A Note on Asynchronous Parallel Processing*

H. Witsenhausen

Asynchronous parallel processing within a given job in an essentially multi-processor computer is considered in a limited framework. The items discussed are:

1. The scope of an element of data must be defined in two dimensions: time and ownership.
2. The operations performed at a fork.
3. The operations that replace the notion of join.
5. That loss of time due to lock-out is not significant.
6. What a programmer's specification of parallelism could be like.
7. Some hardware features that may be helpful.

* This is a slightly edited version of a term paper submitted for subject 4.40 in January 1964. - J.B.D.
CONTENTS

Parallel Processing

Paradigm and Nonsequence Execution

General

Program Structure

Data Equivalence

Program Description

Special Cases

Allocation

Initial Processing

Conditions for Parallel Operation

Public and Private Data

Tarks

Element Definition

Flow Diagrams

Tape Example

Hardware Control of Data Ownership

Migration of Data Ownership

Programmers Description

Matrix Multiplication Example

Loops of the "While" Type

Locked Branches for Scope Termination

Publication, Grabbing and Lock-Out of Data

Locking-In of subprograms

Execution Trees

Look-Ahead

Sharing Processing Capability

Application
Types of Parallel Processing

The distinction between sequential, single processor and parallel, multiprocessor computers is itself not clear.

a) Parallel treatment of several bits and registers within a single processor is not considered parallel processing.

b) Local concurrency in the sense of Codd, which is found in the 360 and in the Stretch look-ahead scheme does not raise the same type of problems as the operation we consider below. (No programmers specification)

c) Synchronous parallel execution, as found in the Solomon computer, is similar to conventional operation in that processors requested are assumed always available and the detailed relative timing of all operations is completely planned. Logically, these are single processor computers with a distributed processor.

d) We shy away from the difficult and very important case where some relations (inequalities) between execution times of some program sequences by processors are known in advance. In such a case a "fork" does not necessarily give a "joint". The processor executing the shorter prong returns unconditionally to the idle processor list while the other processor unconditionally takes the program, secure in the knowledge that the parallel operations have been completed in due time. As opposed to this we assume that no upper limit can be predicted for the time of execution of any program sequence, for instance because any processor may be pulled out from under us by a higher priority interrupt from some other program.

e) Parallel operation of processors assigned to different jobs, on disjoint data, does not lead to the same problems (some analogies occur in case of shared subroutines). It is essentially a scheduling problem which attempts to satisfy optimization criteria instead of being obligated to satisfy constraints.

Summarizing, we distinguish

1. conventional single processor computers
2. ditto with local concurrency
3. distributed processor computers (synchronous multiprocessors)
4. asynchronous multiprocessor computers with interjob concurrency only.
5. ditto with intrajob concurrency
6. ditto with consideration of known execution time relationships
7. ditto with real time constraints for operation with an
   environment (strict inequality constraints).
We address ourselves to case 5, and this in a restricted framework.

II. Framework

In order to concentrate on the questions of most interest to the
writer, a number of simplifying assumptions will be made, unless otherwise
stated, in this note.

1. We seek for a minimum of changes in conventional memories and
processors, such that parallel operation becomes easy to implement, rather than
be a radically new structure (such as ALPS). In particular one should be able
to run any problem sequentially with a single processor in the conventional manner,
desired.

2. We consider a single job. In fact, the multiprogramming concurrency
of several jobs is essential to make parallel processing efficient. Idle processors
must be able to find employment. This scheduling problem is similar to that of
sequential computers with I/O concurrency and is not our concern.

3. While the main reason for parallel operation within a job is to
capitalize its completion as required by the final user, we will not consider the
problems raised by rigid real time constraints imposed on the production of certain
results.

4. I/O operations are not considered; in the parallel processing frame-
work they differ from other program segments solely by the necessity to obtain
access to one or more of a special kind of processing units. In all other respects
they are implicitly included (cf. Conway).

5. No interrupts or traps are considered explicitly though the
possibility of interrupts is one of the factors that make execution time of a
program segment a random variable without upper bound and uncorrelated with that
of other executions.

6. Allocation problems are considered only from the point of view of
who should be saved and to whom it is accessible (the 2-dimensional scope problem)
there it is stored. The latter would be handled by an extension of the
static storage allocation techniques for the single processor case (cf. Van Horn).

**Remarks on Monosequence Execution**

**General**

Monosequence execution of a job means its accomplishment by a string of
active steps, which include conditional branching. It has been shown by
Pen ear a single instruction is logically sufficient:

\[
\begin{align*}
(A) \rightarrow (Y) &\rightarrow (A) ; \quad (A) \rightarrow (Y) \rightarrow (Y) \\
(PC) + 1 &\rightarrow (PC)
\end{align*}
\]

\[\{ A's \ complement \]

A distinction between processor state and other data is not essential.
Multiple address instructions the state of the processor between instruc-
tions is characterized solely by the program counter. The latter could be a
static storage location. The notion of state-word is unnecessary as long as
there are no interrupts. Even with interrupts, provision for multiple program
(TX2, Honeywell 800) does away with the state-word concept. In
machines this concept occurs only because of technological structure:
special location of AC, M2, XR's... makes a difference as to how they
specified (implicitly, tags, ...) and how fast they are accessible, and
more.

The specification of the algorithm carried out by the machine can be
made at various levels.

Representation instruction by instruction is shown in Fig. 1. Without
interrupts and exception traps, and taking all illegal instructions as halts:

**Figure 2.** This is the representation considered by Karp.

Representation at the program level is provided by
- flow diagrams (Rutishauser's, Plan-Kalki1)
- Algorithm schemes (A. Markov, A.A. Lyapounovland, V.I. Yanov)
- Incidence matrices (Karp)
Instruction-by-instruction representation.

Simplified instruction-by-instruction representation.
All use the notion of operator which can be a single instruction or a sequence without conditional branching out of the operator; the terminal control point of an operator is unique.

These schemes do not explicitly account for subroutine linkage because the terminal control point of the last operator of the routine is not unique. To make it unique one must follow it by a series of tests to identify the call point; an unrealistic representation. This can be avoided by returning to the instruction level.

To prepare for discussion of parallel processing we now introduce a specific representation.

### Program Structure

As in the flow diagram approach we consider data as distinct from program. We consider all programs to be pure procedure (by use of indirect addressing, index registers, etc.). Our reason for doing so is not the relocation problem (that we do not consider) but the problems of parallel processing.

Programs are considered as made up of a finite and fixed number $n$ of segments with a unique entry point to each. They are labeled by the integers $1$ to $N$, label $0$ being reserved for halt (or trap to supervisor). When a subroutine call is to be explicitly shown, the calling segment is split, creating a label for the return point. The subroutine entry point has of course its own label.

### Data Equivalence

The program, being rigidly fixed, can be thought of as in a protected read-only store. All other information, including the contents of processor registers, is called data. In a finite machine the data can have a finite number of states only. At any time there are likely to be large numbers of states which are equivalent as far as the execution of the job, or of the current segment, is concerned.

At halt or at end-of-segment time this equivalence has two causes: 1) only a fraction of the words in store are of interest, the others contain immaterial information. 2) when the original data is inappropriate, exception conditions may occur leading to the recording of error flags and possibly a premature halt.
The production situation, as distinct from debugging, all these final states belong to the single garbage class.

In effect then, if the job terminates, we are only interested in the final state within an equivalence. At all previous stages we are only concerned distinguishing those states that will lead to non-equivalent final states, the states that will lead to endless operation are all members of the garbage class.

In some way, an equivalence can be defined for any program segment by considering it as a program in its own right.

Program Description

Each program segment, whenever it is called into operation has access to the current state of the data. Its effect is to map this initial state into a final state and into one of the \( N > 1 \) labels 0 to \( N \). If the label producing algorithm was to lead to anything but one of these, this would be considered a label 0 with error flag. We thus consider the program perfectly debugged for the initial data not necessarily meaningful.

Each segment is defined by two single-valued functions.

\[
\text{segment } i \begin{cases} D_1 = f_i(D_0) \\ L = g_i(D_0) \end{cases} \quad i = 1, \ldots, N
\]

where \( D_0 \) and \( D_1 \) range over the possible data states, while \( L \) is in the set of integers 0 to \( N \).

The description of the program by these \( 2N \) functions is felt to be more general than the description of Yanov. The function \( g_i(D_0) \) partitions the states into \( i + 1 \) classes. The tests of boolean variables constrained by change lists are much more restrictive, even when the maximum of 9 possibilities to which Shaffer's scheme can be extended is taken into account. These are obtained, for a given operator, by combining the 3 possible change conditions of the initial value zero with the 3 similar conditions for the initial value one. The 2 conditions are no change, complementation, either. This still does not account for the fact that the effect of an operator depends on the previous operator among and the initial data.
Special Cases

1. \( g_i(D_0) \) = constant; such a segment is called a "box" (operator) and a string of boxes is a box.

2. \( g_i(D_0) = n_1 \) or \( n_2 \); such a segment is a combination of a box and a two-way branch.

3. \( g_i(D_0) \) is a function only of the label of the previous segment (supplied in \( D_0 \)). This is the case of the subroutine with unconditional return.

4. \( f_i(D_0) = D_0 \) the case of a pure branch, which can always be viewed as successive two-way branches.

Notes: Internally generated traps (overflow, etc.) do not transcend this representation since they amount to replacing by permanent hardware the coding of execution of conditional branches.

The assumption of pure procedure programs means that the functions \( f_i \) and \( g_i \) are fixed for successive passages through the same segment and that no segments can be created.

It is impossible to know the exact equivalence classes at any time without running the problem for various initial states but it is possible to define equivalence classes which are surely fine enough. Assume the error flag to be non program resettable, then all states leading to its setting in a segment are equivalent. Also states are equivalent if they differ only in locations known to be currently meaningless.

Allocation

To allow dynamic storage allocation the programmer must specify the scope of every piece of data. This amounts to specifying its release (erasure). The scopes are nested in Algol for push-down allocation but this is not a necessity.

Parallel Processing

Conditions for Parallel Operation

Given a sequential formulation of an algorithm, what is required to
two successive segments in parallel?

Let \( i \) and \( i + 1 \) be the segments in question, with

\[
\begin{align*}
\begin{cases}
    f_i(D_0) = D_1 \\
    g_i(D_0) = i + 1 \text{ for all } D_0 \text{ of interest}
\end{cases}
\end{align*}
\]

(1)

\[
\begin{align*}
\begin{cases}
    f_{i+1}(D_1) = D_2 = f_{i+1}(f_i(D_0)) = q(D_0) \\
    g_{i+1}(D_1) = g_{i+1}(f_i(D_0)) = p(D_0) = L
\end{cases}
\end{align*}
\]

They amount to the single segment

\[
\begin{align*}
\begin{cases}
    D_2 = q(D_0) \\
    L = p(D_0)
\end{cases}
\end{align*}
\]

(2)

(2) they are equivalent to any segment which results in

\[
\begin{align*}
\begin{cases}
    D_2 = q(D_0) \pmod{\text{modulo equivalence}} \\
    L = p(D_0)
\end{cases}
\end{align*}
\]

(3)

For all \( D_0 \) of interest, segments \( i \) and \( i+1 \) are said to be compatible if (3) is satisfied for any time relation of the individual instructions of the two sequences.

This is the necessary and sufficient condition for their parallel implementation in our framework.

Two other concepts occur in this connection. The segments are said to

commute if

\[
\begin{align*}
\begin{cases}
    f_i(f_{i+1}(D_0)) = f_{i+1}(f_i(D_0)) \pmod{\text{modulo equivalence, for all } D_0 \text{ of interest.}}
\end{cases}
\end{align*}
\]

This is necessary but not sufficient results from very simple examples such as \( 5 + 5 \rightarrow (x) \) and \( (x) + 5 \rightarrow (x) \) which commute but are not compatible if the additions take place in the accumulators of two different processors which fetch and store location \( x \) in two operations which are asynchronous.

Commuting segments can be made compatible by lock-out of data but this destroys the gain in time.

The segments are said to be data-disjoint if the set of locations into which segment \( i \) stores is disjoint from the set of locations accessed (for either reading or writing) by segment \( i+1 \) and vice versa, and this for all cases of interest.
This is a sufficient but not necessary condition, for instance both
segments could use the same location for temporary storage of a single inter-
mediate result which is necessarily the same in both, or they could execute
$x \rightarrow x$ and $x+5 \rightarrow x$ by add-to-memory instructions for which lock-out is automatic.

Thus we have

\[ \text{data-disjoint} \cap \text{compatible} \cap \text{commutation}. \]

At execution time, a check for compatibility could only be made by simulation-
before-execution (to check effective addresses for disjunction or to check
final results) taking more time than sequential execution. The responsibility
rests squarely on the shoulders of the programmer-compiler team.

4. **Public and Private Data**

In a monosequence execution, the same segment can be executed numerous
times (loop, subroutines) usually with different initial data (different
equivalence class) at each passage. In multiprocessing, similarly, the same
segment will have to be executed several times and this by different processors.
It will therefore occur that the same segment is being used simultaneously by
one or more processors. These executions must be compatible, - in practice,
data-disjoint.

This requires:

a) pure procedure segments referencing data by indirect means.
b) the indirect means must include at least one element
   which is distinct for each processor, for instance an
   index register $R$ associated with the physical processor, $P$.

It follows the intermediate results generated by two processors
executing the same segment go into disjoint locations and that cross references
from one processor $P_1$ to the data generated by $P_2$ are in general undesirable
and may be impossible. To find the location of a result generated by $P_2$ we
need to know the state of $R$ in $P_2$ as of the time the result was stored. Even
if $R$ has not been changed by $P_2$ it is impossible for $P_1$ to discover whether
it is physical processor $P_2$ or $P_15$ that is involved. Looking at all processor
program counters is no help, since $P_2$ may already have finished the segment
(under assumption of completely asynchronous time relationships))
In conclusion we must distinguish two types of data: public and private. Public data is accessible to all processors unless a temporary lock-out condition exists. Private data is associated with specific processors and accessible to them. This is not a "look-out" but a long-term memory protection. It can be accomplished by pointers in the processor and protected pages or segments in main memory. In case of interrupt, only pointers need to be stacked.

In main memory, access to any page or segment by any of the proposed machines (Milburn, Van Horn, Dennis) will involve checking that the reference is to public data not locked out or to private data owned or co-owned by this processor. Otherwise transfer to emergency procedures is in order. This should of course only result from error (except for lock-out).

Ownership of private data can be extended to more than one processor when new ones are called to serve, i.e. at fork time.

Forks

Program execution always starts by activating a single processor, providing it with a program counter setting (label of segment) and a pointer to an initially empty set of private data. To get more than a single execution sequence, an instruction in active processors must be possible: the fork. Forks are a new type of macro instruction that can be specified in a program segment. Their function is the creation of a new set of private data and segment label for the benefit of an additional processor. If none is available, a pointer to this data is sent to the processor scheduling system, which hands out such pointers to physical processors as soon as they become available. The fork can optionally specify several priority levels for consideration by the scheduler. If only one job is considered, the scheduler could easily be implemented in hardware. The scheduler must also consider the requirements of other jobs, I/O traffic. It will keep a number of queues, one per priority level, setting the priorities by some algorithm. A processor then becomes necessary for the scheduling job. It can be obtained by interrupting any one of the processors. (The interrupted sequence resumes immediately if a processor is available before the end of the scheduling activity). Alternatively, a special purpose processor can be reserved for scheduling and other supervisory tasks.
Expense for hardware to speed up the red tape involved in forks is justified. It reduces the minimum segment length for which parallelization is worthwhile; large numbers of low priority requests for short segments can be stacked. Even if the majority will ultimately be sequentially processed, due to the limited number of processors, an over-all gain in job completion time will result. This applies only to the highest priority in multiprogramming. It is very questionable whether parallel processing of any but the highest priority job is meaningful. The most likely situation is the presence of one job of extreme urgency and of others forming a background which prevents wastage of processing capability. In any case a lower priority job would have no incentive to fork for short segments under a reasonable scheduling algorithm.

Richards is quoted by Conway as having shown that a single queue of "state words" is not optimum. I do not know his assumptions but it seems plausible that segment length should be taken into consideration.

**Segment Definition**

Let $D_0$ be the public data at the time processor $k$ begins execution of segment $i$. Since other processors are constantly modifying public data, $D_0$ is only defined modulo equivalence for the operations of segment $i$. The equivalence classes for execution of $i$ by $k$ can not be affected by the other processors since they are assumed to be compatible. Let $P$ be the initial state of data private to $k$ (within equivalence for co-owned data). We allow only one fork per segment (segmenting segments where necessary). At termination of segment $i$ we have at most two processors with private data $P_1$ and $P_2$ which are to continue with segments labeled $j_1$ and $j_2$. The public data has undergone a transformation into equivalence class $D$.

We have, modulo equivalence for segment $i$

$$
\begin{cases}
D = f_i(D_0, P) \\
P_1 = h_1^1(D_0, P) \\
j_1 = g_1^1(D_0, P) \\
P_2 = h_1^2(D_0, P) \\
j_2 = g_1^2(D_0, P)
\end{cases}
$$
description is symmetrical with respect to the two prongs of the fork. 

try is introduced if we take the convention that processor \( k \) is to 

\( j_1 \) while a pointer to \( j_2 \) is sent to the scheduler. If 

both requests to go to the scheduler, we can do so by construction of 

\[ f_{j_1}(D, P_1) = D \]

\[ h_{j_1}(D, P_1) = \emptyset \] the empty set

\[ g^1_{j_1}(D, P_1) = o \] label of halt

\[ g^2_{j_1}(D, P_1) = P_1 \]

\[ h^2_{j_1}(D, P_1) = j_3 \] the desired continuation.

Class segments are characterized by \( h^2_{j_1} = \emptyset \) and \( g^2_{j_1} = 0 \), this amounts to the 

processor situation on the cartesian product \( D \times P \) (within equivalence, 

processors modifying \( D \) and co-owned parts of \( P \) are running). In 

\( h_1 = \emptyset \), \( h_2 = \emptyset \) we say that the processor "quits". Instead 

of notifying job completion this means that all private data is released 

(turned to free storage unless co-owned by another processor) and the 

scheduler informed of the processor's availability. Only when the scheduler 

has no requests in store for the job and all processors working on it have 

quit is job completion reached (but see exception under look-ahead below).

### Flow Diagrams

For monosequence processing, diagrams need only the following symbols:

- the box (operator), the junction (or confluence), the two-way branch, the sub-
  - return connector, the halt.

With proper interpretation only two new symbols are necessary for multi-

processing: the fork and the locked branch. Definitions of the flow diagram 

symbols in terms of segments are now given:

- **Box:** \( g^2_1 = o \), \( h^2_1 = \emptyset \), \( g^1_1 = \text{constant} \)

Figure 4a.
**Junction**: indicates equality of successor labels of two boxes; it is not a "join", no waiting is involved. Figure 4B.

**Branch**: $f_i(D, P) = D$; $h_i^1(D, P) = P$; $g_i^1(D, P) = j_1$ or $j_2$

$g_i^2(D, P) = o$; $h_i^2(D, P) = \phi$

Figure 4c.

**Locked Branch**: $g_i^1(D, P) = j_1$ or $j_2$

$g_i^2(D, P) = o$; $h_i^2(D, P) = \phi$

Figure 4d.

The locked branch differs from the combination of a box and a two-way branch in that the elements of data relevant for $f_i$, $h_i^1$, $g_i^1$ are locked to all other processors during execution of the locked branch. In most cases, a single add-to-memory operation is sufficient (with a copy of the same left in the accumulator), lock-out can then be automatic by virtue of the memory access system. In more complex cases the page or segment containing the relevant items has to be marked as locked (not as private) before execution begins and is unlocked at completion. If it is found locked requests are repeated until access is granted. This is feasible because locked operations will only amount to a few instructions and the number of processors fighting for the data is limited by the number of physical processors.

Lock-out of co-owned private data by one of the owners to all others is likewise required for all elements of data relevant to a locked branch.

**Return connector**: $f_i'(D, P) = D$; $h_i^1(D, P) = P$

$g_i^1(D, P) = \phi(P)$; $h_i^2 = \phi$; $g_i^2 = o$

Figure 4e.
Flow diagram for a segment

Block diagram symbols for specialized segments.
Return address is saved as private data as are parameters and
by results, to permit simultaneous multiple execution of the
With nesting and recursion large amounts of private data
are stored temporarily.

\[ s_1^1 = s_1^2 = 0; \quad h_1^1 = h_1^2 = \emptyset \quad f_1(D, P) = P \]

Figure 4f.

\[
\begin{align*}
  f_1(D, P) &= D \\
h_1^1(D, P) &= P \\
g_1^1(D, P) &= \text{constant} = j_1 \\
g_1^2(D, P) &= \text{constant} = j_2
\end{align*}
\]

Figure 4g.

An essential feature of a fork is the creation of an initial set of
private data \( h_1^2(D, P) \) and a label \( j_2 \) for the new processor (this is
similar to Conway's state-word), with a request for scheduler assign-
ment of a physical processor.

**Basic Example**

A basic example is illustrated by Figure 5. Let A, B, C, and D be
independent segments and let the executions required from B and C be compatible.
A public variable \( T = 2 \), before the fork. The locked branch is an
entry at memory of -1 to T; the quit exit is taken unless the result is zero,
and case continuation is with D. This is the usual "join" operation,
the locked branch is a more powerful concept.
Software Control of Data Ownership

For each page of segment a set of data control bits have to be
instance as follows. With four physical processors, we use ten
bits in Fig. 6.* The interpretation of the data control bits is
the following example.

1. Free storage: 000000
   1111

2. Public data: 101111
   1111

3. Public data locked out by
   processor 3: 101111
   0010

4. Private data of yet
   unschedul ed pro ces sor:
   010000
   1111

5. Private data owned by
   processor 3: 000010
   1111

6. Private data co-owned by
   processor 3 and one or more
   yet unschedul ed processors:
   010010
   1111

7. Same as 6, locked out by
   processor 3: 010010
   0010

8. Private data co-owned by
   processors 1 and 3:
   001010
   1111

9. Same locked out by
   processor 3: 001010
   0010

* The lock-out bits can be replaced by bits of a binary number specifying
the locking processor with zero meaning no lock-out.
A basic example.

<table>
<thead>
<tr>
<th>B</th>
<th>S</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>L1</th>
<th>L2</th>
<th>L3</th>
<th>L4</th>
</tr>
</thead>
</table>

Control bits for data ownership.
Separate bits for private ownership and lock-out are necessary, and decisions on what pattern to restore at the end of the branch is lost. With a six bit set, if the locking processor stores the lock bits during locking, restoration of the initial pattern may want to changes in the ownership status of the segment and there is no reason then wait. A 7→9 transition occurs when the scheduler assigns a processor one with this segment on its initial private data list. A transition occurs if Proc. one releases the segment or quits. In the lock-out bits are then immaterial though processor 3 has no way testing, if he was not sole owner (Sole ownership cannot be changed the owners; see discussion below)

Allocation of Data Ownership

Programs begin with a single processor and all public data. The processor can create data private to itself, or of public data, it can operate on both kinds.

At the first fork, it specifies what initial private data the new processor will have (there may be none). To this effect:

a) It creates data which becomes private solely to the new processor.

b) It declares some of its own private data to be common to the processor and to itself.

"Allocation" activity (for lack of a better word) we call a "fork request."

Both processors can produce new forks until the data structure expands.

The scope of elements of data is delimited on one side by public, and fork declaration, and on the other side by the following erasing actions.

Erasure of public data (return to free storage) can be effected by the processor. That this should not cause conflicts is part of the compatibility requirement.

Release of private data owned solely by the processor will cause its owner. A processor can not always know, without a locked branch on the data bits, whether its ownership is sole. Once it has made the data common
another processor the future course is no longer under its control.
if ownership is sole at any time this fact can not be changed with- 
the processor's consent.

Release of private data which is common to other processors (which 
still be unscheduled) only lowers the corresponding data control bit.
the last-but-one processor releases we are back to sole ownership 
finally, when the last processor releases, to erasure.
When a processor quits it automatically releases all its private


Programmers Description of Data Control, Forks, Locked Branches
We modify Algol, to eliminate the block concept; declarations
and releases are explicit and at the whim of the programmer (not nested),
in and end serve only for statement parentheses.
Examples:

declar e public real x; integer array name (1:n)

later
release x

while the array stays on.

Similarly

declar e private y

later
release y

e or perhaps

quit

We use here \hat when we want to underscore the fact that a variable is private,
for exposition purposes.

fork <declaration part> <assignment part>
<priority part> to <designational expression>

For instance

fork declare private \hat{ } common \hat{x}; \hat{x} = \hat{y} + k -1;
priority 5 to lab;
is public, \( \hat{J} \) is known to the current processor but will not be known
new one, \( \hat{I} \) will be known to the new one only, \( \hat{J} \) will be known to both.
the priority level is for the scheduler, it could be an arithmetic
when scheduled the new processor will begin at label "lab,"
time the values of \( \hat{J} \) and \( \hat{J} \) may have changed, that of \( \hat{I} \) not.
At a locked branch

\[
\text{begin } \text{ lock } T; \quad T := T - 1; \quad \text{if } T > 0 \text{ quit else go to lab end}
\]

fork in the basic example is part of a subroutine which can be called
processors, the variable T must be declared private to the enter-
processor who then extends its ownership by a common declaration at the
This shows the necessity of distinguishing between privacy and lock-

---

**Example of Matrix Multiplication**

We presume the following declarations of public data:

- integer \( N \), real array \( A(1:N, 1:N) \), \( B(1:N, 1:N) \), \( C(1:N, 1:N) \)

A possible program is the following which is diagramed in Fig. 7.

\[
\text{declare public integer } T, I, J; T := N^2 + 1
\]

\[
I := 1 \text{ step 1 until } N \text{ do}
\]

\[
\text{for } J := 1 \text{ step 1 until } B \text{ do}
\]

\[
\text{fork declare private integer } \hat{I}, \hat{J}, \hat{L}; \hat{I} := I; \hat{J} := J;
\]

\[
\text{priority 2 to term;}
\]

\[
\text{test: begin lock } T; \quad T := T - 1;
\]

\[
\text{if } T > 0 \text{ quit else go to next end}
\]

\[
\text{term: declare private } \hat{K}; \quad C(\hat{I}, \hat{J}) := 0;
\]

\[
\text{for } \hat{K} := 1 \text{ step 1 until } N \text{ do}
\]

\[
c(\hat{I}, \hat{J}) := c(\hat{I}, \hat{J}) + A(\hat{I}, \hat{K}) \times B(\hat{K}, \hat{J});
\]

\[
\text{release } \hat{K}, \hat{I}, \hat{J};
\]

\[
\text{go to test;}
\]

\[
\text{next: - - - -}
\]

Note that we can just as well have the initial processor quit
conditionally after issuing the \( N^2 \) forks (set initially \( T := N^2 \)).
Flow diagram for matrix multiplication.
any storage positions for $i, j$ must be available? For $k < N^2 + 1$

now, if the scheduler queue system builds up to a maximum of
$k$ we need $k+q$ stores which could be close to $N^2$. To economize
storage one has to test queue length (a public read-only variable)
each fork. If $q$ is too large the processor queues itself with
state priority.


```fortran
if Q > 10 fork declare common all, priority 2 to next; quit
next: ...

```

declare common all" saves all private data (if any-one is involved
example) for the re-emergence of the processor. It is then up to
scheduler.


**Loops of the "While" Type**

Loops of the "while" type differ from the previous case in that the
variable can not be set at entry. Instead we start at one and index
each time the range of the loop is entered as illustrated in Fig. 8.

Fig shows a case where two side computations are done for each traversal
di range. If $B$ is traversed $n$ times, so are $C, D, G$ and $I$. After loop
1 and $J$ will be traversed once when all $n$ executions of $G$ and $I$,
respectively, have been accomplished. Box $K$ is executed once upon comple-
of both $H$ and $J$.


**Use of Locked Branches for Scope Termination**

It is important to release storage space of some large public
as soon as possible, we can use locked branches as shown in Fig. 10.

Possible coding is

```fortran
declare public M; T:=2;
= - - - -
fork declare common all, priority 2 to C;
```
A loop of the "while" type.

Another "while" loop.
b: - - - ; begin lock T; T:=T-1;

diff T=0 go to 1 else 2 end

1: release M; 2: - - -

c: - - - ; begin lock T; T:=T-1;

diff T = 0 go to 3 else 4 end

3: release M; 4: - - -

In private no locked branch is necessary, we simply release M and the data control bits take care of the rest.

Publication, Grabbing and Lock-Out of Data

A processor can publish any of its private data, this being equivalent to declaring a public copy before release; this can be abbreviated X.

A processor can grab public data by declaring a private copy of it before releasing (= erasing) the public data. Abbreviate grab X.

By first grabbing and later publishing data elements we obtain lock- in the ordinary sense, independently of the locked branch concept. It is believed that such lock-outs are rarely justifiable in programming since they tend to nullify the speed advantages of parallel processing.

Linking of Subprograms

A segment of the box type (defined previously) has the property for each entrance of a processor there will be one and only one exit corresponding processor. To analyze complex flow diagrams it helps to unify subprograms with a single entry and exit which have the same property. This condition is that the number of quits always balances the number of before emergence of the processor or of one of its descendants at exit. Just as for segments, these boxes can, in general, be subject to simultaneous executions - namely if a new processor enters before previous one exits.
The box concept can be generalized to the single entry multiple case. The condition is then that for every entry there will be an emergence at one and only one of the exits. Again all forks must be bypassed by quits before the emergence. Different boxes can call the subroutine, provided the subroutine is itself a box (no "side-effects").

**Execution Trees**

To check a program as to compatibility of all segments that can be executed in parallel, it is helpful to find all commutation conditions. These can be found by imagining execution with a single processor with a decision at each fork and each quit, diagrammed as a direct tree graph.

At a fork the number of sequences ready to run increases by unity. The number is shown by the exit degree of the corresponding tree node. At a quit the number decreases by one. Therefore successive tree nodes vary by ± 1 in their exit degree. The root (entry to program) has exit 1, while tips of the leaves (exit degree 0) are reached when all processes have quit. Each arc is a transformation of the data, or operator. Joining the string of operations for every path from root to a tip and equating these strings to be equivalent gives the commutation conditions. Each tree is in general dependent on the initial data, so that a family of trees (forest) has to be considered.

Boxing-in allows to go through this procedure in steps, beginning with the trees for the innermost, simplest boxes. At the next level, the commutation conditions having been checked, these boxes are equivalent to an operator.

**Section 6. Look-ahead**

Definitions of look-ahead programs:

**Definition 1:** The final exit can be reached with some processing still occurring.

**Definition 2:** The program can proceed to completion of its task without having to wait for execution of all the sequences it initiated.
Fig. 10 Use of a locked branch for scope termination.

Fig. 11 Subroutine for look ahead control.
Section 3: The entire program cannot be considered a single box.

A look-ahead sequence is one which may turn out unnecessary; it is followed by a fork and proceeds in "look-ahead status". At a later time the processor will discover whether it has become a necessary or unnecessary sequence. The fork is taken as soon as the data is available, necessity established or disproved at a later time.

The major problem is to communicate the change in status to the look-ahead processor (and its descendants) despite the impossibility of direct processor to processor communications.

The suggested solution uses boolean variables and a hardware feature. A private look-ahead bit LAH is associated with every processor (in its a word") and is normally zero. At a look-ahead fork this bit is set and a pointer is set to two bits LAH and UNN declared common to both of the fork. The subroutine in Fig. 11 is then prepared, partly in ware form. At the fork we set

\[ \text{LAH} = \text{LAH} = 1 \text{ UNN} = 0 \]

necessity is established, set \[ \text{LAH} = 0 \], when it is disproved set \[ \text{UNN} = 1 \].

The subroutine could be run automatically, as long as \[ \text{LAH} = 1 \], by the processor before assignment of a physical processor and then by that processor unt k instructions.

As long as \[ \text{LAH} = 1 \] all descendants of the processor are initiated in look-ahead status, with common ownership of LAH and UNN.

The subroutine action for \[ \text{UNN} = 1 \] can be a jump to the end of the look-ahead sequence.

As an example consider the "while" loop implementation in Fig. 12 where 2, 3, 4, 5, 6 designate boxed subroutines. Box 3 is only in some revolutions, and for which the data because test of predicate P decides its necessity. The method accounts for the case that on the n-th revolution an unnecessary look-sequence started on a previous revolution may still be running.

Note that the arrays indexed with \( C \) are always of small size, regard-

- of \( C \), because the lower \( C \)-values are progressively released. The value could be incremented modulo a sufficiently large integer if very large

lation numbers are expected.
Fig. 12: An example of look-ahead.
A simpler method is available

if P=1 wait for termination of box 3 at the "join"
if P=0 do not begin box 6 without checking (by a common
boolean variable) that box 3 has been stopped.

This way we prevent restart of the loop with old look-ahead sequences
running. It is felt that the complications of the first approach are
more time-wise, especially if the look-ahead-status checking subroutine
is programmed and therefore is called at longer intervals.

Sharing Processing Capability

Some operations which can be accomplished much more rapidly by
hardware than by subroutines do not occur quite often enough for
installation of this hardware in every processor. It seems worthwhile to
provide an appropriate number of these units, one for operation A, 2 for
frequent operation B, etc. For, say, 10 processors. Call it by
"requesters": if all special units are busy, transfer to the subroutine is
possibly delayed.

If the time difference (subroutine-hardware) is considerable, one
may queue requests up to a maximum length after which the subroutine is
possibly delayed.

Applications

An obvious class of applications for this type of multi-processing
is the integration of ordinary differential equation systems. Speed gains by
parallel execution proportional to the order of the system (up to the number of physical
processors) are possible and parallelism is easy to specify. Such problems
often occur under time pressure conditions (missile tracking). The superiority
of the present approach compared to DDA's or a global-type system is the
flexibility of compilation techniques with changes in problem size. The
hard-wired machines must fit the problem to a fixed equipment complement
in a hand-tailored manner.