MIT/LCS/TM-104

THE USE OF QUEUES IN THE PARALLEL DATA FLOW EVALUATION OF "IF-THEN-WHILE" PROGRAMS

Jeffrey Jaffe

May 1978

MIT/LCS/TM-104

THE USE OF QUEUES IN THE PARALLEL DATA FLOW EVALUATION OF "IF-THEN-WHILE" PROGRAMS

Jeffrey Jaffe

May 1978

Massachusetts Institute of Technology
Laboratory for Computer Science

Cambridge Massachusetts 02139

The Use of Queues in the Parallel Data Flow Evaluation of "If-Then-While" Programs

Jeffrey Jaffe *

Abstract

A property of a model of parallel computation is analyzed. We show that the use of queues may speed-up the execution of well formed data flow schemas by an arbitrarily large factor. A general model of data flow computation is presented to provide a framework for the comparison of data flow models. In particular a formal definition of a data flow version of the Computation Graphs of Karp and Miller and the Data Flow Schemas of Dennis are provided within the context of this model.

KEYWORDS: Computation graphs, data dependency programs, data flow computation, if-then-while programs, parallel computation, parallel schemata, queues, well formed data flow schema.

---

* This paper was prepared with the support of National Science Foundation research grant no. MCS77-19754 and a National Science Foundation graduate fellowship.

1. Introduction

This paper studies a property of the data flow models of computation [1,3,4,5,6,7,10,11,12] which are models of parallel asynchronous execution. The property that we analyze is the effect of queues or buffers on the data flow implementation of "if-then-while" programs. These programs may be easily translated into what are called well formed data flow schemas [5,6]. Two properties of well formed data flow schemas are that termination does not depend on the size of the queues used and that at termination the output of any computation is invariant under changes in the queue size. It is shown (in Section 6) that for any integer i, there is some "if-then-while" program, whose parallel evaluation using well formed data flow schemas is sped-up by a factor of i when queues are used. Thus, while queues do not change the final outcomes of computations as they do in other models [10] they can improve efficiency of computation. We will presently describe an example of a well formed data flow schema but leave precise definition for Sections 3 and 4.

The notion of a *data dependency program* is also defined. These programs are a very general model (similar to that of [1]) that are presented to give a common framework for the comparison of different data flow models. In Section 3 it is shown that some of the models that have been studied (certain versions of computation graphs, data flow schemas) are special cases of our data dependency programs.

The following is an example of these well formed data flow schemas, and a discussion of the intuitive meaning of their computation. They are proposed as a natural asynchronous implementation of "if-then-while" programs,

and we illustrate this relationship with the example. Consider the following "while" program:

i:=1

n:=100

while $i \leq n$ do i:=i*i+sqrt(n)

The corresponding well formed data flow schema is given in Figure 1.

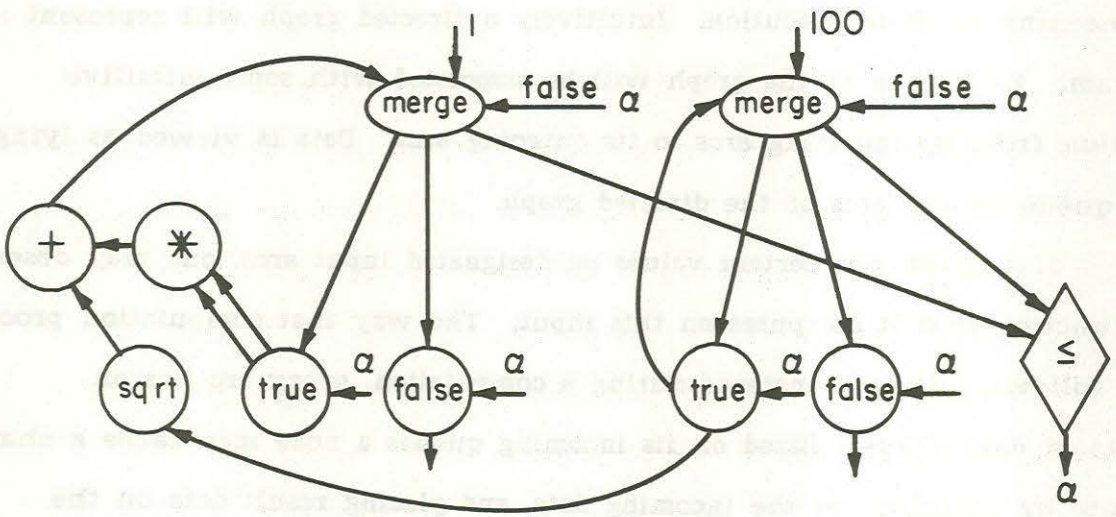

Figure 1.

This is a simple example of an "iteration construct". Intuitively, execution proceeds as follows. Since the third arc that leads to each "merge node" is initially *false* the iteration commences by obtaining the initial values from the second incoming arc to each merge (and not the first arc). If the test evaluates to *true*, the values of the variables are passed through the "true nodes" and one iteration of the while statement is executed. The values continue to circulate until the test evaluates to false. In that case the while statement terminates and outputs the results through the "false nodes".

This computation takes place on a machine with many processors for each type of node, and thus many nodes may be asynchronously executing.

2. Data dependency programs

This section presents the general data flow model of computation. The basic structure of data flow programs is presented along with a definition of the meaning of their execution. Intuitively a directed graph will represent a program. Each node in the graph will be associated with some primitive function from its incoming arcs to its outgoing arcs. Data is viewed as lying in a queue on the arcs of the directed graph.

If a graph has certain values on designated input arcs, one may observe the function that it computes on this input. The way that computation proceeds is as follows. At each instance during a computation, every arc has an associated data queue. Based on its incoming queues a node may cause a change in state by operating on the incoming data, and placing result data on the outgoing arcs. This continues until the state cannot be changed by any node. The result of the computation is then observed by looking at certain designated "output arcs".

We now proceed to the formal definitions, in which the above ideas are expressed in full generality.

Definition 1. A *data dependency program* (DDP), $\mathcal{P}$, consists of:

(1) A directed graph, G, with a node set $M=\{m_1,...,m_{|M|}\}$ and an arc set $A=\{a_1,...,a_{|A|}\}$.

(2) $V=\bigcup_{i=1}^{|A|} V_i$ the *value domain*. V_i is called the *value domain associated with the arc* a_i, and each V_i may be any set. (By abuse of notation

V_a also denotes the value domain associated with an arc a.) Let Q be the set of mappings of each arc $a \in A$ into a finite string over V_a (formally, $Q=\{q \mid q:A \rightarrow V$ where $q(a) \in (V_a)^*\}$). The set Q is called the set of *states*. If $A' \subset A$, $Q(A')$ denotes the set of restrictions of Q to A'.

Notation: If $m \in M$, B denotes the set of incoming arcs to m, and C denotes the set of outgoing arcs from m.

(3) $T=\{t_m \mid m \in M\}$ is a set of *local transition functions* of $\mathcal{P}$ indexed by nodes. The function t_m has domain and range specified by $t_m: Q(B \cup C) \rightarrow Q(B \cup C)$. If $t \in T$, then t depends only on the incoming queues and the size of the outgoing queues. Thus if $q, q' \in Q(B \cup C)$ with $q(B)=q'(B)$ and $|q(c)|=|q'(c)|$ for every $c \in C-B$ (where $|w|$ is the length of the string w) then $t(q)=t(q')$.

Arbitrary data dependency programs may not be determinate [9]. However, the specific models discussed in this paper are determinate.

If $q \in Q$ we let $t_m(q)$ be a shorthand for $t_m(q|(B \cup C))$, called *the value of the local transition function at the node m and state q* (where $q|A'$ is the restriction of q to a set $A' \subset A$).

A DDP $\mathcal{P}$, is *FIFO* if for every node m, and state q, if $t_m(q)=q'$ then for each $b \in B-C$, $q'(b)$ is a suffix of $q(b)$. Intuitively, the meaning a function $t \in T$ (for a FIFO DDP) is that it tells you what is left on incoming arcs, and what is added to arcs that are only outgoing arcs.

The *state* of a DDP, $\mathcal{P}$, is an element $q \in Q$. An *initialized* DDP is a DDP together with an element $q \in Q$ called the *initial state*.

The next state, q', after firing $M' \subset M$ in state q, (sometimes denoted $f(M',q)$), is defined as follows:

(1) If a is neither an incoming arc nor an outgoing arc for any $m \in M'$

then $q'(a)=q(a)$.

(2) If a is an incoming arc of some node $m\in M'$ and is not an outgoing arc of a different $m\in M'$ then $q'(a)=t_m(q)(a)$.

(3) If a is not an incoming arc of any node $m\in M'$, and is an outgoing arc of some $m\in M'$, then $q'(a)=q(a)\cdot t_m(q)(a)$.

(4) If a is an incoming arc of a node $m\in M'$ and an outgoing arc of a different node $n\in M'$, then $q'(a)=t_m(q)(a)\cdot t_n(q)(a)$.

An *execution sequence, e,* for a DDP is a sequence of nodes $M_1,\ldots,M_m$ where $M_i\subset M$ for $i=1,\ldots,m$. The *state of an initialized DDP* (with initial state q) *after an execution sequence* $e=M_1,\ldots,M_m$ is $f(M_m,(\cdots(f(M_2,(f(M_1,q))))\cdots))$. A state $q\in Q$ is said to be a *final state* if $f(\{m\},q)=q$ (for every $m\in M$).

An *I/O DDP* is a DDP with two sets $I,O\subset A$, I and O disjoint, where I is called the *set of input arcs*, and O is called the *set of output arcs*. Each output arc must be an arc that is incident on a node with no outgoing arcs (called an output node). These nodes have the identity function as their local transition function.

Let $\mathcal{P}$ be an initialized I/O FIFO DDP with initial state $q\in Q$. We will describe the meaning of the relation $R(\mathcal{P})$ computed by $\mathcal{P}$ on an input q'. Examine the program $\mathcal{P}$ in its initial state and then replace the initial data words of all input arcs as specified by q'. Execute $\mathcal{P}$ until a final state is reached if one exists. If the value of the output arcs in this final state is q'' then $(q',q'')\in R(\mathcal{P})$. In the interesting case that $\mathcal{P}$ is determinate, the value, q'', is unique if it exists and $\mathcal{P}$ specifies a partial function.

3. Description of Other Models

This section indicates how to describe two data flow models of computation in terms of our data dependency programs. They are a data flow version of the Computation Graphs of Karp and Miller [10], and the Data Flow Schemas of Dennis [5].

If z is a string we denote the prefix of z of length i by z_i and the suffix of z of length $|z|-i$ by z^i.

A node m is said to be *enabled* in a state q if $t_m(q)\neq q|(B\cup C)$.

A node m is said to be an *ordinary computation node* if for each arc $b\in B$ there are integers T_b, W_b (with $T_b\geq W_b$), and for each arc $c\in C$ there is an integer U_c such that:

(1) m is enabled iff $|q(b)|\geq T_b$ for every $b\in B$.

(2) Whenever m is enabled

(i) $t_m(q)(b)=q(b)^{W_b}$ for $b\in B-C$.

(ii) $t_m(q)(a)=q(a)^{W_a}\cdot d(q,a)$ where $d(q,a)$ is a string dependent only on q and a, and $|d(q,a)|=U_a$ for $a\in B\cap C$.

(iii) $|t_m(q)(c)|=U_c$ for every $c\in C-B$.

(3) Let q and q' be two states in which m is enabled such that for each $b\in B$, $q(b)_{T_b}=q'(b)_{T_b}$. Then $t_m(q)(c)=t_m(q')(c)$ for $c\in C-B$ and $d(q,a)=d(q',a)$ for $a\in C\cap B$.

Intuitively what an ordinary computation node does whenever the size of the data queue on each incoming arc b at least equals T_b, is operate on the first T_b elements of arc b, remove W_b of them, and place U_c results on arc c.

A DDP is said to be a *computation graph* if all nodes are ordinary computation nodes. This definition of computation graphs differs from that of Karp and Miller only in the fact that they allow their nodes to depend on some

auxiliary memory. Our computation graphs are thus a pure data flow model of theirs.

A *data flow schema* is an initialized FIFO, I/O, DDP where.

(1) For each arc i in the input set, there is a node m (called an input node) such that

(a) The only outgoing arc from m is i.

(b) There are no incoming arcs to m.

(c) The local transition function on m is $t_m(q)(i)=\epsilon$ for every $q\in Q(\{i\})$.

(2) The nodes of the graph are either input nodes, output nodes, ordinary computation nodes, True gates, False gates, or Merge gates, where the latter three are defined below. Also, for each incoming arc to an ordinary computation node $T=W=1$, and for each outgoing arc from an ordinary computation node $U=1$.

(3) A *True gate* is a node with two incoming arcs a,b and an arbitrary number of outgoing arcs. The value domain associated with b, the second incoming arc is $\{true,false\}$. The value domain associated with all other incoming and outgoing arcs must match each other. The local transition function is given by $t(q(a))=q(a)^1$; $t(q(b))=q(b)^1$; $t(q(c))=q(a)_1$ if $q(b)_1=true$ and $t(q(c))=\epsilon$ if $q(b)_1=false$ where c is any outgoing arc. A *False gate* acts as a True gate with the role of *false* and *true* reversed. A *Merge gate* is a node with three incoming arcs a,b,d and an arbitrary number of outgoing arcs. The value domain associated with d, the third incoming arc is $\{true,false\}$. The value domain associated with other incoming and outgoing arcs must match each other. The local transition function is given by: if $t(q(d))_1=true$ then $t(q(a))=q(a)^1$, $t(q(b))=q(b)$, $t(q(d))=q(d)^1$,

and $q(c)=q(a)_1$ where c is any outgoing arc, and if $t(q(d))_1=false$ then $t(q(a))=q(a)$, $t(q(b))=q(b)^1$, $t(q(d))=q(d)^1$, and $q(c)=q(b)_1$ where c is any outgoing arc.

4. "If-then-while" programs and well formed data flow schemas

This section fills in the details of the formal definition of the well formed data flow schemas. An *I/O well formed data flow schema* is a data flow schema whose graph fits one of the following inductive structural definitions.

(1) any ordinary computation node (which requires k incoming arcs and m outgoing arcs) with k input nodes, m output nodes and arcs from each input node to the ordinary computation node and from the ordinary computation node to each output node.

(2) any conditional schema (as defined below)

(3) any iteration schema (as defined below)

(4) any acyclic composition of non-I/O well formed data flow schemas with the proper number of input nodes and output nodes as in (1).

A *non-I/O well formed data flow schema* is a well formed data flow schema with input nodes and arcs and output nodes and arcs removed.

A sample conditional schema is illustrated in Figure 2. The general conditional schema is composed as follows (refer to figure). P and Q are non-I/O well formed data flow schemas requiring a compatible number of inputs and outputs, and C is a decision structure (to be defined below) with the same number of inputs as P and Q. There are nodes labelled with True, False, and Merge with the quantity of such nodes based on the number of inputs and outputs of P and Q. The data inputs for each merge come one each from P and Q.

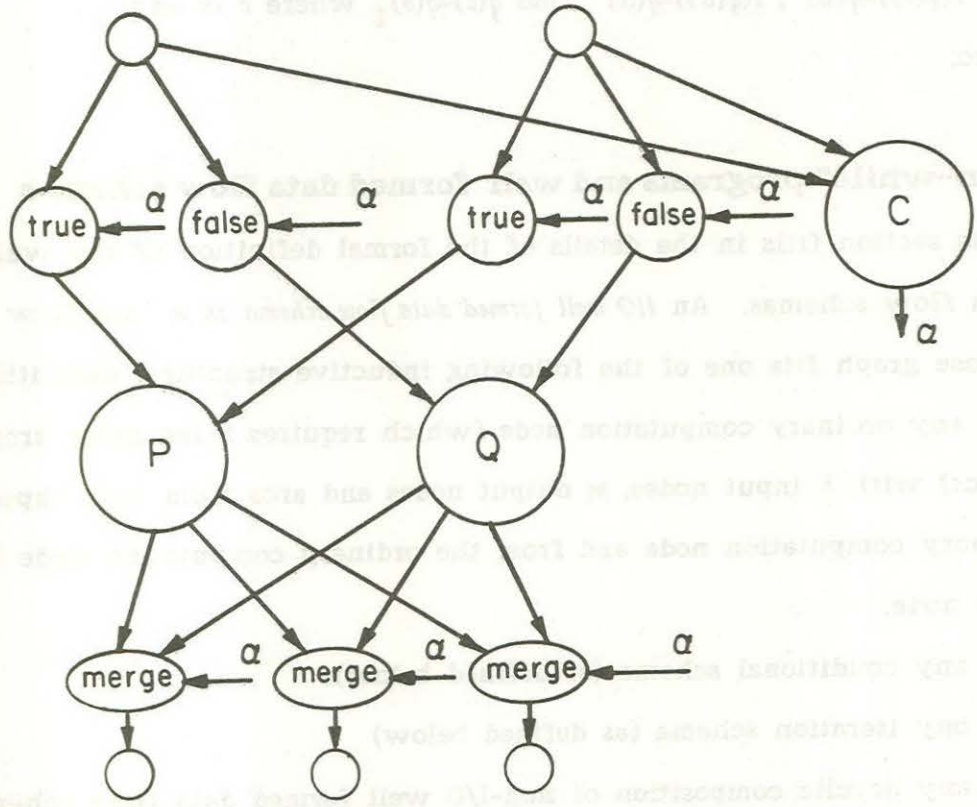

Figure 2.

A sample iteration schemas is illustrated in Figure 3. The general iteration schema is composed as follows (refer to figure). *P* is a non-I/O well formed data flow schema requiring the same number of inputs as it does outputs, and *C* is a decision structure with the same number of inputs as *P*. There are nodes labelled with True, False, and Merge equal to the number of inputs of *P*. The "third" incoming arc to each merge is labelled with *false*. These labellings are the only initializations of arcs permitted in well formed data flow schemas (aside from those of input arcs).

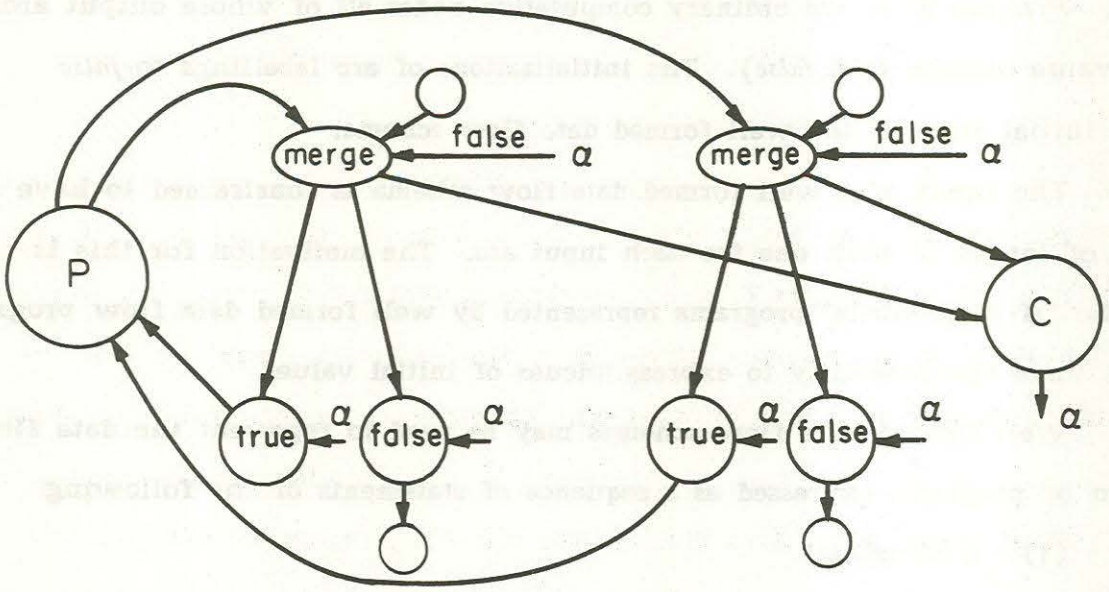

Figure 3.

A decision structure consists of a set of copy, predicate, and boolean nodes. The copy nodes accept inputs and send copies of them to predicate nodes which in turn produce boolean outputs to an acyclic composition of Boolean function nodes. For any set of incoming data there is a unique boolean result from this acyclic composition.

In the above constructs we also require that each node (other than input nodes) have at least one incoming arc.

In the sequel a well formed data flow schema will always refer to an I/O well formed data flow schema.

True, False, and Merge nodes were defined in Section 3. *Copy nodes* are single input ordinary computation nodes that when enabled remove the first element of their incoming queue, and place it at the end of each outgoing

queue. *Predicate nodes* are ordinary computation nodes all of whose output arcs have value domain {*true,false*}. The initializations of arc labellings to *false* is the initial state of the well formed data flow schema.

The input of a well formed data flow schema is constrained to have a string of length at most one for each input arc. The motivation for this is that the "if-then-while" programs represented by well formed data flow programs do not have the capability to express queues of initial values.

Well formed data flow schemas may be used to represent the data flow version of programs expressed as a sequence of statements of the following form: (1) x_j:=<constant>

(2) $x_j := x_k$

(3) $x_j := f_k(x_{i_1}, \ldots, x_{i_n})$

(4) if <Boolean exp> then <program> else <program>

(5) while <boolean exp> do <program>

This is the class of programs refered to as the "if-then-while" programs. Well formed data flow programs represent the parallel evaluation of "if-then-while" programs in a straightforward manner [6]. Arcs between ordinary computation nodes are viewed as representing the variables on the left hand side of definitions of the form (1), (2), and (3); conditional schemas representing statements of the form (4), and iteration schemas representing statements of the form (5).

5. Complexity of data dependency programs

We specify the notion of the number of steps required by a data dependency program. Implicit in this definition is a consideration of how many processors of various types are available, and the size of the queues that are

being used.

Let C be a machine with a fixed (perhaps infinite) number of processors for each type of node. The *parallel complexity* of an I/O, initialized, FIFO, DDP with input q' on the machine C is the length of the shortest execution sequence leading to a final state that does not exceed the processor limitations. Specifically, for each M_i in the execution sequence, the number of times that nodes with the same local transition function appear must not exceed the number of processors for such nodes.

The role of queues in our model may be stated formally as follows. For simplicity we restrict the formal definition to ordinary computation nodes. An ordinary computation node on a machine with queues of size k, is *enabled* iff $|q(b)| \geq T_b$ for every $b \in B$ (as before), and $|q(c)| \leq k - U_c$ for every $c \in C$. The rest of the definition of ordinary computation nodes remains the same. In general, no transition which could increase the size of a data queue past k is allowed. Thus the parallel complexity of a fixed program depends both on the number of processors, and the size of the queues of the machine. Usually the number of processors will be fixed in which case the parallel complexity of a program $\mathcal{P}$ with input q, on a machine with queues of size k will be denoted by $PC^k_{\mathcal{P}}(q)$. The symbol $PC^u_{\mathcal{P}}(q)$ is used when the machine has unbounded queues.

6. Improved efficiency with queues

It is evident that for certain data dependency programs, the size of the queue available causes a change in the partial function computed by a program [9,10]. By induction on the structure of well formed data flow schemas, it may be shown that the partial function computed on any input q',

with $|q'(i)|=1$ for every $i \in I$ is invariant under queue size [9]. This fact only serves to highlight the question of whether queues are useful for speeding up computations. Three results presented herein give an affirmative answer to this question. The later results are stronger than the earlier ones, but require a slightly more complicated variation of our construction. The programs considered will have a single input variable, $x \in \mathbb{N}$.

Theorem 1. For all $i \in \mathbb{N}$ there is a "if-then-while" program, whose translation into a well formed data flow schema $\mathcal{P}$, is at least i times faster if unbounded queues are used. Specifically,
$\lim_{x \to \infty} \inf (PC^{1}_{\mathcal{P}}(x))/(PC^{u}_{\mathcal{P}}(x)) > i$.

Theorem 2. For all $i,k \in \mathbb{N}$ there is a well formed data flow schema $\mathcal{P}$, such that $\lim_{x \to \infty} \inf (PC^{k}_{\mathcal{P}}(x))/(PC^{k+1}_{\mathcal{P}}(x)) > i$.

Finally, in the realistic case when we have exactly r function processors the speed-up is at most r (actually, it may be slightly larger if one counts processors for the gates). In that case:

Theorem 3. For all $k \in \mathbb{N}$ and for all $i < r$ there is a well formed data flow schema, $\mathcal{P}$ such that $\lim_{x \to \infty} \inf (PC^{k}_{\mathcal{P}}(x))/(PC^{k+1}_{\mathcal{P}}(x)) > i$.

Proof of theorems

We now present the proof of the three part theorem. The programs for the three parts are quite similar. Figure 4 is an outline of the well formed data flow schema used in the proofs. We omit the "if-then-while" program for brevity. There are i levels in the general program. The first i-1 are

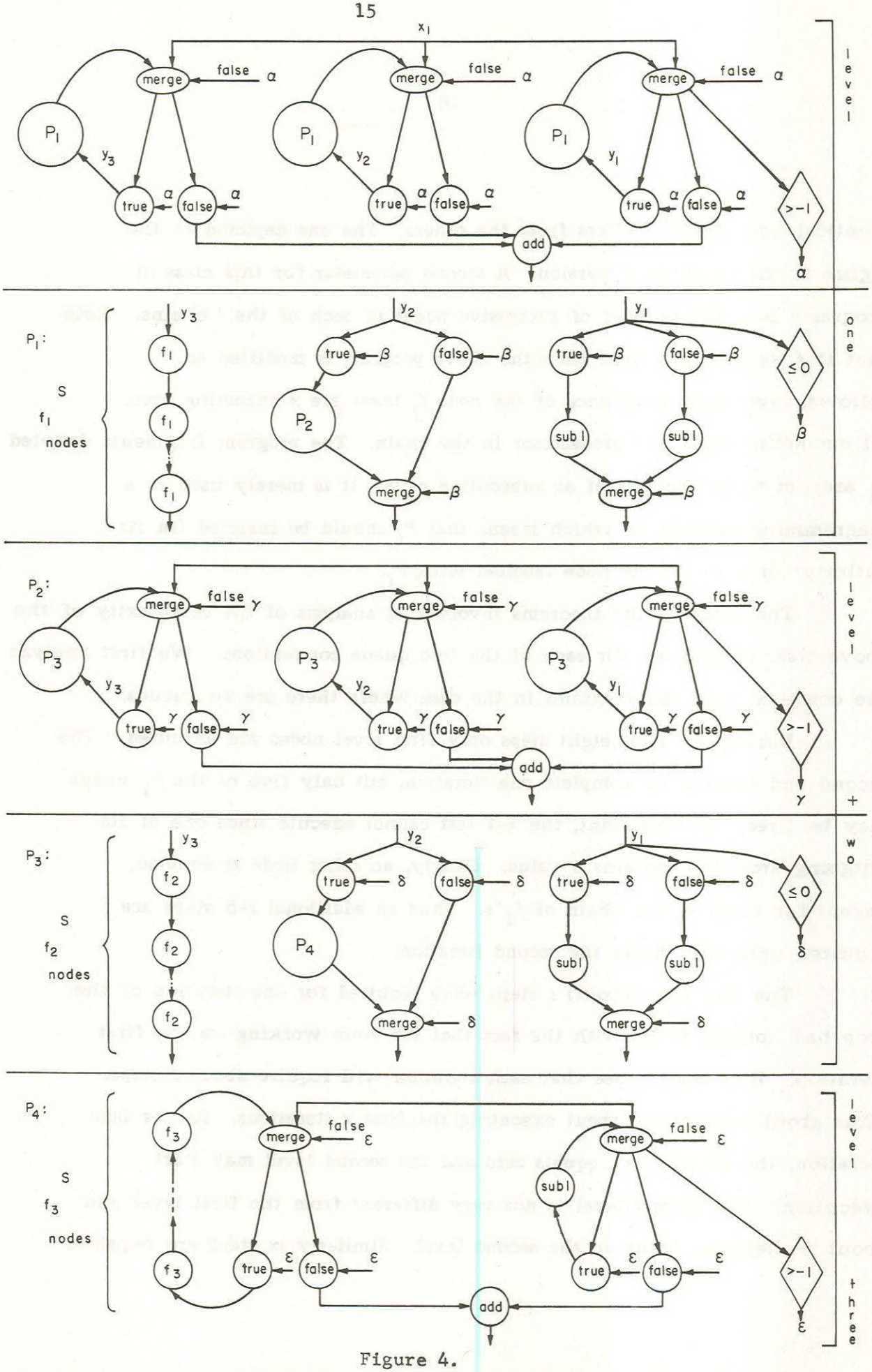
level three
level two
level one
level zero
merge
false
true
add
sub1
>-1
≤0
P1
P2
P3
P4
P1:
P2:
P3:
P4:
S
f1 nodes
f2 nodes
f3 nodes

Figure 4.

identical, and the i^{th} differs from the others. The one depicted in the figure is the three level version. A second parameter for this class of programs is s, the number of successive nodes in each of the i chains. Note that if f_j is an n-ary node then the above program is modified as follows. For each occurrence of the node f_j there are n incoming arcs, all emanating from the predecessor in the chain. The program fragments denoted P_i are not to be thought of as subroutine calls. It is merely used as a diagramming convention which means that P_i should be inserted (in its entirety) in place of the node labelled with P_i.

The proof of the theorems involves an analysis of the complexity of the above class of program for each of the two queue conventions. We first analyze the complexity of the programs in the case where there are no queues.

During the first eight steps only first level nodes are executed. The second and third loops complete one iteration, but only five of the f_1 nodes may be fired. At this point, the >-1 test cannot execute since one of its outgoing arcs still contains a value. Clearly, no other node is enabled, except for those in the chain of f_1's. Thus an additional s-5 steps are required until may begin the second iteration.

The fact that (about) s steps were required for one iteration of the loop had nothing to do with the fact that we were working on the first iteration. It is easy to see that each iteration will require about s steps. Thus about sx steps are spent executing the first x iterations. At the last iteration, the value of y_1 equals zero and the second level may start execution. The second level is not very different from the first level and about sx steps are spent on the second level. Similarly sx steps are required

for the third, . . ., i-1st levels. Finally, it is easy to see that sx steps are required for the i^{th} level. The total execution time for the program is six steps plus lower order terms.

The performance of this program on a machine with unbounded queues follows. The first eight steps are identical to the no queue case. However, the ninth step differs. While still executing the chain of f_1 nodes as part of the first iteration, one may begin working on the second iteration. This is because the test of the iteration is enabled even though not all of its outgoing arcs are void. Continuing the third, . . ., x^{th} iteration in the same manner, it will only take $7x+1$ steps until the value $y_1=0$ is passed through the merge of the iteration. After $7x+5$ steps the second level commences, even as we continue processing the first level. When the subsequent levels of the program are executed, $7x+5$ steps are again required to get from level to level. Thus $(7x+5)(i-1)$ steps are spent until the i^{th} level is reached. At the i^{th} level $(s+3)(x+1)$ steps are spent finishing the iteration. In parallel, this time is also used to complete the execution of the other i-1 levels. Finally, very few steps are spent executing the "add" nodes to complete execution of the program. Thus the total complexity of the program is $7ix+sx$ plus lower order terms.

The ratio is:

$PC^1_{\rho}(x)/PC^u_{\rho}(x)=six/(sx+7ix)=si/(s+7i)=i/(1+(7i/s))$.

Since s was an arbitrary parameter, s may be chosen to be as large as desired, causing the ratio to be arbitrarily close to i. (Actually, to remove low order terms in the denominator x must be chosen to be large.) □

We remark that with a program of i levels the speed-up obtained was about $i/(1+(7i/s))$. A program of i levels and s nodes in the chain of each

level has a total of about is nodes. Thus with a program of size is, we obtained a speed-up of $i/(1+(7i/s))$. Using this, it may be determined how large a program is required in order to get a speed-up of i. For this particular proof a program of size $28i^2$ is needed to get a speed-up of i. This can be seen by looking at a program with $2i$ levels and $s=14i$. Then using the above formulas the speed-up is equal to i and there are $28i^2$ nodes.

Proof of Theorem 2. To compare queues of size k to queues of size $k+1$, first analyze the performance of the program of Figure 4 with input $x=k$. Using a queue of size k, about $7k$ steps are spent running x and y_1 through the iteration k times. If $s>7k$, then at least s steps must be executed before the final iteration of the loop (and thus the beginning of the second level) is started. Continuing in this manner, $s(i-1)$ steps are required until the last level, is started and then sk steps are required just on the execution of the last level. Thus the total complexity with a queue of size k is about $s(i+k)$.

With a size $k+1$ queue the program's behavior is quite different. After $7k$ steps the second level may start since the queues are large enough to accommodate all $k+1$ firings of the test of the iteration. Similarly after $7k(i-1)$ steps the last level begins. Naturally, sk steps are again spent on the last level. Thus the ratio is about:

$$(si+sk)/(7ki+ks)=(i+k)/((7ki/s)+k)$$

This ratio is at least as large as $i/(k+(7ki/s))$. Thus to get a speed-up ratio of i, use a program of $2ki$ levels and $s=14ki$. This uses about $28k^2i^2$ nodes.

The connection between the behavior of the program of Figure 4 on the particular input $x=k$ and a general program that exhibits the above behavior is

made as follows. If a program uses the constant k as the loop counter for any input x, then the behavior of that program for any input is described by the above discussion! □

We may modify the construction to get a slightly better result in program size. The idea is to execute the chain of s actors only on the first time around each iteration. With queues of size k, only about si steps are required since the additional sk steps on the last level are not needed. With queues of size $k+1$ only $10ki+s$ steps are required for the execution of the program (the factor of 10 replaces the 7 due to extra nodes in the new program). The ratio is thus about $si/(10ki+s)=i/(1+(10ki/s))$. To get a ratio of i it now suffices to use a program with $2i$ levels and $s=20ki$ for a total program size of $40ki^2$, i.e. only linear in k.

Proof of Theorem 3. In the case where there are only r processors the maximum speed-up factor is bounded by r (or actually slightly larger if one counts processors for gates) since even with a queue of size one, at least one node is executed at each step. We restrict attention to r level programs.

Using the second program described in the proof of Theorem 2 the ratio for an i level program with s nodes in a chain is $i/(1+(10ki/s))$ even with only r processors (as long as $i \leq r$). To get a speed-up of i ($2i \leq r$) we use the exact same program as above using $2i$ levels. However if $2i > r$ then we must use a slightly different construction. Specifically, we may take advantage of at most r levels. To get a speed-up of i in this case, we use r levels and a value of $s=10kri/(r-i)$ for a total program size of $10kr^2i/(r-i)$. Here we cannot readily take advantage of using many levels since at most r of them may execute at once.

Finally, in our analysis of the case where there are limited number of processors, it is unnecessary to worry about processors for the parallel execution of the various nodes that are not located in the chains. They are few in number (relative to the total number of nodes that are in the chain), and thus even if they are not executed in parallel, the time spent on their execution has little effect on the total complexity. □

7. Practical usage of queue results

The argument may be raised that queues are useful for only a small class of programs. While we cannot provide a complete answer to this objection, a few remarks are relevant. Any program that could be sped-up due to the capability of simultaneously executing three different executions of the body of a while statement benefits from the use of queues. This follows from the fact that three simultaneous executions of the body may not take place if queues are not used.

Of course one might also argue that three simultaneous executions of the body of a "while" is rarely possible in real situations. In that case, there is a different possible useful modification in machine data flow implementation.

There is a mechanism introduced in machine implementations to guarantee safety of programs [8]. Basically, a program is unsafe if a node may obtain all the incoming data needed to execute, but is disabled due to data that remains on outgoing arcs.

Many of the programs for which queues do not help are inherently safe. Specifically, if a constraint is imposed that two executions of the body of a while statement never occur simultaneously, then the safety problem is

eliminated for well formed data flow schemas. This constraint does not hinder parallelism if the simultaneous execution of the body did not help parallelism.

We summarize the above discussion. If a program benefits from three of more simultaneous executions of the body of a "while" - it benefits from queues. If it does not even benefit from two simultaneous executions - we have a potentially simple solution to the safety problem for well formed data flow schemas. Only if a program benefits from exactly two simultaneous executions of the body do we lack a potential suggestion!

Acknowledgements

We would like to thank Albert Meyer for his help in formulating the data dependency model and his many suggestions on the presentation of the results of this paper.

References.

1. Adams, D. A., A Computation Model With Data Flow Sequencing, TR CS 117, Computer Science Dept., Stanford Univ., Dec. 1968.

2. Ashcroft, E., and Manna, Z. "The translation of 'goto' programs to 'while' programs, *Information Processing 71*, North Holland Pub. Co. 1972 pp 250-255.

3. Bahrs, A. "Operation patterns," *Symposium on Theoretical Programming*, Novosibirsk, USSR, August 1972.

4. Dennis, J. B., "Programming generality, parallelism, and computer architecture," *Information Processing 68*, N. Holland Pub. Co., 1969 pp 484-492.

5. Dennis, J. B., "First Version of a Data Flow Procedure Language", MIT LCS

TM61, May 1975.

6. Dennis, J. B., and Fosseen, J. B. "Introduction to Data Flow Schemas" (to appear).

7. Estrin, G., and Turn, R. "Automatic Assignment of Computations in a Variable Structure Computer System", *IEEE Trans. Elect. Comp. EC-12* (1963) pp 755-773.

8. Hack, M., "Analysis of Production Schemata by Petri Nets," MIT MAC TR94, Feb. 1972.

9. Jaffe, J. M. Parallel Computation on Data Flow Machines, PhD Thesis in preparation, Dept of EECS, MIT.

10. Karp, R. M. and Miller, R. E. "Properties of a model for parallel computations: determinacy, termination, queueing," *SIAM J. of Applied Math., 14*, 6 (Nov. 1966) 1390-1411.

11. Kosinski, P. R., "Mathematical Semantics and Data Flow Programming, *3rd POPL Symposium*, pp 175-184.

12. Rodriguez, J. E., A graph model for Parallel Computation, MIT MAC TR64, Cambridge, MA Sept. 1969.