THE SEMANTICS AND COMPLEXITY OF PARALLEL PROGRAMS FOR VECTOR COMPUTATIONS. Part II

by

E.K. BLUM

Abstract

Recent research in parallel numerical computation has tended to focus on the algorithmic level. Less attention has been given to the programming level where algorithm is matched, to some extent, to computer architecture. This two-part paper presents a three-level approach to parallel programming which distinguishes between mathematical algorithm, program and computer architecture. In part I, we motivate our approach by a case study using the Ada language. In part II, a mathematical concept of parallel algorithm is introduced in terms of partial orders. This serves as the basis of a theory of parallel computation which makes possible a precise semantics and a precise criterion of complexity of parallel programs. It also suggests some notation for specifying parallel numerical algorithms. To illustrate the ideas presented in part II, we concentrate in part I on parallel numerical computations which have vector spaces as their central data type and which are intended to be excuted on a multi-processor system. The Ada language, with its task constructs, allows one to program computer algorithms to be executed on multi-processor systems, rather than on "vector (pipelined) architectures". To provide a concrete example of the general problem of programming parallel numerical algorithms for multi-processor computers, part I includes a case study of how Ada can be used to program the solution of a system of linear equations on such computers. The case study includes an analysis of complexity which addresses the cost of data moment and process control/synchronization as well as the usual arithmetic complexity.

Part I appears in an issue of BIT dedicated to Peter Naur on his 60th birthday. It is my pleasure to dedicate part II to Jaco de Bakker on the occasion of his 25th anniversary at the Mathematisch Centrum.

1. Introduction

Many computational systems involve operations on vector quantities which can be executed in a concurrent mode. We consider the specification/design of such systems. In our view, specification takes place on three levels : (1) the mathematical algorithm level; (2)the program level; and (3) the computer architecture level. Actually, this view is an idealization of the practical cases in which it is not always possible to separate the three levels of specification, either logically or in order of performance. Nevertheless, we shall treat them as distinct levels.

We assume that specification starts on level 1 with some mathematical equations expressing the system input-output function. For example, the solution of a system of linear equations is specified by the equations $Ax = b$ and $x = A^{-1}b$, where A is a nonsingular $n \times n$ matrix and b is an n-dimensional vector. The equations involve basic operations on basic sets in one or more data types; e.g. the operations in numerical computations are the arithmetic operations on integers and reals and the boolean operations on {0, 1}. However, as in our example, they may include operations on vectors and matrices. Such a purely mathematical specification may be regarded as taking place on the highest level, which we shall call level 1a. On level 1b, the mathematical equations are transformed into equations which specify a mathematical algorithm. In our example, these are the equations defining the familiar Gauss elimination algorithm. (See part I.) The notion of parallel mathematical algorithm needs to be made more precise and we do this in section 2 in a fairly general context. In numerical algorithms involving vectors, a natural parallelism is often implicit in the operations on components. However, there are usually several different modes of parallel execution possible. To achieve the most efficient usage of the architecture available on level 3, it may be necessary to specify explicitly which parallel mode is to be selected.As yet, there is no standard notation for this kind of specification.

2. The three-level way

As remarked above, we wish to propound the thesis that programming of numerical algorithms is one level of a three-level specification procedure. In this section, we shall make more precise our general conception of levels 1 and 2. On level 1, a mathematical specification of a problem is formulated using mathematical concepts and notations. Level 1 is usually separable into two sub-levels, 1a and 1b. On level 1a, a non-algorithmic description of the problem is given, generally as a set of equations to be solved. The equations involve formulas built up in conventional informal notation from variables, constants and operators in prescribed data types. For numerical problems, it suffices to limit the discussion to what we shall call the numerical data types. These comprise the "standard" types, integer, real, Boolean and real array. The operations in these types include the "standard" arithmetic operations on integers and reals, the algebraic operations on vectors and matrices of linear algebra and the "standard" Boolean operations. We need not be more precise than this for our purposes. In our example, the equations are written in the familiar form $Ax = b$, where A is a given nxn non-singular matrix and b is a given n-vector. The solution can be expressed by the equation $x = A^{-1}b$. It is well-known that this equation does not lead to an efficient algorithm for computing x. If we postulate that A can be factored so that $A = LU$, where L is lower triangular and U is upper triangular, then the first equation becomes $LUx = b$. Setting $Ux = y$, we obtain the equivalent pair of equations $Ly = b$, $Ux = y$, which, with $A = LU$, define the problem in a way that leads to an efficient algorithm. As we know, it is generally necessary to apply a permutation matrix, P, to A before it can be factored. So actually, $PA = LU$ and pivoting must be done in the algorithm. This is the kind of mathematical formulation that should be done on level 1a prior to designing a numerical algorithm.

On level 1b, a mathematical algorithm (m.a.) is designed to compute the solution specified by the level 1a equations. The m.a. is likewise expressed in conventional notation using equations, except that the equations must explicitly define their left-hand sides and, for vector problems, logical quantifiers are used to specify iterations. In this example, the mathematical algorithm is the familiar Gauss elimination method for computing the LU factorization and then solving the triangular algorithms also require Boolean conditions expressed by if-then clauses and Boolean or, and, not operations. The equations and conditions again involve formulas constructed from variables and operators in the data types. Although the variables are to be associated with values in the data types, there is no connotation of storage of values. In particular, there is no notion of "old" value and "new" value of a variable, x. These are level 2 concepts. On level 1. to denote the changing values caused by a recursive iteration, an index is associated with x; e.g. x(i-1) represents the old value of x at the beginning van the ith iteration and x(i) the new value after this iteration. There is no dynamic operation like an assignment, which changes the "state" of a "memory location" for x. Memory is a level 2 concept. Thus, level 1b is essentially applicative (i.e. functional). Although there are formal applicative languages like the lambda calculus and much ongoing research into functional programming, at present we usually rely on traditional mathematical notation enriched with special notation to specify m.a.'s of the kind encountered in vector computations. Since there are no standard notations for specifying the parallel aspects of such m.a.'s, we present some of our own in part I.

What we have called "traditional" or "conventional" mathematical notation has evolved over the last three centuries, with many conventions already present in the works of Descartes, Pascal and Fermat. This notation, with its formulas involving numerical constants, letters for variables, symbols such as "+" for operations, superscripts for exponents and punctuation such a parentheses has been absorbed into most modern programming languages. Given a precise syntax (e.g. in Backus-Naur form) in languages like Fortran, Algol, Pascal and Ada, formulas have a

structure representable by parse-trees. The parse-tree embodies a partial ordering of the "applications" of operations to operands.

The parse tree also specifies a partial order subexpressions which corresponds to an ordering of the functional combinators which perform composition and tupling of functions. The functional structure implicit in a parse-tree defines a "data flow" relation between the results of certain operations and the "input" operands of other operations. Indeed, this non-algorithmic functional aspect of formulas is their essential mathematical meaning. Traditional notation also includes equations, in particular, those of the form "variable = formula". A set of such equations may specify a partial order of applications which is not a tree.

The functional structure of formulas and equations is "static". Algorithms involve something more, namely, a dynamic aspect that translates into a temporal partial ordering of the applications of operations, i.e. a "control-flow". In the classical formal notions of algorithm (e.g. of Turing, Post, Markov), which are not concerned with efficiency and complexity, the control flow is essentially sequential (i.e. totally ordered). To deal with efficiency, complexity and parallelism we shall reformulate the notion of algorithm using partial orders of applications. Although these partial orders are implicit in the traditional notation for formulas and equations and would seem to be a consequence of the data flow ordering, this becomes less obvious when logical conditions and iteration specifications are included. In fact, when level 3 computational constraints are imposed, such as number of processors, speeds of operations and inter-processor communication, the partial order may not be entirely a consequence of the static functional structure of formulas. To prepare for this level 1b, attention shoud be focused on how an m.a. should be composed out of partial orderings of applications of operations to operands. Thus, level 1b is intermediary between the functional approach on level 1 a and the typical programming approach on level 2.

On level 2, the m.a. is transformed into a computer algorithm described by a computer program. It is generally accepted that

most programming languages provide practical counterparts to the classical formal systems for specifying "effective procedures".

A computer algorithm is an effective procedure constructed from a basis of operations provided by the particular languages used to write the program. Although the basis differs from language to language, there is a common core, some of which we have already mentioned (e.g. the standard Boolean and arithmetic operations). Aside from recent functional programming languages, most languages provide an assignment operation which changes the state of an abstract store (memory) associated with the variables in a program. There is also an implicit finite-state control unit, as in a Turing machine, and many of the basis operations (e.g. goto) are for the purpose of manipulating the control state. The transformation of an informal specification of a level 1b mathematical algorithm into a formal computer program specifying a computer algorithm on level 2 is the essential task of the computer programmer.

Most programming languages are not applicative with respect to the operations which manipulate the state of the store and the controls state, since there are no state variables of type state that can be used explicitly in programs. For example, in an application of the assignment operation, the "current" store state is an implicit operand and the "next" store state is an implicit result. A goto is an operation on the implicit control state which has a new control state as result. In the multi-task programs in Ada, the control state can have a complex structure distributed over many concurrent tasks. We note that the synchronization primitives of Ada permit quasi-orders rather than partial orders of applications; i.e. it can happen that deadlocks occur. Presumably, such deadlocked programs are incorrect as far as m.a.'s are concerned. A parallel computer algorithm can be regarded as a system of concurrent "processes". The simplest kind of process is the sequential algorithm. How processes are to be joined together into a system of concurrent processes which interact is still a subject of resarch. In the three-level way, a programmer faced with the problem of transforming a given m.a.

(specified in some informal notation) would analyze the m.a. into purely sequential m.a.'s combined using cominators for composition, paralleling and recursion. These combinators are usually realized as synchronization and control operations on level 2. We illustrate this in our Ada case study.

Finally, on level 3, the computer algorithm is transformed into a specific computer system. A computer system is characterized by its architecture which configures a set of hardware and software components.

Some of the more routine difficulties in programming are being ameliorated by "tools" that increase the skill of the programmer in dealing with complex syntactic details and in manipulating files. However, in complex problems, the major difficulties lie elsewhere and are primarily semantic. From the perspective of the three-level way, we identify the following sources of semantic difficulties in the programming of parallel algorithms, which may occur in the sequential case as well.

1) The mathematical algorithm that a programmer starts with may nog be precisely specified and conventional mathematical notation may be inadequate to the task.

2) The mathematical algorithm may be incompletely specified, in which case the specification must be completed on level 2 as part of the programming.

3) The constructs provided in the programming language may not match the mathematical algorithm constructs, making the level 1b-to-level 2 transformation a complex one.

4) Programs describe computer algorithms to be executed on some real computer system, hence must include computer-oriented specifications (e.g. date storage) that permit them to be translated into a level 3 specification that can be executed. The inclusion of these details is often tantamount to the design of a virtual computer system.

5) The level 2-to-level 3 translation is performed by a compiler and a run-time system that must deal with bounded resources (memory, processors, timing constraints, communication bandwidths, interconnections, etc.) and the problems raised by the general-purpose stored-program concept.

To achive efficient compilation the programmer must often provide further implementation-dependent information on level 2. This many have to be done in an environment that is an extension of the formal framework of the programming language, requiring the programmer to attend to level 3 matters as well.

6) For parallel algorithms there is the further difficulty that there is no generally accepted formal model analogous to those for sequential algorithms.

Ideally, the solution to these difficulties is to eliminate level 2 entirely. The historic trend to higher-level languages, which made great early strides when assembly languages were replaced by Fortran, Algol, Pascal etc. has failed to come even moderately close to this ideal with more recent languages. In fact, for parallel algorithms, the quest for maximum speed and efficiency may make this ideal unattainable. However, it seems possible to greatly reduce the effort on level 2 by attacking the deficiencies on all three levels.

3. Mathematical algorithms

If f is an n-ary operation and $a_1,\ldots,a_n$ are valid operands in its domain, then an application of f to these operands yields a result, $f(a_1,\ldots,a_n)$, in one of the numerical data types. For brevity, we shall use vector operands, writing a for $(a_1,\ldots,a_n)$ and f(a) for the result. We shall also allow vector variables as operands. As a simple example, 2+3 denotes the application of the integer add operation to the operand (2,3) with the result 5. (Usually, we use infix notation.). 3+2 is a different application of + with operand (3,2) and the same result. An expression containing variables will be regarded as (denoting) a family of applications. We call such a family a symbolic application. Thus, x+3 denotes a symbolic application which is the family of all applications consisting of adding 3 to an integer. In general, a symbolic application, r(x), parametrized by $x = (x_1,\ldots,x_n)$ is to be regarded abstractly as a mapping $r: D_1\times\ldots\times D_n \rightarrow Ap$, where x_i varies over data type D_i and Ap is the set of all applications of some operation, ω_r. For any value $v = (v_i,\ldots,v_n)$ with $v_i \in D_i$, the mapping yields an application, r(v), in the family. r(v) consists of operation ω_r, an input operand, In(r(v)), and a result, Res(r(v)). If In(r(v)) is in the domain of ω_r, then we require that $Res(r(v)) = \omega_r(In(r(v)))$ and otherwise Res(r(v)) = $\perp$, where $\perp$ is a special element read as "undefined". Thus, if r(x) is given by the formula x+3, where + is integer addition, then r(2) is the application with In(r(2)) = (2, 3) and Res(r(2)) = 5. So r(2) = (+, (2, 3), 5). On the other hand, In(r($\sqrt{2}$)) is ($\sqrt{2}$, 3) and Res(r($\sqrt{2}$)) = $\perp$. In most of this paper, symbolic applications will be specified by conventional mathematical formulas and equations, but this does not preclude their representation by other means, such as tables, for example. For convenient reference, we summarize the foregoing in a definition.

Definition 1. An application, α, is a triple consisting of an operation ω_α, an input, $In(\alpha)$, and a result, $Res(\alpha)$, such that $Res(\alpha) = \omega_\alpha(In(\alpha))$ whenever $In(\alpha) \in Dom(\omega_\alpha)$ and otherwise $Res(\alpha) = \perp$. A symbolic application is a family of applications all having the same operation.

A symbolic application represents a function's graph, but note that the functions involved may be different even when the same operation is used, as in x+2, x+3 and x+y. In algorithms, applications occur in a dynamic context that must be specified. For example, the formula (2+3)*4 can be interpreted a a dynamic sequence of two applications, 2+3 followed by 5*4. The formula, (3+2)*(2+4) denotes either of the sequences (3+2, 2+4, 5*6) or (2+4, 3+2, 5*6 to compute the result, 30. This ambiguity can be used to advantage in paralle algorithm specification. On level 1a, we interpret a mathematical formula as denoting a function and a partial order of functional (data-flow) dependencies. On level 1b, we shall interpret such a formula as suggesting a partial order (p.o.), of applications in a temporal sense. In simple cases, this p.o. can be represented as a finite tree. In the preceding example of formula (3+2)*(2+4), we have the familiar tree,

Figure 1

(Note the implied order of the operands in such a figure.)

Suppose p and q are two p.o.'s on a set S. We say that p is a refinement of q if $q \subset p$ as binary relations. Let $\mathcal{F}$ be a conventional formula and $\leq_\mathcal{F}$ the p.o of its parse-tree. If p.o.'s p and q are refinements of $\leq_\mathcal{F}$, then executing the applications according to p yields the same results as

executing them according to q. For example, both of the application sequences given above are refinements of the p.o. in Figure 1 and yield the same result. We shall use $\leq$ to denote partial orders. We say that an application, r, is a predecessor of application s in a p.o. if $r \leq s$. It is an immediate predecessor if there is no u such that $r \leq u \leq s$.

A formula that contains a (vector) variable $x = (x_1,\ldots,x_n)$ can be interpreted as denoting a set, F(x), of symbolic applications, each parametrized by zero or more of the x_i variables, and having a partial order specified by the structure of the formula. For example, let f(x) be the formula (3+x)*(x+4). The p.o. which f(x) denotes can be represented by a tree having the structure,

Figure 2

where the parameter x can have any integer value. The set, F(x), is the set {x, 3, 4, 3+x, x+4, (3+x)*(x+4)}. (We shall consider constants and variables to be 0-ary operations.) Note that the application (3+x)*(x+4) is obtained by tupling of the applications 3+x and x+4 followed by composition with the application y*z. Later, we shall replace tupling by a parallelling combinator and composition by a sequencing combinator. The tree represents a partial order, $\leq$, on the symbolic applications in F(x). If v is an element in a data type, then F(v) is the tree obtained by substituting v for x. There is a partial order, $\leq_v$, on F(v) induced by $\leq$ in a natural way; e.g. F(2) and $\leq_2$ are given by Figure 1 in this example.

Partial orders have been considered in a variety of contexts as a basis for the semantics of concurrent processes. What is new in what follows is the choice of algebraic applications, suitably indexed, as the elements of a

partial order in a natural way that leads to a precise definition of mathematical algorithm and algorithmic complexity. This also seems to provide a natural mathematical framework for both the static and dynamic structure of parallel algorithms and their computations. We treat applications and partial orders abstractly, rather than syntactically, to allow for a variety of notations on level 1b, including some which may not fit into the usual algebraic syntax of terms. Furthermore, as we wish to give a dynamic interpretation of applications occurring in a temporal partial order in a computation which may be specified by expressions combined with equations and various logical and control operators, we cannot restrict the orders to trees. For example, although the pair of equations, x = 2+1, y = (3+x)*(x+4) can be combined into one with a single right-side term (tree), they can also be interpreted to specify the following algorithm: compute x as the result of the application 2+1, substitute the result, 3, for x in the formula for y, then do the set of applications {3+3, 3+4} in some order (possibly in parallel) and finally do 6*7. This p.o. of applications is not a tree. It can be depicted by a kind of Hasse diagram as in Figure 3. (Downward paths specify the p.o..)

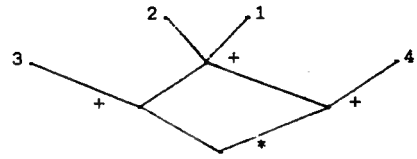


Figure 3

Observe that this p.o. has several <u>minimal</u> elements (1,2,3,4) to start the computation. Compare figure 3 with Figure 4, which is the tree obtained from Figure 2 by tree substitution of 2+1 for x.

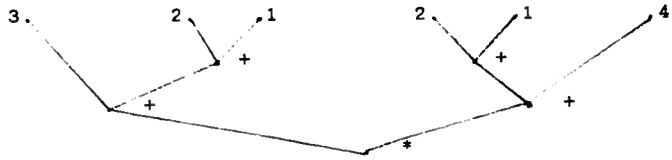


Figure 4

The algorithm depicted in Figure 4 computes the result 42 also, but has two instances of the application 2+1. In transforming this algorithm into a parallel computation, we shall interpret this to mean two executions of the same application. In this case, this is clearly avoidable a priori. However, it is possible that at level 2 such duplication is actually more efficient when executed on two processors if data access operations are included in the complexity analysis. (They can also be included on level 1b by using the identity operation. See below.) In general, such repetition is unavoidable a priori. For example, in the formula (3+(2+x))*((2+y)+4), it may happen that x and y are both set equal to 1 in other equations. To distinguish instances of the same application, we are led to consider indexed sets of applications. The indexing could be done with reference to the syntactic structure of the formulas and equations which specify an algorithm, but, again, we do not wish to restrict the method of indexing to be constrained by conventional syntax as there may be other ways to specify the "control flow". Hence, we shall allow arbitrary posets as indexing sets.

Definition 2. Let $(J, \leq)$ be a poset (the indexing poset) and F(x) a set of symbolic applications, each parametrized by zero or more variables in the sequence of distinct variables, $x = (x_1, x_2, \ldots)$. An algorithmic structure (a.s.) on F(x) is given by an indexing function, ap: J -> F(x), which defines an indexed poset, $(F(x))_J$, of symbolic applications. We denote the a.s. by $((F(x))_J, \leq)$. Further, for each value sequence, v, and $r(x) \in F(x)$, let r(v) be the corresponding application. Define F(v) to be the set of all r(v) such that $Res(r(v)) \neq \perp$. Let $Jv \subset J$ consist of the indices, j, for which $ap(k)(v) \in F(v)$ for all $k \leq j$. Define an indexed poset, $(F(v))_{Jv}$, by the derived indexing function ap_v: Jv -> F(v) given by $ap_v(j) = r(v)$, where r(x) = ap(j). Let $\leq_v$ be the p.o. which is the restriction of $\leq$ to Jv. We call $((F(v)_{Jv}, \leq_v)$ a (parallel) computation structure (pcs) of the a.s. .

Remark. It is convenient to say that application ap(j) precedes ap(k) when $j \leq k$. The variables $x = (x_1, x_2, \ldots)$ which parametrize the symbolic applications in F(x) take values in data types D_i. Each symbolic application, r(x), is a family defined by a mapping, $r: D_{i_1} \times \ldots \times D_{i_n} \rightarrow$ Ap., on finitely many of the data types. We say that r(x) depends on $x_{i_1}, \ldots, x_{i_n}$. It is convenient to regard each variable, x_i, as a symbolic application which is a family of 0-ary applications. For a sequence of values ,v, the application $x_i(v)$ is the 0-ary operation v_i. In most algorithmic structures, the parametrizing variables x_i will be included in the structure as 0-ary applications which are predecessors of certain applications. This will be the case when ordinary formulas are used to specify the structure. We call such x_i input variables. In this paper, we shall assume that all x_i which parametrize an a.s. are input variables. For each (input) value, v, the results of certain maximal applications (if any) can be indexed as outputs. To output an intermediate result (of a non-maximal application), we can force it to be the result of a maximal application simply by adjoining the identity operation.at that point.

An a.s. specifies permissible "control flow" or execution dynamics of a computation. It is also necessary to specify functional structure ("data flow") as would be implicit in a parse-tree. In an a.s., there must also be some relation between the inputs and results of its various applications whereby the needed inputs of any application are the results of applications which are predecessors. This is provided in the next definition.

Definition 3. An algorithm structure $((F(x))_J, \leq_J)$ is said to be a mathematical algorithm (m.a.) if it has the following algorithmic properties:

(1) The set of minimal applications is finite;

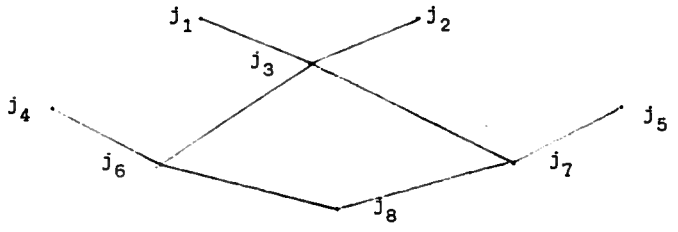
(2) Each application has a finite number of predecesors and immmediate successors;

(3) For each value sequence, v, there exists a data flow function, D_v, defined on J as follows. For $j \in J$, let $r(x) = ap(j)$ have an operation part of arity $n \geq 1$. Then $D_v(j) = (j_1,\ldots,j_n)$, where the j_i are predecessors of j such that $In(r(v)) = (Res(ap_v(j_1)),\ldots,Res(ap_v(j_k)))$.

If $\leq_J$ is total, the m.a. is called sequential and if p is a total order which is a refinement of $\leq_J$, then the m.a. is said to be sequentialized by p.

To illustrate these concepts, let us define an a.s. for the formula (3+(x+y))*((x+y)+4). Let J be the poset given by the diagram in Figure 3a.

Figure 3a



An a.s. is defined by the indexing $ap(j_1) = x$, $ap(j_2) = y$, $ap(j_3) = x+y$, $ap(j_4) = 3$, $ap(j_5) = 4$, $ap(j_6) = 3+(x+y)$, $ap(j_7) = (x+y)+4$, $ap(j_8) = (3+(x+y))*((x+y)+4)$. For v = (2,1), we get the poset F(v) in Figure 3, which is isomorphic to Figure 3a since all symbolic applications are defined for v = (2,1). The data flow function D_v is defined for each j in J as the pair of immediate predecessors when ap(j) has a binary operation.

It follows from algorithmic property 3 that every minimal application is

0-ary, since otherwise it would have at least n ≥ 1 predecessors. From property 1 it follows that there are a finite number of starting points for a computation of an m.a. .From property 2 it follows that at any point in a computation there are only a finite number of parallel paths. We see that an m.a., $((F(x))_J, \leq_J)$, gives rise to a parametrized (by the values of x) family of indexed posets of applications. If we think of each indexed application as residing in a micoprocessor able to execute its operation, each such indexed poset, $((F(v))_{Jv}, \leq_v)$, suggests various possible dynamics (temporal orders) of execution of the applications in it. If $j \leq_v k$, then $ap_v(j)$ should not (usually cannot) be executed later than $ap_v(k)$. This insures that all results needed as inputs to an application are computed prior to its execution. (The microprocessor for $ap_v(j)$ would have to be connected to the one for $ap_v(k)$ to allow the proper data flow.) An execution of all the applications $ap_v(j)$, $j \in$ Jv, satisfying these temporal constraints is a parallel computation of the m.a. for input data v. The p.o. $\leq_J$ allows different dynamical modes of parallel execution. These modes correspond to different refinements of $\leq_J$.

For example, consider the equations x = a+1, y = (3+x)*(x+b). An m.a. which they specify can be represented by a diagram obtained from Figure 3 by replacing the nodes 2 and 4 by a and b respectively. a and b are input variables. Assigning the values 2 and 4 to a and b respectively yields the pcs in Figure 3, which defines the possible parallel computations of the algorithm for these input data. This is a simple example, of course. In the case study,we shall consider a real algorithm in which vector and recursive iterations occur.

As yet, we have said nothing about the effective calculability of the indexing function ap and the data flow function D. Indeed, the symbolic application has not been restricted to be effectively calculable (i.e. computable). We defer such issues to the choice of a particular formal notation for m.a.'s.

The branching produced by execution of Boolean conditions is easily accommodated into the partial order structure of an m.a., but, of course, it should not be confused with the forking produced by parallel execution. One way to include (deterministic) branching in an m.a. is to use the if-then-else operation. If b then f else g can be interpreted applicatively as an operation having three operands, b, f, and g.Thus, the two equations, x = a+5, y = if x > 2 then x+1 else x*3, would specify the m.a. depicted below

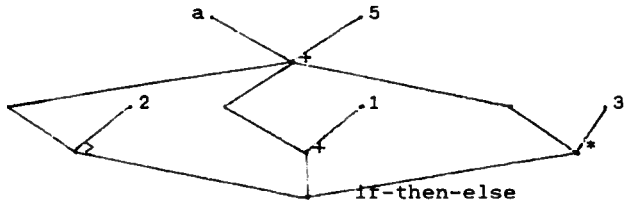
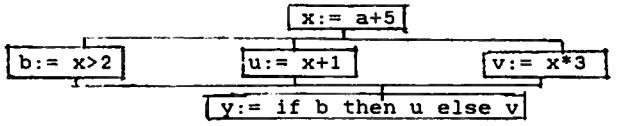


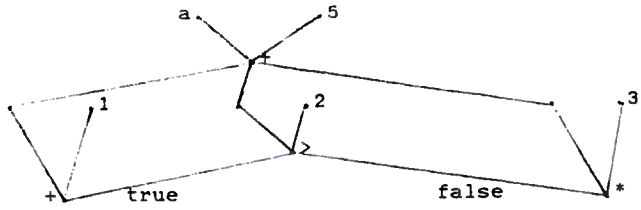
Figure 5

in which if-then-else has the result x+1 if x > 2 and x*3 otherwise, where all x's have been replaced by a+5 (as indicated by the unlabeled nodes). The operations >, +, and * can be applied in any order prior to if-then-else. This parallel interpretation of if-then-else would lead to a level 2 parallel flowchart like



in which there is a three-way fork and join. A more conventional approach to logical branching is simply to consider a symbolic application like x > 2 as denoting two disjoint families of applications, those (like 3 > 2) having the result True and those (like 1 > 2) having the result False. If we denote these families by $r_T(x)$ and $r_F(x)$, then $r_T(3)$ = ((3,2), >, True), $r_F(1)$ = ((1,2), >, False), $r_F(3)$ = ((3,2), >, ⊥) and $r_T(1)$ = ((1,2),>, ⊥). The diagram in Figure 5 would be replaced by the one in Figure 6 below.

Figure 6



The arcs labelled "true" and "false" indicate which application results are to have x+1 as successor and which are to have x*3. Thus, $r_T(x) \leq_J x+1$ and $r_F(x) \leq_J x*3$. According to Definition 3, this p.o. defines an m.a. which has the execution dynamics of the conventional sequential flowchart with a test box for x > 2 having a two-branch exit. Finally, in passing, we note that non-deterministic branching in an m.a. can be done by an application of a "choice" function. For example, Choice(x+1, x+3) would allow either x+1 or x+3 as a result, indicating that either application can be performed at the point of the Choice application. Such non-deterministic algorithms can be defined as sets of m.a.'s. Thus, a formula like Choice(x+1, x+3)*(4+x) would specify a set of two m.a.'s, one given by the formula (x+1)*(4+x) and the other by the formula (x+3)*(4+x) as in Figure 2.

By using the branching combinators (both deterministic ones like if-then-else and non-deterministic ones like Choice) together with other combinators, m.a.'s of a complex structure can be composed out of simple m.a's. It seems possible that the static structure of most m.a.'s in practice can be analyzed and synthesized with relatively few combinators, making possible an algebra of m.a.'s analogous to the algebra of processes in [37]. The following combinators seem most natural: sequencing (or composition), paralleling (or union) and recursion (or iterated composition). Various versions of these combinators have been studied in the context of programming languages, flowcharts and processes [37, 38, 46-48]. We sketch their definition for m.a.'s. (A complete development is the subject of ongoing

Part I:

research. We illustrate their use in the case study.)

In terms of ordering, sequencing is a matter of connecting designated maximal elements (the outputs) of a p.o. to designated minimal elements (the input variables) of another p.o. This is easy to visualize when the Hasse diagrams of the p.o.'s are simple. Intuitively, for two m.a.'s, f and g, we can form a new m.a., f·g, in which designated output nodes of f are connected to designated input nodes of g. For example, suppose we use a conventional equational specification of an m.a. in which f is given by the equation, x = 2+1 and g by y = (3+x)*(x+4). One possible sequencing is given by Figure 3a in which the two instances of the input variable x are replaced by the output node (+) representing the application 2+1. This could be interpreted to mean that the result of 2+1 can be accessed concurrently by 3+x and x+4. To take into account actual data access operations on x (at levels 2 and 3) in which data accesses to x probably occur serially, we may wish to specify an m.a. in which the result of 2+1 (stored in x) is accessed first by x+4 and then by 3+x. To do this, we introduce the identity operation, id , and the application id(x), which has input x and result x. We then form an alternative sequencing, $f \cdot_\sigma g$ shown in Figure 3b.

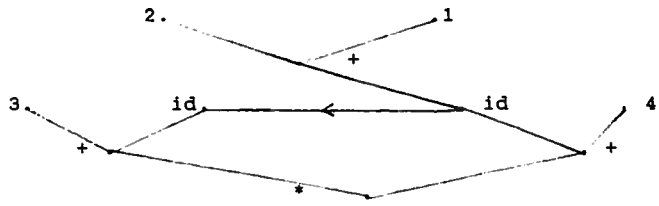


Figure 3b

σ denotes a "connection" mapping which maps the two input (x) nodes of g onto the output (+) node of f. Then the application x is replaced by id. The applications 3+x and x+4 are changed accordingly. Now to specify the serial data accesses, we add a directed edge from one id node to the other, which yields a refinement of the p.o in Figure 3a.

We make these intuitive ideas of sequencing more precise in the following

definition.

Definition 4. Let $f = ((F(x))_J, \leq_J)$ and $g = ((G(y))_I, \leq_I)$ be a.s.'s., with respective indexing functions ap_J and ap_I. We assume $I \cap J = 0$. Let $(r)_{J'}$ be a subfamily of outputs of f and $(y')_I$ a subfamily of input variables of g. Let $\sigma: (y')_I \rightarrow (r)_J$ be a connection function which maps the set $I' \subset I$ of indices of $(y')_I$ into the set of indices $J' \subset J$ of $(r)_J$. Define $G(y|r))_{I|J'}$ to be the indexed family obtained by modifying $(G(y))_I$ as follows: Replace each minimal input variable $ap_I(i) \in (y')_I$ by $ap_J(\sigma(i))$ and replace $i \in I$ by $\sigma(i)$. If $ap_I(i) \in (y')$ is not minimal, replace it by the application which has the identity as its operator and $Res(ap_J(\sigma(i))$ as its input. This yields a new indexing set, $I|J'$, with a p.o. $\leq_{I|J'}$ and a new a.s. $(G(y|r))_{I|J'}$. Let $\leq_{J \cdot I}$ be the p.o. which is the transitive closure of $\leq_J \cup \leq_{I|J'}$. Then the indexed family $(F(x))_J \cup (G(y|r))_{I|J'}$ with $(J \cup I|J', \leq_{J \cdot I})$ as indexing p.o. is an a.s., f·g, called a sequencing of f and g.

In a conventional equational specification of an m.a., the variables on the left sides usually prescribe how a sequencing is to be done. Thus, Figure 3 depicts a sequencing of the elementary algorithm 2+1 with the m.a. shown in Figure 2, as specified by the equations y = (3+x)*(x+4), x =2+1. (Our sequencing combinator for algorithmic structures is somewhat analogous to sequencing of processes in [37], where the variable x is said to be "internalized" and output-input matching is modelled abstractly by monoid morphisms and a "restriction" operation. Processes are level 2 or 3 constructs in our approach.) Also note that definition 4 is independent of the choice of a syntax for m.a. specification. This makes it simple to establish the following basic property of sequencing.

Lemma 1. If f and g are m.a.'s, then so is any sequencing f·g.

Proof. We simply verify the algorithmic properties (1), (2) and (3) in definition 3. Properties (1) and (2) are immediate from the definitions. To establish (3), we simply define a data-flow function, D_{fg}, for f·g in a natural way. Let v be a sequence of input values. For an application, ap, in f we set $D_{fgv}(ap) = D_{fv}(ap)$, where D_{fv} is the data-flow function of f with the input values, v, replacing the input variables of f. For an application, ap, in g, there are two cases. If all inputs of ap are connected to outputs of f, then the results, w, of these outputs for input values v are considered as input values for g and we set $D_{fgv}(ap) = D_{gw}(ap)$, where D_{gw} is the corresponding data-flow function of g. If some input variable, y_i, of g is left unconnected, then the value v_i is used as the input value of y_i in combination with the values w.

We now give the definition of the paralleling combinator.

Definition 5. Let f and g be as in definition 4. The paralleling of f and g is the a.s., f‖g, having (x, y) as input variables and the outputs of both f and g. Its indexing poset is the union I ∪ J with p.o. $\leq_I \cup \leq_J$.

This is the purest kind of parallel combinator, since it does not establish any serial connections between f and g. However, it appears to suffice for the numerical algorithms considered in this paper. Of course, when f‖g is sequenced with another a.s., the outputs of f and g may be commingled.

Lemma 2. If f and g are m.a.'s, then f‖g is an m.a. .

Proof: Immediate from the definitions of m.a. and paralleling.

Definition 6. A recursive iteration on a.s., f, is an an a.s., f^{∞}, obtained by a potentially infinite sequence of sequencings of f with "copies" of itself in which each input of one copy is connected in a uniform way to an output of the preceding copy.

Under appropriate conditions on the sequencing, if f is an m.a., then so

is f^{∞}. We shall not prove such a result here , but illustrate it with a parallel m.a. for the factorial function, n! = n*(n-1)!, starting with 1! = 1. In equation form, we specify parallel n! as follows using some notation explained below.

FACTORIAL(n) is

(y(0) = 1, FACT(0) = 1, x = n);

for i = 0,...,INFINITY loop

(y(i+1) = if x>y(i) then y(i)+1 else 0, FACT(i+1)=y(i)*FACT(i))endloop

We use braces to delimit sets of equations that may be evaluated in parallel, whereas the semi-colon denotes sequencing. The for...loop...endloop notation denotes iterated sequencing of the m.a. delimited by loop and endloop. As in conventional usage, the outputs y(i+1), FACT(i+1) are to replace the inputs y(i), FACT(i) of the next iteration. To represent the p.o. of this m.a., consider the equations in the second pair of braces. These define an m.a., f, with three input variables, x, y, FACT, and a p.o. depicted schematically by the solid lines in Figure 7.1. In Figure 7.2, the solid lines depict a copy of f with inputs replaced by Id nodes, indicating connection to the previous output nodes connected by dashed arcs. Output nodes are also labelled Id.

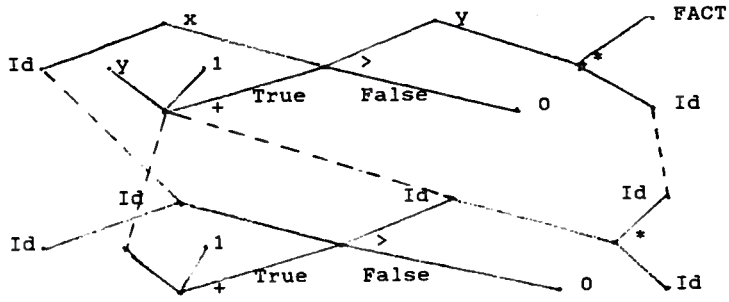


Figure 7.1

Figure 7.2

Together, Figures 7.1, 7.2 depict f·f. An infinite sequence of such connected figures would depict the m.a. f^{∞}, an infinite p.o. which can be defined mathematically as a fixed-point of an appropriate combinator. This is a well-known approach to recursion. (e.g. see [37].) f^{∞} can also be defined set-theoretically, as we have done, as a transitive closure. The construction

of the m.a. for FACTORIAL(n) is completed by the sequencing $f_o \cdot f^{\infty}$, where f_o is the initialization m.a. consisting of the three 0-ary applications n, 1,1 to be connnected to x, y, and FACT in Figure 7.1. Although this is a parallel m.a. for n!, we have used the second interpretation of if-then-else in which x>y denotes two families of applications. Using the value $\perp$ as in Definition 3, each integer value,v, of n determines a finite computation structure, FACTORIAL(v), having v copies of f and the output v!.

We are now able to give a precise definition of the complexity of an algorithm. If an m.a., $((F(x))_J, \leq_J)$, is sequential, it determines a unique computational dynamics for each input value sequence, v, since $\leq_v$ is a total order. The discrete time dynamics can be represented as a sequence of applications (ap(1), ap(2),...), possibly infinite if recursion is involved, where ap(i) is considered to be executed in a time interval $[t_i, t_{i+1})$. If the m.a. is parallel and if $\leq_v$ is not a total order, then any refinement, $\leq_v'$, of $\leq_v$ gives rise to a discrete-time dynamics of execution. We shall define two models of discrete-time dynamics: synchronous and asynchronous. Consider the indexed family $(F(v))_{Jv}$. We partition it into a possibly infinite sequence of disjoint subfamilies S_1, S_2,..., where S_1 is the family of all applications ap(j) such that j is a minimal element of $\leq_v'$, S_2 is the family of ap(j) such that all immediate predecessors of j are (indices of elements) in S_1, S_3 consists of all ap(j) such that j has all its immediate predecessors in $S_1 \cup S_2$ and at least one in S_2 and so on. Thus, S_i contains precisely those ap(j) such that all immediate predecessors of j are in $\cup_{k<i} S_k$ and at least one is in S_{i-1}. Let $t_o < t_1 < \ldots$ be a sequence of time points. In a synchronous dynamics model, all ap(j) in S_i are executed in the interval $[t_{i-1}, t_i)$. Since the ap(j) may not be distinct applications, there can be multiple executions of an application in any interval.

Definition 7. The sequence $S(\leq_v') = (S_1, S_2, \ldots)$ defined above is called the synchronous parallel computation determined by $\leq_v'$ and S_i is its ith step. The synchronous time-complexity of $((F(v))_{J_v}, \leq_v')$ is the length of $S(\leq_v')$. Let R_v be a set of refinements of $\leq_v$. The synchronous time-complexity of the m.a. $((F(x))_J, \leq_J)$ relative to R_v is $C(v) = \min(\text{length } S(\leq_v') : \leq_v' \in R_v)$.

In [49] (Part I), a synchronous model of computation is adopted for measuring complexity. Their model is described in terms of "synchronous processors, each having access to the same storage ... and ... at any parallel step i, any processor may use any input or any element computed by any processor before step i...". Our Definition 7 makes this concept mathematically precise purely in terms of p.o.'s and extends it to include constraints on the p.o.'s. For example, to define the notion of "k-parallelism" [49] we restrict the refinements in R_v to those in which each set S_i has at most k applications. As in most complexity definitions, in numerical algorithms it is usually possible to characterize inputs v according to some size criterion, say size(v), in such a way that complexity is a function of size(v) rather than v itself. For example, in the problem Ax = b, we can take the dimension , n, of the vector space as size(v).

In an asynchronous dynamics model, not all applications in S_i are executed in interval $[t_{i-1}, t_i)$. Some may be delayed waiting for operands which are results of preceding "slow" application executions or may themselves be slow. To model this kind of asynchronous dynamic behavior we introduce a duration function, d:F(v) -> N, which prescribes an integer time duration $d(r(v)) \geq 1$ for each application. Then the completion time for $ap(j) \in (F(v))_{J_v}$ relative to $\leq_v'$ is the integer $t(ap(j))$ given by

$$t(ap(j)) = \max\{t(ap(i)) : i \leq_v' j\} + d(ap(j)).$$

In synchronous dynamics, $d(r(v)) = 1$ for all $r(v)$.

Definition 8. Let $((F(v))_{Jv}, \leq'_v)$ be a parallel computation structure of the m.a. $((F(x))_J, \leq_J)$. Let d_v be a duration function on F(v) and let AS_i be the subfamily of $((F(v))_{Jv}$ consisting of those ap(j) such that t(ap(j)) = i. The sequence (AS_i) is called an asynchronous parallel computation of the m.a.. The length of (AS_i) is the maximum i such that AS_i i nonempty. The asynchronous time-complexity, Ac(v), of the m.a. relative to a set, D, of duration functions and a set, R_v, of refinements of $\leq_v$ is the minimum of the lengths of all asynchronous parallel computations determined b D and R_v.

Most of the theoretical studies of algorithmic complexity are based on the synchronous model of dynamics . Asynchronous complexity studies would probably restrict the set, D, of duration functions. A reasonable restriction would be to functions which prescribe the same duration to all applications having the same operation part or similar operation parts; e.g. all arithmeti operations would have the same duration for all operands. We shall make this kind of restriction in our case study in section 5. We remark, in passing, that if we allow duration functions which are infinite for some applications, then it seems possible to model such properties as safety and liveness of asynchronous computations..This is a matter for future study.

E. K. Blum
Mathematics Department
University of Southern California
Los Angeles 90089, California

March 14, 1989

Acknowledgement: This research was partly supported by N.S.F. grant CCR 8712192