4] (namely v[12]) is obtained via a communication from the right. Why does this follow from the above algorithm? Prior to any communications, each of p4 and p6 has the component value for one of the leaves in its version of E(8). Unlike p6, processor p4 begins communicating in the direction of E(8), so that it has finished the work for E(8) before it receives its final communication from the right.

The strategy just described can be improved in two ways. First: Instead of
sending a component value to a neighbor, processors can send the result of calling \( P1 \) with the component value, so that only one call on \( P1 \) is necessary for a particular component value and this call need only be made by the processor located at that component value. Second: After the first stage of communication, instead of sending the result of calling \( P1 \) with a component value, processors can send the result of calling \( P2 \) with exactly two component values. This latter improvement changes all decomposition trees so that leaves are grouped in pairs. Figure 10 shows the decomposition trees for a 16-component vector for processors \( p4 \) and \( p5 \) when this improvement is used. Once again we see how the flexibility of the nondeterministic divide supports an efficient implementation.

Finally we consider diva calls that contain hyphen notation. If the diva call contains hyphen notation corresponding to one or more scalar parameters of mode \texttt{out}, the values of the vector components are communicated around the ring (using the communication algorithm described above) and calls on \( P1 \) are made by each processor for each component value of \( v \), to obtain the appropriate \texttt{out} scalars to use in calls on \( P2 \) during each stage of computation. The decomposition trees in Figure 10 can still be used, since each processor can be programmed to send two component values to its neighbor in each communication after the first stage. If the diva call contains hyphen notation corresponding to one or more vector parameters of mode \texttt{out}, the values of hyphened scalar \texttt{in} parameters are communicated around the ring (using the communication algorithm described above), calls on \( P1 \) are made by each processor to obtain the component values of the \texttt{out} vectors, any \texttt{out} scalar results from these particular calls on \( P1 \) are ignored, and no additional calls are made on \( P2 \) beyond those described earlier in this paragraph.

5.2.3 Torus

This section briefly considers a torus. A torus is a two-dimensional rectangular mesh with wrap around bidirectional connections: the processor on the bottom of a column can communicate directly with the processor on the top of the same column and the processor farthest right on a row can communicate directly with the processor farthest left on the same row. Thus a torus is the Cartesian product of bidirectional rings.

Here we consider a torus \( T \) having an even number of rows and the same number of columns, for a total of \( n \) processors. \( T \) can compute a reduction in time proportional to \( \sqrt{n} \) by performing a reduction across all rows in parallel and then using the results in a reduction across all columns in parallel. The extra requirement of diva procedures that unhyphen calls on \( P2 \) be made so actual parameters are given in the proper order can be satisfied by a straightforward extension of the implementation for bidirectional rings just described. To obtain the decomposition tree for processor \( p(i,j) \), take the decomposition tree for \( p_j \) in a bidirectional ring with \( \sqrt{n} \) processors and replace each leaf by the decomposition tree for \( p_i \) in a bidirectional ring with \( \sqrt{n} \) processors. (The
preceding section gave two algorithms for bidirectional rings, that lead to dif­
ferent decomposition trees; either algorithm can be used for the rows and either
can be used for the columns.)

Now consider a diva call containing hyphen notation. A torus having an even
number of rows, such as we have considered, clearly has a single ring as a sub­
graph that contains all its processors. Thus the algorithm described at the end
of the preceding section can be used, to compute the $n^2$ results in asymptotically
optimal linear time. Notice that this is asymptotically superior to implementing
such a diva call using a loop of unhyphenated diva calls, where each unhyphened
call is implemented using the algorithm just given for unhyphenated diva calls,
since that would require the product of linear and square-root time.

5.2.4 Hypercube
Let $H$ be a hypercube of dimension $d$ and let $v$ be a vector whose subscripts are
the processor indices of $H$, using the usual ordering for binary numbers. Two
processors of $H$ share a bidirectional communication channel if and only if their
indices differ in exactly one bit position. Two processors of $H$ that don’t share
a channel can communicate with one another, using intermediate processors.
However, channel contention can be minimized during parallel program execu-
tion if each processor at each stage of execution communicates only with an adjacent processor; the implementation described in this section satisfies this property.

Unlike the preceding descriptions for parallel computers, this section does not describe an implementation algorithm for an unhyped diva call in terms of a decomposition tree. For this reason, the section contains a proof that the implementation algorithm corresponds to a decomposition tree.

By the \textit{i}th bit of the code for a binary number we shall mean the \textit{i}th rightmost bit of the code. A standard way to let each processor in \(H\) obtain the result of a reduction is to let each processor compute successive partial reductions using its own value and, for \(i\) from 1 to \(d\), the value of the partial reduction of its neighbor on the communication channel corresponding to the \(i\)th bit of its index. Assuming appropriate calls are made on \(P_1\), correctness for a diva call is ensured by using the following order for the first two actual parameters in the calls on \(P_2\) by a processor: if the processor's index is lower than the neighbor's, first the processor's partial value and then the neighbor's partial value; otherwise, first the neighbor's partial value and then the processor's partial value.

To prove that each processor following this algorithm is using a decomposition tree, the following terminology and notation will be helpful. By a \textit{bit of a component of} \(v\) we shall mean a \textit{bit of the subscript of the component}. Also, for \(1 \leq i \leq d\) and \(j \in \{0, 1\}\), let \(S(i, j)\) denote the ordered sequence of components of \(v\) whose \(i\)th bit is \(j\) and whose remaining 1 bits (if any) occur to the right of the \(i\)th bit of the component. Finally, for \(1 \leq i < d\), let \(T(i, j)\) denote the ordered sequence of components of \(v\) obtained by setting to 1 the \((i + 1)^{\text{st}}\) bit of each component in \(S(i, j)\). (For example, if \(d = 4\), \(i = 2\), and \(j = 1\), then \(S(i, j)\) is the sequence \(v[0010], v[0011]\) and \(T(i, j)\) is the sequence \(v[0110], v[0111]\).)

**LEMMA.** For \(1 \leq i \leq d\), \(S(i, 0)\) and \(S(i, 1)\) are slices of \(v\) and \(S(i, 0)\) immediately precedes \(S(i, 1)\).

**PROOF.** This is clearly true if \(i = 1\), since it says the first component of \(v\) is a slice, the second component of \(v\) is a slice, and the first component immediately precedes the second component. Suppose the lemma is true for \(1 \leq i < d\); we show it is true for \(i + 1\). \(S(i + 1, 0)\) is a slice of \(v\), since it is the sequence obtained by concatenating the adjacent slices \(S(i, 0)\) and \(S(i, 1)\). In addition, \(S(i + 1, 1)\) is a slice of \(v\), since it is the sequence obtained by concatenating \(T(i, 0)\) and \(T(i, 1)\) (which are adjacent slices since they are the ordered sequences obtained by adding \(2^i\) to each subscript of the ordered sequences \(S(i, 0)\) and \(S(i, 1)\), respectively). Finally, \(S(i + 1, 0)\) immediately precedes \(S(i + 1, 1)\). \textbf{QED}

**THEOREM 3.** Each processor following the algorithm described above uses a decomposition tree.

**PROOF.** Let \(i\) be the bit position corresponding to the communication chan-
nel the processor uses at a particular stage of the algorithm. Let \( k \) be the number whose binary code is the same as the index of the processor, except for having all 0 bits in its rightmost \( i \) bit positions. (If the processor index is 1110 and \( i \) is 2, then \( k \) is 12.) The call on \( P_2 \) by the processor combines two results: its own partial result prior to the current stage and the partial result received by the communication at the beginning of the current stage. The first (respectively, second) partial result used in the call on \( P_2 \) corresponds to the ordered sequence obtained by adding \( k \) to each subscript of the ordered sequence \( S(i, 0) \) (respectively, \( S(i, 1) \)). It follows easily from the lemma that the two ordered sequences just defined are slices of \( v \) and the first immediately precedes the second. \textbf{QED}

Figure 11 shows the decomposition trees for processors \( p_4 \) and \( p_5 \). Processor \( p_4 \) never learns, nor needs to learn, the partial result for slices properly contained within the slice \( v[0:3] \). Four communication steps (in general \( d \)) are needed to obtain the final result: these communications lead to the \( P_2 \) call at the nodes labeled 1, 3, 5, and 7. The tree for processor \( p_5 \) is identical. The trees for all other processors are the same, except for the numbering of nodes in the tree.

Now consider a diva call containing hyphen notation. As was true for a torus, a ring subgraph of a hypercube that contains all its processors can be
procedure P (A: in Vector1; B: out Vector2; First, Last: in Integer;
   Data: in Type1; Ans: out Type2) is
   L, R: Type2;
begin
   Pi (A[First], B[First], First, Last, Data, L);
   R := L; -- Assures R has a value in the call on P2
   ShiftAmount := 1;
   for I in 1 .. Log loop
      Shift (L, R, ShiftAmount);
      P2 (L, R, L);
      ShiftAmount := ShiftAmount * 2;
   end loop;
   Ans := L;
end P;

Figure 12: Pseudocode for Sample SIMD Implementation

obtained. The binary reflected Gray code can be used for this purpose. Once
again, the resulting algorithm is linear in the number of processors. A loop of
unhyphened diva calls, where each unhyphened call is implemented according to
the algorithm above for unhyphened calls, would require the product of linear
and logarithmic time.

5.2.5 SIMD

Here we consider a SIMD computer whose processors are connected by a hy­
percube interconnection network. One strategy for implementing the diva call
for

   P (X, Y, 1, N, B, C);

on a SIMD computer is for each processor to make the call

   P (X, Y, MyIndex, MyIndex, B, C);

where the pseudocode for P is in Figure 12, and where only the first processor
uses the resulting value of out parameters. The computer is assumed to permit
data to be shifted from processors with higher processor numbers to processors
with lower numbers by shifts that are powers of 2. We assume the length of the
vectors is a power of 2 and is the number of processors. The notation

   Shift (<data>, <target_variable>, <delta>);

denotes a shift, where the value of <data> is shifted to the local variable
<target_variable> of the processor whose index is <delta> less than that
of the processor executing the instruction. Processors told to shift data to a
processor whose index is less than 0 execute no-op's during the clock cycles
when the remaining processors are executing the shift. The value of $\log$ in Figure 12 is $\log_2$ of the number of processors. The code is to be executed in SIMD fashion. For example, when a block of the code is a branch of an if statement, those processors whose local data dictates not performing the branch would execute no-op’s while the remaining processors execute the branch. For many diva procedures, such blocks of code occurring within the true and false branch of the outer if of the diva procedure contain just a single assignment statement. Thus the use of such no-op’s would not be as extensive for these diva procedures as it would for executing typical MIMD style code on a SIMD architecture. The decomposition tree for the first processor is the same as those shown in Figure 11 for the hypercube, except for the node numbering.

In the implementations described earlier, for rings, torus, and hypercube, all processors received the final answer. In this implementation, the final answer is received only by the first processor. An implementation that lets each processor receive the answer can be obtained via a modification of the implementation described for a hypercube. The MIMD code described for the hypercube is SPMD (single program multiple data) and can thus be adapted for a SIMD computer.

A diva call containing hyphen notation can be implemented as suggested for a hypercube; i.e. using a ring approach. The code for the ring implementation, like that for the hypercube, is SPMD.

### 5.2.6 Keeping Processes Alive

Since the purpose of Section 5.2 is to illustrate the flexibility of the nondeterministic divide, the implementation of just a single diva call has been considered. Often two or more diva calls that use vectors having the same length occur in sequence and, in such situations, the processes that implement the first call can be employed to help carry out the work of the remaining calls in the sequence.

The simplest way to do this is to use a barrier [Jordan 1988; Pratt 1987] at the end of the implementation of a diva call to ensure all work on the diva call is complete before any subsequent work begins. Of course, the equivalent of a barrier is implicit after each instruction on a SIMD computer, so such computers are not considered further in this section.

A barrier is not always needed for the implementations we have considered, assuming the message passing or other low-level primitives used to implement one diva call are programmed so they cannot interfere with the primitives used to implement another diva call. (A send executed by a process implementing the second of two diva calls should never match a receive executed by a process implementing the first call, etc.) For instance, even without a barrier at the end of the implementation of the first diva call in Example 3.6, each value used by a process during the second diva call is appropriate, either because the value is unaffected by the first call (true for N) or because the value used by a process in the second call is received via (direct or indirect) communication from the
process that computed the value (true for component values of \( Z \)). The same is true for Example 3.8 (except that it is component values of Position that are received from the process that computed the value). Often a barrier is unnecessary in the implementation between a pair of diva calls because the only values used by a process in the implementation of the second call which are changed by the first call are changed by the process itself.

Here is an example to illustrate the need for a barrier. Suppose the vector Position has all zero values prior to the first of the calls

\[
\text{FindCount}(X, 1, N, [-], X[-], \text{Position}[-]);
\]

\[
\text{Assign}(Y, 1, N, [-], \text{Position}[2]);
\]

where \text{FindCount} and \text{Assign} are defined in Figures 18 and 19 in the Appendix. The second call assigns Position[2] to each component of Y. Lack of a barrier in the implementation at the end of the first call could result in the first processor's assigning too small a value to its component of Y. A strategy for determining when to use a barrier is beyond the scope of the article.

5.3 Asymptotic Optimality

This final part of Section 5 summarizes the theoretical results obtained and seeks to put them into practical perspective. Unless mentioned otherwise, all diva calls referred to here are on diva procedures whose \( P_1 \) and \( P_2 \) can be executed in constant time. It is straightforward to generalize the asymptotic complexity results so they are expressed in terms of the time complexity of \( P_1 \) and \( P_2 \), but that would introduce unnecessary notational baggage and the case where these two procedures take constant time includes nearly all examples in the article.

\textbf{THEOREM 4.} If each processor works with just a single component of the vectors then, in terms of the length of the vectors:

- An unhyped diva call can be executed
  - in linear time on a unidirectional ring and on a bidirectional ring
  - in square-root time on a torus
  - in logarithmic time on a hypercube and on a SIMD computer with hypercube interconnections.

- A diva call containing hyphen notation can be executed
  - in linear time on a unidirectional ring, a bidirectional ring, a torus, a hypercube, and a SIMD computer with hypercube interconnections.

All times are asymptotically optimal.
PROOF. See the preceding implementation descriptions. For the unhyphenated call, asymptotic optimality follows from the fact that these are the times it takes to communicate values between the most distant pair of processors in the computers. (Bidirectional rings require less time than unidirectional rings, though both take the same asymptotic time.) For the diva call containing hyphen notation, asymptotic optimality follows from there being $n^2$ results to be computed, where $n$ is the length of the vectors. QED

We now proceed to remove the restriction that each processor works with just a single component of the vectors. Also removed are limitations on the length, $n$, of the vectors: that it is even, or the square of an even number, or a power of two. Let $p$ denote the number of processors in the parallel computer. (Up to now $n$ denoted both the number of processors and the length of the vectors.) A uniprocessor implementation, as described in the proof of Theorem 2, permits each processor to work on a slice of a vector whenever we have described the processor working on a single component. For a diva call containing hyphen notation on a distributed memory system, this requires that more information be communicated between processors, which often can be accomplished more efficiently in large bundles of communications, such as by using cut-through switching [Kermani and Kleinrock 1979] or a connection-oriented transport protocol [Tanenbaum 1988]. Each processor can store the result of such communications until ready to use the information.

We view $p$ as a variable; i.e., we view the parallel computer as scalable. We view the minimum and maximum slice length per processor as constants determined by the characteristics of both the particular processor and the particular parallel computer, such as the relative time required for a computation and a communication. In the following theorem, saying that $n$ is large enough to justify using the parallel computer in question means that if the parallel computer is a ring, then $n$ is at least twice the minimum slice length (since a ring has at least two processors), and, if it is a torus, hypercube, or SIMD hypercube, then $n$ is at least four times the minimum slice length.

THEOREM 5. The asymptotic optimality results in Theorem 4 hold for each parallel computer mentioned there, when each processor is given a slice whose length is inclusively between the minimum and maximum slice length allocated to a processor, assuming $n$ is large enough to justify using the parallel computer in question. The maximum slice length need be no more than 4 times the minimum slice length for any such computer. For some values of $n$ for the torus, the maximum slice length must be 4 times the minimum slice length.

PROOF. See Figure 22 in the Appendix.

In spite of the asymptotically optimal results, more practical approaches are sometimes possible. As indicated in Theorem 2, an unhyphenated diva call can
be executed in linear time on a uniprocessor. Since a ring offers no asymptotic advantage for such calls, the theoretical results suggest that in situations where the only computer available is a ring, just a single processor of the ring should be used instead of the entire ring. In addition, a sufficient and rather broad condition for satisfying the hypothesis of Theorem 2 is that no loops or procedure calls occur within P1 and P2 (nor any operation such as exponentiation that might introduce a variable independent of n) and this condition can be checked automatically by the translator.

But using the entire ring for a single unhyphenated diva call could indeed be quite practical when the vectors involved are too large to be processed feasibly by a single processor. Huge vectors are produced routinely by certain scientific applications, such as nuclear accelerator experiments, DNA research, and remote earth sensing via satellite. Consider, for example, the problem of counting ascents for a vector of trillions of data values from a satellite, stored on a large number of tapes. Given the diva algorithm approach, a translator can provide the needed instructions for the members of a large ring of computers to read tapes in parallel, to use the sequential loop in Figure 7(b) on the slice they receive from each tape, and to combine the results as explained in Section 5.2.2. The computers can all be uniprocessors, but this is not required. Some can be workstations, some hypercubes, etc., and it is even conceivable that the implementation could permit the mix to change over the time of the computation. The nondeterministic divide is at a sufficiently high level of abstraction to support quite a variety of implementations.

Of course a ring is also practical when the time required for executing a call on P1 is significant (whether linear time or not), since the calls on P1 could then be performed in parallel.

The fact that the given implementations of diva calls are asymptotically optimal does not imply that using a diva call is itself a magic wand. The author knows of no programming language construct that, when used to build a correct solution to a problem, will guarantee an asymptotically optimal time solution to the problem (except, of course, for a programming construct that yields a constant time algorithm). The user of a for loop rightfully expects the language translator to ensure the loop will be performed on a uniprocessor in linear time, assuming a single iteration of the loop takes constant time. But this does not imply that the optimal solution to the problem being programmed by the for loop cannot have better than linear time: for instance, a loop can be used to find the sum of the numbers from 1 to n, which can be found in constant time by computing n(n+1)/2. This distinction between asymptotically optimal implementation and asymptotically optimal algorithm applies equally to diva calls. In fact, the same example can be used, since an unhyphenated diva call on FindSum of Figure 2 can be used to compute the same sum.

This observation applies to diva calls containing hyphen notation as well. Consider the Plateau Problem [Gries 1981]: given a vector whose component values are in nondecreasing order, compute the length of the longest plateau,
where a "plateau" is a slice whose components have equal values. The problem can be solved using a diva call (unhyphened), whose P1 and P2 take constant time, so a solution can be obtained in logarithmic time on a hypercube and in square-root time on a torus. Now consider the more general problem of finding, for each component of the vector, the length of the longest plateau containing the component. This more general problem can be solved using a diva call containing hyphen notation. The programmer who uses such a diva call can rightfully expect the quadratic number of computations will be obtained in linear time on any of the computers under consideration (a guarantee not afforded the programmer of a doubly nested for loop). But the problem can be solved by a totally different sequential algorithm, which uses an extra vector and has just a linear number of computations to perform. (Make a pass from left to right. Corresponding to and having the same length as each plateau in the first vector, store in the second vector a pattern that begins with zero or more Os and ends with the length of the plateau. Then make a pass from right to left over the second vector, copying each nonzero value found into the adjacent components to its left whose values had been 0.) The point of this example seems to be: before looking for a diva-based solution to a problem that has similar subproblems for each component of a vector, try to find a linear time sequential algorithm for solving the problem. But then, once again, it may not be that simple: if the vector is too large to be processed feasibly by a single processor, the diva approach may be more practical.

6 GENERAL REDUCTION OPERATORS

Given that a function f is known to be associative, a general reduction operator R obtains the reduction R(f, v) of a vector v using the function f. Such a reduction can be implemented in log time on a suitable parallel computer, assuming that each call on the function takes constant time. Note that, like the use of diva procedures, such a computation can be programmed without requiring that the programmer consider parallelism; this is permitted, for example, in Paralation Lisp [Sabot 1988].

To illustrate this technique, consider computing the number of ascents in a sequence S. (This was accomplished in Example 3.3 using the diva procedure FindNumber.) The major steps would be:

1. Define a type AnswerType that is a record having the integer valued fields Count, First, and Last.
2. Declare a new vector V whose range of subscripts is that of S and whose components are of type AnswerType.
3. Initialize the components of V using code similar to the inline call on P1 (i.e., the true branch of the if) in FindNumber as well as using the corresponding values of S.
4. Define a function subprogram \( f \) that takes two arguments of type \texttt{AnswerType} and returns a value of type \texttt{AnswerType} using code similar to inline call on P2 (i.e., the false branch of the if) in \texttt{FindNumber}.

5. Apply the general reduction operator to reduce \( V \) using \( f \).

A comparison of diva algorithms with general reduction operators can be summarized as follows:

1. The use of recursion is well-understood by the sequential programmer. Thus the use of a diva procedure requires that the programmer learn no new semantic rules, except for those relating to the use of the nondeterministic divide. But the sequential programmer can view the semantics of the nondeterministic divide as being simpler than usual divide-and-conquer programming: there is no need to specify a division point if the choice of division point doesn't matter.

2. A separate proof of associativity of \( f \) need not be given when a diva procedure is used, since the associativity of \( f \) follows from the proof that the diva procedure returns the unique value specified by the procedure. On the other hand, when a general reduction operator is used, a separate proof of the associativity of \( f \) is needed. That is, one must prove that \( f(f(x,y),z) = f(x,f(y,z)) \) always holds, which can be complicated. Confronted with such complexity when using a general reduction operator in a program, programmers are tempted to assume associativity, thereby increasing the risk of error. While such a program may yield correct answers when tested, the program may in fact rely on a particular ordering of the combining of partial results that will change, such as when the underlying operating system is changed or when the program is ported to a different computer. It is conceivable that such a programming error could occur with even as simple an application as \texttt{FindNumber}. For example, suppose the ordering used by the underlying implementation had the effect of combining the component values of \( V \) from left to right. Then a programmer using the approach of Figure 13 would obtain correct results and might think that “natural problems like counting ascents are associative”. But other underlying implementations would not always give correct results. (Consider, for example, what would happen if the component values of \( S \) were 20, 30, and 10 and the underlying implementation combined results from right to left.)

3. It is unnatural to include the requirement of associativity in the specification of a function, since in the context of such a specification the function operates on just two values.

4. Strong induction can provide a simple approach to verifying program correctness for the kind of applications considered in the article, as illustrated
Define a record type AnswerType to have Integer fields Count and Value, initialize V so that for all I

\[ V(I).\text{Count} := 0; \]
\[ V(I).\text{Value} := A[I]; \]

and apply a general reduction operator to reduce V using the function:

```plaintext
function f (X, Y: AnswerType) return AnswerType is
    Ans: AnswerType;
    begin
        if X.Value < Y.Value then
            Ans.Count := X.Count + Y.Count + 1;
        else
            Ans.Count := X.Count + Y.Count;
        end if;
        Ans.Value := Y.Value;
        return Ans;
    end f;
```

Figure 13: A Simple but Incorrect Approach to Counting Ascents

in Section 4.1. Such simple use of strong induction relies on a simple conceptualization that larger objects are composed of smaller objects. To adequately support such use of strong induction, the programming language should make such a conceptualization clear to the programmer and this conceptualization should be more abstract than implementation-specific concepts, since for portability a proof of program correctness should be independent of a particular implementation whenever possible. Diva procedures provide just such a simple, yet abstract, conceptualization; namely, the conceptualization of the nondeterministic divide. On the other hand, when a general reduction operator is used, no such conceptualization is made clear by the form in which the technique is expressed.

5. Whenever possible, the form through which a technique is expressed within a programming language should remind the programmer of the underlying assumptions of the technique, even if it does not provide a suitable conceptualization. As observed earlier, when a general reduction operator is used the underlying implementation will assume the associativity of \( f \) but this technique does not provide assistance in proving associativity. Perhaps more importantly, the language form for expressing a general reduction operator does not make clear the associativity assumption itself nor an equivalent to this assumption; the programmer is required to remember this assumption independently of the language form. The language form
for expressing a general reduction operator also does not make clear the lack of any commutativity assumption on \( f \). For example, knowing that important assumptions about \( f \) are not made clear within the language form, a programmer using a general reduction operator could worry that the result of applying \( f \) to the component values of \( V \) with odd indexes and the result of applying \( f \) to the component values of \( V \) with even indexes might be found separately and then combined, thereby precluding many of the applications discussed in the article.

6. As explained at the beginning of this section, general reduction operators require not only that \( f \) be a function, but also that the parameters of \( f \) be two values of the result type of \( f \). As a result it is less than ideal to use such operators for applications in which values need to be used by the function that the function must not modify. The formal parameters \texttt{GivenLoc} and \texttt{Threshold} of diva procedure \texttt{FindRight} in the Appendix are such values. When a general reduction operator is used for this application, either fields containing these values need to be included in the result type, or else a nonlocal access from within \( f \) must be used to obtain these values. In either case, an important fact known to the programmer is being withheld from the translator: these values should not be modified. Using an in parameter in a diva procedure makes this fact explicit.

7. While one can always replace procedures by functions in any language permitting the value returned by a function to be an arbitrary record, the widespread use of procedures demonstrates their greater naturalness for many applications. A diva procedure can have more than one parameter that returns part of the final answer. Here we point out an awkward aspect of Example 3.2, where the programmer is forced to decide among three alternatives: to use an assignment statement like

\[
\text{Value} := \text{Answer.Corr};
\]

in the body of code that includes the diva call, to use an encapsulating procedure simply to hide this detail, or to use the field and record name in all later code accessing the result of the diva call. But since a diva procedure is a procedure, there is a fourth alternative: let the diva procedure have two parameters for returning results, one for returning the location and the other for returning the value. The use of a function in a general reduction operator, on the other hand, requires a record for more than one result value.

8. Only a single vector is reduced by the application of a general reduction operator. Within a conventional semantic model it is thus necessary to create a new vector of records solely to satisfy this requirement for many applications. Such ad-hoc vectors can require additional space and time and reduce the readability of the program and are unnecessary when a
diva procedure is used, since multiple vectors are permitted. (Such ad-hoc vectors can also be avoided by using the noteworthy, unconventional semantics of Paralation Lisp [Sabot 1988].) In addition, the effect of assigning initial values to the components of such an ad-hoc vector can be achieved in the true branch of the if of the diva procedure and this action can be implemented in parallel.

9. Using a general reduction operator can force the programmer to violate the level of abstraction ideally provided by a function in a programming language, since in defining $f$ the programmer often must focus on the particular use of $f$ by a general reduction operator.

7 CONCLUSIONS

The importance of developing effective ways to program parallel computers has been widely acknowledged. Often such computers are programmed by thinking in terms of "who does what when". The complexity of such a low level approach can cause the programmer to overlook race conditions, possible deadlocks, and other problems. One solution to the problem of programming parallel computers is to provide high-level concepts for sequential programming that parallelize well.

This article proposes and investigates the nondeterministic divide as such a high-level concept. The use of the nondeterministic divide can be viewed by the sequential programmer as being simpler than usual divide-and-conquer programming: there is no need to specify a division point if the choice of division point doesn't matter.

The article has the following limitations. The results are applicable only to situations involving data that can be accurately represented within the target computer; for example, associativity of addition and multiplication is assumed. A recommended strategy for a compile-time decision among the various options for implementing a diva call on a uniprocessor is beyond the scope of the article. While the article presents several ways of implementing a diva call on parallel computers, the purpose is to illustrate the flexibility of the nondeterministic divide rather than to provide an exhaustive study, so more research is needed on parallel implementations. A strategy for determining when a barrier is needed at the end of the implementation of a diva call is beyond the scope of the article. Finally, the use of the nondeterministic divide in the article is restricted to applications involving one or more vectors. While this is a significant restriction, vectors support a large class of important applications. They can model successive values of sensors over a fixed time interval, where the resulting data can be scientific (such as the cosmic ray count in a spacecraft), social (such as population), or commercial (such as stock market statistics). Additional one-dimensional phenomena include computer programs, DNA, natural language texts, and manuscripts of music. For some applications, the nondeter-
ministic divide described here can be used with more general data structures. For instance, one two-dimensional array can be copied to another, by treating a two-dimensional array as a vector of vectors. However, either the rows will be treated independently of each other or the columns will be treated independently of each other, so full two-dimensional information is not considered.

The article has shown that diva calls have the major implementation advantages of general reduction operators, yet are free of numerous undesirable features of general reduction operators and thereby enhance program correctness and abstraction. The undesirable features of general reduction operators include the lack of a simple, implementation-independent conceptualization for the combining of function values, the difficulty of showing that a nontrivial function is associative in the absence of a suitable conceptualization, the fact that the language form for expressing a general reduction operator does not make underlying assumptions clear to the programmer, the fact that information known at the programmer's level of abstraction must be withheld sometimes from translators even though such information could be used to uncover errors, and the frequent necessity of using records to combine variables into a single object regardless of whether such use of records is a natural and appropriate abstraction. In addition, unlike general reduction operators, diva calls can be used to assign computed values to the components of a vector.

The article has shown that the nondeterministic divide supports informal reasoning about program correctness, both using strong induction and using a style of hierarchical reasoning analogous to the use of a definite integral, in which one considers a binding of free variables after considering a simpler situation. Diva calls are at a higher level of abstraction than such implementation issues as MIMD versus SIMD, interconnection topology, and the workload of processors. Concerns about implementation details should be orthogonal to concerns about logical correctness.

Let \( P \) be a diva procedure whose \( P_1 \) and \( P_2 \) procedures execute in constant time. The article has shown how a diva call on \( P \) that doesn't contain hyphen notation is implementable on rings, toruses, and hypercubes in asymptotically optimal time for the target topology: linear, square-root, and logarithmic time, respectively. The article has also shown how the quadratic number of results of a diva call on \( P \) containing hyphen notation can be computed by a ring, torus, and hypercube having a linear number of processors in linear time, which is asymptotically optimal. Each processor of the parallel computer is responsible for as large a slice as necessary for a practical grain size, assuming this size is in a constant range determined by the processor and the parallel computer. When practical considerations conflict with considerations of asymptotic complexity and when the vectors are sufficiently long, the nondeterministic divide also provides desirable flexibility by permitting slices to be read from secondary memory by different processors at the same time.

Hardware support for high-speed computing is expected to be increasingly in the form of heterogeneous computing systems [NSF 1992]. Although the arti-
cle has emphasized static allocation of processors for the implementation of the nondeterministic divide, it has also suggested the allocation can be dynamic. Indeed the implementation could use the most appropriate combination of computing resources available at run-time. The level of abstraction provided by the nondeterministic divide may ultimately allow programmers to ignore such run-time considerations, just as programmers can now ignore the run-time considerations of virtual memory (and, if other research projects succeed, will be able to ignore the run-time considerations of distributed memory).

Rather than propose a particular programming construct, the article has used pseudocode. Many results in the article might be possible without the need for any new programming syntax, since a compiler might be able to detect when a program's form satisfies certain restrictions required by diva algorithms (and uses but does not declare a Divide function). Special syntax for diva algorithms would certainly seem to aid both the programmer and the compiler writer. In deciding whether to include such special syntax within a programming language, one is reminded of the advice in Hoare's CSP article [Hoare 1978]:

Where a more elaborate construction ... is frequently useful, has properties which are more simply provable, and can also be implemented more efficiently than the general case, there is a strong reason for including in a programming language a special notation for that construction.

Yet another possibility, now under research by the author, is a user-friendly program development environment that avoids the necessity of adding syntax to a programming language to support diva algorithms, yet enforces the restrictions of diva algorithms when the programmer has indicated this is desirable and assists with informal proofs of correctness.

ACKNOWLEDGEMENTS
The author is grateful for discussions with colleagues at the Institute for Parallel Computation at the University of Virginia during his 1989-90 sabbatical. The author is indebted to his assistants over the past eight years, Chris Hanes, Jeff Michel, Greg Morrisett, and Scott Shauf, who built a prototype translator to assist in developing diva applications. In particular, the description of implementing diva calls on a bidirectional ring is based on Scott Shauf's work with the translator. Dana Richards suggested the final application listed in Figure 3. Michael Holloway served as technical monitor on the NASA grant since 1987.
Appendix

diva procedure FindMaxCorr (A: in FloatVector;
   B: in IntegerVector;
   First, Last: in Integer;
   Chosen: out AnswerType) is
   -- Chosen.Max is assigned the maximum of A[First:Last].
   -- Chosen.Corr is assigned the value of the component of B[First:Last]
   -- corresponding to the first component of A[First:Last] equal to
   -- Chosen.Max.
   L, R: AnswerType; -- Results for left and right slices
   Middle: Integer;
begin
   if First = Last then
      Chosen.Max := A[First];
      Chosen.Corr := B[First];
   else
      Middle := Divide (First, Last);
      FindMaxCorr (A, B, First, Middle, L);
      FindMaxCorr (A, B, Middle + 1, Last, R);
      if L.Max >= R.Max then
         Chosen := L;
      else
         Chosen := R;
      end if;
   end if;
end FindMaxCorr;

Figure 14: A Diva Procedure To Find A Value Corresponding to the Maximum
of a Vector
diva procedure FindNumber (A: in Vector; First, Last: in Integer;
                          Ans: out AnswerType) is
  -- Ans.Count is assigned the number of times a component of
  -- A[First:Last] is less than the next component of A[First:Last].
  -- Ans.First is assigned A[First].
  -- Ans.Last is assigned A[Last].
  L, R: AnswerType; -- Results for left and right slices
  Middle: Integer;
begin
  if First = Last then
    Ans.Count := 0;
    Ans.First := A[First];
    Ans.Last := A[Last];
  else
    Middle := Divide (First, Last);
    FindNumber (A, First, Middle, L);
    FindNumber (A, Middle + 1, Last, R);
    if L.Last < R.First then
      Ans.Count := L.Count + R.Count + 1;
    else
      Ans.Count := L.Count + R.Count;
    end if;
    Ans.First := L.First;
    Ans.Last := R.Last;
  end if;
end FindNumber;

Figure 15: A Diva Procedure to Find the Number of Ascents in a Sequence
The record type AnswerType has Integer fields Max, Pos, MaxFirst, MaxLast, and MaxLastPos, and a Boolean field AllExceed.

diva procedure FindMaxWinningStreak (A, B: in Vector;
First, Last: in Integer;
Ans: out AnswerType) is
   -- Ans.Max is the length of the longest slice in A[First:Last] all of
   -- whose components exceed the corresponding components of B[First:Last]
   -- and, if Ans.Max > 0, then Ans.Pos is the position in A[First:Last]
   -- where the first such slice begins.
   -- Ans.MaxFirst is the length of the longest such slice in A[First:Last]
   -- that starts at A[First].
   -- Ans.MaxLast is the length of the longest such slice in A[First:Last]
   -- that ends at A[Last] and, if Ans.MaxLast > 0, then Ans.MaxLastPos
   -- is the position in A[First:Last] where this slice begins.
   -- Ans.AllExceed is True if and only if all components of A[First:Last]
   -- exceed the corresponding components of B[First:Last].
   L, R: AnswerType; -- Results for left and right slices
   Middle: Integer;
begin
   if First = Last then
      if A[First] > B[First] then
         Ans.Max := 1;
         Ans.MaxFirst := 1;
         Ans.MaxLast := 1;
         Ans.AllExceed := True;
      else
         Ans.Max := 0;
         Ans.MaxFirst := 0;
         Ans.MaxLast := 0;
         Ans.AllExceed := False;
      end if;
      Ans.Pos := First;
      Ans.MaxLastPos := First;
   else
      Middle := Divide (First, Last);
      FindMaxWinningStreak (A, B, First, Middle, L);
      FindMaxWinningStreak (A, B, Middle + 1, Last, R);
      FindMaxWinningStreak2 (L, R, Ans); -- See next page
   end if;
end FindMaxWinningStreak;

Figure 16: A Diva Procedure to Find the Position of the Longest Winning Streak of One Sensor Over Another
procedure FindMaxWinningStreak2 (L, R: in AnswerType; 
   Ans: out AnswerType) is 
begin 
   if L.Max >= R.Max and L.Max >= L.MaxLast + R.MaxFirst then 
      Ans.Max := L.Max; 
      Ans.Pos := L.Pos; 
   else 
      if L.MaxLast + R.MaxFirst >= R.Max and L.MaxLast > 0 then 
         Ans.Max := L.MaxLast + R.MaxFirst; 
         Ans.Pos := L.MaxLastPos; 
      else 
         Ans.Max := R.Max; 
         Ans.Pos := R.Pos; 
      end if; 
   end if; 
   if L.AllExceed then 
      Ans.MaxFirst := L.Max + R.MaxFirst; 
   else 
      Ans.MaxFirst := L.MaxFirst; 
   end if; 
   if R.AllExceed and L.MaxLast > 0 then 
      Ans.MaxLast := L.MaxLast + R.Max; 
      Ans.MaxLastPos := L.MaxLastPos; 
   else 
      Ans.MaxLast := R.MaxLast; 
      Ans.MaxLastPos := R.MaxLastPos; 
   end if; 
   Ans.AllExceed := L.AllExceed and R.AllExceed; 
end FindMaxWinningStreak2;
The record type AnswerType has a Boolean field Found and an integer field Loc.

diva procedure FindRight (A: in Vector;
   First, Last: in Integer;
   GivenLoc: in Integer;
   Threshold: in Integer;
   Ans: out AnswerType) is

   -- Ans.Found is true if and only if A[First:Last] has a component to
   -- the right of A[GivenLoc] greater than Threshold.
   -- If Ans.Found is true, Ans.Loc is the location of the first such
   -- component.
   L, R: AnswerType; -- Results for left and right slices
   Middle: Integer;
begin
   if First = Last then
      Ans.Found := (First > GivenLoc and A[First] > Threshold);
      Ans.Loc := First;
   else
      Middle := Divide (First, Last);
      FindRight (A, First, Middle, GivenLoc, Threshold, L);
      FindRight (A, Middle + 1, Last, GivenLoc, Threshold, R);
      if L.Found then
         Ans := L;
      elsif R.Found then
         Ans := R;
      else
         Ans.Found := False;
      end if;
   end if;
end FindRight;

Figure 17: A Diva Procedure To Find the First Component to the Right Greater
than a Threshold
diva procedure FindCount (A: in Vector; First, Last: in Integer;
GivenLoc: in Integer;
Threshold: in Integer;
Count: out Integer) is
  -- Count is assigned the number of components of A[First:Last] that
  -- are: smaller than Threshold or
  -- equal to Threshold and to the left or at GivenLoc.
  L, R: Integer; -- Results for left and right slices
  Middle: Integer; -- Results for left and right slices
begin
  if First = Last then
    if A[First] < Threshold or
       (A[First] = Threshold and First <= GivenLoc) then
      Count := 1;
    else
      Count := 0;
    end if;
  else
    Middle := Divide (First, Last);
    FindCount (A, First, Middle, GivenLoc, Threshold, L);
    FindCount (A, Middle + 1, Last, GivenLoc, Threshold, R);
    Count := L + R;
  end if;
end FindCount;

Figure 18: A Diva Procedure To Compare Vector Components With a Threshold
diva procedure Assign (A: out Vector; First, Last: in Integer;
   GivenLoc: in Integer;
   GivenValue: in Integer) is
   -- If A[First:Last] has a subscript equal to GivenLoc, then
   -- this component of A[First:Last] receives the value GivenValue.
   Middle: Integer;
begin
if First = Last then
   if First = GivenLoc then
      A[First] := GivenValue;
   end if;
else
   Middle := Divide (First, Last);
   Assign (A, First, Middle, GivenLoc, GivenValue);
   Assign (A, Middle + 1, Last, GivenLoc, GivenValue);
end if;
end Assign;

Figure 19: A Diva Procedure To Assign a Value to a Vector Component
The proof is by strong induction on the length $n$ of the slice $A[\text{First:Last}]$. Let $\text{Assert}(A, B, \text{First}, \text{Last}, \text{Ans})$ be the assertion at the beginning of the procedure, with free variables $A$, $B$, $\text{First}$, $\text{Last}$, and $\text{Ans}$.

**case one:** $n = 1$. Then $\text{First} = \text{Last}$ and $\text{Assert}(A, B, \text{First}, \text{Last}, \text{Ans})$ is easy to check.

**case two:** Otherwise. We show the first part of $\text{Assert}(A, B, \text{First}, \text{Last}, \text{Ans})$. The proofs of the remaining parts are similar. To show:

\[
\text{Ans.Max} \text{ is the length of the longest slice in } A[\text{First:Last}] \text{ all of whose components exceed the corresponding components of } B[\text{First:Last}]
\]

By the induction hypothesis, both $\text{Assert}(A, B, \text{First}, \text{Middle}, L)$ and $\text{Assert}(A, B, \text{Middle} + 1, \text{Last}, R)$ hold, so by substitution

- $L.\text{Max}$ is the length of the longest slice in $A[\text{First:Middle}]$ all of whose components exceed the corresponding components of $B[\text{First:Middle}]$
- $R.\text{Max}$ is the length of the longest slice in $A[\text{Middle}+1:\text{Last}]$ all of whose components exceed the corresponding components of $B[\text{Middle}+1:\text{Last}]$
- $L.\text{MaxLast}$ is the length of the longest such slice in $A[\text{First:Middle}]$ that ends at $A[\text{Middle}]$
- $R.\text{MaxFirst}$ is the length of the longest such slice in $A[\text{Middle}+1:\text{Last}]$ that starts at $A[\text{Middle}+1]$

Also $\text{Ans.Max}$ gets its value from the logic:

\[
\text{if } L.\text{Max} \geq R.\text{Max} \text{ and } L.\text{Max} \geq L.\text{MaxLast} + R.\text{MaxFirst} \text{ then}
\]

\[
\text{Ans.Max} := L.\text{Max};
\]

\[
\text{else}
\]

\[
\text{if } L.\text{MaxLast} + R.\text{MaxFirst} \geq R.\text{Max} \text{ and } L.\text{MaxLast} > 0 \text{ then}
\]

\[
\text{Ans.Max} := L.\text{MaxLast} + R.\text{MaxFirst};
\]

\[
\text{else}
\]

\[
\text{Ans.Max} := R.\text{Max};
\]

\[
\text{end if};
\]

\[
\text{end if};
\]

The result follows from observing that the length of the longest winning streak in $A[\text{First:Last}]$ is the length of the longest winning streak in $A[\text{First:Middle}]$, if this is at least as long as the longest winning streaks in both $A[\text{Middle}+1:\text{Last}]$ and the overlap of $A[\text{First:Middle}]$ and $A[\text{Middle}+1:\text{Last}]$; otherwise it is the longer of the latter two winning streaks.

Figure 20: Informal Proof of Correctness of FindMaxWinningStreak
Let $M$ and $m$ represent the maximum and minimum time, respectively, for executing the conquer block of $P$. Consider any call on $P$ and let $n$ denote the length of the initial vectors of the call. Recall that the true branch of the if of $P$ will be executed $n$ times and the false branch $n-1$ times. Let $T$ be the total time required for all $n + (n - 1)$ determinations of which branch of the if to execute as well as all $n$ executions of the true branch; note that $T$ is independent of the choice of division points. For $1 \leq i \leq n - 1$, let $E_i$ and $M_i$ represent the time required for the $i$th execution of the conquer block for the End and Middle options, respectively, and let $L_i$ and $R_i$ represent the overhead for using a loop and recursion, respectively. (This notation does not ignore the fact that the values of local variables of the $i^{th}$ execution using the End Option will, in general, be different from such values of the $i^{th}$ execution using the Middle Option.) Let $D$ denote the smallest of the differences $R_i - L_i$, for $i$ between $1$ and $n - 1$. The fact that the End Option requires less total time than the Middle Option follows from the inequalities

$$T + \sum_{i=1}^{n-1} (E_i + L_i) \leq T + M(n - 1) + \sum_{i=1}^{n-1} L_i$$

$$< T + (m + D)(n - 1) + \sum_{i=1}^{n-1} L_i$$

$$= T + m(n - 1) + \sum_{i=1}^{n-1} (D + L_i)$$

$$\leq T + m(n - 1) + \sum_{i=1}^{n-1} (R_i - L_i + L_i)$$

$$= T + m(n - 1) + \sum_{i=1}^{n-1} R_i$$

$$\leq T + \sum_{i=1}^{n-1} (M_i + R_i)$$

where the single strict inequality follows from the condition in the hypothesis. The proof is completed by observing that the only property of the Middle Option used in the proof is that this option is implemented using recursion.

Figure 21: Proof of Theorem 1
First we show that for some values of \( n \) for the torus, the maximum slice length must be 4 times the minimum slice length. The smallest torus satisfying the description in Section 5.2.3 has 4 processors and the next to smallest has 16. Thus a vector that is one component too short to permit partitioning among 16 processors, because that would not allocate the minimum slice length to all processors, must be partitioned among 4 processors, which requires that some processors have 4 times as large a slice as the minimum desired slice.

Now let \( a \) be the minimum slice length allocated to a processor, and recall that this is a constant. Let \( p \) be the largest number such that both \( ap \leq n \) and: \( p \) is an even number (for a bidirectional ring), or the square of an even number (for a torus), or a power of two (for a hypercube and SIMD hypercube).

We can conclude that the maximum slice length need be no more than 4 times the minimum slice length for any such computer, by proving that the inequality

\[
ap \leq n < 4ap
\]

holds for all computers considered. For a bidirectional ring, we have \( p \geq 2 \), so

\[
ap \leq n < a(p + 2) = ap + 2a \leq 2ap < 4ap.
\]

To see the second inequality in the above line, use the fact that \( p \) is the largest even number such that \( ap \leq n \). The proof for a unidirectional ring is omitted since it is easier than the proof for the bidirectional ring. For a hypercube and SIMD hypercube

\[
ap \leq n < a(2p) < 4ap.
\]

For a torus

\[
ap \leq n < a(\sqrt{p} + 2)^2 = a(p + 4\sqrt{p} + 4) \leq 4ap
\]

where the final inequality follows from the argument: since \( p \) is the square of an even number, \( p \geq 4 \), so \( 0 \leq (p - 4)(9p - 4) \), hence \( 0 \leq 9p^2 - 40p + 16 \), and thus \( 4\sqrt{p} \leq 3p - 4 \) (square both sides to see this), so \( 4\sqrt{p} + 4 \leq 3p \). This completes the proof of (1).

Finally we see why the asymptotic complexity results continue to hold. Using inequality (1), Theorem 2, and Theorem 4, it is clear that assigning a slice (whose length is bounded by constants) to each processor permits an unhynphened diva call to be performed in \( O(n) \) time on a ring, since the implementation algorithm is \( O(1) + O(p) \) which is \( O(p) \), which is easily seen to be \( O(n) \).

A similar analysis shows that the other asymptotic complexity results continue to hold, since inequality (1) implies that an algorithm that is \( O(\sqrt{p}) \) is \( O(\sqrt{n}) \) and an algorithm that is \( O(\log(p)) \) is \( O(\log(n)) \).

Figure 22: Proof of Theorem 5
REFERENCES

NSF 1992 The National Science Foundation Metacenter. A report prepared for the program advisory committee to the National Science Foundation
Division of Advanced Scientific Computing.

PANCAKE, C. M. 1991 Software support for parallel computing: Where are we headed? Commun. ACM 34, 11, 52-64.


