J. Symbolic Computation (1988) **6**, 65–81

Parallel Scheduling of Recursively Defined Arrays

THOMAS J. MYERS† AND MAYA B. GOKHALE‡

University of Delaware, Delaware, U.S.A.

(*Received* 20 *September* 1986)

This paper describes a new method of automatic generation of concurrent programs which construct arrays defined by sets of recursive equations. We assume that the time of computation of an array element is a linear combination of its indices, and we use integer programming to seek a succession of hyperplanes along which array elements can be computed concurrently. The method can be used to schedule equations involving variable length dependency vectors and mutually recursive arrays. Portions of the work reported here have been implemented in the PS automatic program generation system.

1. Introduction

The PS system deals primarily with specifications which describe arrays, and which are simple enough for a symbolic manipulation process whose output will be very straightforward procedural code. The language is described in Gokhale (1987, 1988), and the current implementation in Torgersen (1986). The methods reported here extend the system in order to generate simple classes of parallel programs by solving a series of (pure, linear) integer programming problems. The type of program generated is a "hyperplane" program: a single time-series of concurrent loop bodies, where the loop bodies contain only conditional assignment to array locations. In the input equations, the arrays are defined via recurrences.

Recurrence relations are a common way of expressing problems in a variety of domains. The recurrence relation is often thought of as defining an iteration in which a new value is a function of certain previously computed values. A great deal of research has gone into parallelising recurrence relations. We have used work by Allen (1983*a*, *b*), Banerjee (1979), Cytron (1986), Kuck (1980, 1981), Midkiff (1986), Polychronopoulos (1986), and Warren (1984). Lamport (1974) introduces hyperplane scheduling, in which the nested loop indices of an iterative program are viewed as defining a single multidimensional $M_1 \times M_2 \ldots \times M_n$ array of events. For some common kinds of computation, we can schedule large numbers of events in parallel by selecting a time axis; at any time t, those events will be scheduled which lie on the hyperplane normal to the time axis at t. When this method is successful, the time required is reduced from the volume of the multidimensional array down to the length of some line through it, provided that enough processors are available. These notions will be carefully described below.

There are many ways of treating equational specifications as programs, and issues of transformability and parallelism crop up frequently in the history of this style of program

† Now at Colgate University, Hamilton, New York.
‡ Supported in part by the Institute for Computer Applications in Science and Engineering while the author was in residence at NASA Langley Research Center and in part by UDRF Grant LTR860114.

development. McCarthy (1960) showed how recursive equations could be directly interpreted; much recent work in algebraic manipulation and parallel interpretation of equational programs has centred around the suggestions of Backus (1978). A great deal of such research has emphasized generality and expressive power for equational languages, such as O'Donnell (1985) in which (although a form of compilation is used) the design emphasis is on semantic completeness. The MODEL system of Prywes *et al.* (1983), in contrast, emphasizes restriction towards a restricted class of equations for which simple, efficient programs can be generated, and the PS system of Gokhale (1987) follows this approach with an additional goal of parallel execution.

We introduce notations for our equations and for the parallel schedules by which we compute the values defined by those equations; examples are chosen to suggest the range of difficulty of scheduling quite simple equations. We adapt an existing scheduling method which handles some simple cases by solving sets of linear (integer) inequalities, but which fails in many cases because the sets are arbitrarily large. We generalize in an *ad hoc* way, using the same inequality-solver to generate "minimal" sets of sample points for each conditional recurrence; several heuristics are possible. The extended method is necessary and sufficient for some single-array problems and many sets of mutually recursive arrays, but the sample sets are sometimes inadequate, leading to incorrect schedules. It turns out that the inequality-solver can be used yet again to complete the sample sets, thus guaranteeing correctness for the resulting parallel algorithm. The limits, value, and reasonable future extensions of the research can then be discussed.

In this paper, we introduce a method to find hyperplane schedules through multidimensional arrays from equational problem descriptions, rather than from iterative programs. In the equations, such arrays are merely functions on subranges of the integers. For example,

$$\begin{aligned} P[n,m] &= 1, \text{ if} n = 0 \vee \mathrm{m} = 0 \\ &= P[n-1,m]+P[n,m-1], \text{ otherwise} \end{aligned}$$

defines a function (Pascal's Triangle) on the range $P[0\ldots n, 0\ldots m]$; the equations specify values and dependencies, but not scheduling. Thus, we are given the equations which define each point $P[i,j]$, possibly as depending on points $P[i',j']$ and we want to find the time t in such a way that (i) dependencies are respected, so that $t(P[i,j])$ is greater than each $t(P[i',j'])$, and (ii) the maximal time (i.e. the last time involved in the computation) is minimal. To make the problem more tractable, we assume that the time-function $t(P[i,j])$ for an array P will always be a linear combination of the subscripts i and j. This "hyperplane" assumption enables us to use simple integer programming techniques to solve a variety of subproblems.

Our conditional equations are structured conventionally, and we will not provide a formal definition for the language. We write a sequence of rules, like the Pascal's Triangle definition, with each rule in the form "item = value, if condition". Each item is a scalar, usually an array component; each value is an expression whose value is to be that of the item, provided that the corresponding condition is True. The rules are to be tried in order, so each presupposes the complement of the conditions of earlier rules. We normally end with a default equation, labelled with the trivial "otherwise" condition assumed to be always True.

The Pascal's Triangle problem can be solved either in a goal-driven (top-down) mode which consists of a depth-first traversal of the computation tree starting with goal node $P(n, m)$, or else in a data-driven (bottom-up) mode starting with the required values

$P(0, i)$ and $P(j, 0)$ for each i in $0 \ldots m$, and each j in $0 \ldots n$. We are working in the data-driven model: our goal is a schedule expressed in pidgin Algol, augmented with a concurrent forall loop structure.

$$\text{forall } i \in 0 \ldots m, j \in 0 \ldots n \text{ do if } i = 0 \text{ or } j = 0 \text{ then } P[i, j] := 1$$

is thus a way of expression $(m+1) \times (n+1)$ simultaneous loop activations, of which all but $m+n+1$ will be vacuous. Such a schedule can contain ordinary for loops, with assignment and reassignment to scalars and array components, but our equational language does not include reassignment. A recurrence over an n-dimensional array is therefore described by an $n+1$-dimensional array in the specification, with one dimension representing time. We assume that the ranges $0 \ldots m$ and $0 \ldots n$ are known: there are equations for which this will not be true, and our methods will not handle such problems. Fortunately, there are also many problems where range determination is trivial; we will simply assume that ranges are explicitly declared.

The next section gives examples for motivation; we will then apply our methods to them.

2. Sample Recurrences: Regular and Ill-structured

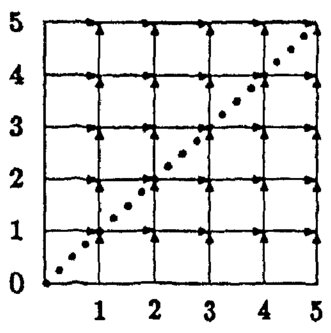
The set of equations shown in Fig. 1 is typical of many problems in numerical analysis in which points are affected only by neighbouring points in a highly regular way, with the boundaries as sole exceptions to the basic rules. A succession of grids is computed (indicated by the k index) in which interior points on a new grid are defined as weighted averages of their neighbours on the previous grid or on the current grid; boundary values are fixed.

This same conceptual structure (highly regular interior with exceptions only at the boundaries) can be seen in a simpler problem from a completely different area. Consider the equations and data-dependency graph for Pascal's Triangle in Fig. 2. The arrows represent vectors of data dependency, or *dependency vectors*. The pattern of dependency is very regular. If the dotted diagonal line represents a time line, all points on a normal to the time line can be computed at one time. The succession of lines (or in the general case hyperplanes) normal to the time line at discrete steps describe an iteration in time. At each step of the iteration, all points on the normal hyperplane can be computed concurrently.

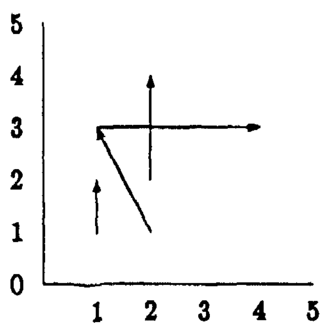
The relaxation algorithm and Pascal's Triangle represent a class of problems which are relatively easy to solve in parallel; our first method will deal only with such problems. The familiar *greatest common divisor* definition on the right of the figure, however, is much less regular. Its dependency arcs do move outwards from the origin, but their structure is not very regular. If there is a "hyperplane of parallelism", it is not immediately discernible. (The *gcd* specification can be written in several ways, with greater or lesser regularity. Some readers may wish to write down a few alternative definitions and follow our methods on them.)

$$\begin{aligned} A_k[i,j] &= 0, \text{ if } i=0 \vee j=0 \vee i=M+1 \vee j=M+1 \\ &= A_{k-1}[i,j] + w(A_k[i,j-1] \\ &\quad + A_k[i-1,j] + A_{k-1}[i,j+1] + A_{k-1}[i+1,j] \\ &\quad - 4 \times A_{k-1}[i,j] \\ &\quad - h^2 \times f(i,j)/4) \text{ otherwise.} \end{aligned}$$

Fig. 1. A Relaxation recurrence.



$P[i,j] = 1$, if $0 \in \{i,j\}$
$= P[i-1,j] + P[i,j-1]$ otherwise



$g[i,j] = i+j$, if $i=0 \vee j=0$
$= g[i-j,j]$ if $i \geq j$
$= g[i,j-i]$ otherwise.

Fig. 2. Pascal's Triangle and greatest common divisor.

Lamport (1974) shows a mechanical method for finding the parallelism in some of these structures. That method depends on the conditions that

1. each point has the same number of dependency vectors, and
2. corresponding vectors for each point have the same length.

This condition holds for the Relaxation and Pascal's Triangle dependencies, but not for *gcd*.

In the next section, we describe a method for finding hyperplanes when the dependency vectors do not necessarily have a constant length. We will lead in to this presentation with a variant on Lamport's method, changed only as far as necessary to adapt to a purely declarative context.

3. Finding Hyperplanes: a Restricted Method and Its Extension

3.1. LINEAR SOLUTIONS FOR ARRAY SCHEDULES WITH FIXED DEPENDENCIES

The fundamental restriction on array scheduling, which no amount of effort on our part will remove, is that no value should be created before the values on which it depends. Thus, in Pascal's Triangle, the time of creation of $P[i,j]$ must in general be greater than the (larger of the two) times of $P[i-1,j]$ and $P[i,j-1]$ are available. Our method finds these times, transforms the equations to include them implicitly, and transforms the surrounding program to use the transformed arrays.

1. *Write a linear time formula for the array, with symbolic coefficients.* For Pascal's Triangle, this gives us the *time equation*

$$t(P[i,j]) = ai + bj.$$

Our first problem is to find usable values for the symbolic coefficients a and b.

2. *Write (strict) dependence inequalities for each recurrence in the array definition.* In this case, the time for $P(i,j]$ must come after the time for $P[i-1,j]$ and also after $P[i,j-1]$; that gives us two dependence inequalities, namely

$$ai+bj > ai+b(j-1) \Rightarrow b > 0 \quad \text{and} \quad ai+bj > a(i-1)+bj \Rightarrow a > 0.$$

3. *Solve the dependence inequalities for the coefficients required.* These are the least integers† a and b for which these dependence inequalities will hold. (This can be trivial in many cases, or can be solved by an integer program otherwise. A variety of methods are applicable (as a basic reference, see Taha, 1975); we have preferred to use the simplex algorithm to find an approximate floating-point solution, then add constraints to force integer solutions.) In this case we get $a = 1$ and $b = 1$.
4. *Substitute the coefficient values into the time formula.* With $t(P[i,j]) = ai + bj$, $a = 1$, $b = 1$, we get $t(P[i,j]) = i + j$. All array elements $P[i,j]$ such that $i+j = t$ will be defined at time t. For given t, these entries comprise a "hyperplane", at right angles to the vector (1, 1) (see Fig. 2). As t is increased from 0 to $t_{\max} = i_{\max} + j_{\max}$‡, we find a sequence of such hyperplanes which cover every point in the array.
5. *Write a preliminary schedule which repeatedly tests each index vector to discover if it is yet time to evaluate the corresponding array element.* For Pascal's Triangle, this is

```
for $t \in 0 \ldots t_{\max}$ do
  forall $i, j$ do
    if $t = i+j$ then
      if $i = 0 \vee j = 0$ then $P[i,j] := 1$
        else $P[i,j] := P[i,j-1] + P[i-1,j]$.
```

6. *Transform the coordinates into a space in which time itself is an array index.* To find a better schedule, we fold the time dimension as one dimension of a transformed array P'. We will define $P'[i',j'] = P[i,j]$ and have $P'[i',j']$ be constructed at time i'. Thus, we transform the coordinates i, j for the array into coordinates i', j' such that $i' = t = i+j$. The j' dimension can be anything not parallel to i'; manipulations are easier if we use j itself. We now have a linear transformation from i, j to i', j', which maps integers into integers. Its inverse will also have that property provided that the mapping is one-to-one as described in Lamport (1974). For the example, we construct
$$i' = i+j,\ j' = j,\ i = i' - j',\ j = j'$$
$$P[i,j] = P'[i',j'] = P'[i+j,j],\ P'[i',j'] = P[i,j] = P[i'-j',j'].$$
7. *Use substitution and simplification to make the transformed array dependent on itself, not on the original.* We now convert the whole problem of evaluating P into one of evaluating P'.
$$\begin{aligned} P'[i',j'] &= P[i'-j',j'] \\ &= 1, \text{ if } i'-j' = 0 \vee j' = 0 \text{ by substitution} \\ &= P[i'-j'-1,j'] + P[i'-j',j'-1] \text{ otherwise} \\ &= P'[i'-1,j'] + P'[i'-1,j'-1] \text{ otherwise by simplification.} \end{aligned}$$
8. *(Trivially) find a schedule for the transformed array.* To schedule the transformed array P', we use its own coordinate i' as (sequential) time and the remaining coordinate j' as (parallel) space:

```
for $i' \in 0 \ldots i'_{\max}$ do
  forall $j'$ do
    if $i'-j' = 0 \vee j' = 0$ then $P'[i',j'] := 1$
      else $P'[i',j'] := P'[i'-1,j'] + P'[i'-1,j'-1]$.
```

† Actually, we are minimising their absolute values.
‡ We will deal later with the problem of finding the range of the transformed time coordinate.

9. *Use substitution to form the transformed top-level program, which refers only to the transformed array.* P was presumably used within some top-level program T: we define the transformed program T' as the result of replacing each term $P[x, y]$ in T by $P'[x+y, y]$†. Further simplification is possible, but the main point has been made: the invertibility of linear transformations, and the use of algebraic simplification on transformed expressions, makes it possible to save a great deal of computation.

Clearly, we have no solution if the dependence inequalities were unsolvable. That simply means that there was no way of scheduling the array as some linear combination of indices. Equally clearly, the preliminary schedule will only be used if the array transformation is unsuccessful; the transformation does not change the length of the time axis, but diminishes the number of active processors required by an order of magnitude. In a wide range of programs such as the Relaxation recurrence above, the transformation will be successful and will not require any ideas beyond those presented in the Pascal's Triangle example. Some will be trivial in that all but one coefficient will be zero, so that the process has the effect of identifying time with an existing dimension, perpendicular to a hyperplane of computation. Non-trivial combinations of indices (as in the example just given) will form hyperplanes which are "diagonal" in the existing array; in such cases, the transformed array's structure will be slightly different from that of the original array, and we might conceivably fail in array transformation if the coordinate transformation is not invertible.

Even when we get a transformed array, it may be larger than the original array; the array P' is twice as large as P. In cases such as Pascal's Triangle or Relaxation, however, we can use the transformed array definition to guide storage reuse. The definition of P did not give us a regular way to reuse the storage space for $P[i, j]$, but $P'[i', *]$ can be thrown away as soon as $P'[i'+1, *]$ is constructed. The MODEL system reported in Prywes (1983), on which ours is based, implements this kind of analysis; it is straightforward, and we will sketch it briefly.

Suppose that we have the recursively defined transformed array P' for which one dimension is time. Suppose also that only the final values in P' are required, or else that elements of P' are to be output as they are produced. Under these assumptions, we can reuse the space occupied by old hyperplanes, putting new ones in their places. Specifically, we can use integer programming once more to solve for the minimal k such that $k-1$ is an upper bound on the time-length of a dependency vector. In the case of Pascal's Triangle the upper bound is clearly 1, so $k=2$. Only two hyperplanes are computationally necessary at any given moment: $k-1$ source hyperplanes and a target hyperplane whose elements are all being simultaneously produced from them.

Now, instead of allocating an array of size $i'_{\max} \times j'_{\max}$, we can allocate an array $k \times j'_{\max}$ and use the following schedule.

```
for $i' \in 0 \ldots i'_{\max}$ do
  forall $j'$ do
    if $i'-j'=0 \vee j'=0$ then $P'[i' \bmod k, j'] := 1$
      else $P'[i' \bmod k, j'] := P'[(i'-1) \bmod k, j'] + P'[(i'-1) \bmod k, j'-1]$.
```

The storage requirements of the computation have now been reduced by a factor of $i'_{\max}/k$, which in many cases (like this one) will be an order of magnitude improvement.

† Alternatively, we could redefine P as a function which merely produces the proper entry from P'.

Our method as described so far primarily reproduces the functionality of Lamport (1974) in a purely equational context, with two improvements due primarily to the context. First, rather than "parallelising" an existing sequential solution, we work on a specification without prior programming effort. Second, our development of a transformed array makes storage reuse straightforward; it is not at all clear how to use this idea on the several kinds of dependencies in procedural code. However, we have not yet shown any method for solving dependency structures which could not be handled by his method.

Now let us consider the third example from section 1: the greatest common divisor definition. This example was chosen because of its highly irregular dependency vectors. This irregularity is reflected in the structure of the dependence inequalities:

$$ai+bj > ai-aj+bj,\ ai+bj > aj-ai+bi \text{ simplify to } 0 > -aj,\ a(2i-j) > b(i-j).$$

By inspection (or by trial and error) we can find that the definition can be scheduled with $a=1$, $b=1$ as before. The same crude schedule or the same transformation can be applied. However, this is a system of two inequations in four unknowns. On the previous example, the i and j subscripts cancelled out, and we could solve a set of inequalities in the symbolic coefficients a and b by an integer program.

This definition is different for two reasons:

- The dependencies are not local, or even of fixed length: dependency vectors in one part of the array will have sizes different from those of other parts of the array. Therefore the inequalities constraining a and b are parametric inequalities using i and j, which cannot be solved by integer programming. From their form, it appears that they should be true for all possible values of i and j. However . . .
- There are different, apparently conflicting, dependencies on different conditionals. The inequality $0 > -aj$ cannot be true throughout the array, since there are points where $j=0$. This did not prevent $a=1$ from being part of a working solution, because $0 > -aj$ was only required at points which had failed the tests $i=0 \vee j=0$. In fact, we only want a dependence inequality to be valid for points (i,j) which would cause that dependence to arise. This clearly depends on the branching structure of the recursive equations used to define the array.

Suppose we are trying to find *gcd(N, M)* using the array g, which (we know in advance) will be of size $N \times M$. The dependence inequality $aj > 0$ must be satisfied at every point (i,j) for which $0 < j < i$. In principle, we could write out $a \cdot 1 > 0$, $a \cdot 2 > 0$, $a \cdot 3 > 0$, and so on up to $a \cdot N > 0$. (Evidently, we must know N as a constant.) Similarly, $a(2i-j) > b(i-j)$ must be satisfied at every point $0 < i \leq j$. We have a total of roughly $(N \times M)/2$ inequalities, which can all be written down explicitly and solved. The process could be mechanised as a partial symbolic evaluation of the *gcd* definition for every i, j point. Remember that the definition is a sequence of equations, each in the form *item* = *value*, if *condition*. Let us define the equation *selected for* a particular (i,j) pair as the first for which the corresponding condition is true. This equation will have one value, with zero or more dependence inequalities involved in the description of that value.

To generate the set of inequalities, we can follow an algorithm such as:

```
for each $i \in 0 \ldots N, j \in 0 \ldots M$ do
  let "item = val, if cond" be the equation selected for $i, j$
  for each dependence inequality in item = val do
    write out that inequality (with constants for $i$ and $j$).
```

Notice that this only works for conditionals which are independent of the contents of the array; they depend upon the indices, i.e. upon *where* we are in the array. For such conditionals, the test can be carried out before the program has any data on which to run because the *cond* can be simplified to *True* or *False* independently of the array's contents.

If the conditionals themselves depend upon the run-time data, then we cannot tell which test is satisfied by a given i, j pair. For example, the conditional equation

$$A[i] = 1 + 2 * A[i-1], \text{ if } A[i-1] \; mod \; 2 = 1$$

has a condition which depends on the value stored in $A[i-1]$. Since we do not know this value before running the program, we cannot tell whether the condition will be true or false and we cannot be sure whether this equation will be selected. In such cases, we have to consider all possible equations which might be selected at run-time. To consider all of them, we must redefine the *selected for* concept: instead of a single equation, it must describe a set. Specifically, an equation will be said to be selected for a particular (i, j) pair if its condition does not simplify to *False* on i, j and no previous test simplifies to *True*. The let statement used above will then become a loop.

After the algorithm above, we are left with a (very large) collection of linear inequalities on a and b. We can pass them to an integer programming system, asking for minimal solutions, but since the collection may be larger than the dependency graph itself, we should not hope for too much. *gcd* provides the simplest case for this kind of analysis: there is one dependence inequality for each (non-boundary) array element, to be computed presumably for the largest possible values of N and M. This is not a very useful solution.

Instead of instantiating each dependence inequality for all possible points at which it might be applied, we will have to take some sort of a sample. For each arm of the conditional, we should select some collection of points (i.e. of values for i and j) at which its dependencies should be satisfied, make linear inequalities by substituting the values into these dependencies, and solve the resulting set. The next question, and the basis for the next refinement of the method, is: what should we use for sample points?

3.2. SAMPLE POINT SELECTION

We will assume that coefficients have been selected for each parameter, and the dependency vectors calculated as before. For each branch in the conditional recursive equation we find a *constraint-set*, i.e. a conjunct of linear comparisons (equalities and inequalities) which arise from the branching structure itself; this will include the negation of the previous conditions. For the g array we are considering, this is awkward because the first line is a disjunction. However, we can split it into a pair of conditional equations, and then we are able to express the branching information by constraint-sets.

$$\begin{array}{lll} & \text{Constraint-set:} & \\ g[i,j] = i+j, \text{ if } i=0 & \{i=0\} & (1) \\ \quad = i+j, \text{ if } j=0 & \{i>0, j=0\} & (2) \\ \quad = g[i-j, j] \text{ if } j<i & \{0<j, j<i\} & (3) \\ \quad = g[j-i, i] \text{ otherwise} & \{0<i, j \geq i\} & (4) \end{array}$$

This *constrained equation set* is the basis for a consistent solution to the scheduling problem, despite the inconsistencies between the dependency vectors under different conditions. The constrained equation set explicitly shows information which was implicit in conditions of the original definition. The constraint-set for each equation defines a set where that equation may actually be used to compute something: outside that set, we do not care whether the dependence inequalities of that equation are satisfied or not.

For example, the dependence inequality $0 > -aj$ derived from equation (3) need only be satisfied when $0 < j \wedge j < i$. Again, we solve for the least i and j which satisfy each constraint. That gives us the base point $p_0 = (1, 2)$, which is our first sample point for equation (3). The inequality $0 > -aj$ from that equation will be instantiated at p_0 as $0 > -2a$. What other points are to be included? For this subproblem, it does not matter; if a is an integer satisfying $0 > -2a$ then $0 > -3a$ will also be true. All we will find about a from equation (3) is that a is a strictly positive integer.

Similarly, the dependence inequality $a(2i-j) > b(i-j)$ from equation (4) need only be instantiated on sample points satisfying $0 < i \wedge j \geq i$. If we find (i_0, j_0) as the least i and j satisfying those, we find $p_0 = (i_0, j_0) = (1, 1)$ for this problem. Instantiating the dependence inequality here leads to $a > 0$. What other points are to be included? Our first strategy will create one for each dimension: p_1 will "stretch" p_0 in the i direction, and p_2 will do the same in the j direction. In other words, p_1 will be the least (i, j) pair satisfying the inequalities that led to p_0 and also satisfying $i > i_0$. Similarly, p_2 will satisfy $j > j_0$. This simple strategy leads to $p_1 = (2, 2)$ and $p_2 = (1, 2)$ which give us the dependency instantiations $2a > 0$ and $0 > -b$, respectively.

Putting all of our inequalities together, we are looking for the least a and b such that $0 > -2a$, $a > 0$, $2a > 0$ and $0 > -b$. There is a lot of redundancy here, but the system is quite finite, and has the solution $a = 1$, $b = 1$ which we found by inspection at the beginning.

At this point, we can restate the whole process, in condensed form. For convenience we will put the sample-point construction process first.

- Generate sample point set
  1. Rewrite the conditional recurrence as a constrained equation set. (If the conditions are expressed as Boolean combinations of linear equalities and inequalities on the indices, then this always works; disjunction is handled through case splitting.)
  2. For each such constraint-set, construct the sample point set. (This includes the least sample point, and the least extensions to it in each dimension.)
- Solve dependence inequalities
  1. Write a linear time-formula for the array evaluation. (This is always trivially possible; just assign coefficients to the indices of the array.)
  2. For each recursive dependence, write the dependence inequality.
  3. Instantiate each dependence inequality on each sample point for its equation.
  4. Solve the set of all instantiations from the previous step to find a time formula for the array evaluation.
  5. Invert the formula, construct and schedule the transformed solution as before.

Clearly this will not always work; there are many recursive definitions in which neither the conditions nor the dependencies can be described by conjunctions of linear formulas.

Other sample point selection strategies are possible. For example, rather than increasing in each dimension, as demonstrated above, we can instead "strengthen" each constraint inequality in each of the constraint sets. The base point would always be the same: in the constraint set $\{0 < j, j < i\}$ for equation (3), we would have $P_0 = (2, 1)$. The extension points would be generated from the constraint sets $\{0+1 < j, j < i\}$ and

$$\begin{aligned} m[k,i] &= G[k,i,k]/G[k,k,k] \text{ if } i>k \\ &= 0, \text{ otherwise} \\ G[k,i,j] &= a[i,j] \text{ if } k=1 \\ &= G[k-1,i,j]-m[k-1,i]*G[k-1,k-1,j] \text{ if } i\geq k \wedge j \geq k \\ &= G[k-1,i,j] \text{ if } i<k \\ &= 0, \text{ otherwise} \end{aligned}$$

The result is G_N.

Fig. 3. Mutual recurrence: Gauss elimination.

$\{0<j, j+1<i\}$; in each case we simply add 1 to the lesser side of the inequation. Somewhat surprisingly, the results are the same for this problem and many others. They are not equivalent, but they are roughly comparable, and neither is entirely satisfactory. (We will explore both incompleteness and inconsistency issues later.)

Another selection heuristic is to include all the corners of the (rectangular) array being defined, with whatever dependence inequalities each is selected for. This was actually our first heuristic; it has the very unpleasant property that it always requires knowledge of the actual values of the upper bounds for the index sets.

4. Multiple Hyperplanes

The previous section demonstrated the method on a single recursively defined array. Essentially the same algorithm can be used for arrays whose definitions involve mutual recurrence. Suppose there are N equations defining N arrays $A_1 \ldots A_N$ which are involved in a mutual recurrence. That is,

$$A_j[i_1, i_2, \ldots] = E(A_j, A_k)$$

where $j, k \in 1 \ldots N$, and the expression E may contain references to both the arrays being defined A_j and to other arrays A_k. Figure 3 shows an example of such a mutual recurrence in a fragment of the specification of Gauss elimination. At each of the k iterations, m is the vector of multipliers for each row of the matrix G. m is defined in terms of G, and G's definition has a reference to m for each array. We construct the time equation for each array and derive from it the constrained inequality set as before. The only novelty here is that we insert an additive constant, a *starting offset*, into the time formula for each array. This means that the time for an n-dimensional array will be expressed as the sum of n terms; it also means that we can express a constant difference between the times for corresponding components of different arrays.

For each array reference on a right-hand side, we get one dependence inequality: there will be two from the definition of m, and three from that of G. If we write that

$$t(m[k,i]) = a+bk+ci \quad \text{and} \quad t(G[k,i,j]) = d+ek+fi+gj,$$

we will end with

$$a+bk+ci > d+ek+fi+gk, \quad \text{if } i>k$$

as the first inequality for m, while the first for G is

$$d+ek+fi+gj > d+e(k-1)+fi+gj, \quad \text{if } i\geq k, j\geq k, k>1.$$

This set of (five) inequalities may be solved by the method of the previous section. Appropriate sample points are chosen for i, j, and k, yielding a set of inequalities with the coefficients a through g as unknowns. The inequalities are solved with an integer program, giving us values for the coefficients, and those values are substituted into the original time inequalities $t(m[k, i])$ and $t(G[k, i, j])$ to give a pair of hyperplanes. The solution hyperplanes are as follows:

$$\begin{aligned} t(m[k, i]) &= a+bk+ci, & t(G[k, i, j]) &= d+ek+fi+gj \\ &= 1+2k, & &= 2k. \end{aligned}$$

Now the problem is to schedule computation of the multiple arrays. A concatenation of schedules for each array (in this example an iterative solution for m followed by an iterative solution for G or vice versa) is incorrect because individual elements of each array depend on elements of the other. Each iteration must include evaluation of elements for every array in the recurrence. A geometric interpretation of this condition is that the hyperplanes are interlaced, a hyperplane for G followed by a hyperplane for m followed by a hyperplane for G and so forth. Therefore, computation of each array must be embedded within a common iteration over time. In addition, each array must be checked to see whether its hyperplane is currently being evaluated. Only if it is on the hyperplane being computed can the array element actually be evaluated.

These two conditions, a common iteration over time and an extra condition to check, are shown in the Gauss schedule.

```
for $t \in 0 \ldots t_{\max}$ do
  forall $i, j, k$ do
    if $t = 2k+1$ then
      if $i > k$ then
        $m[k, i] = G[k, i, k]/G[k, k, k]$
      else $m[k, i] = 0$
    elsif $t = 2k$ then
      if $i = 1$ then $G[k, i, j] = a[i, j]$
    elsif $i \geq k$ and $j \geq k$ then
      $G[k, i, j] = G[k-1, i, j] - m[k-1, i] * G[k-1, k-1, j]$
    elsif $i < k$ then
      $G[k, i, j] = G[k-1, i, j]$
    else $G[k, i, j] = 0.$
```

This follows the form of the first (non-transformed) schedule for Pascal's Triangle. As demonstrated earlier, we may attempt to transform the coordinates to make one dimension of the array a time dimension. This cannot be done with the Gauss example because the inverse mapping does not yield an integer value.

However, it is quite possible to come up with an algorithmically acceptable version. The multiarray hyperplane method will come out with solutions of this form (slightly generalised) for any problem in which array computations must be interleaved, and we can expect to see time formulas which look like this:

$$t(A_1[\ldots]) = nE, \quad t(A_2[\ldots]) = nE+1, \quad \ldots, \quad t(A_n[\ldots]) = nE+(n-1).$$

In this case, E is the expression (like k in the Gauss example) which defines the true time axis, and we can transform the arrays to make time be a dimension of each. (In the Gauss example, this is an identity transformation because k is already a dimension of each.) The constant offsets added to it require only that the schedule specify a sequence of hyperplane activations for each time rather than only one:

```
for $k \in 0 \ldots k_{\max}$ do
  begin
    forall $i, j$ do
      if $k = 1$ then $G[k, i, j] := a[i, j]$
        elsif $i \geq k \wedge j \geq k$ then $G[k, i, j] := \ldots$
        elsif $i < k$ then $G[k, i, j] := G[k-1, i, j]$
        else $G[k, i, j] := 0$;
    forall $i$ do
      if $i > k$ then $m[k, i] := G[k, i, k]/G[k, k, k]$
        else $m[k, i] := 0$
  end
```

This particular method has neither great generality or great elegance, but it covers a common and natural case of mutual recurrence. A great deal can be achieved by such small extensions, each of which arises from some example which cannot be properly scheduled without extension. It seems reasonable to seek a larger generalization, one which will naturally create not only nested blocks but nested for and forall loops, but such recursive applications of the methods will have to be left for future work.

5. Failures and Corrections

5.1. BOUNDARY ADJUSTMENTS

The schedules we have written up to now have assumed that "time" would run from 0 to some undefined maximum value. This is an oversimplification; the same integer-programming techniques which give us our sample points and the schedules themselves can be used to find the starting and ending points for the scheduled loop. As a different example, let us consider finding the number of ways in which a positive integer may be partitioned into a sum of positive integers ($f[n, m]$ is the number of ways in which n can be partitioned into a sum of integers $\geq m$):

$$\begin{aligned} part[n] &= f[n, 1] \\ f[n, m] &= 0, \text{ if } n < m \\ &= 1, \text{ if } n = m \\ &= f[n-m, m] + f[n, m+1] \text{ otherwise.} \end{aligned}$$

The constraint-sets will be $\{n < m\}$, $\{n = m\}$, and $\{n > m\}$; the first two arms of the conditional are non-recursive, but the third requires that we generate sample points (1, 0), (2, 0), and (2, 1). If we assume $t(f[n, m]) = an + bm$ we get the dependence inequalities $0 > -am$ and $0 > b$, so we need a solution for $a > 0$, $2a > 0$, and $b < 0$.

This is easily solved to produce $a = 1$, $b = -1$, so we find that $t(n, m) = n - m$. Unfortunately, this seems to imply a negative time for half the array. It does not matter; we let time range from $t_{\min}$ to $t_{\max}$. It just happens that we have dealt with examples so far in which $t_{\min}$ was 0, but we will actually find $t_{\min}$ and $t_{\max}$ by the same process, which we have skipped over thus far.

Once more, we solve an integer program involving linear inequalities: in this case we already have an expression for time, and we have n and m as index sets (i.e. we know the constants $n_{\min}$, $n_{\max}$, $m_{\min}$ and $m_{\max}$ such that $n_{\min} \leq n \leq n_{\max}$ and $m_{\min} \leq m \leq m_{\max}$. To solve for the minimal (maximal) time, we solve for the i and j which minimise (maximise) the time expression subject to those boundary conditions. In the *part* example, this gives us $t_{\min} = n_{\min} - m_{\max}$ and $t_{\max} = m_{\max} - m_{\min}$.

Similarly, we can note that the proper base value $m_{\min}$ is actually 1; using it throughout the method will make trivial changes in intermediate calculation, but none in the final schedule. (The final array, of course, will be slightly smaller.) Handling details like this is essential to the creation of a working system, but seems to add little to the understanding of the basic problems of equational transformation and scheduling.

5.2. INCOMPLETENESS

The method works successfully on many common examples, but it does fail on examples which are clearly schedulable. Suppose we have $X[0 \ldots N]$ and $Y[0 \ldots N]$ defined as follows:

$$\begin{aligned} X[i] &= 1, \text{ if } i = 0 & \qquad Y[i] &= 0, \text{ if } i = N \\ &= 2 + 3 * X[i-1], \text{ otherwise;} & &= X[i] + Y[i+1] \text{ otherwise.} \end{aligned}$$

Here a "good" schedule will go up X to $X[N]$ and then down Y. We should be able to write this schedule without knowing the specific value associated with N; a time formula such as $t(X[i]) = i$, $t(Y[i]) = 1 - i + 2N$ would do fine.

This solution clearly cannot be generated by the method described; the subterm $2N$ is not a constant that we can handle by integer programming as we propose to use it. Fortunately, there is an easy extension which covers this case and other common cases in which boundaries are symbolic constants: we treat these as parameters of the problem, and generalise our time formulas to

$$t(X[i]) = a + bi + cN \quad \text{and} \quad t(Y[i]) = d + ei + fN.$$

The three dependencies are now $b > 0$, $e < 0$, and

$$d + ei + fN > a + bi + cN.$$

Since N is a boundary, we have the inequality $i \leq N$ from the outset. Sample points $i = 0$, $N = 0$, $i = 1$, $N = 1$ and $i = 0$, $N = 1$ expand the third inequality into

$$d > a, \quad d + e + f > a + b + c, \quad \text{and} \quad d + f > a + c.$$

The solution is $b = 1$, $e = -1$, $d = 1$, $a = 0$, $c = 0$, $f = 2$ as desired. It is important to note that if N were not a boundary, or if we failed to include the inequality $i \leq N$, this solution would not work since we would have to include the sample point $i = 1$, $N = 0$. At that point, the third inequality becomes

$$d + e1 + f0 > a + b1 + c0$$

which for this solution is

$$1 - 1 + 0 > 0 + 1 + 0.$$

In fact, no hyperplane solution to the problem is possible without inclusion of the boundary as a parameter. The same will hold true for many problems in which information moves inwards from an arbitrary boundary.

In general, we cannot claim any completeness properties for the method, but we have found that it is easy to extend with extra information (such as symbolic values for the boundaries).

5.3. INCONSISTENCY

Unfortunately, it is possible to find cases in which the method as described so far returns an incorrect equation for a hyperplane. Consider the following "doctored up" version of the *gcd* specification.†

$$\begin{aligned} g[i,j] &= i+j, \text{ if } i=0 \vee j=0 \\ &= g[i,j-1] \text{ if } i \leq j \\ &= g[j, i+j] \text{ otherwise.} \end{aligned}$$

As a recursive function definition, this will terminate for arbitrary (i,j). Is it schedulable? Perhaps: using $t(i,j) = ai+bj$, we get

$$\begin{array}{lll} ai+bj > ai+bj-b & \text{so } 0 > -b, & \{i>0, j>0, i \leq j\} \\ ai+bj > aj+bi+bj & \text{so } a(i-j) > bi, & \{i>0, j>0, i>j\}. \end{array}$$

For the dependence inequality $0 > -b$, the specific sample points do not matter as long as there is at least one: in this case, the base point (1, 1) is enough. For the second dependence inequality, let us pick the sample points from the constraint-set as we have described. The base point for $\{i>0, j>0, i>j\}$ is (2, 1); extension points are (3, 2) and (3, 1). This yields the set of inequalities $b>0$, $a>2b$, $a>3b$, $2a>3b$. These (augmented by the other equation's requirement that $0>-b$) are satisfied by the point (4, 1), giving a time formula of $4i+j$. Now consider the point (7, 6) which should be computed at time 34. (7, 6) depends on point (6, 13) which is computed at time 37! In other words, the point (7, 6) depends on a point which has not yet been defined at the time (7, 6) is scheduled.

Clearly, such a schedule is in error. The point (7, 6) was not adequately represented by the chosen sample points; the time formula's correctness on those points did not imply that it would work at the point (7, 6). We now extend the method to detect and correct erroneous equations for hyperplanes.

Our basic approach in checking hyperplanes is to search for a counter-example (e.g. the point (7, 6) above) and add it as a sample point. A counter-example in our context is a data point which satisfies a constraint-set, but does not satisfy the associated inequality. If there is no such point, the time formula is correct; we can go on to the next problem. If there is such a point then we can add it as a new sample point (yielding an additional inequality), and re-solve the integer program. If this new integer program can be solved, we will get a new time formula which can then be tested by looking for yet another counter-example. In the worst case, we will end up enumerating each point as a new sample point, but we will never produce an incorrect equation as the final solution because we will always find the counter-example before generating a schedule.

As a first example, we check the original *gcd* hyperplane for correctness. Recall that our constrained linear inequality

$$a(2i-j) > b(i-j) \text{ if } i > 0, j \geq i$$

had a solution $a=1$ and $b=1$. We now negate the inequality while maintaining the same constraint-set $\{i>0, j\geq i\}$: the negated inequality $1(2i-j) \leq 1(i-j)$ simplifies to $i \leq 0$.

This is not compatible with the constraint-set given, so there is no (i,j) which satisfies the negated inequality, and so the time formula is correct.

† The careful reader will note that for certain values of i and j, reference to $g[j, i+j]$ will index non-existent elements of g. This might be treated as a runtime error, but we find it preferable to allow error values within an array. The desired result may still be well defined.

Next, we apply the correctness test to the modified *gcd*.

- We will start with the tentative solution of $a = 4$ and $b = 1$ as the coefficients of the time formula for a point (i, j).
- Now we attempt to find a counter-example to some dependence inequality from this arm of the conditional. Picking the last dependence inequality, we are solving for an (i, j) pair such that $4(i-j) \leq i$ and $i > j$. If a suitable point can be found, we have found an (i, j) pair whose computation precedes (or is simultaneous with) that of some pair on which it depends.
- The least solution to this pair of inequalities is (4, 3), whose time of computation is supposed to be 19. Note that (4, 3) depends on (3, 7), whose time of computation is also 19.
- Since a contradiction has been found, we add (4, 3) to the list of sample points and re-solve.

In this case, adding the $(i, j) = (4, 3)$ point will not really solve the problem. Adding this point to the sample space will give the solution $a = 5$ and $b = 1$, for which a contradiction can be found by the (i, j) pair (5, 4). We will end up enumerating the entire space of (i, j).

If, instead of finding the *least* (i, j) satisfying the reversed inequality, we look for the *greatest* values of i and j, we can easily find the correct hyperplane on the first iteration.

Let $i, j \in (1, 100)$. Then, solving $4(i-j) \leq i$, if $i > j$ for the greatest (i, j) gives us (100, 99) as a new sample point. When we add this sample point, we get the inequalities $b > 0$, $a > 2b$, $a > 3b$, $2a > 3b$, $a > 100b$. Solving this set for the least a and b yields $a = 101$ and $b = 1$, giving us the hyperplane $101i+j$ and the inequality $101(i-j) > i$. When we attempt to find a contradiction,

$$101(i-j) \leq i, \text{ if } i > j$$

there is no solution, so we know that the hyperplane is correct.

Note that this last refinement of using the greatest (i, j) pair requires that we know the ranges of i and j, which was not required for the earlier examples. The approach of treating the size as a symbolic parameter no longer works: the hyperplane schedule we would really like to see would have the form

$$t(g[i, j]) = 101i+j = (N+1)i+j,$$

in which N and i are multiplied together; we no longer have a strictly linear structure unless N is available as a fixed constant.

However, at this point we are working with pseudo-parallel expressions of sequential algorithms; the dependency graph for g really describes a nested-loop structure

```
for $i \in 0 \ldots N$ do
  for $j \in 0 \ldots N$ do
    if $i = 0 \vee j = 0 \vee i \leq j$ then $g[i, j] := i+j$
      else $g[i, j] := g[j, i+j]$.
```

Such a schedule goes beyond the scope of this paper, but examination of it does clarify what hyperplanes can and cannot do. The hyperplane construction can find schedules for a surprising variety of recursive specifications, including many that do not appear linearly structured at first glance. However, the schedule generated is "flat", not recursive, and cannot deal with a lexicographic (rather than geometric) ordering of dimensions. It remains to be seen whether there are natural extensions of the hyperplane method which can make full use of the nested-block structure of the programming language in the schedules which it generates.

6. Conclusions and Future Work

In this paper we have presented a new method for synthesizing parallel schedules for recursive equations. The method can be used in the presence of irregularly structured dependency vectors and mutually recursive equations: it is clearly more general than previous techniques, in non-trivial ways. To evaluate the method, we must consider what it can handle, how expensive it is, and how good (i.e. how efficient) are the results.

- *Generality.* We are constrained by our basic assumption, that the time-formula is to be linear. If no linear formula exists, then we cannot find one. If the recursive dependencies are described by linear inequalities on the indices, we can apply the method: it will not generate an incorrect schedule. If the conditions of branching are also described by linear inequalities on the indices, then the sample points we generate will be required by the problem: we will not fail unless success is impossible, i.e. unless there is no solution for inequalities which any correct (linear) schedule must satisfy. There are many solvable classes between the uniformly linear problems which we can handle and the completely general (trivially undecidable) case, but this is a nicely comprehensible class which includes a great many useful programs.
- *Schedule-generation efficiency.* In principle, our method is extremely expensive, because it depends on integer programming techniques. Even with linear inequalities, this can be hard to handle, and in the worst case (with multiple corrections required) we can end up with a number of inequalities greater than the number of entries in the arrays to be scheduled. In practice, however, we find that algebraic simplification can reduce the number of distinct inequalities to the extent that there is relatively little work to do.
- *Schedule efficiency.* Communications costs will depend on architecture as well as problem structure. Ignoring them, we find a processors/time tradeoff. When we find a hyperplane schedule we reduce time-complexity from the volume of an $N+1$-dimensional array to the number of N-dimensional hyperplanes which need to be computed, i.e. to the length of a line through that array. The numerical improvement thus depends on the shape of the array, which in general will not be known at the time our algorithms are used. To realise this gain, we require a number of processors proportional to the volume of the (largest) hyperplane crossing the array. Nonetheless, we can end up with a gain in space-complexity over straightforward programming, if there is an upper bound on the number of hyperplanes which must exist simultaneously.

Portions of our method have been implemented in the PS automatic program generator as reported in Gokhale (1988), Samet (1984), and Torgersen (1986).

Our primary problems now are divided into four areas.

- We hope to develop a sample-point strategy which will prevent the method from finding initial incorrect hyperplanes. Since we detect and correct these already, this is not of any immediate practical importance, but we hope that by doing so we can get some insight into the range of usable schedule generations. We conjecture that this can be achieved by considering the convex set of points represented by each constraint-set, and using an "earliest" vertex (a vertex for which the time value is minimal) as base point; the extension points should then be found along the edges going out from this point.

- We would like to generalize the scheduling process to create nested blocks of code when necessary. A matrix in which the beginning of each row depends on the end of its predecessor should turn into a nested loop structure, not a single hyperplane. We cannot deal with this at all yet.
- We would like to deal with dependencies which satisfy an ordering, but do not come from simple sums and differences; *a gcd* definition based on

$$gcd[i, j] = gcd[j \bmod i, i]$$

is beyond our present methods, as is

$$V[i] = V[L[i]] + V[R[i]].$$

- We would like to incorporate other transformations. For example, it can be quite simple to decompose a regular array structure into blocks for processor allocation; we have worked on this as a transformation on the specifications with some success.

We gratefully acknowledge the contribution of Dr J. F. Eastham, whose original insight into the problem led to this work.

References

Allen, J. (1983*a*). Dependence analysis for subscripted variables and its application to program transformations. PhD dissertation, Rice University.

Allen, J., Kennedy, K., Porterfield, C. & Warren, J. (1983*b*). Conversion of Control Dependence to Data Dependence. ACM reprint 0-89791-090-7/83/001/0177.

Backus, J. (1978). Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM* **21**, (8), 613–641.

Banerjee, U., Chen, S. C., Kuck, D. J. & Towle, R. A. (1979). Time and parallel processing bounds for Fortran-like loops. *IEEE Trans. Comp.*, Sept.

Cytron, R. (1986). Doacross: beyond vectorization for multiprocessors. *Int. Conf. Parallel Processing*, pp. 836–844.

Gokhale, M. (1987). Algorithm specification is a very high level language. *IEEE Fall Joint Computer Conference, 1987.*

Gokhale, M. (1988). PS technical reference manual. University of Delaware Technical Report.

Henzinger, T. (1986). Denotational equivalence of goal-driven and data-driven interpretation of applicative programs. MS thesis, University of Delaware, August.

Kuck, D., Kahn, R. H., Padua, D. A. (1981*a*). Dependence graphs and compiler optimization. *Proc. 8th Ann. Symp. Principles of Programming Languages.*

Kuck, D., Leasure, B. & Wolfe, N. (1980). Analysis and transformation of programs for parallel computation. *Proc. 4th Int. Computer Software and Applications Conference*, IEEE.

Lamport, L. (1974). The parallel execution of Do loops. *Comm. ACM* **17**, (2), 83–93.

McCarthy, J. (1960). Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM* **3**, (4), 184–195.

Midkiff, S. P. & Padua, D. A. (1986). Compiler generated synchronization for Do loops. *Int. Conf. Parallel Processing*, pp. 544–551.

O'Donnell, M. J. (1985). *Equational Logic as a Programming Language*. Cambridge, MA: MIT Press.

Polychronopoulos, C. D., Kuck, P. J. & Padua, D. A. (1986). Optimal processor allocation for nested parallel loops. *Int. Conf. Parallel Processing*, pp. 519–527.

Prywes, N., *et al.* (1983). Compilation of nonprocedural specifications into computer programs. *IEEE Trans. Software Engineering*, May.

Samet, S. (1984). STARS: Storage And Retrieval System for the PS language. M.S. thesis, University of Delaware Computer Science Department.

Taha, H. A. (1975). *Integer Programming: Theory, Applications, and Computations*. New York: Academic Press.

Torgersen, T. (1986). PS implementation manual. University of Delaware Technical Report.

Warren, J. (1984). A hierarchical basis for reordering transformations. ACM reprint 0-89791-125-3/84/001/0272.