The Nondeterministic Divide

Arthur Charlesworth

IPC-TR-90-005
November 30, 1990

Institute for Parallel Computation
School of Engineering and Applied Science
University of Virginia
Charlottesville, VA 22903

and

Department of Mathematics and Computer Science
University of Richmond
Richmond, VA 23173

This research was supported by the National Aeronautics and Space Administration, under grant number NAG-1-774, by the Jet Propulsion Laboratory, under grant number 95722, and by sabbatical funding from the University of Richmond.

The Nondeterministic Divide

Arthur Charlesworth
University of Richmond and University of Virginia

Abstract

The *nondeterministic divide* partitions a vector into two non-empty slices by allowing the point of division to be chosen nondeterministically. 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 components is only permitted during activations in which the vector parameters have unit length. The notion of a diva algorithm is formulated precisely as a *diva call*, a restricted call on a sequential procedure. Diva calls are proven to be intimately related to associativity. Numerous applications of diva calls are given and strategies are described for translating a diva call into code for a variety of parallel computers. Thus diva algorithms separate logical correctness concerns from implementation concerns.

Categories and Subject Descriptors: D.1.3 [Programming Techniques]: Concurrent Programming; D.3.3 [Programming Languages]: Language Constructs

General Terms: Languages

Additional Key Words and Phrases: Divide-and-conquer, nondeterminism, recursion, parallel programming, associativity, reduction.

Contents

1 INTRODUCTION

The architecture of computers, once restricted to having a single processor for the execution of instructions, is rapidly evolving in diverse directions. Recent multiple instruction stream, multiple data stream (MIMD) computers have from a few tens of processors that use a common bus for accessing shared memory[1] to a few hundred processors that employ a more sophisticated interconnection network between processors and memory.[2] Current distributed memory MIMD computers have from the low hundreds to a thousand processors,[3] while the number of fine-grained processors in current single instruction stream, multiple data stream (SIMD) machines is in the tens of thousands.[4] Hybrids of approaches are also appearing, such as architectures that combine both the MIMD and SIMD approach [Dun90] and the use of a pool of processors by a system of workstations to perform tasks requiring multiple processors [MT86]. Even a wide-area network of heterogeneous workstations may be viewed as a single distributed computing system [BST89].

A paramount question is how such diverse computers, and computers resulting from additional evolution, should be programmed. As always with software, program correctness is a key issue. Other significant considerations include support for abstraction, the cost of software development and maintenance, ease of portability, and, when translation is by compilation, the efficiency of the compiler and the efficiency of the resulting object code.

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 machines, processor trees on MIMD hypercubes, and shifts or pointer doubling on SIMD machines. 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.[5] Support for using less trivial programmer-defined associative functions in computing reductions of vectors is also provided in several languages for parallel computing, such as iPSC/2 Fortran and C [Int89] and the innovative and less conventional languages Connection Machine Lisp [SH86] and Paralation Lisp [Sab88]. Within conventional semantic models, the level of abstraction provided by such general reduction operators is less than ideal when nontrivial programmer-defined functions are used. This is due to such deficiencies as 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

[1] e.g., Sequent Symmetry, Encore Multimax
[2] e.g., BBN Butterfly, IBM RP3
[3] e.g., Intel iPSC/2, NCube/10
[4] e.g., Connection Machine, MassPar MP-1
[5] Language support for reductions was provided much earlier by APL [Ive62].

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 compilers and syntax-directed editors 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. 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. Finally, additional applications are possible if the associativity requirement is generalized to a requirement that the function simply be associative relative to the sequences of interest; this generalization is also efficiently implementable, yet such a more general requirement is even less natural to include in the specification of the function.

A leading strategy for programming both sequential and parallel computers is the divide-and-conquer approach, 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 paper a restricted form of divide-and-conquer sequential algorithm is shown to have the major implementation advantages of general reduction operators, due to an intimate relationship between such an algorithm and associativity. 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. In addition, unlike general reduction operators, such an an algorithm can be used to assign computed values to the components of a vector.

2 THE NONDETERMINISTIC DIVIDE

Turning to the foreigner who wanted to be his new aide, the king remarked: "Your trial task is to devise a strategy that my workers can use to find the exact weight of wood in this worn-out scepter, since I plan to replace it as quickly as possible with a gold one having precisely the same weight. Built into niches of each room of this castle are balancing scales sensitive enough to weigh at least one small diamond. I will not tell you their various capacities, except to say that this scepter greatly exceeds the capacity of any single scale."

Immediately the candidate responded: "I was given a similar task in my previous homeland and the strategy I developed there will work here as well: Break the scepter into two pieces. Each piece is now like the original wood, so keep breaking wood into two pieces until each piece of wood can be weighed on a scale. Add all the weights obtained. There is no need for me to specify where each piece of wood is to 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."

The *nondeterministic divide* partitions a vector into two nonempty slices[6] by allowing the point of division to be chosen nondeterministically; that is, the point of division is not specified by the language or the programmer. 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 components is only permitted during activations in which the vector parameters have unit length.

The following ideas form the basis for this paper:

1. The semantics of a diva algorithm are natural and easy for the sequential programmer to understand.

2. An algorithm that has been demonstrated to be diva can be executed in time proportional to $\log n$, where n is the length of the vectors used as input by the algorithm, if the computation within a single activation takes constant time and parameter passing takes constant time. This can be accomplished by implementing the nondeterministic divide as a divide at the middle of a vector and letting multiple processors use a binary combining tree to share in the computation of the algorithm.

3. The nondeterministic divide can be implemented as a divide just before the last component of a vector (or just after the first component) thereby

[6] A slice of a vector is a sequence of consecutive components of the vector [Uni83].

yielding a sequential loop. Such a loop is often more efficient than the use of recursion to implement a diva algorithm on a sequential computer. Thus during a single use of a diva algorithm, some of the recursive activations – such as those near the beginning of execution of the algorithm – can be carried out in parallel and the rest can be implemented sequentially without the overhead of recursive calls. This facilitates an implementation in which processors do some of the work on the algorithm independently and then combine their partial results. If the divide were defined to be a deterministic divide at the middle, an implementation using a sequential loop (without a stack) would not be possible, in general.

4. By requiring that a programmer use restricted syntax, which we refer to as a *diva call*, to describe a diva algorithm, the demonstration mentioned in 2 can be performed by a syntax-directed editor or a compiler.

5. The nondeterministic divide is fully consistent with the strong induction naturally used in verifying a recursive algorithm. For this reason diva algorithms are relatively easy to verify.

6. The associativity of a function can be proven to be equivalent to the assertion that a diva call naturally related to the function returns a unique value. As mentioned earlier, the associativity of a function is intimately related to the efficiency of implementing operations involving the function on parallel computers.

7. Additional applications are possible if the associativity requirement is weakened to a requirement that the function be associative relative to the sequences of interest. This generalization is also efficiently implementable, and is equivalent to the assertion that a diva call naturally related to the function returns a unique value when the vectors used as input by the diva call satisfy certain properties.

8. Since there are $n - 1$ choices for division point in an n-element vector, the nondeterministic divide can be implemented so that tolerance for faults in channels on a parallel computer is provided.

9. Using a natural choice of syntax for a diva algorithm has the effect of abstracting away issues that are irrelevant to the logical correctness of the algorithm in question. Such issues include whether a loop will be used for sequential implementation, whether parallelism will be used to implement the algorithm and, if so, whether distributed or shared memory will be used, whether the target computer is MIMD or SIMD, whether processors will do some of the combining work independently before combining their partial results, and what technique will be used in combining partial results from different processors (a combining tree of processors, left or right shifts, pointer doubling, or other technique). As the parable of the scales

suggests, such issues also include whether the parallel computer used has heterogeneous processors and, if so, what their respective capabilities are. The resulting high-level syntax results in code that is more portable and easier to read, write, and verify than approaches failing to abstract away such details.

10. Diva calls can be used to perform a variety of computations. A few of the many examples discussed in this paper are: assigning computed values to a vector using corresponding values of other vectors, computing a reduction of a vector using an associative operation, computing the number of runs in a vector, computing the maximum sum among the nonempty slices of a vector, computing the value of a component of one vector corresponding to the maximum of another vector, computing the length of the longest identical corresponding slice of two vectors, computing the first record, in a vector of records, whose fields have the most number of matches with the corresponding fields of a key, and sorting a vector using merge sort. On the other hand, quicksort is an example of a divide-and-conquer strategy that does not have a natural treatment using diva procedures, since the division of a vector by quicksort occurs only after some initial processing of the vector.

The development of diva procedures arose naturally from a project to implement multiway rendezvous activities in log time whenever possible on suitable parallel computers [Cha86, Cha87, Cha89]. Often when a diva call appears within the sequential code of a multiway rendezvous, it is possible to use the processes participating in the multiway rendezvous to compute the results of the diva call, rather than activating new processes. This idea has been encorporated into a translator for a language, Adam, supporting the use of the multiway rendezvous [Cha90a].

The organization of the rest of the paper is as follows: Compiler-enforceable syntax for a diva algorithm, in terms of diva calls, is given in Section 3. Examples of diva calls are given in Section 4 and informal verification of diva calls is illustrated in Section 5. Section 6 contains a characterization of associativity (and of associativity relative to a sequence) in terms of certain diva procedure calls. Implementing diva calls on sequential computers is considered in Section 7, and Section 8 considers the implementation of diva calls on parallel computers. A comparison of using diva procedures with using general reduction operators is discussed in Section 9 and conclusions are given in Section 10. The results of this paper 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.

For concreteness a particular syntax is given in this paper to express diva algorithms. However, the purpose of this paper is to propose and investigate a programming language *concept*, the nondeterministic divide, rather than any

particular *construct* to support the concept.[7] Ada is used as the base language in this paper, but alternate syntax for diva algorithms compatible with any one of numerous modern programming languages could be designed. Ada is used as the base language because the examples given can then be both general and simple, since programmer-accessible attributes (such as `LENGTH` and `FIRST`) and generics are available. In addition, most readers either will be familiar with Ada or have ready access to an Ada reference, such as [Bar84] and [Uni83]. In view of Ada's existing complexity, the author is not recommending that Ada be extended with syntax to support diva algorithms. Furthermore, a construct should be fully integrated within a language during the initial design of the language, since retrofitting a construct to an existing language may require some awkward restrictions to avoid undesirable interactions with other features of the language.

[7] The distinction between a language concept and a language construct can be illustrated by the concept of a definite iteration loop, which is supported by a variety of constructs in programming languages, which differ, among other ways, in restrictions placed on the choice of index type and step size.

3 DIVA CALLS

Diva algorithms need to be formulated in terms of a programming language construct in order to accomplish several of the goals of this paper, such as describing an algorithm that translates the diva approach into programs and making it clear that a compiler could demonstrate adherence to the diva approach.

The construct we use to represent diva algorithms is a procedure call. Procedure calls, rather than function calls, are used because procedure calls are more general than side-effect-free functions. Permitting diva function calls would also be useful in an actual programming language. However, defining and using diva function calls is a straightforward extension of this study, and since the concern of this paper is not with syntax *per se*, no further consideration will be given to diva function calls in the paper.

Diva calls will be defined as restricted calls on diva procedures [Cha89].

DEFINITION 1. A *diva procedure* `P` is an Ada procedure, except that `diva` precedes `procedure` in the declaration, restrictions are placed on the ordering and kind of formal parameters, and the body of the procedure must be a single `if` statement in a restricted form of Ada as indicated in Figure 1, where `<mode>` is one of `in`, `out`, or `in out`, and square brackets enclose an optional unit. Except for type names, access to nonlocals from within `P` is prohibited; thus objects used within `P`, including subprograms, must be declared within `P`. Any choice of identifiers used in `P` that is different from the reserved words is possible; of course, the identifiers must be distinct, except for the type identifiers. Two new attributes, `INITIAL` and `FINAL`, are defined by the implementation for each `A1`, ..., `Am` in an activation of a diva procedure, access to these attributes is only permitted inside the two recursive calls on the diva procedure, and the attributes satisfy the following conditions for each `i` and `j` between *1* and *m*:

1. `Ai` equals `Ai'INITIAL` concatenated with `Ai'FINAL`,
2. both `Ai'INITIAL` and `Ai'FINAL` have nonempty ranges, and
3. the range of `Ai'INITIAL` equals the range of `Aj'INITIAL` and the range of `Ai'FINAL` equals the range of `Aj'FINAL`. □

Notice that the syntax rules of a diva procedure are compiler-enforceable. The requirement that subprograms used within `P` be declared within `P` permits a compiler to disallow additional recursive calls on `P` as a result of executing this Ada code and helps to ensure that the recursive calls are made before any computation. The syntax does not permit the programmer to specify the choice of division point used for dividing `Ai` into `Ai'INITIAL` and `Ai'FINAL`; thus the choice of division point is determined by the implementation.

The term "dynamic parameter" will be used in this paper to refer to the vectors `A1`, ..., `Am`. Ada requires that each formal parameter be declared using

```
diva procedure P
 (A1: <mode> array(INTEGER range <>) of BASE_TYPE1;  ...
  Am: <mode> array(INTEGER range <>) of BASE_TYPEm
  [ ; IN_PARM: in IN_PARM_TYPE ]
  [ ; PARM: [ in ] out PARM_TYPE ] ) is

  [ INITIAL_PARM, FINAL_PARM: PARM_TYPE; ]
  ... Other local declarations, not containing any access to A1, ..., Am.

begin
  if A1'LENGTH = 1 then
    ... Ada code not using an attribute of A1, .., Am other than A1'FIRST.
  else
    P(A1'INITIAL, ..., Am'INITIAL [, IN_PARM ] [, INITIAL_PARM ]);
    P(A1'FINAL,   ..., Am'FINAL   [, IN_PARM ] [, FINAL_PARM   ]);
    ... Ada code not containing any access to A1, ..., Am.
  end if;
end P;
```

Figure 1: Syntax for a Diva Procedure

a named type. Declaring the dynamic parameters using an anonymous type is for simplicity of exposition in this paper and, in fact, when diva procedures are translated into Ada in sections 7 and 8, named types (with unconstrained ranges) are used. For the nonrecursive call on `P` the unconstrained range of each dynamic parameter in a diva procedure is bound to a constrained range in the usual manner for Ada; for the recursive calls, the underlying implementation supplies the constrained range, ensuring that conditions 1) through 3) of Definition 1 are satisfied. Note that the intuitive meaning of condition 3) is that the same division point is used for multiple dynamic parameters.

The restriction to just two non-dynamic parameters, which often must be records, simplifies the exposition of this paper and should be removed in a construct designed for a programming language due to the awkwardness sometimes caused by having to use records. Since our concern is not with syntax *per se*, providing a context-free grammar to describe less restrictive syntax is not needed in this paper and would only add unnecessary complexity to the description of the translation algorithms in sections 7 and 8. More restrictive syntax that would permit only a single dynamic parameter is also possible, but such a restriction could give rise to significant space inefficiency, as discussed at the end of Section 4. Additional rationale for the syntax restrictions is given at the end of sections 8.1 and 8.2.

NOTE: The keyword `diva` supports a one-pass compiler, makes such a procedure easier to read and understand, and supports the use of a syntax directed

editor; thus the programmer need not be burdened with syntax rules. (This keyword is otherwise unnecessary, since a compiler can recognize the intent of using a diva procedure by the use of the `INITIAL` and `FINAL` attributes.) One way to choose syntax that does not rely on the base language having an analog to programmer-accessible attributes would be to require that the false branch of the `if` have the form

```
divide
  A1 into INITIAL1 and FINAL1;
  ...
  Am into INITIALm and FINALm;
recur
  P (INITIAL1, ..., INITIALm [, IN_PARM] [, INITIAL_PARM]);
  P (FINAL1,   ..., FINALm   [, IN_PARM] [, FINAL_PARM]);
conquer
  ...
```

The programmer would not declare the `INITIALi`'s and `FINALi`'s; these declarations would occur implicitly in the `divide` block and the names thereby declared could only be used in the `divide` and `recur` blocks.

DEFINITION 2. A *diva call* is a call on a diva procedure `P` such that all actual arguments used in the call that correspond to dynamic parameters have the same ranges. □

The reason for this condition (and the reason for formulating diva algorithms as procedure calls rather than as procedures) is to ensure that the same division point can be used for all of the dynamic parameters during any given recursive activation of the procedure resulting from the call. The form of diva procedures defined by Definition 1 can be checked at compile-time but the condition in Definition 2 cannot be checked until run-time for actual arguments whose range is unknown at compile-time.[8]

[8] An alternate definition of diva procedure would permit this condition to be satisfied automatically: the alternate definition would be like Definition 1 except that the single range of the dynamic parameters would be a generic parameter.

4 EXAMPLES

The following standard guidelines for writing procedures are assumed in selecting and presenting examples in this section:

1. A procedure `P` should be enclosed in an encapsulating procedure whenever either (a) results need to be computed at the end of calls on `P` and the user of `P` does not need to know how the results are computed or (b) `P` uses auxiliary parameters that do not need to be known by the user of `P`.
2. Whenever writing recursive procedures or loops, programmers should avoid having intermediate results computed if there is a way to compute such results just once at the end.

Since writing encapsulating procedures is straightforward, for brevity such procedures are not included in this paper.

EXAMPLE 1: *Compute the + reduction of a vector.* The sum of the elements of an integer vector `X` can be assigned to an integer variable `TOTAL` using the call

```
FIND_SUM (X, TOTAL);
```

where `FIND_SUM` is declared in Figure 2. □

If the values of vector components need to be modified before a reduction takes place, this can be accomplished within the true branch of the `if`; for example, the sum of the squares of components of `A` can be obtained by rewriting the assignment in the true branch of `FIND_SUM` as

```
SUM := A(A'FIRST) ** 2;
```

An even simpler use of diva procedures is to use the true branch of the `if` to assign values to an array `A` and to let the false branch of the `if` contain only the recursive calls. For example, a diva procedure `ASSIGN_SQUARES` could assign the squares of the index values of `A` to `A`; the `if` statement would be simply

```
if A'LENGTH = 1 then
  A(A'FIRST) := A'FIRST ** 2;
else
  ASSIGN_SQUARES (A'INITIAL);
  ASSIGN_SQUARES (A'FINAL);
end if;
```

Similarly values could be assigned to each component of one or more vectors, based on computations involving the index of the component, possibly including the use of corresponding values of other vectors.

Diva procedures are compatible with Ada generics. For example, a generic diva procedure `REDUCE` with parameters `BASE_TYPE` and `FUNCT` could be defined very much like `FIND_SUM`, so as to support all reductions: After the recursive calls on `REDUCE`, the assignment

```
diva procedure FIND_SUM (A: in array(INTEGER range <>) of INTEGER;
                         SUM: out INTEGER) is
  -- SUM is assigned the + reduction of A
  INITIAL_SUM, FINAL_SUM: INTEGER;
begin
  if A'LENGTH = 1 then
    SUM := A(A'FIRST);
  else
    FIND_SUM (A'INITIAL, INITIAL_SUM);
    FIND_SUM (A'FINAL,   FINAL_SUM);
    SUM := INITIAL_SUM + FINAL_SUM;
  end if;
end FIND_SUM;
```

Figure 2: A Diva Procedure to Find the + Reduction

```
ANSWER := FUNCT (INITIAL_ANSWER, FINAL_ANSWER);
```

would appear. The function `F` corresponding to the parameter `FUNCT` of `REDUCE` clearly does not need to be commutative, since the first argument value always corresponds to lower subscripts of the vector `A` than the second argument value. This supports finding reductions using noncommutative operations, such as matrix multiplication. `F` need only be an associative, binary operation; i.e. `F(X,F(Y,Z)) = F(F(X,Y),Z)` for all `X`, `Y`, and `Z` in the domain of `F`. Commonly used examples of such functions include "+", "*", "and", "or", and user-defined `MAX` and `MIN`.

The next example uses the fact that the two slices of a vector produced by the nondeterministic divide are adjacent to each other. The number of runs in a sequence is used in a testing the randomness of the sequence.[Hoe62]

EXAMPLE 2: *Compute the number of runs in a sequence.* The number of runs in a finite sequence is one greater than the number of times a term appears in the sequence that is larger than the next term in the sequence. Thus the number of runs in the sequence `S` can be computed as follows, where `ANSWER` is a record consisting of `INTEGER` fields `COUNT`, `FIRST`, and `LAST`:

```
COUNT_RUNS (S, ANSWER);
NUMBER_RUNS := ANSWER.COUNT + 1;
```

where `COUNT_RUNS` is declared in Figure 3. □

```
diva procedure COUNT_RUNS (S: in array(INTEGER range <>) of INTEGER;
                           ANS: out ANSWER_TYPE) is
  -- ANS.COUNT is assigned the number of times a component of S is
  --   greater than the next component of S.
  -- ANS.FIRST is assigned the value of the first component of S.
  -- ANS.LAST is assigned the value of the last component of S.
  L, R: ANSWER_TYPE;  -- results of left and right slices
begin
  if S'LENGTH = 1 then
    ANS.COUNT := 0;
    ANS.FIRST := S(S'FIRST);
    ANS.LAST  := S(S'FIRST);
  else
    COUNT_RUNS (S'INITIAL, L);
    COUNT_RUNS (S'FINAL,   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 COUNT_RUNS;
```

Figure 3: A Diva Procedure to Help Find the Number of Runs in a Sequence

The false branch of the outer `if` statement in many diva procedures, such as `COUNT_RUNS`, could be programmed more simply as just the two recursive calls followed by a list of assignment statements, given the existence of conditional expressions such as those permitted in Algol-60 and C. In fact, this is true for all diva procedures in this paper for which code is given.

Notice that `ANS.FIRST` and `ANS.LAST` in `COUNT_RUNS` are used to preserve information about the 2-element overlap between adjacent slices of the sequence. A similar diva procedure can be used to count the number of peaks in a sequence of distinct terms, such as identification numbers; a term of the sequence is a "peak" if it is greater than both its predecessor and successor. Information about the 4-element overlap between adjacent slices of the sequence needs to be preserved in order to examine each consecutive triple of terms in the sequence. Such a diva procedure is related to finding the number of alternating runs [Knu73] in a sequence. The next example also illustrates the use of overlap information. This problem is found in [Gri81].

EXAMPLE 3: *Find the length of the longest plateau of a nondecreasing sequence.* A "plateau" is a slice all of whose components have equal values. The length `LENGTH` of the longest plateau of a nondecreasing sequence `S` can be computed as follows, where `ANSWER` is a record consisting of integer fields `LEN`, `FIRST_LEN`, `LAST_LEN`, `FIRST`, and `LAST`:

```
FIND_LENGTH (S, ANSWER);
LENGTH := ANSWER.LEN;
```

where `FIND_LENGTH` is declared in Figure 4. □

A diva procedure similar to `FIND_LENGTH` can be written to compute the maximum sum among the nonempty slices of a given vector of integers. For this example `ANS` should have four fields, which are used to return: the maximum sum among the nonempty slices of `A`, the sum of all component values of `A`, the maximum sum among the nonempty slices of `A` that start at the first component of A, and the maximum sum among the nonempty slices of `A` that end at the last component of A. This problem and its history are the focus of the chapter on algorithm design techniques in [Ben86][9]. The feasibility of using a diva procedure to solve this problem was suggested by Dana Richards.

EXAMPLE 4: *Find a component value of one vector corresponding to the maximum of another vector.* The component value `Y_VALUE` of an integer vector `Y` corresponding to the first position where the maximum of a floating point vector `X` occurs can be computed as follows, where `ANSWER` is a record consisting of a floating point field `MAX` and an integer field `CORR`:

```
FIND_MAX_CORR (X, Y, ANSWER);
Y_VALUE := ANSWER.CORR;
```

where `FIND_MAX_CORR` is declared in Figure 5. □

A diva procedure `FIND_MIN_CORR`, like `FIND_MAX_CORR` except that minimum plays the role of maximum, can be written with few changes. Both `FIND_MAX_CORR` and `FIND_MIN_CORR` support programming the farthest insertion heuristic algorithm for the Euclidean traveling salesman problem [RSL74]. (A node occurring at the maximum distance from the nodes of the current partial tour is the next node chosen to add to the partial tour, and to add this node most cheaply, it is necessary to find a node in the current partial tour occurring at the minimum distance from this chosen node.) Another example of the use of `FIND_MIN_CORR` is in programming the Prim-Dijkstra algorithm for finding

[9] Actually empty slices are included in the statement of the problem in [Ben86] so the desired answer when all component values of the vector are negative is zero rather than the largest value of the vector. The solution to this variant of the problem can be obtained by taking the maximum of the maximum sum and zero upon return from the diva call.

```
diva procedure FIND_LENGTH (A: in DYNAMIC_VECTOR;
                            ANS: ANSWER_TYPE) is
  -- ANS.LEN is the length of the longest plateau in A.
  -- ANS.FIRST_LEN is the length of the longest plateau in A
  --   that starts at the first component of A.
  -- ANS.LAST_LEN is the length of the longest plateau in A
  --   that ends at the last component of A.
  -- ANS.FIRST is the value of the first component of A.
  -- ANS.LAST is the value of the last component of A.

  L, R: ANSWER_TYPE;  -- results of left and right slices
  ... declaration of MAX function

begin
  if A'LENGTH = 1 then
    ANS.LEN       := 1;
    ANS.FIRST_LEN := 1;
    ANS.LAST_LEN  := 1;
    ANS.FIRST     := A(A'FIRST);
    ANS.LAST      := A(A'FIRST);
  else
    FIND_LENGTH (A'INITIAL, L);
    FIND_LENGTH (A'FINAL,   R);
    if L.LAST = R.FIRST then
      ANS.LEN := MAX (MAX (L.LEN, R.LEN), L.LAST_LEN + R.FIRST_LEN);
    else
      ANS.LEN := MAX (L.LEN, R.LEN);
    end if;
    if L.FIRST = R.FIRST then
      ANS.FIRST_LEN := L.LEN + R.FIRST_LEN;
    else
      ANS.FIRST_LEN := L.FIRST_LEN;
    end if;
    if L.LAST = R.LAST then
      ANS.LAST_LEN := L.LAST_LEN + R.LEN;
    else
      ANS.LAST_LEN := R.LAST_LEN;
    end if;
    ANS.FIRST := L.FIRST;
    ANS.LAST  := R.LAST;
  end if;
end FIND_LENGTH;
```

Figure 4: A Diva Procedure To Find The Length of the Longest Plateau

```
diva procedure FIND_MAX_CORR (A: in array(INTEGER range <>) of FLOAT;
                              B: in array(INTEGER range <>) of INTEGER;
                              CHOSEN: out ANSWER_TYPE) is
  -- CHOSEN.MAX is assigned the maximum of A and CHOSEN.CORR is assigned
  -- the value of the component of B corresponding to the first
  -- component of A equal to CHOSEN.MAX.
  L, R: ANSWER_TYPE; -- results of left and right slices
begin
  if A'LENGTH = 1 then
    CHOSEN.MAX  := A(A'FIRST);
    CHOSEN.CORR := B(A'FIRST);
  else
    FIND_MAX_CORR (A'INITIAL, B'INITIAL, L);
    FIND_MAX_CORR (A'FINAL,   B'FINAL,   R);
    if L.MAX >= R.MAX then
      CHOSEN := L;
    else
      CHOSEN := R;
    end if;
  end if;
end FIND_MAX_CORR;
```

Figure 5: A Diva Procedure To Find A Value Corresponding to the Maximum of a Vector

a minimum spanning tree [Pri57, Dij59]. (A node occurring at the minimum distance from the nodes of a partial spanning tree is iteratively added to the partial spanning tree.)

A straightforward modification of the diva procedure `FIND_MAX_CORR` yields a single generic diva procedure supporting both `FIND_MAX_CORR` and `FIND_MIN_CORR`: A functional parameter `FUNCT` would be used along with the use of the test

```
L.A = FUNCT (L.A, R.A)
```

in comparing `L.A` and `R.A`. A function corresponding to `FUNCT` would be required to be an associative, binary operation that returns the value of one of its two arguments as its result. There are numerous such functions, in addition to the maximum and minimum functions. For example, `FUNCT (X, Y)` could return the value of `Y` if `X` has zero value and return the value of `X` otherwise; when instantiated with this function, the generic diva procedure would yield a diva procedure that returns the value of the component of one vector `B` corresponding to the first nonzero component value of another vector `A` (and returns the last component value of `B` if all component values of `A` equal zero). However, such a generic diva procedure should be used judiciously. It follows from the observations in Section 6 that program correctness is much easier to maintain if a separate diva procedure (not having a functional parameter) is written when the function corresponding to `FUNCT` would be nontrivial.

It is easy to see that a diva procedure similar to `FIND_MAX_CORR`, but using only a single dynamic parameter, can be written to compute the index of the first component of an array whose value satisfies a particular property, such as being maximal, minimal, nonzero, or positive. Of course, if desired, the index of the last such component could be found instead.

None of the examples of diva procedures given so far uses the fact that a diva procedure can have a scalar parameter of mode `in`. An example is a diva procedure `LOOKUP`, which uses the `in` parameter to store a search key for an unordered table lookup. `LOOKUP`, which can be obtained as a straightforward modification of `FIND_MAX_CORR`, has two dynamic `in` parameters `A` and `B`, an `in` parameter `KEY`, and an `out` parameter `ANS` having two fields `FOUND` and `VALUE`. The diva procedure `LOOKUP` sets `ANS.FOUND` to true if and only if `KEY` equals the value of a component of `A`; if `ANS.FOUND`, then `ANS.VALUE` equals the component value of `B` corresponding to the first such component value of `A`. There are many natural variants of this example, some of which involve an associative search. One, suggested by Dana Richards, is the retrieval of the first record, in a vector of records, whose fields have the most number of matches with the corresponding fields of a key.

A diva procedure can be written to compute the length of the longest identical corresponding slice of two (or any predefined number of) given vectors. The overlap between two slices can be considered without reexamining the contents of the overlap by using parameters that keep track of the length of the longest

such slice, the length of the longest such slice starting at the first component, and the length of the longest such slice ending at the last component.

Other simple applications of using more than a single vector in a diva algorithm include computing the inner product of two vectors, computing the square of the distance between two points in n-dimensional Euclidean space, and computing the Hamming distance between two vectors of bits. It is easy to see that the merge sort, but not quicksort, can be written as a diva call. The processing of a vector performed by quicksort before making recursive calls is not permitted within a diva procedure. Unlike earlier examples, the vector corresponding to the non-dynamic parameter containing the computed answer in the diva call on a merge sort diva procedure must have the same length as the length of the dynamic parameter. The merge sort will be considered further in Section 7.

Before going on to the next section, we explain why diva procedures are permitted to have more than a single dynamic parameter. Consider a situation in which `X` and `Y` in Example 4 are used elsewhere in the program in calls on diva procedures having one, but not both, of `X` and `Y` as actual arguments corresponding to dynamic parameters and possibly having additional dynamic parameters. If only a single dynamic parameter were permitted in diva procedures, such diva procedures could be written by defining a vector `Z` whose components are records containing a field for each of the corresponding components of: the vectors `X` and `Y` and each vector used in any diva call with either `X` or `Y`. The number of fields existing in each component of `Z` would be static, whereas the number of fields needed in different parts of the program would vary. Diva calls are most useful when the range of such vectors is large. Since parameter passing for array parameters may be implemented by copy [Uni83, 6.2.7], the Ada programmer will realize that it could be quite costly to use `Z` as an actual argument when not all fields are needed.[10] An alternative in this situation would be to define several different vectors of records for different portions of the program, but since some of the fields in the records of such vectors would contain the same information, this alternative would require copying information from one such vector of records to another. On the other hand, the fact that diva procedures can have several dynamic parameters permits space utilization to be appropriate to a particular usage, within a conventional semantic model, including the use of a vector of records when appropriate. This storage advantage is also provided by the "fields" of a paralation [Sab88]: such "fields" can be added and deleted according to the needs of different parts of the program. The semantics of the noteworthy paralation model are much richer and more powerful than conventional semantics.

[10] Similarly, the implementation of diva calls involving `Z` on a distributed memory computer could be quite wasteful of storage space and/or communiciation time.

5 VERIFICATION OF DIVA CALLS

In view of the role recursion plays in diva procedures, the natural way to verify such a procedure is to use strong induction on the length of the dynamic parameters in the call. This approach is illustrated by the following informal proof that the diva procedure in Example 1 assigns to `SUM` the + reduction of `A`.

NOTE: The + reduction of an array having a single component is defined to be the value of the single component.

`INFORMAL VERIFICATION OF FIND_SUM:`

`case one:` `A'LENGTH = 1`. Since `SUM` is assigned the value of the single component of `A`, `SUM` is assigned the + reduction of `A`.

`case two:` `A'LENGTH > 1`. By 1) and 2) of Definition 1, the length of both `A'INITIAL` and `A'FINAL` is less than the length of `A` so we may apply the induction hypothesis and conclude that, after the two recursive calls, `INITIAL_SUM` is assigned the + reduction of `A'INITIAL` and `FINAL_SUM` is assigned the + reduction of `A'FINAL`. By 1) of Definition 1, the values of the components of `A` are the values of the components of `A'INITIAL` followed by the values of the components of `A'FINAL`. It is thus clear that `SUM` is assigned the + reduction of `A`. □

The only reason verifying other diva procedures is more complex is due to the use of more complex parameters and more complex Ada in the true branch and false branch (after the recursive calls) of the `if`, rather than to any additional complexity of the diva concept itself. For example, here is a sketch of an informal proof of the diva procedure `FIND_LENGTH` of Example 3.

INFORMAL VERIFICATION OF FIND_LENGTH:

case one: A'LENGTH = 1. The value of the single component of A is easily seen to satisfy the specified value of each field of ANS.

case two: A'LENGTH > 1. By 1) and 2) of Definition 1, the length of both A'INITIAL and A'FINAL is less than the length of A so we may apply the induction hypothesis and conclude that, after the two recursive calls:

- L.LEN is the length of the longest plateau in A'INITIAL.
- L.FIRST_LEN is the length of the longest plateau in A'INITIAL that starts at the first component of A'INITIAL.
- L.LAST_LEN is the length of the longest plateau in A'INITIAL that ends at the last component of A'INITIAL.
- L.FIRST is the value of the first component of A'INITIAL.
- L.LAST is the value of the last component of A'INITIAL.

and

- R.LEN is the length of the longest plateau in A'FINAL.
- R.FIRST_LEN is the length of the longest plateau in A'FINAL that starts at the first component of A'FINAL.
- R.LAST_LEN is the length of the longest plateau in A'FINAL that ends at the last component of A'FINAL.
- R.FIRST is the value of the first component of A'FINAL.
- R.LAST is the value of the last component of A'FINAL.

By 1) of Definition 1, the values of the components of A are the values of the components of A'INITIAL followed by the values of the components of A'FINAL. If L.LAST = R.FIRST, then L.LAST_LEN + R.FIRST_LEN is the length of a plateau in A so ANS.LEN = max (max (L.LEN, R.LEN), L.LAST_LEN + R.FIRST_LEN) is the length of the longest plateau in A; otherwise no plateau extends from the end of A'INITIAL to the beginning of A'FINAL, so ANS.LEN = max (L.LEN, R.LEN) is the length of the longest plateau in A. If L.FIRST = R.FIRST, then a plateau in A extends from the beginning of A'INITIAL into A'FINAL so ANS.FIRST_LEN = L.LEN + R.FIRST_LEN is the length of the longest plateau starting at the first component of A; otherwise no plateau extends from the beginning of A'INITIAL into A'FINAL so ANS.FIRST_LEN = L.FIRST_LEN is the length of the longest plateau starting at the first component of A. If L.LAST = R.LAST, then a plateau extends from the end of A'INITIAL to the end of A'FINAL so ANS.LAST = L.LAST_LEN + R.LEN is the length of the longest plateau ending at the last component of A; otherwise no

plateau extends from the end of `A'INITIAL` to the end of `A'FINAL` so `ANS.LAST` = `R.LAST_LEN` is the length of the longest plateau ending at the last component of `A`. Finally, it is clear that `ANS.FIRST` and `ANS.LAST` are assigned the specified values. □

Note that 6 cases are considered in verifying the code in the false branch of the `if` of `FIND_LENGTH` after the recursive calls. We shall return to this observation in the next section. The reader is encouraged to provide similar proofs for the diva procedures given in Examples 2 and 4; the latter will need to use 3) of Definition 1 as well as 1) and 2).

Notice that the inability of assuming anything about the choice of division points does not make these proofs more complicated, since the nondeterministic divide is fully compatible with such use of strong induction.

6 DIVA PROCEDURES AND ASSOCIATIVITY

As mentioned in the introduction, associative operations are well-known to be efficiently implementable on a variety of parallel computers. This section shows how diva procedures are intimately related to associative operations. To support additional applications, associativity is weakened to associativity relative to a sequence. Operations that are associative relative to a sequence are also efficiently implementable and diva procedures are shown to be intimately related to such operations as well.

The section concludes by showing how the use of diva procedures can be simpler than using a general reduction operator to combine the results of a programmer-defined function.

6.1 Characterizing Associativity

Throughout this paper the notation $< x_1, \cdots, x_n >$ denotes a sequence of n elements and $\circ$ denotes string concatenation. By a "slice" of a sequence we shall mean a sequence of consecutive terms of the given sequence.

DEFINITION 3. Let s be a finite sequence of elements of E and let f be a function from $E \times E$ to E. The *f-reduction* of the sequence $< x > \circ s$, denoted $f_r(< x > \circ s)$, is defined to equal x if s is the null string and to equal $f(x, f_r(s))$ otherwise.

DEFINITION 4. Let s be a sequence of elements of E and let f be a function from $E \times E$ to E. To say that f is *associative relative to* s means that $f(f_r(s_1), f(f_r(s_2), f_r(s_3))) = f(f(f_r(s_1), f_r(s_2)), f_r(s_3))$ holds for any nonnull finite sequences s_1, s_2, and s_3 such that $s_1 \circ s_2 \circ s_3$ is a slice of s.

The concept of associativity relative to a sequence does not depend on the arbitrary convention of grouping from the right and is a strict generalization both of associativity and of associativity relative to consecutive triples of terms of a sequence. In fact, the function that would map `L` and `R` into `ANS` in Figure 4 is a nonassociative function that is associative relative to any sequence of records of type `ANSWER_TYPE` corresponding to a nondecreasing sequence of integers, using the natural correspondence defined in the true branch of the `if` in Figure 4. The results stated in this paragraph as well as the next two theorems are proven in [Cha90b].

RELATIVE ASSOCIATIVITY CHARACTERIZATION THEOREM. *Let u be a nonnull finite sequence of elements of a set E, let f be a function from $E \times E$ to E, let S be the set of slices of u, and let R denote the smallest subset of $S \times E$ such that*

(a) *for each term x of u, R contains the pair $(< x >, x)$ and*

(b) *for each s in S, R contains each pair $(s, f(t', t''))$, where (s', t') and (s'', t'') are in R and s is s' concatenated with s''.*

Then the following statements are equivalent:

1. *f is associative relative to u.*
2. *R is a function from S to E.*
3. *R is a function from S to E that maps each s in S to $f_r(s)$.*□

ASSOCIATIVITY CHARACTERIZATION THEOREM. *Let S denote the set of nonnull finite sequences of elements of a set E, let f be a function from $E \times E$ to E, and let R denote the smallest subset of $S \times E$ such that*

(a) *for each x in E, R contains the pair $(< x >, x)$ and*

(b) *for each s in S, R contains each pair $(s, f(t', t''))$, where (s', t') and (s'', t'') are in R and s is s' concatenated with s''.*

Then the following statements are equivalent:

1. *f is associative.*
2. *R is a function from S to E.*
3. *R is a function from S to E that maps each s in S to $f_r(s)$.*□

The following consequence of the characterization theorems describes an intimate relationship between diva procedures and associativity.

COROLLARY. *Let* `P` *be a diva procedure whose parameter list has the form*

```
A: in array(INTEGER range <>) of BASE_TYPE1;
PARM: [in] out BASE_TYPE2
```

and such that an assignment to `PARM` *is made during each call on* `P`*. Let f denote the function that combines the results from* `A'INITIAL` *and* `A'FINAL` *in the false branch of the* `if` *of* `P` *into* `PARM`*. Then:*

1. *f is associative if and only if the value assigned to* PARM *by a call on* P *is uniquely determined by the value of the actual argument corresponding to* A.

2. *f is associative relative to the sequence of component values of a vector* X *if and only if, for any slice* S *of* X, *the value assigned to* PARM *by a call on* P *with actual argument* S *is uniquely determined by the component values of* S.

PROOF. Proof of (1). It suffices to reduce the proof to the case where BASE_TYPE1 and BASE_TYPE2 are the same and the true branch of the if of P simply assigns A(A'FIRST) to PARM: For ACT can then be applied, letting E be the set of values of type BASE_TYPE2 and letting R be the set of all pairs (X′, Y′), where X′ is the sequence of component values of an array X of base type BASE_TYPE2 and range type a subrange of INTEGER and where Y′ is the value of a variable Y after the call P(X,Y). To reduce the proof to this case, for each possible actual argument X corresponding to the dynamic parameter A of P, let X″ denote the array, having the same range as X and whose base type is BASE_TYPE2, whose component values are obtained by applying the true branch of the if of P to each component of X. Let P″ denote the diva procedure obtained by replacing the first parameter declaration of P by

```
A: in array(INTEGER range <>) of BASE_TYPE2;
```

and by replacing the true branch of the if of P with

```
PARM := A(A'FIRST);
```

Note that the function f in the hypothesis is also the function that combines the results from A'INITIAL and A'FINAL in the false branch of the if of P″ into PARM. Finally note that the value assigned to PARM by a call on P is uniquely determined by the value of X if and only if the value assigned to PARM by a call on P″ is uniquely determined by the value of X″.

Proof of (2). It suffices to reduce the proof to the case where BASE_TYPE1 and BASE_TYPE2 are the same and the true branch of the if of P simply assigns A(A'FIRST) to PARM: For RACT can then be applied, letting E be the set of values of type BASE_TYPE2 and letting R be the set of all pairs (S′, Y′), where S′ is the sequence of component values of a slice S of an array X of base type BASE_TYPE2 and range type a subrange of INTEGER and where Y′ is the value of a variable Y after the call P(S,Y). To reduce the proof to this case, for each slice S of X, let S″ denote the array, having the same range as S and whose base type is BASE_TYPE2, whose component values are obtained by applying the true branch of the if of P to each component of S. Let P″ denote the diva procedure obtained by replacing the first parameter declaration of P by

```
A: in array(INTEGER range <>) of BASE_TYPE2;
```

and by replacing the true branch of the if of P with

```
PARM := A(A'FIRST);
```

Note that the function f in the hypothesis is also the function that combines the results from `A'INITIAL` and `A'FINAL` in the false branch of the `if` of `P''` into `PARM`. Finally note that the value assigned to `PARM` by a call on `P` is uniquely determined by the value of `S` if and only if the value assigned to `PARM` by a call on `P''` is uniquely determined by the value of `S''`. □

It is indeed possible for a diva procedure to satisfy the first sentence of the corollary, yet have a nonassociative combining function f. For a diva procedure `NONDETERM`, having formal parameters `A` and `PARM`, can be written that assigns to fields of `PARM` the length `LEN_INITIAL` of the `INITIAL` slice of the dynamic parameters used in its first recursive call, the length `LEN_FINAL` of the `FINAL` slice used in its second recursive call, and the total length `LEN` of the dynamic parameter `A`, respectively. Clearly the value assigned to `PARM` by a call on `NONDETERM` is *not* always uniquely determined by the actual argument corresponding to `A`, since this value depends on the choice of division points. The diva procedure `NONDETERM` also illustrates the final comment in the third point of Section 2: If the divide in a diva procedure were defined to be a deterministic divide at the middle, then `LEN_INITIAL` and `LEN_FINAL` would be guaranteed to differ by at most 1, so a sequential loop (without a stack) could not be used to implement this procedure for initial dynamic parameters of length 4 or greater. (The fact that through such a diva procedure a programmer can obtain information about the choice of division points made during a particular execution is no violation of nondeterminism, of course.)

For all diva procedures `P` considered in this paper that satisfy the hypothesis of the corollary, with the obvious exception of `NONDETERM` in the preceding paragraph, the uniqueness of the value assigned to `PARM` would immediately follow from the natural specification of `P` so the proof that `P` satisfies its specification would immediately imply that f is associative.

6.2 Proving Associativity

Many applications of diva procedures can be computed using a general reduction operator, instead of using a diva procedure. When a general reduction operator is used to produce a unique value, it is necessary to demonstrate both that the value returned by the function f satisfies the relevant properties and that f is associative (perhaps relative to a sequence). In the remainder of this section we consider how such a proof of associativity can be more complex than the entire proof of the diva procedure.

Let f be the combining function implicit in the false branch of the `if` of a diva procedure `P` after the recursive calls and let p denote the number of cases used in proving the correctness of a single application of f. Note that p can be less than the number of logical paths in the false branch of the `if` of

P. For example, the number of logical paths in the false branch of the `if` of `FIND_LENGTH` is 24, since $(4+2)\times 2\times 2 = 24$, yet p has the value 6 for the proof of correctness of this code segment given in Section 5.

Demonstrating the associativity of f via a straightforward case analysis based on the proof of correctness of a single application of f requires p^4 cases. For, given three records `X`, `Y`, and `Z` of the same type as `PARM`, for each of the p possible relationships between the values of `Y` and `Z`, there are p possible relationships between the resulting computed value and the value of `X` and for each of these p^2 cases there are p^2 similar cases to consider in which the values of `X` and `Y` are combined before a comparison is made with the value of `Z`. Thus when p is 2 or larger, the number of such cases is much smaller when a diva procedure is used than when a general reduction operator is used: p instead of p^4.

Sometimes the number of such cases can be reduced by considering which relationships among the data are actually relevant in view of the code within f. For example, let f be the combining function for the false branch of the diva procedure `COUNT_RUNS` in Example 2. A straightforward proof of correctness of a single application of f would involve 2 cases, so for such a proof, p is 2 and p^4 is 16. However, in determining the effect of $f(\mathtt{Y},\mathtt{Z})$ and then $f(\mathtt{X}, f(\mathtt{Y},\mathtt{Z}))$ and $f(\mathtt{X},\mathtt{Y})$ and then $f(f(\mathtt{X},\mathtt{Y}),\mathtt{Z})$, the conditionals and assignments within f are such that combinations of the following cases are sufficient:

```
variables compared      case 1     case 2
------------------      ------     ------
X.LAST and Y.FIRST        >        otherwise
Y.LAST and Z.FIRST        >        otherwise
```

Thus 4 cases are sufficient, an improvement over p^4 (but not an improvement over p). For applications having more than a trivial number of kinds of conditionals within the logic of f, such as counting the number of peaks in a sequence of distinct terms, such a consideration of relevant relationships requires significant effort that is unnecessary when the diva approach is used and can still result in many more cases in the proof.

Another approach to proving associativity is to rewrite the requirements for the definition of associativity to hold, perhaps using conditional expressions, before performing a case analysis. Such a proof of relative associativity for the combining function f for the false branch of `FIND_LENGTH` appears in the appendix to [Cha90b] and is also significantly more complex than the entire proof of the diva procedure `FIND_LENGTH` given in Section 5.

A more extensive comparison of the use of diva procedures and general reduction operators appears in Section 9.

7 SEQUENTIAL IMPLEMENTATION OF DIVA CALLS

Implementing diva calls on sequential computers is considered in this section and 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 dynamic parameters is n, and assume that n is a power of 2. 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 dynamic parameter range), and a straightforward strong induction argument shows that the number of executions of the false branch is $n - 1$. To see the latter,[11] assume the result holds when the length of the dynamic parameters 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 vectors at the middle of each vector and to make the recursive calls specified in the diva procedure; we shall refer to this option as the *middle option*. Another option is to implement a diva call using a loop, by choosing the division point of vectors just before the end of each vector; we shall refer to this option as the *end option*. Note that the end option is different from tail recursion, although they can both be implemented using a loop, since tail recursion requires that a recursive call occur last in a subprogram.

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 simple theorem illustrates this fact. In the statement of the theorem, the term *conquer block* refers to the portion of the false branch of the `if` of the diva procedure after the recursive calls.

[11] This fact is intuitively obvious, since to break a piece of wood into n pieces requires $n - 1$ vertical division points, regardless of the location of the division points.

THEOREM. *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 additional time required for two recursive calls on* `P` *(beyond that required for a loop). Then choosing division points for the dynamic parameters 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. 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 dynamic parameters 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 i between *1* and n - *1*, let E_i and M_i represent the time required for the ith 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

$$\begin{aligned}
T + \sum_{i=1}^{n-1}(E_i + L_i) &\leq T + M(n-1) + \sum_{i=1}^{n-1} L_i \\
&< T + (m+D)(n-1) + \sum_{i=1}^{n-1} L_i \\
&= T + m(n-1) + \sum_{i=1}^{n-1}(D + L_i) \\
&\leq T + m(n-1) + \sum_{i=1}^{n-1}(R_i - L_i + L_i) \\
&= T + m(n-1) + \sum_{i=1}^{n-1} R_i \\
&\leq T + \sum_{i=1}^{n-1}(M_i + R_i)
\end{aligned}$$

where the single strict inequality follows from the condition in the hypothesis. Finally note that the only property of the middle option used in this proof is

that the middle option requires the use of recursion. □

The additional time required for two recursive calls beyond that required for a loop is typically more than the time for executing a single addition. Based on this assumption, the diva procedures in Examples 1, 3, and 4 clearly satisfy the condition in this theorem, since the time required for executing the conquer block of each is constant, and the diva procedure in Example 2 also satisfies the condition, since the only variability in time for executing its conquer block is due to the possible execution of a single `+ 1`. On the other hand, the diva procedure for a merge sort would fail to satisfy this condition; the merge sort is considered further near the end of this section.

One strategy for a sequential implementation of a diva procedure `P` is to translate the diva procedure into the code for the procedure `P` of Figure 7. Since both procedures have the same name and formal parameter list, no modification is needed to the code for a diva call. The compiler is assumed to determine a value `MY_NUM_ITERATIONS` for `P`, which is 1 if the middle option is chosen and is the length of the dynamic parameters in the nonrecursive call on `P` if the end option is chosen. Of course, the target code in Figure 7 could be optimized so no test on `MY_NUM_ITERATIONS` is included for these two extremes, so no run-time cost need be incurred. For generality, Figure 7 assumes the parameter list of `P` is as given in Figure 1, with `IN_PARM` and `PARM` included, and with `PARM` having the mode `in out`, except that we assume the m actual arrays corresponding to the dynamic parameters in the diva procedure have been declared to be constrained versions of `DYNAMIC_VECTOR1`, ..., `DYNAMIC_VECTORm`. If any names used in the source version of `P` conflict with newly introduced names, such as `FIRST` and `LAST`, a suffix should be added to the new name to avoid such a conflict. If the diva procedure contains neither an `out` nor an `in out` parameter `PARM`, the assignments of `PARM` to `INITIAL_PARM` and `FINAL_PARM` in Figure 7 should be omitted. The procedure `P_1` for the diva procedure `FIND_SUM` of Example 1, i.e. `FIND_SUM_1`, is illustrated as part of Figure 10 in the next section.

Note that in computing the cost of recursion in `P` over the cost of using a loop when Figure 7 is the basis for the translation, the form of parameters used should be that of procedure `P_1`, where parameters to indicate the current range are used, rather than vectors whose range is unconstrained, and where the parameter `IN_PARM` has been eliminated. The theorem stated in this section 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 `if` is the code that will be executed to compute the value of `FINAL_PARM`. This additional information can be very useful.

As an illustration, consider Figure 8, which contains the true branch of the `if` of Figure 7 when the diva procedure `P` is `COUNT_RUNS` of Example 2. In performing data flow analysis, each variable of type `ANSWER_TYPE` could be treated as three separate variables. Straightforward data flow analysis would permit the elimination of variables whose values are not used or whose values are guar-

```
procedure P (A1: in out DYNAMIC_VECTOR1; ...  Am: in out DYNAMIC_VECTORm;
             IN_PARM: in IN_PARM_TYPE; PARM: in out PARM_TYPE) is

  procedure P_1 (FIRST, LAST: in INTEGER;  -- limits of range
                 PARM:        in out PARM_TYPE) is
    MID: INTEGER;
    MY_SECTION_LEN: INTEGER := LAST - FIRST + 1;
    INITIAL_PARM, FINAL_PARM: PARM_TYPE;
    ... Place any additional local declarations of the source's P here.
  begin
    if MY_SECTION_LEN <= MY_NUM_ITERATIONS then
      ... The true branch of the if of the source's P is placed here, with
      ... A'FIRST replaced by FIRST.
      for I in FIRST+ 1 .. FIRST + MY_SECTION_LEN - 1 loop
        INITIAL_PARM := PARM;
        ... The true branch of the if of the source's P is placed here, with
        ... A'FIRST replaced by I.
        FINAL_PARM := PARM;
        ... The false branch of the if of the source's P (after the recursive
        ... calls) is placed here.
      end loop;
    else
      MID := (FIRST + LAST) / 2;
      P_1 (FIRST,   MID,  INITIAL_PARM);
      P_1 (MID + 1, LAST, FINAL_PARM);
      ... The false branch of the if of the source's P (after the recursive
      ... calls) is placed here.
    end if;
  end P_1;

begin -- P
  P_1 (A1'FIRST, A1'LAST, PARM);
end P;
```

Figure 6: Translation of a Diva Procedure For a Sequential Computer

```
ANS.COUNT := 0;
ANS.FIRST := S(FIRST);
ANS.LAST  := S(FIRST);
for I in FIRST + 1 .. FIRST + MY_SECTION_LEN - 1 loop
  L := ANS;
  ANS.COUNT := 0;
  ANS.FIRST := S(I);
  ANS.LAST  := S(I);
  R := ANS;
  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 loop;
```

Figure 7: Code To Be Optimized

anteed to be zero, the replacement of `R.FIRST` and `R.LAST` by `S(FIRST + I)`, and the elimination of unnecessary code such as

```
else
  L.COUNT := L.COUNT;
```

It is easy to see that such straightforward transformations permit the sequential loop to be replaced by

```
L.COUNT := 0;
L.FIRST := S(FIRST);
L.LAST := S(FIRST);
for I in FIRST + 1 .. FIRST + MY_SECTION_LEN - 1 loop
  if L.LAST > S(I) then
    L.COUNT := L.COUNT + 1;
  end if;
  L.LAST := S(I);
end loop;
```

Based on more sophisticated data flow analysis, this code could then be replaced by

```
L.COUNT := 0;
L.FIRST := S(FIRST);
for I in FIRST + 1 .. FIRST + MY_SECTION_LEN - 1 loop
  if S(I - 1) > S(I) then
    L.COUNT := L.COUNT + 1;
  end if;
end loop;
L.LAST := S(FIRST + MY_SECTION_LEN - 1);
```

A comparison of times required for the middle and end options should be made only after optimizations, such as those in the preceding paragraph, have been performed.

The merge sort is a noteworthy example in which the end option (which yields the straight insertion sort) is more efficient for dynamic parameters of very short length and the middle option is more efficient for dynamic parameters of longer length, since the conquer block of the merge sort would require time proportional to the length of the dynamic parameters to carry out the merge. Expecting a compiler to determine analytically for a diva procedure such as merge sort a value of `MY_NUM_ITERATIONS` strictly between 1 and the length of the dynamic parameters seems unreasonable but there are at least two other possibilities. First, a general algorithm could be made available to the compiler, which could use characteristics of the conquer block in question as parameters in the algorithm. Such an algorithm could either be heuristic or obtained as the result of an analytical study. Second, the compiler could calculate an approximate value of `MY_NUM_ITERATIONS` by running several compile-time tests involving vectors whose components are assigned random values.

There are a variety of reasonable options in addition to the middle and end options, such as choosing the division point after the first component of the dynamic parameters. Of course, when a compiler is unable to determine which option is superior, it could use the middle option, with the nondeterministic divide leading to no run-time cost (nor savings) over that typically expected by programmers for a divide-and-conquer algorithm. In the discussion so far, the choice of option does not depend on the values of the actual arguments. This restriction could be relaxed and the values of parameters, other than the dynamic parameters, could help to guide the choice of option.

Developing sophisticated techniques for a compile-time decision among such options is the subject of further research. For the purpose of this paper, 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 important diva procedures for which a straightforward compiler could eliminate unnecessary overhead due to recursion.

8 PARALLEL IMPLEMENTATION OF DIVA CALLS

Since it is well-known that associative operations can be implemented efficiently using a variety of techniques and since diva calls have been shown in Section 6 to be intimately related to associative operations, it is intuitively clear that diva calls can be implemented efficiently on a variety of parallel computers. For the sake of illustration, strategies for implementing diva calls on several classes of parallel computers are discussed in this section. These strategies are similar to strategies for implementing reductions on such computers.

We first present an algorithm that translates diva calls into code for certain MIMD computers and then discuss in Section 8.3 how the algorithm could be modified for SIMD computers. For simplicity, the algorithm assumes the number of processors is a power of 2.

The basic implementation strategy is to let worker processes obtain partial results and then to combine these partial results using access to shared memory (for shared memory MIMD architectures), message passing (for distributed memory MIMD architectures), or shifts (for SIMD architectures). Discussing how to generate actual code for particular parallel computers would lack generality and would introduce unnecessary details. Thus a high-level language, Ada, is used to describe the target code and the form of Ada rendezvous used is one in which the accept block simply assigns values to parameters. Such a simple Ada rendezvous could be implemented using access to shared memory on a shared memory computer and using message passing on a distributed memory computer. In discussing SIMD computers, we shall explicitly show how the use of such an Ada rendezvous can be replaced by the use of shifts. There is no simple way to characterize hybrids of the three major classes of parallel architectures, shared memory MIMD, distributed memory MIMD, and SIMD. Thus target code for hybrids will not be presented. However, the various translation options for a simple Ada rendezvous suggest the feasibility of adapting the target code for a hybrid architecture.

Combining the partial results of a diva call using an MIMD computer can involve the participation of a large number of processors either accessing certain memory locations (if shared memory is used) or communicating simultaneously (if memory is distributed). Ideally, shared memory computers are programmed without considering the actual physical location of memory; however, contention for nonlocal memory can lead to serious degradation when a large number of processors attempt to access the same memory location simultaneously [YTL86, RT86]. Similarly, whereas the use of worm-hole routing [Dal86] permits distributed multicomputers to be programmed without considering the distance (in communication hops) between processors, contention for channels can lead to system degradation when many processors are involved in simultaneous communication. To avoid degradation due to memory contention or

channel contention, the interconnections within the target computer need to be considered by a diva call translation algorithm. Although a variety of communication schemes can form the basis for such a translation algorithm, to present a specific algorithm we shall assume the following condition is satisfied by the MIMD computers considered:

> Processes can be assigned indexes in such a way that, for all i and j less than the number of processes, an Ada rendezvous whose accept block simply assigns values to parameters can be implemented efficiently so that information can be passed from the process with index i to the process with index j whenever the binary code for j can be obtained from that of i by flipping a single 0 bit; this flipped bit must be to the right of the rightmost 1 bit in the code for i if the code for i has a 1 bit.

For example, when the number of processes is 8, the condition requires that a simple Ada rendezvous can be implemented efficiently:

from the process with index	to the process(es) with index
000	001, 010, and 100
010	011
100	101 and 110
110	111

Notice that a hypercube topology provides direct connections between pairs of processor/memory nodes with the above indexes.

8.1 Translation for MIMD Shared Memory Computers

The existence of shared memory is assumed in this section. This assumption will be removed in Section 8.2.

`TRANSLATION ALGORITHM:`

Let `P` be a diva procedure having dynamic parameters `A1`, ..., `Am` and additional parameters `IN_PARM` and `PARM`, where the dynamic parameters have been declared to be constrained versions of `DYNAMIC_VECTOR1`, ..., `DYNAMIC_VECTORm`, respectively. The diva procedure `P` can be implemented on an MIMD shared memory computer by translating it into the code for the procedure `P` given in Figure 9, where the following additional notation is used: A worker is responsible for computations involving the slices of the dynamic parameters of length `MY_GRAIN_SIZE` starting with subscript `MY_SECTION_START`. The array `MY_INDEX_TO_CALL` is to store the list of integers whose binary codes are obtained by flipping 0 bits in the binary code for `MY_INDEX` as explained above;

the length of this list is assigned to the variable `MY_NUM_OF_CALLS` and this list is arranged in ascending order. The variable `MY_NUM_ITERATIONS` should be assigned 1 if the middle option is chosen (see Section 7) and assigned the value of `MY_GRAIN_SIZE` if the end option is chosen. □

The translation algorithm would translate the source code for `FIND_SUM` in of Example 1 into the code for the procedure `FIND_SUM` of Figure 10.

```
procedure P (A1: in out DYNAMIC_VECTOR1; ...  Am: in out DYNAMIC_VECTORm;
             IN_PARM: in IN_PARM_TYPE; PARM: in out PARM_TYPE) is

  task type WORKER_TYPE is
    entry COMBINE (FINAL_PARM: out PARM_TYPE);
  end WORKER_TYPE;
  WORKER: array(0..NUM_OF_WORKERS-1) of WORKER_TYPE;

  task body WORKER_TYPE is
    MY_INDEX, MY_SECTION_START, MY_GRAIN_SIZE, MY_NUM_ITERATIONS,
    MY_NUM_TO_CALL: INTEGER;
    ... Place the declaration of the array MY_INDEX_TO_CALL and the local
    ... declarations of the source's P here.
    ... Place the declaration of procedure P_1 of Figure 7 here.
  begin -- WORKER_TYPE
    ... Assign appropriate value to task index MY_INDEX. (See [Bur85].)
    ... Determine the values of MY_SECTION_START, MY_GRAIN_SIZE,
    ... MY_NUM_ITERATIONS, MY_NUM_TO_CALL, and MY_INDEX_TO_CALL.
    -- Perform computations independent of the other workers:
    P_1 (MY_SECTION_START, MY_SECTION_START + MY_GRAIN_SIZE - 1, INITIAL_PARM);
    -- Combine local results with the results from other workers:
    if NUM_OF_WORKERS > 1 then
      for I in 1 .. MY_NUM_TO_CALL loop
        WORKER(MY_INDEX_TO_CALL(I)).COMBINE (FINAL_PARM);
        ... The false branch of the if of the source's P (after the recursive
        ... calls) is placed here, with each return statement replaced by a
        ... goto statement with target END_FALSE_BRANCH.
        <<END_FALSE_BRANCH>>
        INITIAL_PARM := PARM;
      end loop;
      if MY_INDEX = 0 then
        PARM := INITIAL_PARM; -- result assigned to in out parameter
      else
        accept COMBINE (FINAL_PARM: out PARM_TYPE) do
          FINAL_PARM := INITIAL_PARM;
        end COMBINE;
      end if;
    else
      PARM := INITIAL_PARM; -- result assigned to in out parameter
    end if;
  end WORKER_TYPE;

begin -- P
  null;  -- activate the array of WORKERs
end P;
```

Figure 9: Translation of Diva Procedure for MIMD Shared Memory Computer

```
procedure FIND_SUM (A: in DYNAMIC_VECTOR; SUM: out INTEGER) is

   task type WORKER_TYPE is
     entry COMBINE (FINAL_SUM: out INTEGER);
   end WORKER_TYPE;
   WORKER: array(0..NUM_OF_WORKERS-1) of WORKER_TYPE;

   task body WORKER_TYPE is
     MY_INDEX, MY_SECTION_START, MY_GRAIN_SIZE, MY_NUM_ITERATIONS,
     MY_NUM_TO_CALL: INTEGER;
     ... Place the declaration of the array MY_INDEX_TO_CALL here.
     INITIAL_SUM, FINAL_SUM: INTEGER;

     procedure FIND_SUM_1 (FIRST, LAST: in INTEGER;   -- limits of range
                           SUM: out INTEGER) is
       MID: INTEGER;
       MY_SECTION_LEN: INTEGER := LAST - FIRST + 1;
       INITIAL_SUM, FINAL_SUM: INTEGER;
     begin -- FIND_SUM_1
       if MY_SECTION_LEN <= MY_NUM_ITERATIONS then
         SUM := A(FIRST);
         for I in FIRST + 1 .. FIRST + MY_SECTION_LEN - 1 loop
           INITIAL_SUM := SUM;
           SUM := A(I);
           FINAL_SUM := SUM;
           SUM := INITIAL_SUM + FINAL_SUM;
         end loop;
       else
         MID := (FIRST + LAST) / 2;
         FIND_SUM_1 (FIRST,   MID,  INITIAL_SUM);
         FIND_SUM_1 (MID + 1, LAST, FINAL_SUM);
         SUM := INITIAL_SUM + FINAL_SUM;
       end if;
     end FIND_SUM_1;
```

Figure 10 (continued on next page)

```
   begin -- WORKER_TYPE
     ... Assign appropriate value to task index MY_INDEX. (See [Bur85].)
     ... Determine the values of MY_SECTION_START, MY_GRAIN_SIZE,
     ... MY_NUM_ITERATIONS, MY_NUM_TO_CALL, and MY_INDEX_TO_CALL.
     -- Perform computations independent of the other workers:
     FIND_SUM_1 (MY_SECTION_START, MY_SECTION_START + MY_GRAIN_SIZE - 1,
                 INITIAL_SUM);
     -- Combine local results with the results from other workers:
     if NUM_OF_WORKERS > 1 then
       for I in 1 .. MY_NUM_TO_CALL loop
         WORKER(MY_INDEX_TO_CALL(I)).COMBINE (FINAL_SUM);
         SUM := INITIAL_SUM + FINAL_SUM;
         INITIAL_SUM := SUM;
       end loop;
       if MY_INDEX = 0 then
         SUM := INITIAL_SUM; -- result assigned to in out parameter
       else
         accept COMBINE (FINAL_SUM: out INTEGER) do
           FINAL_SUM := INITIAL_SUM;
         end COMBINE;
       end if;
     else
       SUM := INITIAL_SUM; -- result assigned to in out parameter
     end if;
   end WORKER_TYPE;

begin -- FIND_SUM
  null;  -- activate the array of WORKERs
end FIND_SUM;
```

Figure 10: Ada Code Produced by the Translation of Example 1

An optimizing compiler could enhance the efficiency of the sequential phase of the target code by removing the use of many temporary variables. For example, using data flow analysis an optimizing compiler could replace the code in Figure 10 for

```
SUM := A(FIRST);
for I in FIRST + 1 .. FIRST + MY_SECTION_LEN - 1 loop
  INITIAL_SUM := SUM;
  SUM := A(I);
  FINAL_SUM := SUM;
  SUM := INITIAL_SUM + FINAL_SUM;
end loop;
```

with the code for

```
SUM := A(FIRST);
for I in FIRST + 1 .. FIRST + MY_SECTION_LEN - 1 loop
  SUM := SUM + A(I);
end loop;
```

since the values of `INITIAL_SUM` and `FINAL_SUM` are not needed after the loop.

Let n denote the length of the dynamic parameters in the nonrecursive call on `P` and suppose the true branch of the `if` of `P` takes time proportional to t_1, the false branch takes time proportional to t_2, and the cost of parameter passing takes constant time. Then it is easy to see that the execution of `P` takes time proportional to $t_2(\log n) + t_1$ when executed with n processors using the implementation described in this section.

The purpose of presenting the implementation algorithm in this section is simply to make it clear that diva calls can be translated automatically so that the resulting code can be executed by an MIMD computer for which each processor performs independent work before results of different processors are combined. More efficient implementations are certainly possible. For example, if several diva calls use initial dynamic parameters of the same length, a single set of worker processes would suffice for such calls, so the overhead of activating and initializing such processes can be spread over several applications.

Some of the restrictions on the syntax of diva procedures are to support an implementation strategy such as that given in this section. Note that when `PARM` is used in a diva procedure, it is actually the values of local variables `INITIAL_PARM` and `FINAL_PARM` that must be used in the recursive calls, and the syntax prohibits assigning values to these local variables during initialization. This syntactic restriction facilitates letting the workers execute the true branch of the `if` without there being any recursive calls. (Note that, although the purpose of `PARM` is to provide output from the diva call, `PARM` is permitted to be an `in out` parameter. This is because Ada rules do not allow read access to the value of an `out` parameter so restricting `PARM` to being an `out` parameter would sometimes require a large number of extra local variables.)

Since the value of PARM is not used in the recursive calls, IN_PARM is necessary to permit non-dynamic input information to be communicated to each activation of the true branch of the if. Such input to a diva procedure can include a table of initialization information, computed within an encapsulating procedure (of the kind mentioned at the beginning of Section 4) using the value of other non-dynamic inputs. Note that the only actual argument in the recursive calls on P that can play the role of IN_PARM is IN_PARM itself, so the value of this parameter is the same for each activation of a particular nonrecursive call on the diva procedure and thus the executions of the true branch can take place without there being any recursive calls.

8.2 Translation for Distributed Memory

The sample target code given in Figure 9 assumes the shared memory model of parallel computing; for example, the vectors used in the translated diva procedure are nonlocal to the workers. When distributed memory is used, a large vector, such as a vector corresponding to a dynamic parameter, should be maintained as an ordered collection of local arrays within the local memories. Other nonlocal data used in the target code of Figure 9 can be replicated on local memories. The message passing to implement such local arrays and replicated data should be accomplished prior to the beginning of execution of the diva call whenever possible. Permitting distributed memory computers to be programmed at a level supporting the illusion of shared memory is a related but separate research problem that is being investigated by numerous researchers; for example, see [CK88, KMR90, Li86, RSW88, SCMB90]. Thus the focus of this section is on activities within the diva call itself, rather than on distributing vectors and other data among the local memories.

The target code given in Figure 9 can be modified for distributed memory as follows. For each i in 1..m, within each worker declare a local array LOCAL_Ai having the range

```
MY_SECTION_START .. MY_SECTION_START + MY_GRAIN_SIZE - 1
```

and base type BASE_TYPEi. The values of the corresponding slice of Ai should be assigned to LOCAL_Ai and use of Ai should be replaced by LOCAL_Ai. A local variable LOCAL_IN_PARM corresponding to IN_PARM should similarly be declared and used. All use of the Ada rendezvous should be implemented by message passing and the two assignments of INITIAL_PARM to PARM should be replaced by code that sends the value of INITIAL_VALUE to the executor of the diva call. Finally, if any of the dynamic parameters are out (or in out and a value is assigned) the new value of the corresponding vector should be reflected in changed values of local arrays.

Two of the restrictions on the syntax of a diva procedure facilitate the kind of implementation given in this section. The prohibition on accessing nonlocals from within a diva procedure reduces the dependence of Figure 9 on the shared

memory model of parallel computing and the restriction on accessing a dynamic parameter within the false branch of the `if` after the recursive calls permits a processor to execute this code without having to access components of such an array, which may be outside the local memory of the processor. It is no violation of the syntax rules for `IN_PARM_TYPE` to be a named array type with unconstrained range. However, since parameter passing for an array may be implemented by copy [Uni83, 6.2.7], the Ada programmer will realize that it could be quite costly to use such a parameter to get around the restriction on accessing dynamic parameters mentioned at the beginning of this paragraph.

8.3 Translation for SIMD Computers

A modification of the target code in Figure 9 produces code for SIMD computers, such as the ICL DAP [ICL80, Per87] and the Connection Machine [HS86], that permit data to be shifted from processors with higher processor numbers to processors with lower numbers by shifts that are powers of 2. This modification is straightforward since the message passing in Figure 9 can be replaced by shifting. In this section we assume that the length of the initial actual arguments corresponding to the dynamic parameters is a power of 2 and is at least as large as the number of processors.

Let the notation

```
SHIFT (<data>, <target_variable>, <distance>);
```

represent a shift, where the value of `<data>` is to be shifted to the local variable `<target_variable>` of the processor whose index is `<distance>` less than that of the processor executing the instruction and assume that 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 modification can be described as follows. The translator should assign the same value to each instance of `MY_GRAIN_SIZE`, the same value to each instance of `MY_NUM_ITERATIONS`, and log_2 of the number of processors to each instance of `MY_NUM_OF_CALLS`. Remove the declaration of `MY_INDEX_TO_CALL` and the determination of its value from the target code. Replace the lines

```
for I in 1 .. MY_NUM_TO_CALL loop
  WORKER(MY_INDEX_TO_CALL(I)).COMBINE (FINAL_PARM);
  ... The false branch of the if of P (after the recursive calls)
  ... is placed here, with each return statement replaced by a
  ... goto statement with target END_FALSE_BRANCH.
  <<END_FALSE_BRANCH>>
  INITIAL_PARM := PARM;
end loop;
if MY_INDEX = 0 then
  PARM := INITIAL_PARM; -- result assigned to in out parameter
else
  accept COMBINE (FINAL_PARM: out PARM_TYPE) do
    FINAL_PARM := INITIAL_PARM;
  end COMBINE;
end if;
```

with the lines

```
SHIFT_AMOUNT := 1;
for I in 1 .. MY_NUM_TO_CALL loop
  SHIFT (INITIAL_PARM, FINAL_PARM, SHIFT_AMOUNT);
  ... The false branch of the if of P (after the recursive calls)
  ... is placed here, with each return statement replaced by a
  ... goto statement with target END_FALSE_BRANCH.
  <<END_FALSE_BRANCH>>
  INITIAL_PARM := PARM;
  SHIFT_AMOUNT := SHIFT_AMOUNT * 2;
end loop;
if MY_INDEX = 0 then
  PARM := INITIAL_PARM; -- result assigned to in out parameter
end if;
```

Of course this 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.[12] 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 an SIMD architecture.

8.4 The Independence of Logical Correctness from Implementation Details

A variety of implementation possibilities can be used for implementing diva calls. As in Section 7, diva calls can be implemented sequentially using recur-

[12] Recall the observation about conditional expressions made after Example 2 in Section 4.

sion based on choosing division points at the middle, using a loop based on choosing division points just before the end, using a combination of these two options, or using another option. Diva calls can be implemented on parallel computers with a variety of grain sizes, with each processor using a sequential implementation of diva calls before results are combined; combining of results from different processors can be obtained through the use of a processor tree with communication via access to shared memory (Section 8.1) or via message-passing (Section 8.2), through the use of shifts (Section 8.3), or through the use of another technique (such as pointer doubling [HS86]). The allocation of processors for the execution of the diva call can be either static or dynamic and tolerance for faulty channels can be programmed into a parallel implementation, since there are $n-1$ possible choices for division point for vectors of length n. Finally, when multiple processors are used, they need not be identical.

As shown in Section 5, all implementation issues are abstracted away from the text of a diva procedure, so that in verifying the logical correctness of a diva procedure a programmer is allowed to think at a simpler, higher-level of abstraction. Concerns about implementation details should be orthogonal to concerns about logical correctness. Implementation details can be specified independently as qualifiers in the compilation command. Such implementation details could help obtain more efficient use of the target computer, yet the source code would retain the existing advantages of diva procedures relating to logical simplicity and portability.

9 COMPARISON WITH REDUCTION OPERATORS

Connection Machine Lisp [SH86] and Paralation Lisp [Sab88], recent innovative and unconventional languages for parallel computing, permit the programmer to reduce a vector `V` using a programmer-defined 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. The technique used is to define a binary function f (that takes two values of the base type of `V` and returns a value of the base type of `V`) and then to use f and `V` as arguments in a call on a general reduction operator. Note that, like the use of diva procedures, such a computation can be programmed without requiring that the programmer consider parallelism. Other approaches, such as that of iPSC/2 Fortran and C [Int89] are only permitted within parallel code and assume a particular target architecture.[13]

To illustrate this technique, consider computing the number of times a term appears in a finite sequence `S` that is greater than the next term of the sequence. (Recall that this was accomplished in Example 2 using the diva procedure `COUNT_RUNS`.) The major steps, using Ada syntax, would be:

1. Define a type

```
type ANSWER_TYPE is
  record
    COUNT,
    FIRST,
    LAST:    INTEGER;
  end record;
```

2. Declare a new vector

```
V: array(S'RANGE) of ANSWER_TYPE;
```

3. Initialize the components of `V` using code similar to the true branch of the `if` in `COUNT_RUNS` as well as using the corresponding values of `S`.

4. Define a function subprogram f that takes two arguments of type `ANSWER_TYPE` and returns a value of type `ANSWER_TYPE` using code similar to the false branch of the `if` in `COUNT_RUNS`.

5. Apply the general reduction operator to reduce `V` using f.

[13] In addition, the iPSC/2 software requires that f be commutative, but this requirement can be overcome [Cha].

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, as shown in Section 6. On the other hand, when a general reduction operator is used, a separate proof of the associativity of f is needed. As shown in Section 6, such a proof of associativity, when carried out using a straightforward case analysis based on the proof of correctness of a single application of f, requires p^4 cases, where p is the number of cases used in proving the correctness of a single application of f. Other approaches can also be significantly more complex than proving the correctness of a diva procedure.

   Confronted with such complexity when using a general reduction operator in a program, some programmers may be 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 `COUNT_RUNS`. 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 11 would obtain correct results and might assume naively that "counting runs is obviously 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, 10, and 30 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. As discussed in [Cha90b], a proof technique based on strong induction can be used to prove the associativity of f using the same number of cases as

```
Define a type ANSWER_TYPE as

   type ANSWER_TYPE is
     record
       COUNT,
       VALUE:   INTEGER;
     end record;

initialize V so that for all I

   V(I).COUNT := 0;
   V(I).VALUE := S(I);

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


function f (X, Y: ANSWER_TYPE) return ANSWER_TYPE is
  ANS: ANSWER_TYPE:
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 10: A Simple but Incorrect Approach to Counting Runs

the proof of the corresponding diva procedure. However, typically f operates on and returns records, where the initial sequence of such records that is being reduced is related to, but different from, the sequence of interest. Thus typically two sequences must be referred to within the proof of associativity. Additional applications are possible if the associativity requirement is generalized to a requirement that the function simply be associative relative to the sequences of interest [Cha90b]. Such a requirement involving a sequence is even less natural to include in the specification of the function.

5. Strong induction can provide a simple approach to verifying program correctness for the kind of applications considered in this paper, as illustrated in Section 5. Such simple use of strong induction relies on a simple con-

ceptualization 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.

6. Whenever possible the form in 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 this paper.

7. Programmers should be motivated by at least an informal proof of correctness of strategy. Both informal and formal proofs of correctness are, of course, subject to human error. When it is clear that the result calculated by a particular diva call is independent of the choice of division points, a diva call can be (partially) tested and debugged on a sequential computer using small initial dynamic parameters with division points selected at the middle of the dynamic parameters, an environment involving debugging tools familiar to the sequential programmer.

8. 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. This requirement creates a disadvantage of using such operators for applications in which values need to be used by the function that the function should not modify. Consider, for example, the diva procedure `LOOKUP`, mentioned after Example 4, which

uses an `in` parameter to store the value of a search key to be used in an unordered table lookup. When a general reduction operator is used for this application, either a field containing the search key would need to be included in the *result* type, even though this value must not be changed by a call on f, or else a nonlocal access from within f must be used to obtain the value of the search key. In either case, an important fact known to the programmer, that the value of the search key should not be inadvertently modified, is being withheld from compilers and syntax-directed editors. Using an `in` parameter in a diva procedure makes this fact explicit.

9. While it is always possible to 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. As mentioned in Section 3, the requirement in this paper that diva procedures have just a single non-dynamic `out` or `in out` parameter is made only to simplify the exposition of this paper and should be removed in a construct designed for a programming language, since it is unnatural to be forced to use a record simply to satisfy an arbitrary restriction. For instance, in Example 4 this restriction forces the programmer to decide among three alternatives: to use an assignment statement like

   ```
   Y_VALUE := 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.[14] A diva construct designed for a programming language could allow the elimination of such awkwardness, but the use of a function in a general reduction operator would require a record for more than one result value.

10. 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. The disadvantage of creating such typically large ad-hoc vectors, in addition to the awkwardness of using records when none may be required, is discussed at the end of Section 4. Such ad-hoc vectors are unnecessary when a diva procedure is used, since multiple dynamic parameters are permitted.[15] In addition, the effect of assigning initial

[14] Even if the result of the diva call is used just once or twice, using a field and record name may not be the natural approach. Consider, for example, finding the variance of a vector by using a single diva procedure to return both the the sum of the components of the vector and the sum of the squares of components; the natural approach would be to have two parameters for the sums rather than a single record parameter with fields for the two sums.

[15] Recall that such ad-hoc vectors can also be avoided by using the noteworthy, unconventional semantics of Paralation Lisp.

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.

11. 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.

10 CONCLUSIONS

The importance of developing effective ways to program parallel computers has been widely acknowledged. All too 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 paper 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.

Although the purpose of this paper is to propose a language concept, rather than a specific language construct, for concreteness the concept is studied here in terms of one possible construct, the diva procedure call. Diva calls are shown to have the major implementation advantages of general reduction operators, due to an intimate relationship between a diva call and associativity, yet to be free of numerous undesirable features of general reduction operators and thereby to 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 compilers and syntax-directed editors 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.

Diva calls have been shown to be relatively easy to verify, applicable to a variety of applications, and implementable on parallel computers in log time. Yet diva calls are at a higher level of abstraction than such parallel programming issues as MIMD versus SIMD, the workload of processors, and distributed versus shared memory. Thus diva calls provide one approach to simplifying the programming of parallel computers.

In deciding whether to include a notation or construct supporting the nondeterministic divide within a programming language, one is reminded of the advice in Hoare's CSP article [Hoa78]:

> 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.

Additional research being pursued by the author includes the extension of diva procedures to permit the use of multidimensional dynamic parameters and the continued enhancement of the current translator for a language incorporating diva procedures.

ACKNOWLEDGEMENTS: The author is grateful to Greg Morrisett and Jeff Michel for serving as assistants on this language design project for four years. Terry Pratt critiqued an earlier version of this paper and the author is also grateful to Jim French, Paul Reynolds, and Dana Richards for helpful advice.

11 REFERENCES

[BST89] Bal, H. E., Steiner, J. G., and Tanenbaum, A. S. Programming languages for distributed computing systems. *Computing Surveys.* 21, 3 (Sept. 1989), 261-322.

[Bar84] Barnes, J. G. P. *Programming in Ada.* Addison-Wesley, Reading, Mass., 1984.

[Ben86] Bentley, J., *Programming Pearls.* Addison-Wesley, Reading, Mass., 1986.

[Bur85] Burns, A. Efficient initialisation routines for multiprocessor systems programmed in Ada. *Ada Letters*, 5, 1 (July - Aug 1985), 55-60.

[CK88] Callahan, D. and Kennedy, K. Compiling programs for distributed memory multiprocessors. *J. of Supercomputing*, 2, (1988), 151-169.

[Cha86] Charlesworth, A. The design and implementation of a preprocessor for Ada to provide support for the multiway accept, NASA Langley Research Center grant proposal NAG-1-774, Dec. 1986.

[Cha87] Charlesworth, A. The multiway rendezvous. *ACM Trans. Program. Lang. Syst.*, 9, 2, (July 1987), 350-366.

[Cha89] Charlesworth, A. On a class of divide and conquer algorithms. Preliminary report. *Abstracts of the Amer. Math. Soc.*, 10, 6 (Nov. 1989), 492.

[Cha90a] Charlesworth, A. *Adam Language Reference Manual*, preliminary version 7/30/90, Dept. Math. and Comp. Sci., U. of Richmond, Va., July 1990.

[Cha90b] Charlesworth, A. A characterization of associativity, Tech. Rep. IPC-TR-90-007, Inst. for Parallel Computation, U. of Virginia, Charlottesville, Nov. 1990.

[Cha] Charlesworth, A. Programming a class of divide-and-conquer algorithms on the Intel iPSC/2. Paper in progress.

[Dal86] Dally, W. J. *A VLSI Architecture for Concurrent Data Structures.* Ph.D. Dissertation, Dept. of Computer Science, Calif. Instit. of Tech., Tech. Rep. 5209, 1986.

[Dij59] Dijkstra, E. A note on two problems in connexion with graphs. *Numersiche Mathematik* 1 (1959), 269-271.

[Dun90] Duncan, R. A survey of parallel computer architectures. *Computer* 23, 2 (Feb. 1990), 5-16.

[Gri81] Gries, D. *The Science of Programming.* Springer-Verlag, New York, 1981.

[HS86] Hillis, W. D. and Steele, G. L., Jr. Data parallel algorithms. *Comm. ACM*, 29, 12, (Dec. 1986), 1170-1183.

[Hoe62] Hoel, P. G. *Introduction to Mathematical Statistics*, John Wiley, New York, 1962.

[Hoa78] Hoare, C. A. R. Communicating sequential processes. *Comm. ACM*, 21, 8, (Aug. 1978), 666-677.

[ICL80] International Computers Ltd. *DAP Fortran language.* ICL Technical Pub. 6755, 1980.

[Int89] Intel Corporation, *iPSC/2 Programmer's Reference Manual.* Beaverton, Or., Oct. 1989.

[Ive62] Iverson, K. E., *A Programming Language.* John Wiley and Sons, Inc., New York, 1962.

[KMR90] Koebel, C., Mehrotra, P., and Rosendale, J. V. Supporting shared data structures on distributed memory architectures. *SIGPLAN NOTICES*, 25, 3 (Mar. 1990), 177-186. (*Proc. Second ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*)

[Knu73] Knuth, D. E. *The Art of Computer Programming, Vol. 3: Sorting and Searching.* Addison-Wesley, Reading, Mass., 1973.

[Li86] Li, Kai. *Shared Virtual Memory on Loosely Coupled Multiprocessors.* Ph. D. thesis, Yale University, New Haven, Conn., Sept. 1986.

[MT86] Mullender, S. J. and Tanenbaum, A. S. The design of a capability-based distributed operating system. *Computer J.*, 29, 4 (1986), 289-300.

[Per87] Perrott, R. H. *Parallel Programming.* Addison-Wesley, Reading, Mass., 1987.

[Pri57] Prim, R. C. Shortest connection networks and some generalizations. *Bell System Tech. J.* 36 (1957), 1389-1401.

[RT86] Rettberg, R. and Thomas, R. Contention is no obstacle to shared-memory multiprocessing. *Comm. ACM*, 29, 12 (Dec. 1986), 1202-1212.

[RSW88] Rosing, M., Schnabel, R. B., and Weaver, R. P. Dino: Summary and examples. *Proc. Third Conf. on Hypercube Concurrent Computers and Appl.* (1988), 472-481.

[RSL74] Rosenkrantz, D., Stearns, R., and Lewis, P. Approximate algorithms for the traveling salesperson problem. *Proc. 15th Annual IEEE Symp. on Switching and Automata Theory.* 1974, 33-42.

[Sab88] Sabot, G. *The Paralation Model: Architecture-Independent Parallel Programming.* MIT Press, Cambridge, 1988.

[SCMB90] Saltz, J., Crowley, K., Mirchandaney, R., and Berryman, H. Run-time scheduling and execution of loops on message passing machines. *J. of Parallel and Distributed Computing*, to appear.

[SH86] Steele, G. L., Jr. and Hillis, W. D. Connection Machine Lisp: Fine-grained parallel symbolic processing. *Proc. 1986 ACM Conference on Lisp and Functional Programming.* Aug. 1986, 279-297.

[Qui83] Quinn, M. J. *The Design and Analysis of Algorithms and Data Structures for the Efficient Solution of Graph Theoretic Problems on MIMD Computers.* Ph. D. Dissertation, Dept. of Computer Science, Wash. State Univ., Pullman, Wash., 1983.

[Uni83] United States Department of Defense. *Reference Manual for the Ada Programming Language.* ANSI/MIL-STD-1815A-1983, American National Standards Institute, 1983.

[YTL86] Yew, P. C., Tzeng, N. S., and Lawrie, D. H. Distributing hot spot addressing in large-scale multiprocessors. *Proc. of the 1986 Intl. Conf. on Parallel Processing* IEEE Press, New York, 1986, 51-58.