PIPELINING PERFORMANCE
OF STRUCTURED NETWORKS

by

Fred M. Tonge
1. Introduction*

System architectures for interconnecting large numbers of processors are being studied widely [AH75, TH75]. Of particular interest in such architectures is the exploitation of concurrency among the processors. This concurrency can be either parallelism, in which different parts of a single data case are processed at the same time, or pipelining, in which different processes are carried out simultaneously on successive data cases. Many problems involving file and vector processing can be viewed as pipelining problems.

Both structured and arbitrarily interconnected networks of processors have been proposed. The asserted advantage of structured networks is that they are easier to comprehend, control, and so utilize effectively (arguments analogous to those made for structured programming [DA72, KO74]). The arguments for arbitrary networks are that they allow greater flexibility of interconnection and so can be more efficient in many cases.

Also, network architectures are being studied incorporating processor activation through data flow [AR77, AR78, DA77, DE74, DE75, SY77] and through flow of control [AH75, BR77, TH75].

In all of these cases, attention must be given to developing languages for expressing procedures and rules for allocating procedures (and their component parts) to processors so as to obtain maximum concurrency.

This paper presents a particular approach to specifying procedures (process descriptions) for networks and to allocating these process descriptions to processors.

*Work on structured process decomposition has involved Robert Barton and Richard Cowan of the Burroughs Corporation and Randall Flint of U.C. Irvine as well as the author, and all have contributed to developing the concepts presented here. This particular study was suggested by a conjecture of Barton that structuring networks need not degrade throughput. Cowan’s comments on an earlier draft greatly simplified the proof in section 7. This study was supported in part by the Burroughs Corporation and by National Science Foundation grant no. MCS77-02715.
The major result of this paper is a proof that, within our assumptions:

networks constructed using a small set of structured process connectives can achieve at least as good throughput (pipelining performance) as arbitrarily interconnected networks.

2. Structured Process Description

The approach to process description of interest here involves successive structured decomposition of processes, with the resulting process network being mapped (allocated) onto interconnected processors [C067,T076]. These structured process descriptions represent parallelism directly, and can be implemented so as to enhance pipelining. Although control-flow implementations are possible, it is natural to view such processor networks as data-driven.

This approach includes a base language in which atomic processes are described, a set of inter-process connectives, and a philosophy for the allocation of processes to a network of processors. We envision that the base language, inter-process connectives and allocation scheme would be implemented in each processor, and so would be executed as "machine language". Under these conditions, process descriptions would be realized as programs almost directly, subject only to the level of translation done in simple assemblers. In this paper we concentrate on the inter-process connection aspect of the system, and do not develop the details of any particular base language, processor network, and accompanying allocation philosophy.

A process description may be an atomic description (e.g., expressed in base language), or may be decomposed into two component process descriptions joined by a serial (S), parallel (P), or alternative (X) connective (figure 1).

(Sab)  (Pab)  (Xab)

Figure 1.
In serial decomposition, process \( a \) is followed by process \( b \). In parallel decomposition, processes \( a \) and \( b \) are independent, each with a copy of the input data set and with an output data set formed as a union of their individual outputs. The two processes may be carried out in either order, or in parallel if sufficient resources are available. In alternative decomposition, either process \( a \) or process \( b \), but not both, is performed as selected by a predicate \( q \) on the input data set. These connectives are assumed order-preserving; that is, successive data cases remain in their input order on output.

Thus, a process description network (which is itself a process) can be viewed as arising from the interconnection of atomic process descriptions using these connectives, or from the repeated decomposition of a single "top-level" process description into pairs of interconnected components, down to the atomic process level.

The structured process description networks so derived are equivalent to Dijkstra's d-charts [DA72,KO74] with the addition of parallelism and the omission of cycles. In structured process decomposition, the equivalent of cycles is achieved by the use of labels on process descriptions and the notational connective \( I \). A process description of the form \( (I \text{ label}) \) is replaced at the time of allocation by a copy of the process description named by that label. Since allocation is flow-driven (that is, the components of an allocated process description are themselves allocated only with the first invocation of the process), recursion is limited by the particular data sequence at hand. The process description equivalent of the D-chart cycle is given in figure 2. (\( \text{idem} \) is the identity process.)

\[\text{proc:}\]

\[\begin{array}{c}
\text{idem} \\
\text{q} \\
\text{a}
\end{array}\]

\[\begin{array}{c}
\text{a} \\
\text{I proc}
\end{array}\]

\[\text{\textit{D-chart cycle}}\]

\[\text{process description equivalent}\]

\[\text{Figure 2.}\]
As an example, a process description network is given in figure 3 for computing the square root \( s \) of a number \( n \) using \( t \) iterations of the Newton-Raphson approximation method. An informal pseudo-text is used in defining atomic processes, which are assumed to consist of unconditional expressions on input values producing output values.

\[
\begin{align*}
\text{Figure 3.}
\end{align*}
\]

This network can also be described by a process expression as follows (here using atomic process labels in place of the actual expressions):

\[
(S \triangleq (X (S \circ i : (S \circ (X (S (P \circ e) \circ f : (I \circ j)) \circ g)))) \circ h))
\]

\section{Preliminary Discussion}

In the following, we consider networks involving processes connected sequentially, in parallel, or with choice among components. Networks constructed using only the inter-process connectives given above are called \textit{structured} networks. Networks constructed with greater freedom of interconnection (see below) are called \textit{arbitrary} networks. Both structured and arbitrary networks are acyclic. The equivalent of cycles is achieved using the include connective.

More precisely, we define five primitive interconnections from which both structured and arbitrary networks are formed. These are illustrated in figure 4.
1. **sequential** interconnection, a single output to input line.

2. **fork** interconnection, a single line splitting to two lines. In a data flow interpretation, a data case arriving at the fork would be copied, and copies sent along both lines.

3. **join** interconnection, with two lines joining to form a single line. In a data flow interpretation, data cases from the two lines would be combined into a single data case. (Either buffering could be provided at the join to gather the matching data cases or neither preceding process could release its output data until both data cases were ready.)

4. **branch** interconnection, a single line selected to one of two lines depending upon some predicate on the data case. (The value of the predicate is computed in an earlier process, so that the predicate switch itself need only examine a single boolean value.)

5. **merge** interconnection, output from one of two lines switched to a single line. The merge acts an arbiter to insure that data cases on the two lines are not intermixed.

In structured networks, fork and join interconnections occur only in matching pairs, as do branch and merge interconnections. In arbitrary networks these interconnections can occur as desired (within the assumptions given below).

![Figure 4](attachment:image_url)
The following assumptions hold for both structured and arbitrary networks:

1. As many processors are available as are needed to decompose the process description completely.

2. Communication and switching times are negligible. Performance is determined by processing time and by time waiting for computation of preceding processes.

3. Predicate switching time is negligible. (That is, predicates are evaluated as part of an earlier process, and switching time at a branch is negligible.)

4. The data values needed as inputs for one invocation of a process or produced as outputs from one invocation, here called a data case, are transmitted together as a single message.

5. Networks are acyclic.

6. Networks are "well-formed", in the sense that any data case entering a network will eventually complete processing and no parts or copies of the data case will be left in the network [G072, LA77].

7. Networks are order-preserving, with output data cases produced in the same order that input data cases are accepted. For structured networks, this is assured since the individual connectives are order-preserving. For arbitrary networks, we assume (if needed) an instantaneous ordering process appended to the network.

4. Performance measures

The following deterministic measures of process characteristics are of interest:

\[ e(i) \]: elapsed time of process \( i \) from acceptance of input data case to production of corresponding output data case. For atomic processes, this is the actual processing time.

\[ c(i) \]: capacity of process \( i \) (number of input data cases that can be accepted before production of an output data case).
p(i): average period of process i (time between successive output cases at maximum pipelining) at steady state (assuming sufficient input).

The concept of period (the inverse of throughput) is basic to this study of pipelining performance. The period of a process depends on its input as well as on the process itself. If input arrives too slowly to keep the process active, then the input period determines the process output period. If input arrives more frequently than the process can handle, process characteristics determine the output period. In either case, the process may alter the relative timing of data cases within the data sequence.

A regular input data sequence is one in which a pattern of inputs reoccurs over time. One occurrence of this pattern is called a pattern segment. For networks involving predicates, different data cases may follow different paths through the network depending on the particular data values. In that case, we consider the period of the network to be the average period over the data cases of a pattern segment (one repetition of the pattern).

As an example, consider the arbitrary network of figure 5 (the numbers given above each process are processing times), and consider an input sequence with a three data case pattern segment as follows (data cases here being characterized by their predicate values):

```
A (q\land r) (\neg q) (q\land r) (q\land r) (\neg q) ...
```

![Figure 5](image)

A simulation of the above input data sequence through the network of figure 5 is given in figure 6, assuming all data cases are available as needed. The period of the network for this pattern segment is 5/3 time units per data case (average). That is, a 3-data-case segment completes every 5 time units.
From the above definitions, performance measures for the structured connectives can be derived.

atomic process:  
\[ e(i) = \text{atomic processing time} \]
\[ c(i) = 1 \quad \text{(by assumption)} \]
\[ p(i) = \frac{e(i)}{c(i)} = e(i) \]

serial, \((Sab)\):  
\[ e(S) = e(a) + e(b) \]
\[ c(S) = c(a) + c(b) \]
\[ p(S) = \max(p(a), p(b)) \]

parallel, \((Pab)\):  
\[ e(P) = \max(e(a), e(b)) \]
\[ c(P) = \min(c(a), c(b)) \]
\[ p(P) = \max(p(a), p(b), \frac{e(P)}{c(P)}) \]

These performance measures are directly extendable to connective definitions involving more than two components.

Performance characteristics of the alternative connective depend on the sequence of predicate values selected by successive data cases, and so can be analyzed only for specific situations. For regular data sequences a period for the pattern segment as a whole may be found through analysis of the specific pattern. One particular type of irregular sequence, in which one value of the predicate occurs very rarely (as in an error condition or exception-handling) may be approximated by considering the dominant value only.

The include connective is merely notational, and for performance analysis purposes must be replaced by the included description. (This is identical to the execution graph of Arvind and Gostelow [AR77, AR78].)

For one specific use of the alternative connective, performance measures may be defined across all data patterns.
Define the path of a data case through a network to be the sequence of processes through which the data case moves in the network. For networks without predicates, all data cases have the same path. For networks with predicates, the path of a data case is characterized by a logical expression of predicate values.

Define the load, \( w(i) \), on process \( i \) with respect to a particular pattern as follows:

\[
w(i) = \text{the ratio of data cases in the pattern segment whose path includes process } i \text{ to the total number of data cases in the segment.}
\]

Then, for the alternative interconnection of process \( a \) and the identity process (an atomic process with \( e(\text{idem})=0 \) and \( c(\text{idem})=1 \)), the period may be defined as:

\[
\text{alternative,}(X_\text{a idem}) = p(X) = p(a) * w(a)
\]

Since the period is here an average for a pattern segment, the relative timing of input data cases within the segment must be such as to provide inputs as needed. Note that the general definition of period holds only where the input period does not limit throughput.

5. Equivalent Networks

we can partition the atomic processes of a network into two sets: those which compute at most only the values of predicates used within the network, and all other. We call the latter active processes.

A network \( k \) is equivalent to a network \( j \) if:

- for every data case, the active processes in the path for network \( k \) are the active processes in the path for network \( j \);
- for every data case, the ordering of active processes in the path for network \( k \) is consistent with the ordering of those processes in network \( j \);
- the predicates of networks \( k \) and \( j \) are composed from the same set of logical variables; and
- for every data case, the output for network \( k \) is exactly that for network \( j \).

For example, the two structured networks of figure 7 are equivalent to the arbitrary network of figure 5, and to each other.
The period of any structured network without predicates can be reduced to that of the maximum of atomic component periods using the following observation:

For serial connectives, the period is already that of the maximum component.

For parallel connectives, the period is defined as the maximum of component periods or (larger component elapsed time)/(smaller component capacity). In the latter case, construct an equivalent network by inserting sufficient buffer capacity (identity processes) in series with the smaller capacity component to equalize the capacity of the two parallel processes. Then, again, the maximum component period becomes the period of the connective. (Figure 8 illustrates such an enhancement, reducing the network period from 4 to 3.)

A straightforward procedure for decreasing the period of any...
structured network without predicates to the minimum attainable follows from the above observation. (The introduction of buffer processes to increase pipelining throughput has been noted previously [PA76].)

The introduction of buffer processes (additional capacity) to decrease the period and so increase throughput of a parallel process description is a specific example of a more general principle. In any network in which the outputs of several processes must be coordinated (as the two output data cases of parallel components must be combined into a single output case), a delay could be introduced into the flow of data cases, possibly resulting in forced idle (waiting) time for subsequent processes. If the process whose period determines overall throughput is forced to wait for input, such enforced idleness would increase the overall period. More generally, any process which alters the time pattern of input data cases in producing output data cases may introduce idle time for the maximum period process among its successors. And in general, buffers may be introduced to eliminate that idleness.

While networks without predicates produce a regularly spaced output pattern (assuming the input period is less than that of the network), predicate networks can introduce irregular respacings into the time pattern of outputs. It is the need to insure that respacings in a predicate network do not cause subsequent idleness of a "bottleneck" (maximum period) process which motivates the following lemma.

Decoupling Lemma:

Any process $x$ may be replaced by an equivalent process $y$ (with possibly greater capacity) which preserves both the period of the original process and the timing of its input sequence.

Proof:

Process $y$ consists of the serial connection of process $x$ with a buffer process of sufficient capacity. The buffer process collects (at most) all data cases in a segment of the output sequence of $x$ and then releases them with the time spacing of the input sequence. (Again, this assumes that the input period is less than or equal to that of process $x$.)

In practice it is usually not desired to reproduce the input sequence timing exactly, but rather to provide enough buffering to guarantee that subsequent processes can operate at maximum throughput.

One interpretation of the lemma is that, if all data cases in an input sequence segment are available "at the same time", then all output can also be made available to the
structured network without predicates to the minimum attainable follows from the above observation. (The introduction of buffer processes to increase pipelining throughput has been noted previously [PA76].)

The introduction of buffer processes (additional capacity) to decrease the period and so increase throughput of a parallel process description is a specific example of a more general principle. In any network in which the outputs of several processes must be coordinated (as the two output data cases of parallel components must be combined into a single output case), a delay could be introduced into the flow of data cases, possibly resulting in forced idle (waiting) time for subsequent processes. If the process whose period determines overall throughput is forced to wait for input, such enforced idleness would increase the overall period. More generally, any process which alters the time pattern of input data cases in producing output data cases may introduce idle time for the maximum period process among its successors. And in general, buffers may be introduced to eliminate that idleness.

While networks without predicates produce a regularly spaced output pattern (assuming the input period is less than that of the network), predicate networks can introduce irregular respacings into the time pattern of outputs. It is the need to insure that respacings in a predicate network do not cause subsequent idleness of a "bottleneck" (maximum period) process which motivates the following lemma.

Decoupling Lemma:

Any process \( x \) may be replaced by an equivalent process \( y \) (with possibly greater capacity) which preserves both the period of the original process and the timing of its input sequence.

Proof:

Process \( y \) consists of the serial connection of process \( x \) with a buffer process of sufficient capacity. The buffer process collects (at most) all data cases in a segment of the output sequence of \( x \) and then releases them with the time spacing of the input sequence. (Again, this assumes that the input period is less than or equal to that of process \( x \).)

In practice it is usually not desired to reproduce the input sequence timing exactly, but rather to provide enough buffering to guarantee that subsequent processes can operate at maximum throughput.

One interpretation of the lemma is that, if all data cases in an input sequence segment are available "at the same time", then all output can also be made available to the
next process in series "at the same time" without increasing
the period.

(The introduction of buffers suggested by the decoupling
lemma is only one means of altering a process description
network to achieve maximum throughput. The addition of
buffers to increase the capacity of one of two parallel
processes is another. Similarly, in some cases buffers can
be added to one of two processes in alternation to decrease
their overall period. (In order-preserving networks, the
outputs of alternative processes must be coordinated in a
way similar to those of parallel processes.) Also, the
period of a bottleneck process may be reduced by one-half by
alternating (duplexing) two copies of the original process.
The relative desirability of these approaches to increasing
throughput depends in part on the physical characteristics
of the processors on which the process network is
implemented. If processors generally have large amounts of
storage, then introducing buffers may have little real cost.
If processors generally have small amounts of storage, then
duplexing processes may be desirable.)

7. Performance of Equivalent Networks

We consider here the elapsed time and period (throughput)
performance of equivalent arbitrary and structured networks.

Well-known Result:

There exist arbitrary networks for which no
equivalent structured network has an equal or smaller
elapsed time.

Example:

For the network of atomic processes given in figure
9, there are twelve possible equivalent structured
networks without additional processes, of which four
are shown in figure 10. (Of the remaining eight,
four are variations on the serial order of case iv,
and four are derived from case iii by serializing one
of the two parallel sections.) As can be seen, no
equivalent structured network has an elapsed time as
small as that of the arbitrary network.
Figure 9.

Case 1.

Case ii.

Case iii.

Case iv.

Figure 10.
Theorem:

For any arbitrary network, there exists for any regular input sequence an equivalent structured network with a period at least as small as that of the arbitrary network.

Proof:

1. The period of a network must be at least as great as the maximum of the products of the periods of its component processes multiplied by their loads in the regular pattern segment.

\[ p(\text{arb}) \geq \max(p(i)w(i); i \in \text{arb}) \]

2. Since the network contains no cycles, component processes can be numbered with consecutive integers starting with 1 such that any process is assigned a higher number than any of its predecessors. (This is a common technique in critical path algorithms, for example [KE61].)

3. Since the network contains no cycles, there are only a finite number of paths from the beginning to each process, and those paths can be traced. (Techniques for such path tracing are common in path analysis [AL76, F076].) A predicate expression consisting of the intersection of those predicates in the arbitrary network which must be true for a given path can be associated with that path. A predicate expression, q(i), which is the union of the expressions for all paths leading to atomic process u(i) can be associated with that process.

4. Each atomic process u(i) may be replaced by an equivalent process v(i), where

\[ v(i) = (S z(i) (X u(i) (iden))) \]

and z(i) is a process for computing q(i).

5. The processes (numbered, say, 1 through m) can now be organized into an equivalent structured network of the form:

\[ (S v(1) (S v(2) ... (S v(m-1) v(m)) ... ) ) \]

The numbering procedure guarantees that the ordering of the structured network is consistent with that of the arbitrary network. The predicate of the alternative connective containing process u(i) is that predicate derived
from all paths to \( u(i) \) in the arbitrary network.

6. By the decoupling lemma, sufficient buffering can be added to each alternative, treated as a process, to ensure that its outputs occur in a time pattern which will not force waiting in later processes. Then, the period of the structured network will be the larger of the periods of the first component or the remaining network, or, recursively:

\[
p_{\text{str}} = \max(p(1) \times w(1), \max(p(2) \times w(2), \ldots))
\]

by noting that the equivalent structured network as defined above can generally be simplified as follows:

- If \( q(i) \) is true, then \( v(i) = u(i) \);
- If \( q(i) \) is a predicate of the original network or its negation, then \( z(i) \) can be omitted;
- Any remaining \( z(i) \)'s need only be computed once.

Figure 11 shows a minimum period network equivalent to the network of figure 5.

![Diagram](image)

Figure 11.

In most cases, equivalent structured predicate networks exist with processes on both branches of many alternatives, and often requiring fewer buffers for maximum throughput. Such an equivalent network for the network of figure 11 is given in figure 12. (The two additional buffers are required to maintain throughput while preserving order.) This network has a period for the example pattern segment of 4/3 time units per data case, and thus achieves better throughput (smaller period) than the arbitrary network of figure 5.
From the standpoint of pipelining throughput, the only reduction in performance incurred by expressing predicate process descriptions in a structured manner need be that associated with computing additional compound predicates from already computed predicates. And only in those cases where the computation of one $A$ or $v$ exceeds that of the bottleneck (maximum period) process will it affect throughput. (These compound predicates are associated with the occurrence in the arbitrary network of merges not paired with branches, and are reminiscent of the additional control variables found necessary by Bonn and Jacopini [Bo66].)

(Note that while the organization of processes in the structured network is independent of the input pattern, the number and placement of buffers so as to achieve maximum throughput varies with the particular input pattern. Upper limits on buffer requirements for a network may be derived by assuming the "worst possible" pattern for each alternative, or, for a particular input pattern, by static analysis of the alternatives. In almost all cases, these limits will be too severe.)

9. Concluding Remarks

The conclusion drawn from this analysis is that there need be no reduction in throughput (pipelining performance) from constructing process description networks using only a small set of structured process interconnections. Indeed, in some cases better pipelining performance is achieved.

While some of the assumptions underlying the analysis may seem rather distant from physical reality, they do not necessarily affect the conclusion. Consider the assumption of no communication delays. For this assumption to alter the conclusion, one would have to argue that communication delays will be greater in structured than in arbitrary networks. Since the magnitude of communication delays is closely related to the scheme for allocating processes to processing elements (see, for example, [M78]), this argument would imply that a mechanized allocation process can more efficiently allocate an arbitrarily interconnected network of processes than a structured one. This
implication seems contrary both to intuition and to experience with the management of data structures. Further, from a pipelining standpoint, interprocess communication channels are equivalent to buffer processes. Only if their "period" is greater than that of the maximum process do they affect throughput. (This is not to say that communication delays have no effect on performance, or that various inter-processor connection schemes should not be studied, but only that there is no reason to believe that structured networks would suffer disproportionately from those delays.)

Similar arguments can be made concerning the assumption of sufficiently many processors. Again, at least an informal analysis suggests that whatever schemes are developed for reallocating processors as they are needed are unlikely to be less efficient for structured (well-organized and consistent) process descriptions.

Another area for further research is that of dynamic introduction of buffers for performance enhancement. A dynamic process based on ongoing performance would seem both feasible and desirable. Such tuning would most likely also improve throughput in those cases where a regular input pattern "almost" exists. And such a procedure would not be dependent on advance knowledge of the data sequence. Although appropriate introduction of buffers can also enhance the throughput of arbitrary networks, it would seem likely that the dynamic analysis of structured networks, like their static analysis, would be more straightforward.

Finally, we have refrained from presenting some obvious algorithms for converting arbitrary process descriptions into equivalent structured descriptions. The implication of this paper is that process descriptions should be thought about and constructed in a structured manner from the beginning.
References


