Response Time of Parallel Programs

Richard J. Lipton
Frederick G. Sayward

Research Report #108

June 1977

Response Time of Parallel Programs

Richard J. Lipton

Frederick G. Sayward

Computer Science Department

Yale University

520 Dunham Laboratory

10 Hillhouse Avenue

New Haven, Connecticut 06520

ABSTRACT

The response time of a parallel program is defined to be the maximum delay between successive activities of an event. Response times are dependent on two factors: the parallel program's structure and the program's scheduler policies. It is shown that under weak assumptions about the scheduler policy, the imposition of an N-fair policy in which each event gets a chance to execute at least every N scheduler steps, the response time becomes dependent only on program structure: either the response time is infinite or it is linear in N (i.e., $\leq cN$ for some $c>0$). Also presented are decision procedures for determining whether or not the response time is infinite and for determining the exact linear relationship in N (i.e., the minimum c).

This work is sponsored in part by ONR Grant N00014-75-C-0752.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

Field	Entry
1. REPORT NUMBER	108
2. GOVT ACCESSION NO.	
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle)	Response Time of Parallel Programs
5. TYPE OF REPORT & PERIOD COVERED	Technical
6. PERFORMING ORG. REPORT NUMBER	
7. AUTHOR(s)	Richard J. Lipton Frederick G. Sayward
8. CONTRACT OR GRANT NUMBER(s)	N00014-75-C-0752
9. PERFORMING ORGANIZATION NAME AND ADDRESS	Yale University Department of Computer Science 10 Hillhouse Ave., New Haven, CT 06520
10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS	
11. CONTROLLING OFFICE NAME AND ADDRESS	Office of Naval Research Information Systems Program Arlington, VA 22217
12. REPORT DATE	June 1977
13. NUMBER OF PAGES	
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	
15. SECURITY CLASS. (of this report)	Unclassified
15a. DECLASSIFICATION/DOWNGRADING SCHEDULE	
16. DISTRIBUTION STATEMENT (of this Report)	Distribution of this report is unlimited.
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)	
18. SUPPLEMENTARY NOTES	
19. KEY WORDS (Continue on reverse side if necessary and identify by block number)	N-fair; parallel programs; response time; schedulers; scheduling policies; vector addition systems

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The response time of a parallel program is defined to be the maximum delay between successive activities of an event. Response times are dependent on two factors: the parallel program's structure and the program's scheduler policies. It is shown that under weak assumptions about the scheduler policy, the imposition of an N-fair policy in which each event gets a chance to execute at least every N sheduler steps, the response time becomes dependent only on program structure: either the response time is infinite or it is linear in N (i.e., $\leq cN$ for some $c>0$). Also presented are decision procedures for determining whether or not the response time

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

1.0 *Introduction*

The response time of a parallel program is the maximum time that an event in the program may ever wait for a chance to execute. Response time is clearly important in realtime programs: large or unbounded reponse time may cause the program to fail. Even nonrealtime programs may be seriously degraded if the response time is too large.

Response time of a parallel program is not easily computed. Often it is only determined by empirical observation. The fundamental question addressed in this paper is:

How can one compute the response time of a parallel program?

Previous studies of this question [1,4,8] have shown that, under certain assumptions about how programs are scheduled, one can show that particular events execute infinitely often. While this type of information is useful, there are situations where it ia inadequate: e.g., in a realtime program for data aquistion, knowing that an event will eventually execute does not guarantee that data (for example) will not be lost.

In order to get a more useful analysis of the response time of a parallel program, i.e., to avoid answers of the form

"...executes infinitely often."

we will make stronger assumptions about how our parallel programs get scheduled. In all of the previous work very weak scheduling assumptions have been made. Here we will assume instead that we have scheduling where each event of the parallel program gets a chance to try to execute at least every N scheduling steps (N>0), in the worst case. To avoid scheduling

anomalies, it is necessary that N be at least as large as the number of events in the program. Even in this case, of course, some events may get their chances faster than others; however, no event ever waits longer than N steps to be "looked" at by the scheduler.

Clearly, an event may *not* be able to execute every N steps; it may have to wait for some other event to occur. In particular, if r is the response time of some event, then $r > N$ is possible. A basic question is then:

As a function of N, what values can r take?

For example, can r be N square, i.e., can r grow non-linearly in N? The answers to these questions are contained in the *response time theorem*: as N grows, either

(1) the response time of an event becomes infinite (i.e., in the worst case it can wait forever), or

(2) the response time is linear in N (i.e., it is bounded by cN, for some constant c).

To fully describe the response time behavior of a parallel program we must consider the question of how one can compute the smallest such c (our main theorem gives an upper bound) after having determined that the response time is finite? The answers to these questions are given by providing decision procedures for the following questions:

(1) given an event e of a parallel program, can e ever have infinite response time?

(2) given a constant $c>0$, is the response time of $e \leq cN$ for all N?

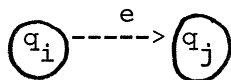
These questions are reduced to questions about suitably encoded vector addition systems [10].

The remainder of this paper has the following organization. In section 2 we give a formal model of parallel programs and their computations and show how this model relates to the parallel programming notations found in the literature. The scheduler of a parallel program is presented in section 3. A scheduler is shown to be just an alternative characterization of a program's computations. In section 4 we introduce the scheduler restrictions necessary for our main result. The response time definition and the response time theorem are given in section 5. In section 6 this result is shown to include as special cases several schedulers used in actual systems. In section 7 the afore-mentioned decision procedures are presented.

2.0 *Parallel Programs*

One of the problems in the area of parallel programming is that the literature in this area is filled with definitions of the form: "a parallel program is...". Of course, we must also supply such a definition: however, we will try to make our definitions simple enough so that they can model a wide variety of situations.

A *parallel program* P, as used here, is a finite directed graph G, a distinguished node q_1 of G, and edges which are labelled with elements from a finite set E. Intuitively, the nodes of G are the *states* of P, while the elements of E are the *events* of P. If



is an edge, then we have the following semantics:

"If P is in state q_i, then event e can occur resulting in P going into state q_1."

Clearly, so far, P is nothing more than a finite state diagram. As an aside, while our main theorem depends on the assumption of finite state, our basic definitions and indeed several of our results can be generalized to allow infinite state parallel programs.

Formally, a *parallel program* P is a 4-tuple $P = (Q,E,q_1,\tau)$ where:

(1) Q is a finite set of *states*, denote $Q = \{q_1,q_2,\ldots,q_n\}$.

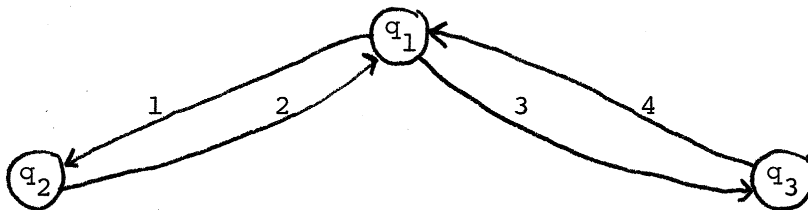
(2) E is a finite set of *events*, denote $E = \{1,2,\ldots,m\}$.

(3) q_1 is a distinguished *start state*.

(4) τ is the *state transition function*: $\tau:Q \times E \dashrightarrow Q$.

It should be noted that this definition is by no means novel. It links well with path expressions [1] and many other such definitions. Also, note that we have deliberately defined a parallel program to be a rather unstructured object. The usual notions of process, semaphores, instruction counters and so forth, are implicit rather than explicit.

As an example of a parallel program, consider the following directed graph, which we will call example 1:



This corresponds to a parallel program represented in the semaphore notation

of [5] as follows:

```
semaphore s (initially 1);
    parbegin
        repeat 1:  P(s);  2:V(s) forever;
        repeat 3:  P(s);  4:V(s) forever;
    parend;
```

Indeed, our model of parallel programs is capable of representing the control aspects of any parallel program which uses bounded value semaphores.

2.1 *Parallel Program Computations*

In order to study the response time of parallel programs, it is necessary to introduce the notion of an event blocking. Thus our definition of a parallel program's computations must include both event execution and event blocking. To this end, let the elements of E be called *event executions*. Then the elements of the following set are called *event blockings*:

$$E' = \{e' \mid e \text{ is in } E\}$$

The elements of the following set are called *event activities*:

$$EA = E \cup E'$$

We will define a parallel program's computations to be certain finite and infinite strings over EA. Intuitively, an event e may execute whenever the program's control is in a state q where e is eligible to execute (i.e.,

τ(q,e) is defined). An event e may block whenever the program is in a state where e cannot execute, the program has passed through a state where e could have executed but didn't, and e hasn't executed in the meantime. To formally define those strings over EA which satisfy this intuitive notion, we introduce the following function on EA*:

Definition: The function *state*:EA* ---> Q is defined as follows:

(1) *state*($\wedge$) = q_1

(2) For e in E and x in EA*, *state*(xe) = τ(*state*(x),e)

(3) For e' in E' and x in EA*, *state*(xe') = *state*(x) only if

(a) τ(*state*(x),e) is undefined, and

(b) for some event f, x=yfz such that τ(*state*(y),e) is defined and *not substr*(fz,e).

where *substr* is the usual substring predicate.

Note that the ways in which *state* can be undefined correspond to illegal event executions and blockings. For example, if P is in state q and τ(q,e) is undefined, then e is not eligible to execute. Likewise, if τ(q,e) is defined, then e can't block. We are now in ready to define the computations of a parallel program.

Definition: The *computations* of a parallel program P are members of the set C, the union of the following two sets:

(1) CF = {x in EA* | *state*(x) is defined}.

(2) CI = {x an infinite string over EA | *state*(y) is defined for all finite prefixes y of x}.

The set CF is called the set of *finite computations* of P and CI the *infinite*

computations. Note that CI may be empty and that C is closed under finite prefix.

For later use, we distinguish a (possibly empty) subset of the finite computations:

Definition: A finite computation x in CF is called *terminating* if for all events e of P both *state*(xe) and *state*(xe') are undefined. A computation terminates when no further event activity is possible.

The following are examples of legal and illegal computations, in terms of regular expressions, for the parallel program presented above.

Legal

(1) (12 + 34)* - no event ever blocks

(2) 13'2(12)* - event 3 remains blocked forever

(3) 123(1')* - event 1 is forever blocking

Illegal

(1) (2' + 4')+ - events 2 and 4 may never block

(2) 12341' - event 1 is ineligible to block since it can execute.

Note that example 1 has no terminating computations.

2.2 *Parallel Program Total State*

At any point during the execution of a parallel program P a (possibly empty) subset of the events will be blocked. We define the total state of the program to be the state of P's control coupled with the subset of currently blocked events.

Definition: A *total state* of a parallel program P is a member of the set $T = \{(q,B) \mid q \text{ is in } Q \text{ and } B \text{ is a subset of } E\}$.

Note that T is finite for finite state parallel programs. Given any computation we can compute the total state via the following function:

Definition: The *total state function* *tstate*:C ---> T is defined as:

(1) $tstate(\Lambda) = (q_1,\Phi)$

(2) Let xf be in C, B a subset of E, q in Q and *tstate*(x) = (q,B).

(a) If f = e then $tstate(xf) = (\tau(q,e),B-\{e\})$.

(b) If f = e' then $tstate(xf) = (q,B \cup \{e\})$.

3.0 *Parallel Program Schedulers*

We have defined the computations of a parallel program P to be sequences of event executions and blockings. Which particular computation is produced by the execution of P is determined by the decisions made in an agent entirely external to P; namely, by the *scheduler*. The scheduler maintains a data structure that contains information such as the state in which P's control lies and the blocking status of P's events. We will call this data structure the *scheduler state*. A *scheduler step* consists of the scheduler determining which events are eligible for event activity, using a *scheduling policy* to determine which one of those events will execute or block, and then reflecting this decision by appropriate changes to the data structure (i.e., making a scheduler state transition). The scheduler repeats this cycle as long as there are events eligible for event activity.

In this section we will formally define the scheduler of a parallel program independently of any scheduling policies. We show that this is

just an equivalent characterization of a parallel program's computations. Thus, in subsequent sections when scheduling policies are introduced, we will be effectively restricting the computations that parallel programs produce.

3.1 *Scheduler State*

Let P be a parallel program having n states and m events. The scheduler state will consist of three types of information:

(1) The *program state*.

(2) For each event, a *delay* which indicates the number of scheduler steps which have passed since the event's last activity.

(3) An *event status set* which indicates whether or not an event is eligible to *block*.

Accordingly, we have the following formal definition:

Definition: Let P be a parallel program having m events. A *scheduler state* S is an element of the set $SS = Q \times D \times B$ where:

(1) Q is the state set of P.

(2) $D = NN \times NN \times \ldots \times NN$ (m times) where $NN = \{0,1,2,\ldots\}$.

(3) $B = \{0,1\} \times \{0,1\} \times \ldots \times \{0,1\}$ (m times).

The *ith* member of D indicates the delay of the *ith* event and the *ith* member of B indicates the blocking status of the *ith* event, with 0 indicating ineligibility.

In order to facilitate future presentation, we now introduce several projection functions on scheduler states. Let $S = (q;d_1,d_2,\ldots,d_m;b_1,b_2,\ldots,b_m)$ be an arbitrary element of SS. We have

(1) *pstate*:SS ---> Q by *pstate*(S) = q.

(2) *delay*:SS × E ---> NN by *delay*(S,i) = d_i.

(3) *blocked*:SS × E ---> {*true*,*false*} by *blocked*(S,i) = {if b_i=1 then *true* else *false*}.

(4) *blockedset*:SS ---> 2(E) by *blockedset*(S) = {i | *blocked*(S,i)} where 2(E) is the power set of E.

(5) *totalstate*:SS ---> T by *totalstate*(S) = (q,*blockedset*(S)).

3.2 *Schedules and the Scheduler*

A *schedule* for a parallel program P will be the non-empty sequence of scheduler state that correspond to a particular computation of P and the *scheduler* of P will be all schedules. We will show that, appropriately defined, P's scheduler is isomorphic to P's computation set.

Definition: Let P be a parallel program which has m event. Let $Z = S_1,S_2\ldots$ be a finite or infinite sequence of scheduler states. Then Z is a *schedule* for P if and only if

(1) $S_1 = (q_1;\ 0,0,\ldots 0;\ 0,0,\ldots,0)$ (2_m zeroes)

(2) For i>1, let $S_i = (q;\ d_1,\ldots,d_m;\ b_1,\ldots,b_m)$ and $S_{i+1} = (q';\ d_1',\ldots,d_m';\ b_1',\ldots,b_m')$. Exactly one of the following two cases must hold:

(a) There is an event e in P such that

(i) $\tau(q,e)=q'$.

(ii) $d_j' = \{\text{if } j{=}e \text{ then } 0 \text{ else } d_j + 1\}$

(iii) $b_j' = \{\text{if } j{=}e \text{ then } 0 \text{ else } \{\text{if } \tau(q,j) \text{ is defined then } 1 \text{ else } b_j\}\}'$.

In this case we say e *executes* and denote by $S_i\ R(e)\ S_{i+1}$.

(b) There is an event e in P such that

(i) $\tau(q,e)$ is undefined and $q' = q$.

(ii) $d_j' = \{\text{if } j{=}e \text{ then } 0 \text{ else } d_j + 1\}$.

(iii) $b_j' = \{\text{if } j{=}e \text{ then } 1 \text{ else } b_j\}$.

In this case we say e *blocks* and denote by $S_i\ R(e')\ S_{i+1}$.

Definition: Let P be a parallel program. Then the *scheduler* for P is the set S = {Z a sequence of scheduler states | Z is a schedule for P}.

Theorem: Let P be a parallel program. Then the set of P's computations C is isomorphic to P's scheduler *S*.

Proof:

We only sketch the proof. Define the function *makesch*:C ---> *S* as follows:

(1) *makesch*(Λ) = (q_1; 0,0,...0; 0,0,...,0)

(2) For xf in C where f is in EA and *makesch*(x) = S, *makesch*(xf) = S' such that S R(f) S'.

It should be clear that *makesch* is well-defined, one-to-one, and onto.

Notation: We let *makecomp* denote the inverse function of *makesch*.

As with computations, we will talk of finite, infinite, and terminating schedules.

4.0 *Initial Scheduler Policies*

In this section we introduce three scheduling policies, the first two are common in the literature - the third new, which allow us to develop our concept of response time.

4.1 *The Busy Wait Free POlicy*

Recall that in example 1 we had z = 123(1')* as a legal computation in which event 1 is forever blocking. Although the program is technically executing, it is essentially doing nothing. This phenomenon has been dubbed *busy wait* [5] and great care has been taken to avoid it in the design of operating systems [3,6,9]. Hence, our first scheduling policy will be a "busy wait free" policy.

Definition: Let $Z = S_1 S_2 \ldots$ be a schedule for a parallel program P. Z is called *busy wait free* if for all $i \geq 1$, S_i R(e') S_{i+1} implies *not* *blocked*(S_i, e).

Intuitively, under the busy wait free policy once an event e blocks e may not block again until e has executed at least once. This rules out 123(1')* as a computation but 1231'(43)* is still legal. The busy wait free scheduler for P is then

Definition: The set $SF = \{Z \text{ in } S \mid Z \text{ is busy wait free}\}$ is called the *busy wait free scheduler* for P'.

and the allowable computations under the busy wait free policy are

Definition: The members of the set $CF = \{makecomp(Z) \mid Z \text{ is in } SF\}$ are

called the *busy wait free computations* of P.

The following result is immediate from the definitions of busy wait free schedules and computations.

Lemma 1: w in C is in CF if and only if for all events e and decompositions w = xe'ye'z, we have *substr*(y,e).

4.2 *The Release Policy*

As noted above, even with the busy wait free policy we have z = 1231'(43)* as a legal computation for example 1. In z event 1 blocks but is never released (i.e., it never executes again even though it is capable of doing so). This is, in general, unacceptable. For example, event 1 could represent a data recording process and we would want it to eventually be executed if it has data to record. Satisfying this criterion has been called showing that an event executes "infinitely often" (if it is capable of doing so) [1,4,11]. Necessary for showing that an event executes infinitely often is the imposition of a "release" scheduling policy.

Definition: Let Z = S_1 S_2 ... be a busy wait free schedule for a parallel program P. Z is called a *release* schedule if for all i≥1 and arbitrary distinct events e and f, we have S_i R(e) S_{i+1}, *not blocked*(S_i,e), and *blocked*(S_i,f) imply (*pstate*(S_i),f) is undefined.

Intuitively, under the release scheduling policy when there is a choice between executing either a blocked or non-blocked event the blocked event is chosen. Thus 1231'(43)* is ruled out as a computation for example 1 since for the second and subsequent executions of event 3 the blocked

event 1 could have been executed. We now have

Definition: The set *SFR* = {Z in *SF* | Z is a release schedule} is called the *release scheduler* for P.

and the allowable computations under the busy wait free scheduling policy are:

Definition: The members of the set CFR = {*makecomp*(Z) | Z is in *SFR*} are called the *release computations* of P.

The following result is immediate from the definitions of release schedules and computations:

Lemma 2: w in CF is in CFR if and only if for all distinct events e and f and decompositions w = xey, we have xfy in CF and *blocked*(x,f) imply *blocked*(x,e).

Here, *blocked*(x,e) is the expected predicate on *tstate*(x).

4.3 *The N-Fair Policy*

Under the release scheduling policy we still have z = (1231'(43)*4)* as a legal computation for example 1. In z event 1 executes infinitely often but from any blocking of 1 to its subsequent execution an arbitrary number of scheduler steps may pass. In certain applications this would be intolerable. To remedy this situation we introduce an "N-fair" scheduling policy.

Definition: Let $Z = S_1 S_2 \ldots$ be a release schedule for a parallel program P. Let N be a fixed integer ≥ 1. Z is called an *N-fair schedule* if for

all i≥1 and all e in E, *not blocked*(S_i,e) implies *delay*$(S_i,e) \le N$.

Intuitively, under the N-fair scheduling policy events which are not blocked will undergo event activity (execute or block) within N scheduler steps from the point of their last execution. Of course, blocked events may have to wait longer than N scheduler steps or forever, depending on the structures of the particular program. Thus in z event 1 would remain blocked for at most N/2 executions of event 3 since the N-fair policy coupled with the release policy would force the scheduler to consider event 1 at that time. We now have the following definitions:

Definition: For fixed N≥1, the set *SN* = {Z in *SR* | Z is an N-fair schedule} is called the *N-fair scheduler* for P.

and the allowable computations under the N-fair scheduling policy are:

Definition: For fixed N≥1, the members of the set CN = {*makecomp*(Z) | Z is in SN} are called the *N-fair computations* of P.

The following results are immediate from the definitions of N-fair schedules and computations:

Lemma 3: For fixed N≥1, w in CFR is in CN if and only if for all events e in E and decompositions w = xyz with |y|>N, *not substr*(y,e), and *not substr*(y,e'), we have *blocked*(x,e).

Here |y| denotes the length of the string y.

Lemma 4: For fixed M > N ≥ 1, we have CN is a subset of CM.

Before proceeding, we present a lemma which will be crucial in proving

our response time results.

Lemma 5: For a fixed $N \geq 1$, let x be in CN with the following properties:

(1) $x = yz$ with $|z| = M > N$.

(2) *tstate*(y) = *tstate*(z).

Then for all $i \geq 1$, x_i is in CM where $x_i = yzz...z$ (i copies of z).

Proof:

Note that by the determinism of τ we have *tstate*(x) = *tstate*(x_i) for all $i \geq 1$. For $i=1$, $x_i = x$ is in CM by Lemma 4. Fix $i>1$.

(1) If x_i is not in C, then we contradict x being in C.

(2) If x_i is not in CF, then we contradict Lemma 1.

(3) If x_i is not in CR, then we contradict Lemma 2.

(4) Suppose that event e is the reason why x_i is not in CM. There are three subcases:

   (a) If *substr*(z,e), then we contradict Lemma 3.

   (b) If *not substr*(z,e) and *blocked*(x,e), then we contradict Lemma 3.

   (c) If *not substr*(z,e) and *not blocked*(x,e), then we contradict xz being in CN.

4.3.1 *Implementation Considerations*

Implementing the busy wait free and release scheduling policies is a rather trivial task: the decisions to be made in a scheduler step can be determined entirely from the scheduler state independently of past or

future decisions (i.e., the scheduler would be a Markov process). Note, however, that this is not true when an N-fair policy is in effect. When making a decision on event activity the scheduler must consider not only past decisions (i.e., event delays) but also the structure of the parallel program under consideration since a faulty decision might make violation of the N-fair policy inevitable. Thus, some degree of "lookahead" must be done. While this can always be done for finite state parallel programs, there will be some infinite state programs which require infinite lookahead and thus N-fair scheduling becomes impossible.

As can readily be seen in example 1, low values of N can severely restrict the scheduler. For example, under 2-fair scheduling there are only four computations: 12, 13', 34, and 31'. Since each computation is non-terminating, we have an anomalous situation. In general, we should choose N at least as large as the number of events in the program.

5.0 *Response Time of Parallel Programs*

Recall in the computation z = (1231'(43)*4)* for example 1, under N-fair scheduling once event 1 blocks it will wait at most N scheduler steps to execute (i.e., respond). We call this time of waiting the *response time* of an event. We will be concerned with the *worst case* response time of an event for all possible schedules since a parallel program with acceptable worst case behavior is acceptable in general.

In most applications we would like all events to have finite response times. Moreover, we would like these finite response times to be "acceptable" in some sense. Suppose we have a two event parallel program P in which it is known that both events, say e and f, have finite response time for all

values of N. Suppose further that event e has acceptable response time r(e) for $N = t$ but f's response time is unacceptable for $N < 5t$. Hence, we must adopt a 5t-fair scheduling policy to have any hope that both events will have acceptable response time. A basic question is: how is event e's response time affected by this increase in N? In this section we answer the question by showing that e's response time will increase only linearly in N.

We have the following definitions:

Definition: Let e be any event of a parallel program P and for $N \geq 1$, let $Z = S_1 S_2 \ldots$ be in *SN*. The *response time* of e in Z, denote $r(e,N,Z)$, is

case 1: Z is a terminating schedule with S_n the final scheduler state and *blocked*(S_n,e). Then $r(e,N,Z)$ is infinity.

case 2: Otherwise,

$$r(e,N,Z) = \max\{delay(S_i,e) \mid i \geq 1\}.$$

The *N-response time* of e, denote $r(e,N)$, is

$$r(e,N) = \max\{r(e,N,Z) \mid Z \text{ is in } SN\}.$$

Hence, there are two ways that $r(e,N)$ might be infinite: the program could terminate with e blocked, or e might block and never execute again in spite of the fact that the program never terminates. This latter condition has been defined as "individual starvation" [7].

The following result is immediate from the definitions:

Lemma 6: Let e be any event of a parallel program P and for $N \geq 1$, let

$Z = S_1\ S_2 \ldots$ be in *SN*. If $r(e,N,Z) > N$ then e is blocked in Z for $r(e,N,Z)$ consecutive scheduler steps.

5.1 *Response Time Theorem*

We first prove the following lemma, which holds for general string systems.

Lemma 7: Let A be a finite set, w in A*, and $N \geq 2$. If $|w| \geq 2|A|N + 2$, then there exist an a in A such that w can be decomposed as w = xayaz with $|y| > N$.

Proof: (by induction on $|A|$)

If $|A| = 1$ then $|w| \geq 2N + 2$. The form of w must be w = aya where $|y| \geq 2N > N$.

Assume the result holds for $|A| < k$, for fixed $k \geq 2$. If $|A| = k$ then $|w| \geq 2kN + 2 > 2N + 2$. Assume a is the first character of w and decompose w as w = axy where $|x| = N + 1$. We have two cases:

(1) If *substr*(y,a) we are done.

(2) If *not substr*(y,a) then y is in $(A-\{a\})^*$ and $|A-\{a\}| = k - 1$. We have

$$|a| + |x| + |y| \geq 2kN + 2.$$

Thus

$$|y| \geq 2kN + 2 - 1 - N - 1 = 2kN - N.$$

Since $2 - N \leq 0$, we have

$$|y| \geq 2kN - N + 2 - N = 2(k - 1)N + 2.$$

By the induction hypothesis on y, there is a b in A-{a} such that y can be decomposed as $y = x'by'bz'$ and $|y'| > N$.

We are now ready to prove our main result.

Response Time Theorem: Let P be a parallel program having n states and m events. For any event e of P either:

(1) There exist $N \geq 2$ such that for all $M \geq N$ $r(e,M)$ is infinity, or

(2) For all $N \geq 2$ there is a constant $c > 0$ such that $r(e,N) \leq cN$.

Proof:

Assume that $r(e,N)$ is finite for all $N \geq 2$. Suppose there is an $M \geq 2$ such that for all constants $c > 0$ we have $r(e,M) > cM$.

Let $d = n2^m = |T|$ be the number of total states of P and look at the constant $c' = 2d + 1$. There must be a schedule Z in *SM* such that $r(e,M,Z) > c'M$. Let $Z = S_1 S_2 \ldots$ and let $Y =$ *totalstate*(S_1) *totalstate*$(S_2) \ldots$

By Lemma 6 we can decompose Y as $Y = X_1 X_2 X_3$ where $|X_2| = c'M$ and e is always blocked in X_2. Thus,

$$|X_2| = c'M = (2d + 1)M = 2dM + M \geq 2dM + 2.$$

Applying Lemma 7 to X_2, there is a total state X such that $X_2 = X_4 X X_5 X X_6$ and $|X_5| > M$. Clearly, e is blocked in X.

Rewriting Y, we have $Y = X_1 X_4 X X_5 X X_6 X_3$ as the sequence of P's total states which correspond to the schedule Z.

Let $L1 = |X_1 X_4 X| - 1 \geq 0$ and $L2 = |X_5 X| - 1 > M$. Look at the com-

putation corresponding to the schedule Z: $z = makecomp(Z)$. We can decompose z as $z = z_1\ z_2\ z_3$ where $|z_1| = L1$ and $|z_2| = L2$.

By the above arguments we have $tstate(z_1) = tstate(z_2)$, $blocked(z_1,e)$, and e doesn't execute in z_2 (i.e., *not substr*(z_2,e)). Also, z_1z_2 is in CM and $|z_2| = L2 > M$. Hence, by Lemma 5, $z_1\ z_2\ z_2\ z_2 \ldots$ is in $C^{L}2$ and it follows that $r(e,L2)$ is infinity. Thus, with this contradiction, $r(e,M) \le cM$ for all $M \geq 2$.

As an interesting sidelight of this proof we have established an upper bound on c to be $1 + n2^{m+1}$.

6.0 *Additional Scheduler Policies*

We have defined only the minimum amount of scheduler policies needed to proof the response time theorem. Observe that the N-fair scheduler will make an arbitrary choice when more than one blocked event is capable of executing. Because of this, it is possible that an event may have an infinite response time even though it is always capable of executing. A way to avoid this is by a *FIFO scheduling* policy, as has been suggested in [6,9]. We have

Definition: Let P be a parallel program and for fixed $N \ge 2$, let $Z = S_1\ S_2 \ldots$ be in *SN*. Z is called an *N-fair FIFO schedule* if for all $i > 1$ and arbitrary distinct events e and f, the following holds: $S_i\ R(e)\ S_{i+1}$, $blocked(S_i,f)$, and $\tau(pstate(S_i),f)$ defined imply $delay(S_i,e) \ge delay(S_i,f)$.

In this definition note that the release policy guarantees $blocked(S_i,e)$. Intuitively, in FIFO scheduling when there is a choice of executing several

blocked events, an event which has been blocked for a maximum number of scheduler steps is chosen. Since the FIFO policy is a restriction of N-fair scheduling, we have the following corollary:

Corollary 1: The response time theorem holds under an N-fair FIFO scheduling policy.

In certain applications it is desirable that the choice among blocked event be made on the importance of the events rather than on the egalitarian FIFO rule. This called *priority scheduling* [3]. Each event is given a priority as follows:

Definition: Let P be a parallel program. A *priority function* is a total mapping $\theta:E \longrightarrow \mathbb{N}$.

When several blocked events are capable of executing, the choice is made on the basis of maximum priority:

Definition: Let P be a parallel program with priority function θ. For fixed $N \geq 2$, let $Z = S_1 S_2 \ldots$ be in *SN*. Z is called an *N-fair θ priority schedule* if for all $i \geq 1$ and arbitrary distinct events e and f, the following holds: S_i R(e) S_{i+1}, *blocked*(S_i,f), and τ(*pstate*(S_i),f) defined imply $\theta(e) \geq \theta(f)$.

Note, unlike FIFO scheduling, it is possible under priority scheduling for an event to have infinite response time even though it is always capable of executing. Since priority scheduling is a restriction of the N-fair policy, we have

Corollary 2: The response time theorem holds under an N-fair priority

scheduling policy.

7.0 *Response Time Decision Procedures*

There are two additional questions we must answer in order to completely describe the response time behavior of a parallel program:

(1) Given an event e of a parallel program P, is the response time of e ever infinity?

(2) What is the minimum constant $c>0$ which describes the linear growth of $r(e,N)$ as N grows?

We will answer these questions by providing decision procedures for the following:

(a) Does there exist an $N \geq 2$ such that $r(e,N)$ is infinite?

(b) Given $c>0$ is $r(e,N) \leq cN$ for all $N \geq 2$?

The answer to question 1 follows directly from (a). Question (2) is answered by (b) and the observation in section 5 that the minimum constant is bounded above by $1 + n2^{m+1}$.

The decision procedures for (a) and (b) will be by reduction to questions about suitably encoded vector addition systems.

7.1 *Vector Addition Systems*

In this section we briefly review the definition of vector addition systems [10], their decision procedures that we will use, and relate these systems to our definition of a parallel program's scheduler.

Definition: A *vector addition system* of degree k, denote VAS, is a 2-tuple W = (v,V) where:

(1) The *start vector* v is in NN^k = NN X NN X ... X NN (k times).

(2) V is a finite set of *vectors*, each in ZZ X ZZ X ... X ZZ (k times) where ZZ = {...,-2,-1,0,1,2,...}.

Definition: The *reachability set* of a VAS W, denote R(W), is a subset of NN^k recursively defined as follows:

(1) v is in R(W).

(2) for x in R(W) and w in W, $x+w$ is in R(W) iff. $x+w \geq 0$.

We will be using the following two problems which are concerned with the reachability set of a VAS.

Definition: The *boundedness problem*: given an arbitrary $x \geq 0$ is there a y in R(W) such that $y \geq x$?

Definition: The *reachability problem*: given an arbitrary $x \geq 0$, is x in R(W)?

A decision procedure for the boundedness problem can be found in [10]. The decidability of the reachability problem has recently been claimed in [13].

The following lemma, given without proof, provides a link between the scheduler of a parallel program and vector addition systems. The lemma holds since vector addition systems can represent any finite state control [10].

Lemma 8: Let P be a parallel program and *SR* its release (and busy wait

free) scheduler. Then there is a VAS W and a homomorphism h:*SR* ---> W such that Z in *SR* implies h(Z) is in R(W).

7.2 *A High Level VAS Language*

Rather than work with vectors of integers, it will be more convenient and convincing to give the VAS reductions in terms of a "high-level" non-deterministic VAS language. This approach has been previously used in [12].

There are five statement types in the language: initialization of variables, assignment, non-deterministic branch, testing the finite state control, and updating the finite state control. All but the first statement types may have a statement label. The syntax and semantics are as follows:

Initilization

var $v_1=a_1, v_2=a_2, \ldots, v_n=a_n$

The distinct variables $v_1, v_2, \ldots, v_n$ are initilized to the respective natural numbers $a_1, a_2, \ldots, a_n$. Variables not initilized start at zero.

Assignment

$v_1 \longleftarrow v_1 + c_1, \ldots, v_n \longleftarrow v_n + c_n$

where the v's are distinct variables and the c's are integers. The assignment can take place only if $x_i + c_i \geq 0$ *for all* i. Otherwise, the VAS computation terminates.

Guessing

guess$(s_1, s_2, \ldots, s_n)$

This statement causes a non-deterministic branch to one of the statements

labelled $s_1, s_2, \ldots, s_n$. If n=1 then the branch is deterministic.

Testing Event Activity

event(character)

This statement is used to see which events are eligible for event activity. It returns a list of eligible events, each prefixed by the supplied character. If no event activity is possible (i.e., the parallel program has terminated), then the list consists exclusively of the supplied character. For example, if events 1 and 2 can execute and event 3 can block, then *event*(s) returns s_1, s_2, s_3. If no event activity can take place, the list would be s. *Event* is always used in conjunction with the *guess* statement, e.g., *guess*(*event*(s)).

Testing for a Blocked Event

blocked(e,s)

This statement causes a branch to statement s if event e is blocked. If e isn't blocked then the statement acts like a no-op.

Updating the Control

update(f)

Here f is either an event execution (e) or an event blocking (e'). This statement reflects in the finite state control the result of event activity f.

Globally, VAS programs are listed one statement per line and execution commences at the first statement. Execution proceeds sequentially until a *guess* is encountered, whence several non-deterministic computations may be spawned. A computation may terminate in the ways listed above or by executing the last statement in the list (when it is not a *guess* statement).

Although not listed above, we have also a *no-op* statement with the obvious semantics.

7.3 *VAS Reductions*

We now show that the questions posed above are reducible to questions about suitable VAS systems.

Theorem 2: Let P be a parallel program having n states and m events. For an event e of P the question of whether or not there is an $N \geq 2$ such that $r(e,N)$ is infinity is reducible to a boundedness question.

Proof:

The vector addition system will have the following coordinates:

$\langle$finite state control, local control$,d_1,\ldots,d_m,M_1,\ldots,M_m,B\rangle$

To facilitate the presentation of the VAS program we will employ obvious abbreviations described in comments and the following two macros:

```
zerodelay(<1>,<2>)
 l: d<1> <--- d<1> - 1,M<1> <--- M<1> + 1
    guess(l,<2>)
```

The macro takes two string inputs and does the usual concatenation. Its function is to try to set the delay counting variable d<1> to zero.

```
sucdelay(<1>)
    blocked(<1>,l)
    d<1> <--- d<1> + 1,M<1> <--- M<1> - 1
 l: no-op
```

The purpose of this macro is to increase the delay count variable d<1> of a non-blocked event.

The VAS program is as follows:

var M_1=1,M_2=1,...,M_m=1

comment: guess N

10: M_1 <--- M_1 + 1,...,M_m <--- M_m + 1

guess(10,20)

comment: Simulate P - guess to start phase 3 whenever e blocks

20: *guess*(*event*(2))

2: *guess*(20)

comment: the following group of statements is repeated for $1 \le i \le m$.

2i: *zerodelay*(i,2ii)

2ii: *update*(i)

comment: the following statement is repeated for each j not equal to i.

sucdelay(j)

guess(20)

comment: the following group of statements is repeated for each i not equal to e.

2i': *zerodelay*(i,2ii')

2ii': *update*(i')

comment: the following statement is repeated for each j not equal to i.

sucdelay(j)

guess(20)

2e': *zerodelay*(e,2ee')

2ee': *update*(e')

comment: the following statement is repeated for each j not equal to e.

sucdelay(j)

guess(20,30)

comment: simulate P assuming that e never executes again.

30: B <--- B + 1

guess(*event*(3))

3: *guess*(30)

comment: as in phase 2, the following group of statements is repeated. However, here it is repeated for each i not equal to e.

3i: *zerodelay*(i,3ii)

3ii: *update*(i)

comment: the following statement is repeated for each j not equal to i.

sucdelay(j)

guess(30)

comment: the following group of statements is repeated for each i not equal to e.

3i': *zerodelay*(i,3ii')

3ii': *update*(i')

comment: the following statement is repeated for each j not equal to i.

sucdelay(j)

guess(30)

comment: by busy wait free, 3e' is impossible.

3e: *guess*(3e)

The result follows since r(e,N) is always finite iff. B is bounded.

Several comments are in order about this VAS program. In phase 1, by non-determinism, every value of $N \geq 2$ is considered. In phase 2, the N-fair execution of the parallel program is simulated. An event activity

as dictated by the finite state control is non-deterministically chosen and appropriate event delay counts are performed on the d variables. The variable pairs d_i and M_i play a crucial role in that they force only N-fair computations to be considered. Note that $d_i + M_i = N$ is invariant. When an event is executed or blocked we try to set d_i to zero. The crucial part of the simulation is to observe that even if d_i isn't set exactly to zero (it will be in some computation) we still get N-fair computations since M-fair computations are N-fair computations for $M<N$. Similarly, d_i is increased by 1 whenever event i is blocked and another event activity takes place.

Phase 3 is started non-deterministically whenever event e blocks in phase 2. The purpose of phase 3 is to assume e will never execute again and reflect this in variable B. If e does execute, then phase 3 loops forever and B is bounded. If the parallel program terminates (i.e., no event activity with e blocked) or e is never executed again, the B grows unboundedly.

For the second VAS reduction we will simply modify the above VAS program.

Theorem 3: Given $c>0$, whether or not $r(e,N) \leq cN$ for all $N\geq 2$ is reducible to a reachability question.

Proof:

A variable D is added to the above VAS program with an initial value of $D = c + 1$. Statement 10 is changed to

10: M_1 <--- $M_1 + 1, \ldots, M_m$ <--- $M_m + 1$, D <--- D + c

Hence, after phase 1 completes D has a value of cN + 1.

A fourth phase is added at the end of the program as follows

```
40: D <--- D - 1,B <--- B - 1
    guess(40)
```

The purpose of the fourth phase is to see if B ever is $\geq$ D. If so, then there must be some VAS computation in which B = D and thus r(e,N) > cN. This happens only when D = B = 0 can be reached. It remains only to make changes to the VAS program to non-deterministically start phase four. They are:

(1) Change statement 2 to 2: *guess*(40).

(2) Change each *guess*(30) in phase 3 to *guess*(30,40).

Hence r(e,N) > cN iff. D = B = 0 is reached.

8.0 *Conclusions*

We have introduced the notion of an N-fair scheduling policy as a condition which allows the development of theoretical results on the response time behavior of parallel programs. We have shown that for any event either the response time is infinite or it is linear in the choice of N, that one can determine which is the case, and that one can compute the exact linear relationship in the finite case.

Although the methods used would seems to indicate that computing the exact response time behavior of a parallel program is an intractable task, the development of heuristics for computing good upper bounds on response time is under investigation.

References

[1] J. Cadiou and J. Levy.
"Mechanizable Proofs about Parallel Programs."
Fourteenth Symposium on Switching and Automata, Oct. 1973.

[2] R. H. Campbell and A. N. Habermann.
"The Specification of Process Synchronization by Path Expressions."
Proc. Int. Symp. on Operating Systems Theory and Practice, April 1974.

[3] E. G. Coffman and P. J. Denning.
Operating Systems Theory.
Prentice Hall, Englewood Cliffs, 1973.

[4] E. S. Cohen.
"A Semantic Model for Parallel Systems with Scheduling."
Proc. Second Symp. on Principles of Programming Languages, 87-94.

[5] E. W. Dijkstra.
"Cooperating Sequential Processes."
In *Programming Languages*, ed. F. Genuys, Academic Press, New York, 1968, 43-112.

[6] E. W. Dijkstra.
"The Structure of the THE Multiprogramming System."
CACM 17,10 (May 1968) 341-347.

[7] E. W. Dijkstra.
"Hierarchical Ordering of Sequential Processes."
ACTA Informatica 1,2 (1971) 115-138.

[8] P. J. Gilbert and W. J. Chandler.
"Interference Between Communicating Parallel Processes."
Comm. of the ACM 15,6 (June 1972) 427-437.

[9] C. A. R. Hoare.
"Monitors: An Operating System Structuring Concept."
CACM 17,10 (Oct. 1974) 549-562.

[10] R. Karp and R. Miller.
"Parallel Programming Schemata."
J. Computer and Systems Science, May 1969, 147-195.

[11] R. J. Lipton.
"On Synchronizing Primitive Systems."
Proc. Sixth Annual Symp. on the Theory of Computing, May 1974.

[12] R. J. Lipton.
"The Reachability Problem Requires Exponential Space."
Yale University Dept. of Comp. Sci. Research Report #62, Jan. 1976.

[13] G. S. Sacerdote and R. L. Tenney.
"An algorithm for the reachability problem for vector addition systems."
abstract, 76T-E61, *Notices of the AMS* 23,6 (Oct. 1976) A595.