EXPRESSIONS FOR TIME AND SPACE

IN A

RECURSIVE REALIZATION OF PARALLELISM

Fred M. Tonge
Department of
Information and Computer Science

University of California, Irvine

TECHNICAL REPORT No. 79

May, 1976

EXPRESSIONS FOR TIME AND SPACE IN A RECURSIVE REALIZATION OF PARALLELISM

parallel processing, performance measurements, program modularity, resource allocation

INTRODUCTION

In this paper we study the time and space behavior of a prototypical form of parallelism achieved through recursion. The processes whose behavior is studied are expressed as binary decompositions of processes. Within this context we develop expressions for space used and elapsed time, with and without exercising maximum admissible parallelism.

DECOMPOSITION AND THE PROCESSOR AS RESOURCE ALLOCATOR*

Assume that a procedure is expressed as (defined by) a decomposition into two component procedures. Henceforth, we call such procedures process descriptions. (More completely, a process description consists of a process expression giving the decomposition into component process descriptions and a process body giving inputs, outputs, and resource requirements.) At some level this decomposition must terminate, for example, by expression of a process description as a segment of code in some programming language (here called the base language). Then, excepting the

---

*I am indebted to Bob Barton and Don Lyle of the Burroughs Corporation for my introduction to the ideas in this and the following section.

interpretation of base language when it is encountered, we can view the action of the processor "executing" process descriptions as strictly one of resource allocation--that is, of making available to the component process descriptions the resources that they need.

We define a process as consisting of a process description, values of the process description's inputs, and an allocation of sufficient resources to meet the process description's resource requirements. The processor then "executes" the process by reallocating those resources to the component process descriptions as required and providing to the components the appropriate inputs, thus converting those process descriptions into processes enabled for execution. In general, the resources referred to here are processor-memory combinations; that is, the "main" processor is itself composed of processor-memory components. A particular version of this architecture is described in more detail in [TO76]. For our purposes, it is sufficient to describe the possible forms of decomposition, and then to consider an issue arising from the assertion that associated with each process description must be a statement of its resource requirments.

FORMS FOR DECOMPOSITION

We consider the following four forms by which a process description is decomposed into its components (see Figure 1).

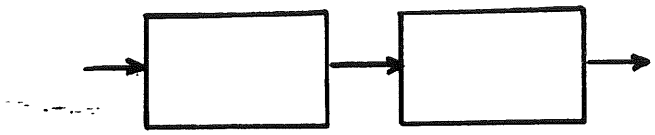
(a) Serial. The two components in specified order

(b) Parallel. The two components in either order, or even simultaneously if permitted by the resources available.

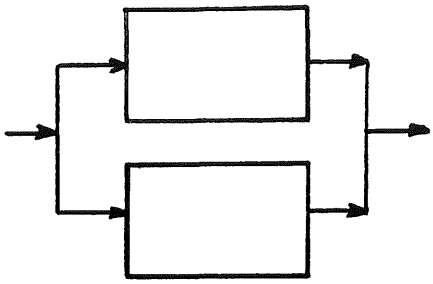
(c) Selection. One of the two components, as selected by a particular data input. The other component is not executed.

(d) Cyclic. One component, then the second, then the first again, repeating until termination is indicated by a particular data output of the "first" component.

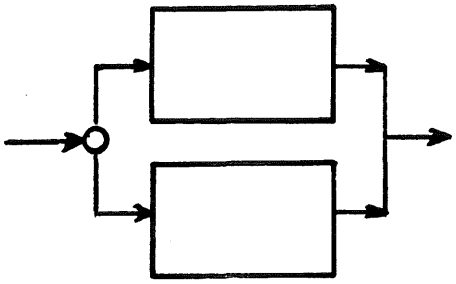
Several of these forms may be generalized in a straightforward way to more than two components, but such generalization may not be desirable from a design-aesthetic standpoint.



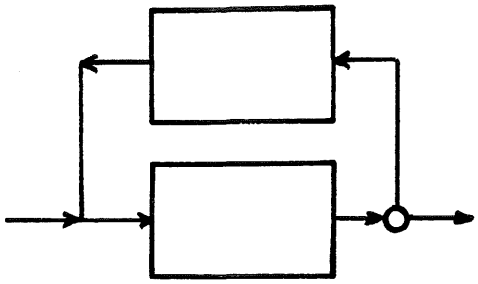
(a) Serial



(b) Parallel



(c) Selection



(d) Cyclic

Figure 1. Forms for Decompositon

ATTAINING PARALLELISM

How then to attain parallelism within such an architecture? Consider for example the addition of two vectors to form a third. It would be desirable to perform as many of the component additions as possible in parallel. Or consider the problem of finding the largest value in a sequence of numbers. Again, it would be desirable to consider a number of subsequences in parallel.

Note that the attainable amount of parallelism depends upon the dimension of the data at hand, and so cannot be specified at the time that the process description is written. Thus, a generalized version of the parallel form, in which the number of components in parallel was specified when the process description was written, does not solve this problem (except in the trivial sense that a "guaranteed large enough" number of components could be specified for all cases). And even then some additional processing would be needed for dividing up the data among component process descriptions and terminating unneeded ones.

Two approaches seem possible for attaining parallelism in this case: splitting the data set into two parts to be handled (recursively) in parallel, and providing a process (possibly in the base language) by which the procedure-writer can request allocation of an arbitrary number of components. This latter approach is that used in [AR75]. It does require the incorporation of resource

allocation requests in the base language or some equivalent (e.g., specific types of components such as "allocators"), and it also requires development of methods for specifying additional data inputs needed to uniquely identify and coordinate the parallel components (for example, in vector additions, the specification of which vector entry is to be added by this component). This approach can be viewed as requiring that part of the input data for parallel procedures be generated by the control structure. In an attempt to keep the resource allocation and process control structure as simple as possible, we have chosen to explore the other alternative.

A major problem with the recursive splitting approach to attaining parallelism is that allowing recursion violates the requirement that each process description have an associated statement of resource requirements. However, if we allow the resource requirement to be stated as a base language expresssion dependent on input values to the process description, we can then ask whether such an expression can be found for recursive splitting. (Such expressions will be necessary for stating the resource requirements of iteration as well as recursion. Iteration is not considered further here.) The straightforward recurrence relation associated with a recursive procedure is of no use, as it is not easily computable without carrying out an equivalent recursion. The remainder of this paper, after an example, is devoted to showing that:

(1) a particular easily implemented strategy for recursive splitting can be shown to be the best attainable, and

(2) the corresponding resource expressions are simple functions of the number of items being processed.

AN EXAMPLE OF RECURSIVE SPLITTING

As an example of recursive splitting, we consider the problem of finding the largest value in a list of numbers. The basic approach is to split the list repeatedly into two sublists until we have lists of length one or two, find the larger value in these smaller lists, and repeatedly recombine by selecting the larger of recombined pairs. More specifically, the FINDMAX process description can be sketched out as shown in Figure 2. This same process description can be represented as a decomposition tree as given in Figure 3.

In the following sections we derive expressions for time and space requirements for such a process description as a function of the number of items being processed. The same expressions, with appropriate coefficient values, describe any recursive splitting process description using this splitting strategy. And following this approach similar expressions can be derived for other specific process descriptions. Thus, a procedure-writer following this approach can use a standard resource expression to the extent that he is willing to use this standard recursive splitting strategy.

```
FINDMAX is INITIALIZATION followed by RECURSION.

   INITIALIZATION is base language (to initialize
    data structures).

   RECURSION is DETERMINE_CASE followed by PROCESS_CASE.

      DETERMINE_CASE is base language (to determine
       whether length of list is two or less).

      PROCESS_CASE is a selection of SIMPLE_CASE
       or MORE_RECURSION_CASE.

         SIMPLE_CASE is DETERMINE_NUMBER
         followed by PROCESS_NUMBER.

            DETERMINE_NUMBER is base language
             (to determine whether length of list
             is less than two).

            PROCESS_NUMBER is a selection of
             ONE or TWO.

               ONE is base language (to return
                none or one items).

               TWO is base language (to return
                larger of two items on list).

         MORE_RECURSION_CASE is SPLIT followed
          by PROCESS_SPLIT.

            SPLIT is base language (to divide
             the list into two parts).

            PROCESS_SPLIT is INITIATE_RECURSIONS
             followed by RECOMBINE.

               INITIATE_RECUSIONS is RECURSION
                in parallel with RECURSION.

               RECOMBINE is base language
                (to return larger of values
                from two recursions).
```

Figure 2. FINDMAX Defined in Standard Recursive Form

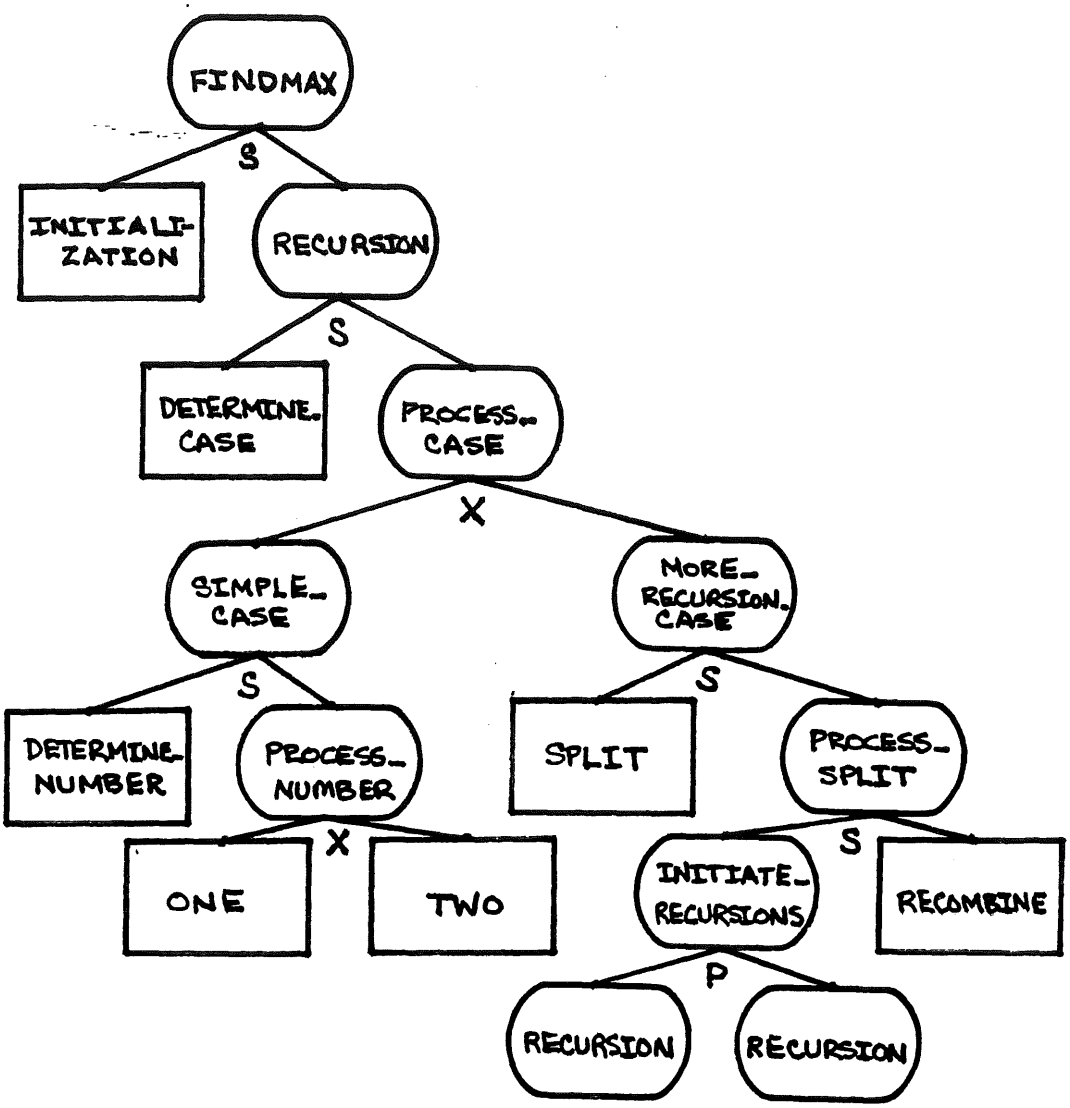


Figure 3. Decomposition Tree for FINDMAX

SPLIT TREES AND K-SPLIT

We can represent the recursive execution of a process by a tree at each of whose nodes is indicated the number of items to be processed. We call such a tree a split tree. Several split trees, for different numbers of items and different splitting strategies, are shown in Figure 4.

Because the terminal nodes of split trees denote one or two items (we call these one-leaves and two-leaves), all nodes have either 0 or 2 descendents. In the following section we make use of some well-known results for binary trees with each interior node having exactly 2 descendents.*

a) For such trees with m leaves, the total number of nodes, independent of the shape of the tree, is:

$$l(m) = 2*m-1.$$

b) For such trees with k leaves, where $k=2\uparrow g$, g a positive integer, and with the leaves below any interior node evenly divided between the two subtrees of that node, the depth of the tree is:.

$$b(k) = \log(k)+1 = g+1.$$

c) For such trees, $d(k)=\log(k)+1$ is the minimum possible depth, and any tree of $m>k$ leaves must have $d(m)>d(k)$. In particular, the minimum depth for such a tree of m leaves is g+1, where $2\uparrow(g-1)<m\leq 2\uparrow g$, or:

---

*We use the following notation. [m]=largest integer not greater than m; log(m)=log of m to the base 2.

$$e(m) = [\log(m-1)]+2.$$

In a later section we demonstrate that a particular splitting strategy, which we call k-split, is optimal for a large class of resource expressions. In k-split, the n items being processed are divided into two groups, one of k items and one of (n-k) items, when k is the largest integer less than n that is an integral power of 2 -- that is, $k=2\uparrow[\log(n)]$. In the case that n is itself an integral power of 2, the two resulting groups are of n/2 items each. Examples (b), (d), and (f) in Figure 4 are split trees produced by k-split.

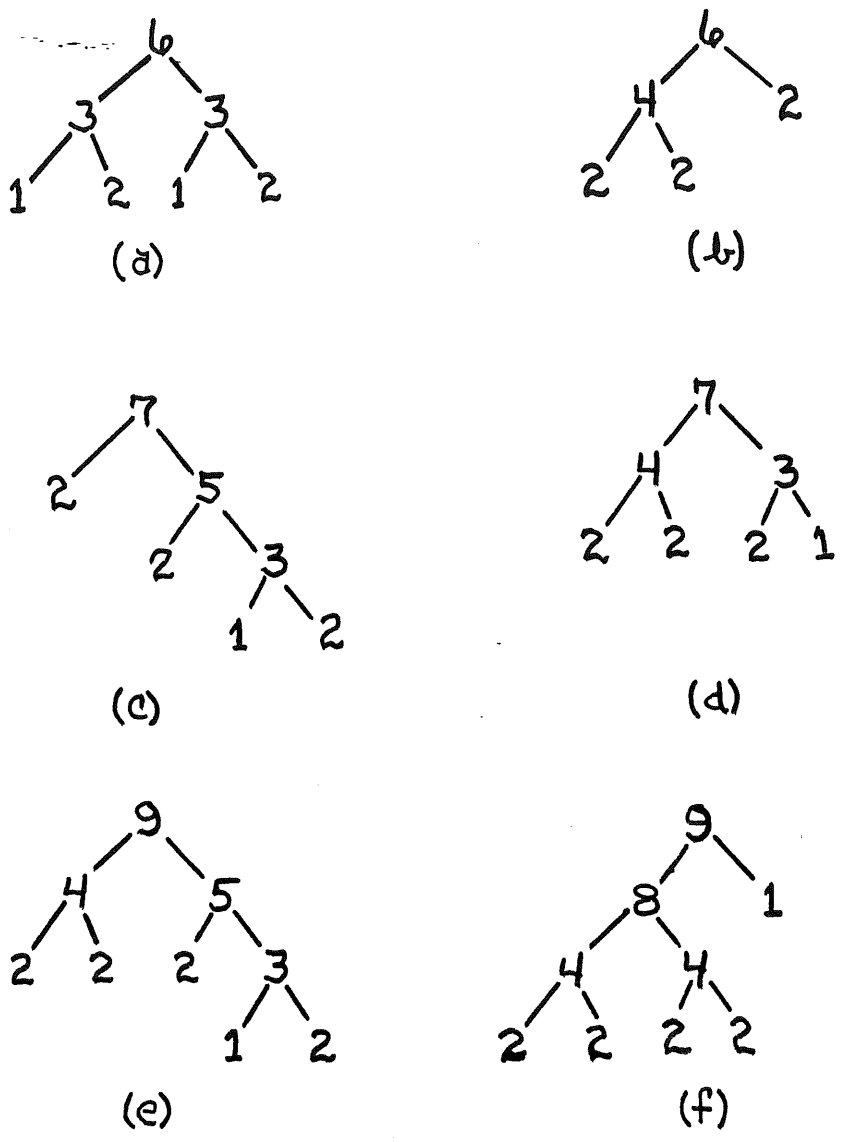


Figure 4. Examples of Split Trees

DERIVATION OF RESOURCE EXPRESSIONS

Development of resource requirement expressions for such a recursively defined process can proceed in three steps:

specification of the general form of expressions for time and space with and without parallelism;

analysis of the cost coefficients of those expressions;

analysis of the parameters of those expressions as determined by the splitting strategy employed.

In these derivations we use the following notation for the characteristics of a split tree of n items:

s1(n) = number of one-leaves

s2(n) = number of two-leaves

r(n) = number of interior nodes

d(n) = depth

General Form of Expressions

Elapsed time, minimum parallelism:

In the case of no parallelism, elasped time is simply the sum of the times to process each item or split each group of items (thus, summing over the nodes in the split tree). Since there may be different processing costs associated with one-leaves, two-leaves, and interior nodes, the total cost is:

$$C(n)=C1*s1(n)+C2*s2(n)+Cr*r(n)+Cf$$

Space used, minimum parallelism:

In this case only one of the two components of a decomposition will be active and assigned space at any time, and so the maximum space required is that of the root and the larger of its components. Repeating this reasoning for the larger component, we see that the maximum space with no parallelism depends on the longest sequence of interior nodes (plus one leaf) in the split tree; that is, on the depth of the split tree. Therefore:

$$K(n)=Kr*(d(n)-1)+\max(K1,K2)+Kf$$

The term max(K1,K2) arises in those cases where the longest sequence of nodes terminates in both a one-leaf and a two-leaf. For the remainder of this analysis, we make the simplifying assumption that $K2 \geq K1$, resulting in:

$$K(n)=Kr*(d(n)-1)+K2+Kf$$

(This simplication overstates K(n) for n=1 if K2>K1.)

Similar analyses show the following expressions to hold for the maximum parallelism cases.

Elapsed time, maximum parallelism:

$$Q(n) = Qr*(d(n)-1)+Q2+Qf$$

Space used, maximum parallelism:

$$J(n)=J1*s1(n)+J2*s2(n)+Jr*r(n)+Jf$$

Cost Coefficients

We assume here that each component has the same cost, independent of the type of decomposition or base language code. When working with a specific implementation of this general architecture, more precise costs can be determined.

Given this assumption, the costs can be determined directly from a decomposition tree for the standard recursive splitting strategy. Figures 5 and 6 indicate the derivation of cost coefficients for the first two cases below. The other coefficients are derived in the same manner.

Elasped time, minimum parallelism:

$$C1 = 7;\ C2 = 7;\ Cr = 8;\ Cf = 2$$

Space used, minimum parallelism:

$$K1 = 5;\ K2 = 5;\ Kr = 5;\ Kf = 1$$

Elapsed time, maximum parallelism;

$$Q1 = 7;\ Q2 = 7;\ Qr = 8;\ Qf = 2$$

Space used, maximum parallelism:

$$J1 = 5;\ J2 = 5;\ Jr = 5;\ Jf = 1$$

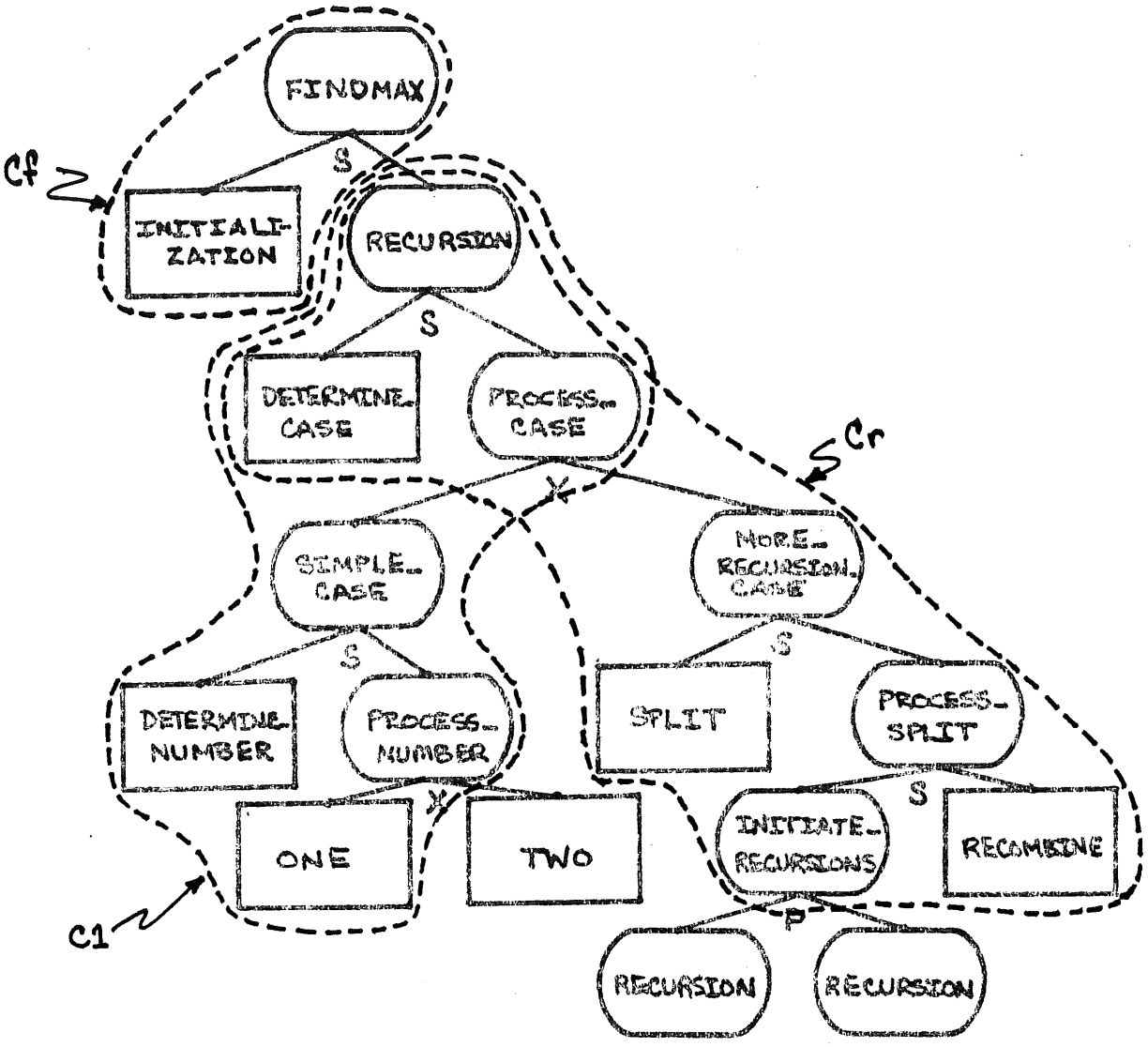


Figure 5. Cost Coefficients for Case of Elapsed Time, Minimum Parallelism

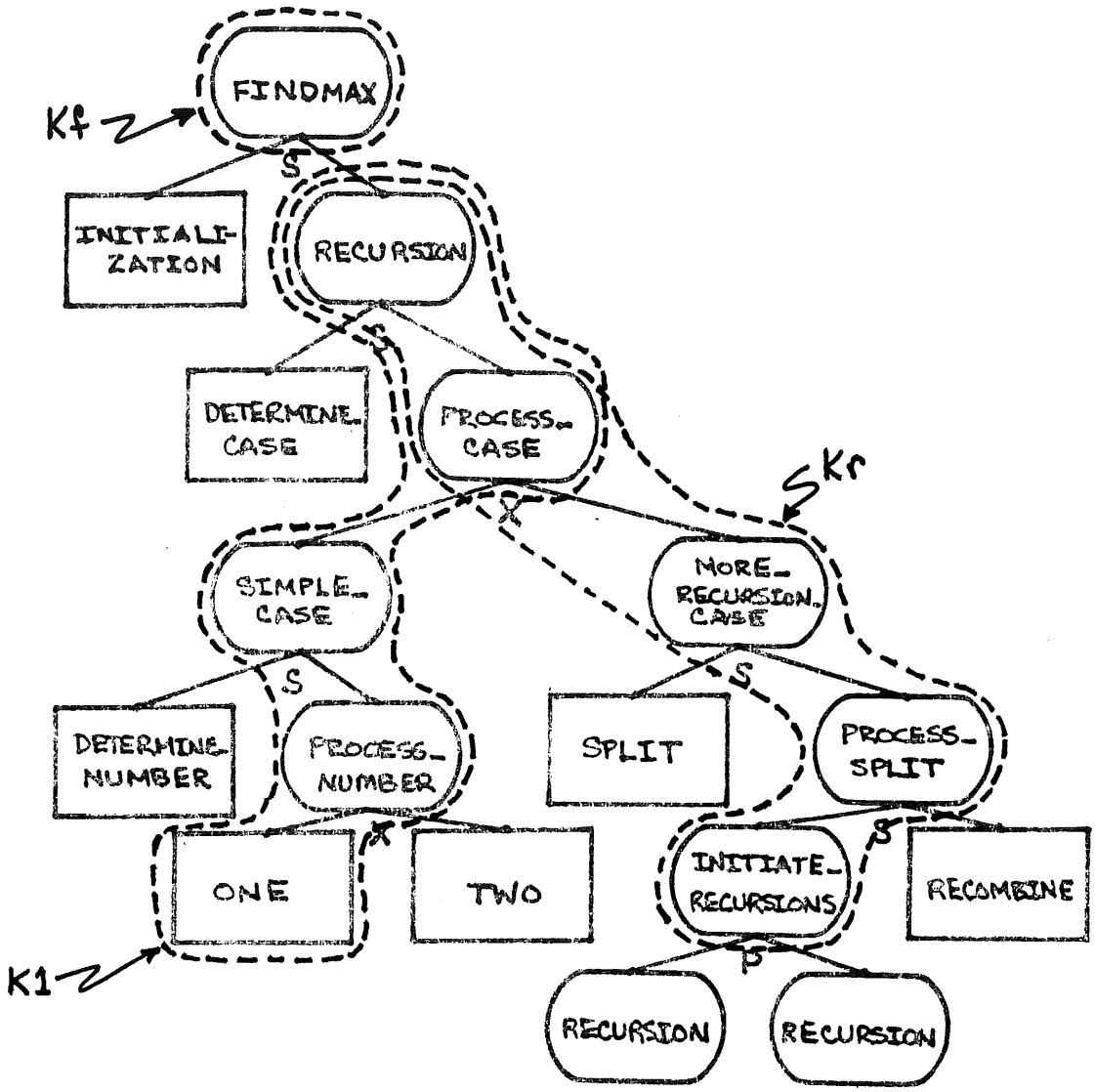


Figure 6. Cost Coefficients for Case of Space Used, Minimum Parallelism

Parameters (for k-split)

One-leaves:

Since k-split repeatedly constructs groups of size $k=2\uparrow g$ until that is no longer possible (i.e., a single item is left), there is at most a single one-leaf (when n is odd). Thus,

$$s1(n) = n-2*[n/2]$$

Two-leaves:

Subtracting the number of one-leaves from n, we find

$$s2(n) = [n/2]$$

Interior nodes:

Using result a) given earlier,

$$r(n) = [(n-1)/2]$$

Depth:

We consider two cases.

(1) If n=k, there will be k/2 leaves in the split tree and, from result b) earlier, $d(n)= \log(k) = \log(n)=\log(n/2)+1$. Also,

$$n = n > n/2, \quad \text{or}$$

$$n > (n-1) \geq n/2, \quad \text{or}$$

$$\log(n) > \log(n-1) \geq \log(n/2), \quad \text{or}$$

$$[\log(n-1)] = \log(n/2),$$

since log(n) and log(n/2) are consecutive integers. Substituting above,

$$d(n) = [\log(n-1)]+1$$

(2) If n>k, then d(n) = max(d(k),d(n-k))+1. Since k is defined such that n>k>n/2, it follows that (n-k)<(n-n/2)=n/2, and so k>(n-k). From result c) earlier and repeated application of the above reasoning until (n-k)=1 or 2, it follows that $d(k) \geq d(n-k)$.

Therefore, d(n)=d(k)+1.

The split tree for k items will have k/2 leaves and so, from result 2) earlier, d(k)=log(k). Then,

$$2*k > n > k, \quad \text{or}$$

$$2*k > (n-1) \geq k, \quad \text{or}$$

$$\log(2*k) > \log(n-1) \geq \log(k), \quad \text{or}$$

$$[\log(n-1)] = \log(k).$$

Substituting above,

$$d(n) = [\log(n-1)] + 1$$

SUMMARY OF RESOURCE EXPRESSIONS

The expressions developed for time and space resources under recursive splitting using k-split are summarized in Figure 7.

The cost coefficients used in that figure reflect a particular simplifying assumption and would be changed for any specific implementation. The magnitude of the increase in space used with parallelism, and also of the decrease in elapsed time, is of the order $n/[\log(n-1)]$. For the units of measure assumed here, the space-time tradeoff is approximately one-to-one.

These four cases are of interest even in a single processor environment. For example, space used with minimum parallelism corresponds to the usual serial processing while space used with maximum parallelism corresponds to pipelining.

Elapsed time, minimum parallelism:

$$C(n) = 7*(n-2*[n/2]) + 7*[n/2] + 8*[(n-1)/2] + 2$$

or, $C(n) = 7.5*n-6$ if n is even

$= 7.5*n+1.5$ if n is odd

Space used, minimum parallelism:

$$K(n) = 5*[\log(n-1)] + 5 + 1$$

$$= 5*[\log(n-1)] + 6$$

Elapsed time, maximum parallelism:

$$Q(n) = 8*[\log(n-1)] + 7+ 2$$

$$= 8*[\log(n-1)] + 9$$

Space used, maximum parallelism:

$$J(n) = 5*(n-2*[n/2]) + 5*[n/2] + 5*[(n-1)/2] + 1$$

or, $J(n) = 5*n-4$ if n is even

$= 5*n+1$ if n is odd

Figure 7. Summary of Resource Expressions

OPTIMALITY OF K-SPLIT

Consider those splitting strategies which generate two descendents at each interior node and which terminate in either one-leaves or two-leaves.

K-split as defined generates for a set of n data items either n/2 two-leaves (if n is even) or (n-1)/2 two-leaves and 1 one-leaf (if n is odd). Clearly this is the minimum number of one-leaves attainable. Since replacing two-leaves with one-leaves must result in two additional one-leaves for each two-leaf replaced, this is also the minimum number of total leaves attainable. That is, k-split results in the minimum number of leaves attainable.

From result a) given earlier, it follows that number of interior nodes is the number of leaves less one. Thus, a splitting strategy which minimizes the number of leaves also minimizes the number of interior nodes. Therefore, k-split results in the minimum number of interior nodes attainable.

From result b) given earlier, we see that for n=k, k-split generates the minimum depth tree. For n>k, the resulting tree will have n/2 leaves if n is even and (n+1)/2 leaves if n is odd. We analyze the two cases separately, using result c) given earlier.

For n even: $2\uparrow(g-1)<n/2\leq 2\uparrow g$ is equivalent to

$2\uparrow g < n \leq 2\uparrow(g+1)$ which is equivalent to

$2\uparrow g < (n-1) < 2\uparrow(g+1)$ since n and $2\uparrow g$ are integer.

Thus, $g = [\log(n-1)]$ and minimum attainable depth = $[\log(n-1)]+1$

For n odd: $2\uparrow(g-1) < (n+1)/2 \leq 2\uparrow g$ is equivalent to

$2\uparrow g < (n+1) \leq 2\uparrow(g+1)$ which is equivalent to

$2\uparrow g \leq n < 2\uparrow(g+1)$ since n and $2\uparrow g$ are integer, and that is equivalent to

$2\uparrow g < n < 2\uparrow(g+1)$ since n is odd, or to

$2\uparrow g \leq (n-1) < 2\uparrow(g+1)$

Again, minimum attainable depth = $[\log(n-1)]+1$.

Since this is precisely the expression for the depth of a tree generated by k-split, it follows that k-split results in the the minimum depth of tree attainable.

Thus, k-split is optimal for any linear resource expression in number of leaves, number of interior nodes, and depth. It is also optional for expressions which distinguish one-leaves and two-leaves as long as $C2 \leq 2*C1+Cr$, $Q2 \leq 2*Q1+Q2$, etc.

GENERALIZATION OF RECURSIVE SPLITTING

The binary recursive splitting approach presented above can be generalized for splitting the data items into more than two subsets. In the following, we consider the case of a p-way split, with up to q items handled (in base language) at each leaf. We do not fully develop a proof of optimality here, but rather sketch out the results needed for such a proof.

The general form of resource expressions in this case is given by the following two examples:

$$C(n) = (\text{sum},i=1 \text{ to } q: Ci*si(n)) + Cr*r(n) + Cf$$

$$K(n) = (\text{max},i=1 \text{ to } q: Ki*si(n)) + Cr*(d(n)-1) + Kf$$

where, for example, Ci is the generalization of C1,C2,..., and si(n) is the generalization of s1(n), s2(n),... .

The results for full binary trees are extended as follows for p-way trees in which each interior node has exactly p descendents.

a) For such trees with m leaves, the total number of nodes is:

$$l(m,p) = (p*m-1)/(p-1)$$

b) For such trees with k leaves, $k=p\uparrow g$ for g a positive integer and with the leaves below any interior node evenly divided among its subtrees, the depth of the tree is:

$$b(k,p) = \log(k)/\log(p) + 1$$

c) For such a tree with m leaves, m>k, the depth is:

$$e(m,p) = [\log(m-1)/\log(p)]+2$$

Note that the number of leaves in a p-way tree must be an integer of the form $1+j*(p-1)$ for $j=0,1,2,...$

Next we consider the minimum number of leaves necessary to represent n data items, given that at most q items may be handled at each leaf. Note that q must be $\geq (p-1)$ or some cases will not be defined. (For example, for p=4 and q=2, what split tree would represent the case of 3 data items?)

The number of split tree leaves for n data items, given at most q items per leaf, is at least $[(n-1)/q]+1$. However, this number of leaves may not be realizable in a p-way split, in which case the number of leaves must be increased to the next acceptable number of leaves. Taking into account the leaves forced because of the need to split p ways, the number of leaves for n data items is:

$$m(n,p,q) = (p-1)*[([(n-1)/q]-1)/(p-1)]+p.$$

(For p=q=2, the case presented earlier, this simplifies to $[(n-1)/2]+1$.)

Finally, we note that this minimum number of leaves does not always uniquely determine the number of one-leaves, two leaves,...,p-leaves. For example, for p=q=3 and n=5, there must be at least 3 leaves, but these can represent item partitions of 3,1,1 or 2,2,1. Further, it is true that if the cost of a leaf is proportional to the number of

items, the total cost of these partitioned items is independent of how they are partitioned into a fixed number of groups.

Thus, we conclude that the splitting strategy equivalent to k-split in the case of p-way splitting with at most q items per node is one of possibly several that achieve the minimum number of leaves. One such strategy is given by the following algorithm for splitting n items into (1 or) p groups:

```
If n≤q
   then done
   else if n = q*p↑g for positive integer g
     then split into p groups of n/p items each
     else begin
        for j=1 thru p-1
           begin create group of
              min(max(q*p↑g for positive integer g),
                 n-(p-j));
              decrease n by size of group
           end;
         create group of size n
     end;
```

CONCLUSION

In the preceding we present a standardized approach to attaining parallelism through recursive splitting. We show that simple expressions can be derived for the time and space resources requirements of this approach. These resource expressions are simple functions of the number of items being processed, and so could be used to control resource allocation -- in particular to implement recursive processing in a decomposition scheme where no process description would be initiated unless the required resources were available for allocation.

We prove that for binary splitting a particular strategy, k-split, is optimal for a wide variety of linear cost equations. And we generalize this result to p-way splitting.

This standard recursive approach can be adapted to many problems and these resource expressions used directly. In those cases where another recursion scheme is used, similar expressions can be developed.

REFERENCES

[Ar75] Arvind and Gostelow, K.P., A New Interpreter for Data Flow Schemas and Its Implications for Computer Architecture, Technical Report No.72, October 1975, Department of Information and Computer Science, U.C. Irvine.

[To76] Tonge, F.M., An Introductory Programming System Based on Structured Decomposition, Department of Information and Computer Science, U.C. Irvine, forthcoming.