Science of Computer Programming 8 (1987) 213-229
North-Holland

APPLICATIONS OF A STRATEGY FOR DESIGNING DIVIDE-AND-CONQUER ALGORITHMS

Douglas R. SMITH
Kestrel Institute, Palo Alto, CA 94304, U.S.A.

Communicated by J. Darlington
Received March 1985
Revised May 1986

Abstract. A strategy for designing divide-and-conquer algorithms that was originally presented in a previous article is extended and applied to several new problems. The extension involves techniques for modifying the original specification based on specific kinds of failures that can occur during the design process. We derive several divide-and-conquer algorithms that are substantially more efficient than previously known algorithms. This paper also emphasizes the naturalness with which divide-and-conquer algorithms can be transformed into a parallel format. One problem explored is to find the maximum sum over all rectangular subregions of a given matrix of integers. For an $n \times n$ matrix there is a straightforward $O(n^6)$ enumeration algorithm. We derive a $O(n^3)$ divide-and-conquer algorithm, then show that it can be executed in $O(\log^2 n)$ time in parallel and, furthermore, with pipelining of inputs it can be executed with $O(1)$ time between successive outputs.

1. Introduction

In this paper we extend and apply a strategy for designing divide-and-conquer algorithms that was originally presented in [8]. The extension to the strategy involves techniques for modifying the original specification based on specific kinds of failures that can occur during the design process. The problem we deal with has one- and two-dimensional versions, called 1D-MAXSUM and 2D-MAXSUM respectively or MAXSUM generically. 1D-MAXSUM involves finding the maximum over the sums of all subarrays of a given array of integers. 2D-MAXSUM involves finding the maximum over the sums of all rectangular subregions of a matrix of integers. We derive two linear time divide-and-conquer algorithms for the 1D-MAXSUM problem. One of these runs in $O(\log n)$ time on a parallel processor. With pipelining of successive inputs, the time between the appearance of successive outputs is reduced to a constant. Then we derive a $O(n^3)$ time divide-and-conquer algorithm for the 2D-MAXSUM problem. This algorithm can be executed in $O(\log^2 n)$ time on a parallel mechanism. With pipelining of successive input matrices the time between the appearance of successive solutions is again reduced to a constant. These algorithms were previously unknown and the $O(n^3)$ algorithm for 2D-MAXSUM

and all of the parallel algorithms are substantially faster than previously known algorithms.

In Section 2 we present the extended design strategy and use it to derive a simple but optimal divide-and-conquer algorithm for 1D-MAXSUM. In Sections 3 and 4 we derive 'balanced' divide-and-conquer algorithms for 1D-MAXSUM and 2D-MAXSUM and discuss their transformation into parallel algorithms.

2. A strategy for designing divide-and-conquer algorithms

In this section we present a strategy for designing divide-and-conquer algorithms and illustrate it by deriving an algorithm for 1D-MAXSUM. The first step is to obtain a formal specification of the problem that we desire to solve. A formal specification for 1D-MAXSUM is

$$\text{1D-MAXSUM1}(A[1\,..\,n]) = best \text{ such that}$$

$$1 \leq n \;\Rightarrow\; best = \max_{1 \leq j \leq k \leq n} \sum_{p=j}^{k} A[p]$$

which can be read as follows. The problem is named 1D-MAXSUM1[1] and takes as input an array A with domain $1\,..\,n$ and codomain integers. For simplicity of notation we will omit typing information on variables—such information should be clear from the context. The output is called *best* and is specified to be an integer satisfying the input/output relation

$$1 \leq n \;\Rightarrow\; best = \max_{1 \leq j \leq k \leq n} \sum_{p=j}^{k} A[p].$$

The antecedent, $1 \leq n$, called the *input condition*, expresses any assumptions about the inputs. Here we are assuming that the input array is nonempty. The consequent,

$$best = \max_{1 \leq j \leq k \leq n} \sum_{p=j}^{k} A[p],$$

called the *output condition*, expresses the conditions under which an element of the output domain is a feasible solution to the problem posed by the input. Here, the output *best* is required to be the maximum over the sums of all subarrays of A between 1 and n. For example, if $A[1\,..\,5] = \langle -1, 3, -2, 5, -3\rangle$ then the maximum sum subarray is $A[2\,..\,4] = \langle 3, -2, 5\rangle$. The maximum sum subarray of $A[1\,..\,5] = \langle -1, 3, -2, -5, -3\rangle$ is $A[2\,..\,2] = \langle 3\rangle$.

[1] The following notational conventions will be used: specification names are fully capitalized and set in Roman, operators are indicated by capitalizing their first letter, and program scheme operators are further indicated by italics.

Intuitively, divide-and-conquer algorithms behave as follows: If the input is primitive, then a solution is obtained directly, by simple code. Otherwise a solution is obtained by decomposing the input into parts, independently solving the parts, then composing the results. Program termination is guaranteed by requiring that the parts be smaller than the input with respect to a suitable well-founded ordering. We restrict our attention to divide-and-conquer algorithms that have the following general form:

```
DC(x0)::=
  if Primitive(x0)
  then DC := Directly-Solve(x0)
  else begin
    ⟨x1, x2⟩ := Decompose(x0);
    z1, z2 := G(x1), H(x2);
    DC := Compose(z1, z2)
    end
```

where at least one of G and H is DC. If one is not DC, then it typically can be taken to be Id, the identity function. We refer to *Decompose* as the decomposition operator, G and H as the component operators, *Compose* as the composition operator, *Primitive* as the control predicate, and *Directly-Solve* as the primitive operator.

The following algorithm is an instance of this program scheme:

```
1d-Maxsum2(A[1 .. n])::=
  if n = 1
  then 1d-Maxsum2 := ⟨A[1], A[1]⟩
  else begin
    ⟨best1, right1⟩, u := 1d-Maxsum2(A[1 .. n − 1]), A[n];
    1d-Maxsum2 := ⟨max(best1, right1 + u, u), max(right1 + u, u)⟩
    end
```

1d-Maxsum2 actually solves a slightly stronger form of the 1D-MAXSUM1 problem and is the first algorithm we will derive. On input $A[1 \,..\, n]$ it produces a 2-tuple of values $\langle best, right \rangle$ where *best* is the value satisfying the specification 1D-MAXSUM1. When $A[1 \,..\, n]$ has length 1, the solution is directly computed by $\langle A[1], A[1] \rangle$. Otherwise $A[1 \,..\, n]$ is decomposed into $A[1 \,..\, n-1]$ and $A[n]$ which are then processed by a recursive call and the identity function respectively. The resulting intermediate solutions are composed by $\langle \max(best_1, right_1 + u, u), \max(right_1 + u, u) \rangle$. Here and in later algorithms we omit an explicit call to a decomposition operator when it is simpler to perform the decomposition implicitly in the code.

The main difficulty in designing an instance of the divide-and-conquer scheme for a particular problem lies in constructing decomposition, component, and composition operators that work together. The following strategy extends the strategy

called DS1 in [8]. Steps 2 and 3 contain extensions for handling failures that can arise in applying this strategy. Both typically involve modifying the original specification then retracing earlier steps using the modified specification.

Step 1. Choose a simple decomposition operator.

Step 2. Let the component operators be either of DC or Id depending on the output types of the decomposition operator. If there is a mismatch between the outputs of the decomposition operator and the inputs expected by the component operators, then generalize the input domain and input/output relation of the component operators to eliminate this mismatch. If the original specification is modified, then return to Step 1.

Step 3. Derive the control predicate based on the conditions under which the decomposition operator can be successfully applied.

Step 4. Solve for the input/output relation of the composition operator using a constraint expressing the correctness of the divide-and-conquer scheme. If there is a mismatch between the inputs of the composition operator and the outputs generated by the component operators, then generalize the output domain and input/output relation of the component operators to eliminate this mismatch. If the original specification is modified, then return to Step 1. Otherwise, use the derived input/output relation to set up a specification for the composition operator and construct an algorithm for it.

Step 5. Design an algorithm for the primitive operator.

A more formal treatment of this and other strategies may be found in [8, 9]. This strategy seems to lead most frequently to useful algorithms because in the formal structure of divide-and-conquer algorithms [7] there are more constraints on the decomposition operator than on the component and composition operators. Consequently it is easier to satisfy the decomposition constraints first.

'Solving for' the input/output relation of the composition operator is the most challenging step and bears further explanation. Let $O_{Decompose}$, O_{DC}, O_G, O_H, and $O_{Compose}$ denote the output conditions for *Decompose*, *DC*, *G*, *H*, and *Compose* respectively. The following constraint relates the output conditions of the subalgorithms to the output condition of the whole divide-and-conquer algorithm:

$$O_{Decompose}(x_0, x_1, x_2) \wedge O_G(x_1, z_1) \wedge O_H(x_2, z_2) \wedge O_{Compose}(z_0, z_1, z_2) \Rightarrow O_{DC}(x_0, z_0). \quad (1)$$

The intuitive meaning of (1) is that if input x_0 decomposes into a 2-tuple $\langle x_1, x_2 \rangle$, and z_1 and z_2 are solutions to inputs x_1 and x_2 respectively, and furthermore solutions z_1 and z_2 can be composed to form solution z_0, then z_0 is guaranteed to be a solution to input x_0. This constraint is used like an equation in five unknowns; given O_{DC} from the original specification we supply expressions for $O_{Decompose}$, O_G, and O_H, then reason backwards from the consequent to an expression over the program variables z_0, z_1, and z_2. This derived expression is taken as the input/output condition of *Compose*.

Returning to the 1D-MAXSUM1 problem, there are several ways to decompose the input array $A[1\,..\,n]$. One way is to split $A[1\,..\,n]$ into a subarray of length $n-1$ and a single element, for example, into $A[1\,..\,n-1]$ and $A[n]$. This choice leads to an algorithm like 1d-Maxsum2 and will be explored below. Another way is to split $A[1\,..\,n]$ into subarrays of roughly equal length, for example, into $A[1\,..\,n/2]$ and $A[n/2+1\,..\,n]$. This choice will be explored in Section 3.

Let us call the target algorithm 1d-Maxsum1 and choose to decompose $A[1\,..\,n]$ into $A[1\,..\,n-1]$ and $A[n]$. How can the decomposed inputs be treated by the component operators? $A[1\,..\,n-1]$ is an array segment so a recursive call seems appropriate. The other subinput, $A[n]$, is an integer, so we might treat it as an instance of a distinct problem and use the identity operator in processing it. Thus let G be 1d-Maxsum1 and H be Id. However to pass $A[1\,..\,n-1]$ to 1d-Maxsum1 we must ensure that it satisfies the input condition, i.e., that $1 \leqslant n-1$. This will be so when $2 \leqslant n$. Consequently the decomposition operator should not be invoked on inputs on inputs of length one and the control predicate is taken to be $n = 1$.

At this point we have the partially instantiated scheme

```
1d-Maxsum2(A[1 .. n]) ::=
  if n = 1
  then 1d-Maxsum2 := Directly-Solve(A(1 .. n]);
  else begin
    best1, u := 1d-Maxsum2(A[1 .. n − 1]), A[n];
    1d-Maxsum2 := Compose(best1, u)
    end
```

where *Directly-Solve* and *Compose* remain to be specified. In order to derive an input/output relation for the composition operator the following instance of (1) is set up:

$$\text{Decompose}(A[1\,..\,n]) = \langle A[1\,..\,n-1],\, A[n]\rangle$$

$$\wedge\ best_1 = \max_{1\leqslant j\leqslant k\leqslant n-1} \sum_{p=j}^{k} A[p] \ \wedge\ u = A[n] \ \wedge\ O_{Compose}(best_0, best_1, u)$$

$$\Rightarrow\ best_0 = \max_{1\leqslant j\leqslant k\leqslant n} \sum_{p=j}^{k} A[p]. \tag{2}$$

To set up (2) we used the following substitutions:

- 1d-Maxsum1 replaces G,
- the identity function replaces H,
- $A[1\,..\,n]$, $A[1\,..\,n-1]$, $A[n]$ replace x_0, x_1, x_2 respectively,
- $best_0$, $best_1$, u replace z_0, z_1, z_2 respectively,
- Decompose $(A[1\,..\,n]) = \langle A[1\,..\,n-1], A[n]\rangle$ replaces $O_{Decompose}(x_0, x_1, x_2)$,
- $best_1 = \max_{1\leqslant j\leqslant k\leqslant n-1} \sum_{p=j}^{k} A[p]$ replaces $O_G(x_1, z_1)$,
- $u = A[n]$ replaces $O_H(x_2, z_2)$.

Intuitively, (2) is to be interpreted as follows: $A[1\,..\,n]$ is decomposed into subinputs $A[1\,..\,n-1]$ and $A[n]$, and $best_1$, the output of the recursive call, is the maximum over the sums of all subarrays of $A[1\,..\,n-1]$ and u, the output of the other component operator, is $\mathrm{Id}(A[n])$, and $best_0$, the output of the whole divide-and-conquer algorithm, can be composed from $best_1$ and u according to the (unknown) relation $O_{Compose}(best_0, best_1, u)$, then $best_0$ is the maximum over the sums of all subarrays of $A[1\,..\,n]$. We use this formula to reason backwards from the consequent to input/output specification of the composition operator as follows:

$$\begin{aligned} best_0 &= \max_{1\leq j\leq k\leq n} \sum_{p=j}^{k} A[p] \\ &= \max\left(\max_{1\leq j\leq k\leq n-1} \sum_{p=j}^{k} A[p],\ \max_{1\leq j\leq n-1} \sum_{p=j}^{n-1} A[p]+A[n],\ A[n] \right) \\ &= \max\left(best_1,\ \max_{1\leq j\leq n-1} \sum_{p=j}^{n-1} A[p]+u,\ u \right). \end{aligned}$$

Here we are left with a way to compose $best_0$ from $best_1$, u, and the additional term

$$\max_{1\leq j\leq n-1} \sum_{p=j}^{n-1} A[p] \tag{3}$$

which denotes the maximum over all sums of subarrays of $A[1\,..\,n-1]$ containing $A[n-1]$. The difficulty is that in our program scheme the only inputs to *Compose* are the outputs from the component operators, namely $best_1$ and u, thus *Compose* does not have sufficient information to compute (3). *Compose* could be given an additional argument, such as $A[1\,..\,n-1]$, the input to the recursive call. However having *Compose* compute (3) could be computationally expensive and would involve a more elaborate program scheme. Another approach is to modify one of the component operators so that it makes (3) available as an input to *Compose*. Since only the recursive call has the requisite input ($A[1\,..\,n-1]$) to compute (3), we add a new output value to the specification 1D-MAXSUM1 as follows:

$$\text{1D-MAXSUM2}(A[1\,..\,n]) = \langle best, right\rangle \text{ such that}$$

$$1\leq n \Rightarrow best = \max_{1\leq j\leq k\leq n} \sum_{p=j}^{k} A[p] \wedge right = \max_{1\leq j\leq n} \sum_{p=j}^{n} A[p].$$

Now we go back and redo the derivation using the same choice of decomposition operator. Again setting up an instance of (1) we have

$$\begin{aligned} &\mathrm{Decompose}(A[1\,..\,n]) = \langle A[1\,..\,n-1], A[n]\rangle \\ &\wedge best_1 = \max_{1\leq j\leq k\leq n-1} \sum_{p=j}^{k} A[p] \wedge right_1 = \max_{1\leq j\leq n-1} \sum_{p=j}^{n-1} A[p] \\ &\wedge u = A[n] \wedge O_{Compose}(best_0, right_0, best_1, right_1, u) \\ &\Rightarrow best_0 = \max_{1\leq j\leq k\leq n} \sum_{p=j}^{k} A[p] \wedge right_0 = \max_{1\leq j\leq n} \sum_{p=j}^{n} A[p]. \end{aligned}$$

This formula differs from (2) in that the substitutions for O_G and O_{DC} now reflect 1D-MAXSUM2 (vice 1D-MAXSUM1). Notice also that the set of arguments to $O_{Compose}$ has been expanded to reflect the modified specification. The composition operator now takes $best_1$, $right_1$, and u as inputs and produces $best_0$ and $right_0$ as outputs. Carrying through a derivation analogous to the one above, we have

$$\begin{aligned} best_0 &= \max_{1 \le j \le k \le n} \sum_{p=j}^{k} A[p] \\ &= \max\left(\max_{1 \le j \le k \le n-1} \sum_{p=j}^{k} A[p], \max_{1 \le j \le n-1} \sum_{p=j}^{n-1} A[p] + A[n], A[n] \right) \\ &= \max\left(best_1, \max_{1 \le j \le n-1} \sum_{p=j}^{n-1} A[p] + A[n], A[n] \right) \\ &= \max(best_1, right_1 + u, u) \end{aligned}$$

and

$$\begin{aligned} right_0 &= \max_{1 \le j \le n} \sum_{p=j}^{m} A[p] \\ &= \max\left(\max_{1 \le j \le n-1} \sum_{p=j}^{n-1} A[p] + A[n], A[n] \right) \\ &= \max(right_1 + u, u). \end{aligned}$$

This time we have been able to deduce a relation over the variables $best_0$, $right_0$, $best_1$, $right_1$, and u which gives us the following input/output relation for composition operator:

$$best_0 = max(best_1, right_1 + u, u) \ \wedge \ right_0 = max(right_1 + u, u).$$

From this relation it is easy to construct the composition operator

$$\text{Compose}(right_1, u, best_1) ::= \langle \max(best_1, right_1 + u, u), \max(right_1 + u, u) \rangle.$$

Since the decomposition operator can only be applied when the input array segment has at least two elements, we terminate the recursion when $n = 1$. In this case we have

$$best_0 = \max_{1 \le j \le k \le 1} \sum_{p=j}^{k} A[p] = A[1] \quad \text{and} \quad right_0 = \max_{1 \le j \le 1} \sum_{p=j}^{1} A[p] = A[1],$$

so the primitive operator computes $\langle A[1], A[1] \rangle$. Instantiating the various operations derived above into the divide-and-conquer program scheme we obtain the program 1d-Maxsum2. This algorithm requires linear time and linear space. 1d-Maxsum2

can be transformed into an equivalent iterative form which consumes only linear time and constant space:

```
1d-Maxsum2-iter(A[1 . . n]) ::=
  begin
    best, right := A[1], A[1];
    for i = 2 to n do
      begin
        right := max(right + A[i], A[i]);
        best := max(best, right)
        end;
    1d-Maxsum2-iter := best
    end.
```

3. Deriving a balanced divide-and-conquer algorithm for 1D-MAXSUM

The divide-and-conquer idea is at its best when we can decompose problems into roughly equal sized subproblems—a technique called balancing. When subproblems have roughly equal size we can envision great speedups due to executing the subproblems in parallel. The above derivation of 1d-Maxsum2 suggests that we might create a balanced divide-and-conquer algorithm by decomposing the input $A[1 . . n]$ into subproblems $A[1 . . mid]$ and $A[mid + 1 . . n]$ where $mid = n/2$. Since each subproblem has the same type as the original problem we process both of them recursively. Thus both G and H are DC. Intuitively, we will decompose the input array in half, find the maximum sum subarray in each half, then use them to compute the maximum sum subarray of the whole. As the reader may suspect, it will turn out that more information will be needed in order to compute the solution for the whole array than just the solutions to the two halves.

When we start to set up an instance of the Strong Problem Reduction Principle we find a mismatch between the output of the decomposition operator and the input required by one of the recursive calls. In particular the recursive call corresponding to H in the scheme expects input $A[1 . . n]$ where n may vary, whereas it receives $A[mid + 1 . . n]$ from *Decompose*. This mismatch suggests that we modify the specification 1D-MAXSUM1 so that the lower bound on the range of the input array may vary:

$$1D\text{-}MAXSUM3(A[i . . m]) = best \text{ such that}$$

$$i \leq m \Rightarrow best = \max_{i \leq j \leq k \leq m} \sum_{p=j}^{k} A[p].$$

Now we are obliged to generalize the decomposition operator also, since the input domain has changed. Let $decompose(A[i . . m]) = \langle A[i . . mid],\ A[mid + 1 . . m] \rangle$

where $mid = (i+m)/2$. Again since the decomposed inputs have the same type as the initial input let both of the component operators be recursive calls to 1d-Maxsum3. Setting up an instance of (1) we have

$$\text{Decompose}(A[i\,..\,m]) = \langle A[i\,..\,mid], A[mid+1\,..\,m]\rangle \wedge mid = (i+m)/2$$
$$\wedge\ best_1 = \max_{i \le j \le k \le mid} \sum_{p=j}^{k} A[p] \ \wedge\ best_2 = \max_{mid+1 \le j \le k \le m} \sum_{p=j}^{k} A[p]$$
$$\wedge\ O_{Compose}(best_0, best_1, best_2) \Rightarrow best_0 = \max_{i \le j \le k \le m} \sum_{p=j}^{k} A[p]. \tag{4}$$

Next we attempt to derive an input/output relation for the composition operator by reasoning backwards from the consequent of (4) to a relation over the variables $best_0$, $best_1$, and $best_2$.

$$\begin{aligned} beat_0 &= \max_{i \le j \le k \le m} \sum_{p=j}^{k} A[p] \\ &= \max\Bigg(\max_{i \le j \le k \le mid} \sum_{p=j}^{k} A[p],\ \max_{mid+1 \le j \le k \le m} \sum_{p=j}^{k} A[p], \\ &\qquad \max_{i \le j \le mid} \sum_{p=j}^{mid} A[p] + \max_{mid+1 \le j \le m} \sum_{m=mid+1}^{j} A[p] \Bigg) \\ &= \max\Bigg(best_1, best_2, \max_{i \le j \le mid} \sum_{p=j}^{mid} A[p] + \max_{mid+1 \le j \le m} \sum_{m=mid+1}^{j} A[p] \Bigg). \end{aligned}$$

The resulting derived relation is expressed not only over $best_0$, $best_1$, and $best_2$, but also over the complex expressions

$$\max_{i \le j \le mid} \sum_{p=j}^{mid} A[p] \quad \text{and} \quad \max_{mid+1 \le j \le m} \sum_{m=mid+1}^{j} A[p].$$

These sums compute the maximum sum over all subarrays starting at the left and right ends of the input array respectively. Since the only information that *Compose* has available to it is the output of the recursive calls, namely $best_1$ and $best_2$, this suggests that we add new output variables to the specification 1D-MAXSUM3:

$$\text{1D-MAXSUM4}(A[i\,..\,m]) = \langle left, best, right\rangle \text{ such that}$$
$$i \le m \Rightarrow left = \max_{i \le j \le m} \sum_{p=i}^{j} A[p] \ \wedge\ best = \max_{i \le j \le k \le m} \sum_{p=j}^{k} A[p]$$
$$\wedge\ right = \max_{i \le j \le m} \sum_{p=j}^{m} A[p].$$

Carrying out the derivation this time results in the following input/output relations for the composition operator:

$$left_0 = \max\Bigg(left_1, \sum_{m=i}^{mid} A[p] + left_2 \Bigg),$$
$$best_0 = \max(best_1, best_2, right_1 + left_2),$$
$$right_0 = \max\Bigg(right_2, right_1 + \sum_{m=mid+1}^{m} A[p] \Bigg).$$

Alas, there is yet another complex expression in these derived relations which computes the sum of all elements in an input array. Adding another output variable to the 1D-MAXSUM4 specification results in the strengthened specification:

$$\text{1D-MAXSUM5}(A[i\,..\,m]) = \langle left, best, sum, right\rangle \text{ such that}$$

$$i \leqslant m \Rightarrow left = \max_{i \leqslant j \leqslant m} \sum_{p=i}^{j} A[p] \wedge best = \max_{i \leqslant j \leqslant k \leqslant m} \sum_{p=j}^{k} A[p]$$

$$\wedge\ sum = \sum_{p=i}^{m} A[p] \wedge right = \max_{i \leqslant j \leqslant m} \sum_{p=j}^{m} A[p].$$

Carrying out the derivation this time results in the following output relations for the composition operator:

$$left_0 = max(left_1, sum_1 + left_2), \qquad best_0 = max(best_1, best_2, right_1 + left_2),$$
$$sum_0 = sum_1 + sum_2, \qquad right_0 = max(right_2, right_1 + sum_2).$$

The control predicate and primitive operator are derived as in the previous example. The resulting algorithm is

```
1d-Maxsum5(A[i..m]) ::=
  if i = m
  then 1d-Maxsum5 := ⟨A[i], A[i], A[i], A[i]⟩
  else begin
  ⟨left1, best1, sum1, right1⟩ := 1d-Maxsum5(A[i..(i+m)/2]);
  ⟨left2, best2, sum2, right2⟩ := 1d-Maxsum5(A[((i+m)/2)+1..m]);
  1d-Maxsum5 := ⟨max(left1, sum1 + left2),
                 max(best1, best2, right1 + left2),
                 sum1 + sum2, max(right2, right1 + sum2)⟩
  end.
```

To use 1d-Maxsum5 to solve 1D-MAXSUM we simply select the second component of the tuple returned by 1d-Maxsum5.

Since the decomposition and composition operators take constant time, the time complexity of 1d-Maxsum5 can be described by the recurrence relation

$$T(n) = \begin{cases} O(1) & \text{if } n = 1, \\ 2T(n/2) + O(1) & \text{if } n > 1, \end{cases}$$

which has solution $T(n) = O(n)$. However since the recursive calls are independent of each other, they can be computed concurrently. In parallel 1d-Maxsum5 has time complexity

$$T(n) = \begin{cases} O(1) & \text{if } n = 1, \\ T(n/2) + O(1) & \text{if } n > 1, \end{cases}$$

which is $T(n) = O(\log n)$.

Further speedups are possible for this algorithm. For a fixed input size application a bottom-up version of 1d-Maxsum5 could be implemented in hardware. The recursion can be repeatedly unfolded resulting in a complete binary tree with the

processing of individual array elements taking place at the leaf nodes and composition operations taking place at the internal nodes (Fig. 1(a)). Each leaf node receives an element of the array and solves it according to the base case of the recursion (Fig. 1(b)). Each internal node passes up to its parent the composition of the solutions passed up from its two children (Fig. 1(c)). Notice that inputs are supplied to the bottom of the tree and the solutions come out the top. There is no need to wait for the solution to one input to come out before supplying another input to the leaf nodes. If we should need to process a succession of input arrays then a processing structure of this kind could be pipelined. Each level of the processing tree would be working on one input array. The time between the appearance of the solutions to successive inputs is just the time it takes to pass the computation up one level of the tree. Since the internal processing nodes take constant time, the net time between the appearance of successive solutions is a constant.

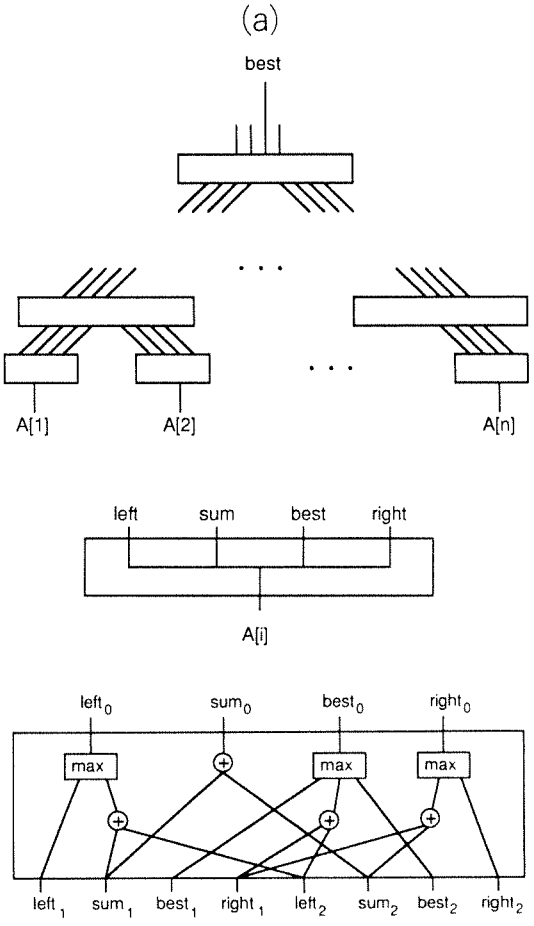

Fig. 1(a). Bottom-up computation of 1d-Maxsum5. (b) Primitive operator. (c) Composition operator.

4. Deriving a balanced divide-and-conquer algorithm for 2D-MAXSUM

The above derivations provide an approach to solving the two-dimensional version of the MAXSUM problem. It is specified by

$$\text{2D-MAXSUM1}(M[1\,..\,n, 1\,..\,n]) = best \text{ such that}$$

$$1 \leq n \Rightarrow best = \max_{\substack{1 \leq a \leq b \leq n \\ 1 \leq c \leq d \leq n}} \sum_{p=a}^{b} \sum_{q=c}^{d} M(p, q).$$

That is, given a nonempty matrix M, find the maximum sum over all rectangular subregions of M. Since an $n \times n$ matrix has $O(n^4)$ rectangular subregions and summing each subregion takes $O(n^2)$ time, there is a straightforward enumeration algorithm which runs in $O(n^6)$ time. A better idea is to create a balanced divide-and-conquer algorithm based on alternately dividing the matrix horizontally and vertically into two submatrices as in Fig. 2. Another essentially equivalent approach is to decompose an $n \times n$ matrix into four $n/2 \times n/2$ submatrices. The first choice of decomposition operator soon leads us to generalize the specification to variable upper and lower bounds along both dimensions of the input matrix:

$$\text{2D-MAXSUM2}(M[i\,..\,j, k\,..\,m]) = best \text{ such that}$$

$$i \leq j \wedge k \leq m \Rightarrow best = \max_{\substack{i \leq a \leq b \leq j \\ k \leq c \leq d \leq m}} \sum_{p=a}^{b} \sum_{q=c}^{d} M(p, q).$$

When the formal manipulations are carried out the need arises to introduce more variables into the output of the problem. In addition to *best*, the cost of the least cost rectangular subregion of the matrix, we need (see Fig. 3):

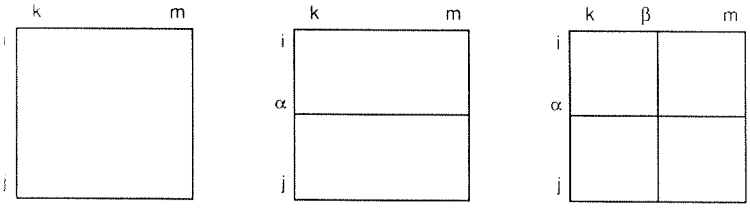

Fig. 2. Decomposing a matrix.

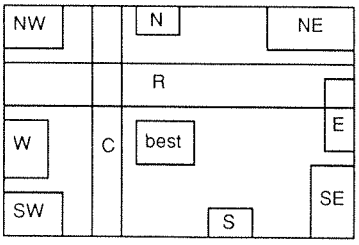

Fig. 3. The output of 2d-Maxsum3.

(1) matrices N, E, S, and W (North, South, East, and West respectively) where, for example, $N(c, d)$ is the least cost rectangular subregion extending between columns c and d, and from the top row downwards;

(2) matrices NE, SE, SW, and NW (NorthEast, etc.) where, for example, $NE(a, c)$ is the sum of the rectangular subregion ranging from the upper left corner of the matrix down to row a and out to column c;

(3) matrices R and C (Row and Column respectively) which give the sums of the horizontal and vertical bands of the matrix. For example, $R(a, b)$ is the sum of all the elements in rows a through b inclusive.

The resulting specification is

$$\begin{aligned}
&\text{2D-MAXSUM3}(M[i..j, k..m]) \\
&\quad = \langle best, NE, N, NW, W, SW, S, SE, E, R, C\rangle \\
&\qquad \text{such that } i \leq j \wedge k \leq m \\
&\Rightarrow best = \max_{\substack{i\leq a\leq b\leq j \\ k\leq c\leq d\leq m}} \sum_{p=a}^{b}\sum_{q=c}^{d} M(p,q) \\
&\qquad \wedge \forall a, b[i \leq a \leq b \leq j \Rightarrow W(a,b) = \max_{k\leq c\leq m} \sum_{p=a}^{b}\sum_{q=k}^{c} M(p,q) \\
&\qquad\qquad \wedge E(a,b) = \max_{k\leq c\leq m} \sum_{p=a}^{b}\sum_{q=c}^{m} M(p,q) \\
&\qquad\qquad \wedge R(a,b) = \sum_{p=a}^{b}\sum_{q=k}^{m} M(p,q)] \\
&\qquad \wedge \forall c, d[k \leq c \leq d \leq m \Rightarrow N(c,d) = \max_{i\leq a\leq j} \sum_{p=i}^{a}\sum_{q=c}^{d} M(p,q) \\
&\qquad\qquad \wedge S(c,d) = \max_{i\leq a\leq j} \sum_{p=a}^{j}\sum_{q=c}^{d} M(p,q) \\
&\qquad\qquad \wedge C(c,d) = \sum_{p=i}^{j}\sum_{q=c}^{d} M(p,q)] \\
&\qquad \wedge \forall a, c[i \leq a \leq j \wedge k \leq c \leq m \\
&\qquad\qquad \Rightarrow NW(a,c) = \sum_{p=i}^{a}\sum_{q=k}^{c} M(p,q) \\
&\qquad\qquad \wedge NE(a,c) = \sum_{p=i}^{a}\sum_{q=c}^{m} M(p,q) \\
&\qquad\qquad \wedge SE(a,c) = \sum_{p=a}^{j}\sum_{q=c}^{m} M(p,q) \\
&\qquad\qquad \wedge SW(a,c) = \sum_{p=a}^{j}\sum_{q=k}^{c} M(p,q)].
\end{aligned}$$

The composition operator for a horizontal split has the form (using subscript i to denote the outputs produced by solving submatrix i of the initial matrix and letting $\alpha = i + j/2$):

$$best_0 = \max(best_1, best_2, \max_{k \le c \le d \le m} (S_1(c, d) + N_2(c, d))),$$

$$N_0(c, d) = \max(N_1(c, d), C_1(c, d) + N_2(c, d)) \quad \forall c, d: k \le c \le d \le m,$$

$$E_0(a, b) = \begin{cases} E_1(a, b) & \text{if } i \le a \le b \le \alpha, \\ \max_{k \le c \le m}(SE_1(a, c) + NE_2(b, c)) & \text{if } i \le a \le \alpha < b \le j, \\ E_2(a, b) & \text{if } \alpha + 1 \le a \le b \le j \end{cases}$$

(S_0 and W_0 are similar to N_0 and E_0 respectively),

$$R_0(a, b) = \begin{cases} R_1(a, b) & \text{if } i \le a \le b \le \alpha, \\ R_1(a, \alpha) + R_2(\alpha + 1, b) & \text{if } i \le a \le \alpha < b \le j, \\ R_2(a, b) & \text{if } \alpha + 1 \le a \le b \le j \end{cases}$$

(C_0 is similar to R_0),

$$NE_0(a, c) = \begin{cases} NE_1(a, c) & \text{if } i \le a \le \alpha, \\ NE_1(\alpha, c) + NE_2(a, c) & \text{if } \alpha + 1 \le a \le j \end{cases}$$

(SE, SW, and NW are similar to NE).

The composition operator for a vertical split is analogous.

The complexity of the resulting algorithm, called 2d-Maxsum3, is figured as follows. Let the original matrix be square and of size $n \times n$. The computation of *best* requires $O(n^2)$ time, N_0 and S_0 require $O(n^2)$ time (each of n^2 elements requires constant time, for a total of $O(n^2)$), E_0 and W_0 require $O(n^3)$ time (some of the n^2 elements require constant time and $O(n^2)$ elements require $O(n)$ time, for a total of $O(n^3)$ time), NE_0, SE_0, SW_0, and NW_0 require $O(n^2)$ time (each of n^2 elements requires constant time, for a total of $O(n^2)$), and R_0 and C_0 require $O(n^2)$ time (each of n^2 elements requires constant time, for a total of $O(n^2)$). Thus the composition step requires $O(n^3)$ time. 2d-Maxsum3 decomposes its $n \times n$ input into two subproblems each of size $(n/2) \times n$. The decomposition takes constant time. So the recurrence describing the complexity of 2d-Maxsum3 is

$$T(1, 1) = O(1),$$

$$T(n, n) = 2T(n/2, n) + O(n^3),$$

$$T(n/2, n) = 2T(n/2, n/2) + O(n^3),$$

or simply

$$T(n, n) = 4T(n/2, n/2) + O(n^3)$$

which has solution $T(n, n) = O(n^3)$. Since any sequential algorithm requires $O(n^2)$ time just to scan the input matrix, 2d-Maxsum3 may not be asymptotically optimal.

However, to the author's knowledge it is the fastest known sequential algorithm for this problem.

A considerable speedup can be obtained by executing both the recursive calls and the composition operation in parallel. Most of the elements in the output matrices require constant time to compute, however there are several exceptions. A computation such as

$$E_0(a, b) = \max_{k \leq c \leq m} (SE_1(a, c) + NE_2(b, c))$$

can be computed in O(log n) time—in parallel compute each of the sums $(SE_1(a, c) + NE_2(b, c))$ for $k \leq c \leq m$, then find the maximum sum in O(log n) time using a tournament elimination structure. Another exception is $best_0$ which requires maximizing over $O(n^2)$ sums. Using a structure similar to that for E_0, $best_0$ can be computed in $O(\log n^2) = O(\log n)$ time. One further factor which must be accounted for is the cost of communicating data. Most of the inputs to the composition operator are used only a constant number of times (thereby incurring constant communication cost). Some however, such as $R_1(a, \alpha)$ and $NE_1(\alpha, c)$, are used $O(n)$ times thereby incurring a O(log n) communication cost (each such datum is replicated via a branching communication structure of height O(log n)). In sum, the cost of distributing the input data and computing output values for the composition operation can be performed in O(log n) time. The recurrence for such a parallel implementation is

$$T(n) = \begin{cases} O(1) & \text{if } n = 1, \\ T(n/2) + O(\log n) & \text{if } n > 1 \end{cases}$$

which is $T(n) = O(\log^2 n)$.

If we are interested in processing a sequence of matrices, as for example in real-time image processing, then a bottom-up computation admits pipelining in a manner analogous to the bottom-up computation of 1d-Maxsum5 in Fig. 1. In order to pipeline the composition operator care must be taken to add extra communication structure so that all data flowing through the operator arrive synchronously at the next level regardless of the kind of processing involved. With pipelining 2d-Maxsum3 can produce successive outputs at constant intervals.

Since the inputs and outputs of 2d-Maxsum3 are naturally 2-dimensional, one possible way to realize this pipelined computation in hardware would be in terms of a 3-dimensional structure. The input matrix would enter the 'bottom' plane and the outputs would exit the 'top' plane. Fortunately the amount of data produced and number of addition and maximization operations required are roughly the same on each level of the tree-structured computation. In particular, at each level a total of at most $11n^2$ data are produced, and $O(n^3)$ addition and maximization operations are performed. The need to perform $O(n^3)$ simultaneous additions at each level would seem to be the main bottleneck in terms of the size of the planes. The third dimension gives added flexibility to the routing of data communication lines at the expense of a more difficult fabrication problem. While these requirements probably

exceed our present technology, the speed and direction of research in fabrication methods suggests that it may be feasible to construct special-purpose chips of this kind in the near future.

5. Concluding remarks

One characteristic of our strategy for designing divide-and-conquer algorithms is that each step can be precisely specified by a postcondition. This allows us to treat certain kinds of failures to achieve the postcondition of a given step via contingency rules associated with the strategy. We have presented and illustrated two such rules in this paper. The two divide-and-conquer strategies which were implemented in the CYPRESS system [9] included another example of a contingency rule. There the failure to design a primitive operator (Step 5 in the strategy of Section 3) led to the reformulation of the input condition of the initial specification. This rule was used to automatically discover and supply missing input conditions.

The notion of modifying a specification based on difficulties which arise during algorithm derivation has received prior attention. Manna and Waldinger [5] give an example in the context of deriving a pattern-matching algorithm. Dershowitz [2] provides a rule for strengthening a specification but does not provide conditions under which it is useful to do so. Our technique of adding new output variables to a specification is related to Paige's finite differencing technique [6] and the notion of strengthening the loop invariant during the derivation of while loops [4]. The idea expressed in these references is to introduce a new variable, say C, whose value is maintained equal to some complex expression e. This technique is primarily used for optimizing a looping program. In contrast, our use of strengthening is intended to facilitate design. Also, in the context of divide-and-conquer algorithms we are able to specify more sharply the conditions under which strengthening is required and how to go about doing it.

The MAXSUM problem originally emerged from an approach to pattern recognition devised by Grenander [3]. Bentley [1] presents a sequence of increasingly efficient algorithms for the 1D-MAXSUM problem culminating in a iterative algorithm similar to 1d-Maxsum2-iter. Gries [4] formally derives a similar algorithm. The sequence of derivations presented in this paper were undertaken in response to Bentley's challenge [1] to improve on the $O(n^6)$ time algorithm for 2D-MAXSUM.

One point of this paper is that we were able to use our formal design strategy to derive algorithms which were previously unknown (1d-Maxsum5 and 2d-Maxsum3) and asymptotically faster than previously known algorithms (the parallel version of 1d-Maxsum5 and the sequential and parallel versions of 2d-Maxsum3). Another point of this paper concerns the naturalness with which divide-and-conquer algorithms can be transformed into a parallel format. Such transformations can result in a tremendous reduction in time complexity. We showed how the linear time 1d-Maxsum5 could be computed in logarithmic time in parallel and in constant

time with pipelining, and how the $O(n^3)$ time 2d-Maxsum3 could be executed in $O(\log^2 n)$ time in parallel and in constant time with pipelining.

Acknowledgment

I would like to thank Richard King for discussions of this material and for pointing out possibilities for parallelism in 2d-Maxsum3. A referee pointed out further possibilities. This research was supported by the Office of Naval Research under Contract N00014-84-C-0473.

References

[1] J. Bentley, Programming pearls, *Comm. ACM* **27** (1984) 865–871.
[2] N. Dershowitz, *The Evolution of Programs* (Birkhauser, Boston, 1983).
[3] U. Grenander, *Pattern Analysis* (Springer, New York, 1978).
[4] D. Gries, A note on a standard strategy for developing loop invariants and loops, *Sci. Comput. Programming* **2** (1982) 207–214.
[5] Z. Manna and R. Waldinger, Knowledge and reasoning in program synthesis, *Artificial Intelligence* **6** (1975) 175–208
[6] R. Paige and S. Koenig, Finite differencing of computable expressions, *ACM TOPLAS* **4** (1982) 402–454.
[7] D.R. Smith, The structure of divide and conquer algorithms, Technical Report NPS 52-83-002, Naval Postgraduate School, Monterey, CA, 1983.
[8] D.R. Smith, The design of divide-and-conquer algorithms, *Sci. Comput. Programming* **5** (1985) 37–58.
[9] D.R. Smith, Top-down synthesis of divide-and-conquer algorithms, *Artificial Intelligence* **27** (1985) 43–96.