A Hybrid Approach for Efficient Dataflow Computing

Yang-Chang Hong T. H. Payne
Department of Mathematics and Computer Science
University of California, Riverside, CA 92521

ABSTRACT
A hybrid graph model for dataflow computation is presented that integrates the concepts of "tagged-token" graphs with those of "static" graphs. This model is based on partitioning the tagged-token graph into highly connected pieces that become a basic unit for activation and execution. A piece is activated like a tagged-token node. Execution of a piece consists of the firing of nodes within the piece according to static firing rules. An individual piece supports multiple concurrent activations by propagating successive wavefronts of tokens through the piece in a pipeline fashion. Pieces are restricted to be acyclic, so that they can be pipelined. This hybrid model was developed to investigate the interaction of static and tagged-token dataflow computation. Hybrid computation is presented from the derivation of the model, through a discussion of the problems of pipelining certain kinds of graph nodes (e.g., merging operators and function invocators), to a logical description of a data-driven processor array implementing the model.

1. INTRODUCTION
A dataflow graph is a directed graph with nodes representing functional operators and arcs specifying the data dependencies among those operators. A dataflow graph executes by firing nodes; a node fires whenever it is enabled. Several graph models of dataflow computation have been proposed [1-3, 7, 8, 19, 20, 22, 23]. A survey of these can be found in [7].

At least two methods have been proposed for ensuring the determinate execution of dataflow graphs. In the static method proposed by J. Dennis [8, 9], an acknowledge arc is added from each node to its producer nodes. These acknowledge arcs ensure that no arc will ever contain more than one token and that the selection of tokens is completely determined. In the tagged-token method proposed by Arvind [2, 3], the same node descriptions are shared by different activations of the graph. To avoid the mixing of tokens belonging to separate activations, different tags are attached to tokens that belong to different activations.

An individual static graph supports multiple concurrent activations by propagating successive wavefronts of tokens through the graph in a pipeline fashion. Activations of a graph must proceed in the order of initiation. Consequently, static graphs do not support recursion. Also, activations cannot pipeline any cyclic portion (loop) of a graph. Only one activation is allowed inside a loop at a time. When one activation exits from the the loop, it allows the next one to enter, thereby ensuring that activations exit from the loop in the same order as they entered. There is no way to guarantee that this order be maintained at the output of the loop if more than one concurrent activation is allowed, since not every activation takes the same number of iteration cycles. Maintaining this order is necessary when the results from the loop must be synchronized with those from the other portion of the graph. Furthermore, if a particular activation of a graph requires a long time to compute, i.e., it is a slow activation, then subsequent activations will be slowed down thereby degrading the performance of the pipeline.

In the tagged-token model, activations need not proceed in order -- they can be interwoven thereby supporting recursion. This feature guarantees that slow activations do not degrade the performance of the system. Many concurrent activations are allowed inside the loop, and each can achieve its inherent parallelism during the execution of the loop. To implement this model, processors must have some form of associative memory for the matching of tokens on the basis of their tags at every action (node). Associative matching is an expensive operation in terms both of hardware and of time.

The two models, static and tagged-token, described above are the extremes where there are no token-matching and where full matching is required, respectively. The major advantages of tagged-token graphs over static graphs occur in the areas of recursion and iteration. This paper shows how these advantages can be achieved without going to full matching of tokens, and how these can be efficiently implemented on a data-driven processor array.

2. THE HYBRID MODEL
The hybrid model proposed here is based on partitioning of tagged-token graphs into highly connected pieces. This partitioning is feasible because of the locality of effect and freedom from side effects, which are well-known properties of dataflow programs.

The pieces of a partition are the basic unit for activation and execution. Tokens carry tags for the enabling of pieces. A piece is activated when all of its input nodes -- those that receive data from other pieces -- have received external input with the same tag. A piece is activated dynamically, like a tagged-token node of Arvind [2, 3]. Upon activation (i.e., firing), the set of input tokens are propagated as a wavefront through the piece according to Dennis' firing rules [8, 9]. An individual piece supports multiple concurrent activations by pipelining these wavefronts. Pieces are restricted to be acyclic, so that they can be pipelined.

Following the Dennis model, each node within a piece reports the consumption of each token to its producer, informing the producer that it may place a new token on the corresponding output arc. Output nodes -- those that deliver
data to other pieces -- need not be informed of the consumption of previously produced tokens in order to fire. But, input nodes of a piece report to an "imaginary" producer for the piece, called the piece's monitor. The monitor submits a new activation to the piece whenever it has a complete matched set of input tokens and has received all the required acknowledgements from the piece's input nodes. Note that the activations of a piece need not be executed in order of arrival (i.e., completion of their matched sets of initial tokens). But, once executing, they must proceed in a pipeline fashion, exactly as in the Dennis model. We are implicitly assuming unbounded buffering capacity for the monitors, as is commonly done for nodes in the tagged-token model.

When pipelining of a piece is not required, one can reduce acknowledgement overhead by not having each consumer node acknowledge its producer node. To submit a new activation to such a piece, one must ensure that all tokens of a previous activation have left the piece [21].

A pure graph node is enabled when there is a token on each of its input arcs and fires by removing one token from each input arc and placing one (additional) token on each output arc. Pieces may also contain nodes for achieving the selective routing of tokens required in conditional and iterative subgraphs. A distributor has a distinguished input arc called its control that receives a Boolean value, $T$ or $F$. The value at the control determines whether a copy of the input token is placed on the $T$ or on $F$ output arc. A merger has two inputs and one output. It fires whenever it receives a token on either input arc. When it fires, it simply copies the input token to its output arc. If it receives two tokens, one on each input arc, each token fires the merger in turn.

We form conditional and iterative subgraphs in the usual well-structured manner similar to those in [8] (see Figure 1). We do not allow any other occurrences of mergers and distributors. The $D$ node increments the iteration number in the token tag (see Section 4.2) by one, and the $D^\$ resets this number to zero.

### 2.1 Pipelining Mergers

Assume that $X$ and $Y$ are wavefronts of tokens corresponding to two activations of some piece. Say that $X$ precedes $Y$ in execution. To ensure the correctness of their results, these two wavefronts must not get interwoven. Otherwise, subsequent computations could involve a mixture of tokens from $X$ and $Y$. Specifically, such interweaving could occur at a merger node, since a token from $Y$ could arrive at one input before a token from $X$ arrived at the other, thus allowing a portion of $Y$ to get ahead of the corresponding portion of $X$. Note that it would be alright for $Y$ to completely overtake $X$, as long as it did so uniformly, across the entire wavefront.

For example, a piece, say $G$, containing a conditional subgraph has a potential race condition -- it may happen that the portion of $X$ routed through the conditional is routed through the "true" part, while the corresponding portion of $Y$ is routed through the "false" part that happens to be shorter, thus allowing this portion of $Y$ to overtake the corresponding portion of $X$. Obviously, the output of the conditional subgraph must be re-synchronized with the rest of the wavefront. To do this we replace the bank of mergers by a monitor, whose input ports have fan-in greater than one, and put all consumers of the mergers, their consumers, their consumers'
consumers, etc. into a new piece $G''$. The monitor for $G''$ will do the requisite merging and will enforce the necessary synchronization to re-form the wavefront and prevent mixed input at the downstream nodes in $G''$. The rest of $G$ is put into another piece, $G'$. See Figure 2.

Note that the races incurred in a piece containing a conditional subgraph can also be overcome by replacing the set of mergers by a set of selectors (i.e., mergers with a control input) with their control input controlled by the predicate that controls the set of distributors of the subgraph. This approach has been adopted by J. Dennis [8]. These control arcs limit pipelining of the piece, but this effect can be overcome by balancing token flow with the addition of identity operators or token buffers. A detailed description of this approach can be found in [5]. This mechanism can be used to prevent overly fine partitioning that can result from the previous approach but does not generalize to iterative subgraphs.

Figure 3. (a) A Piece Containing an Iteration Subgraph; and (b) a Schema for (a).

A similar problem arises in the case of iterative subgraphs. If the iterated portion of $X$ takes more cycles than the corresponding portion of $Y$, then the result of the loop on that portion of $Y$ may be produced first, out of order. This problem can be overcome by putting all consumers of the loop, their consumers, their consumers’ consumers, etc., into a new piece $G''$, as was done for the conditional case above. But, additional synchronization is required at the top of the loop to prevent interweaving of individual iterations. So we replace the bank of mergers by a monitor, and put the rest of the loop except its feed-back arc into another piece, $G'''$. The monitor for $G'''$ synchronizes the wavefront for each iteration. Note that the input ports of the monitor for this piece have fan-in two: a loop-input arc and a feed-back arc. See Figure 3.

2.2 Pipelining Function Invocators

An apply operator is the dataflow analog of a procedure call. It takes as input a description of a graph $H$ to be invoked and a set of input arguments for $H$. Upon firing, it consumes the input arguments and places them on the input arcs of $H$. Eventually, $H$ sends its computed results to an apply$^{-1}$ operator where it are treated as the output tokens of the apply node, but the firing of the apply node is complete as soon as it has submitted its tokens to $H$.

Consider a piece, say $G$, containing a single pair of apply and apply$^{-1}$ nodes. The apply$^{-1}$ node must be put into a piece all by itself. It needs to be monitored, since $H$ may consist of many pieces, and hence its output tokens may not be received by the apply$^{-1}$ node in coherent wavefronts. Also, since a given wavefront in $G$ may generate several concurrent invocations of $H$, it is necessary for the apply node to equip each argument set with fresh context tags, different from those of the calling context. The apply$^{-1}$ node restores the context of the outputs to that of the caller. These outputs will then need to be resynchronized with the rest of the calling wavefront.

Figure 4. (a) A Piece Containing a Pair of apply and apply$^{-1}$; and (b) a Schema for (a).

Let $G''$ be the set of (direct and indirect) consumer nodes of the apply$^{-1}$ node. Note that when a wavefront reaches the apply node, it hangs up at the boundary of $G''$ until the corresponding firing of the apply$^{-1}$. So, the wavefront corresponding to a recursive activation of $G$ would hang up when it reached a producer for a node in $G''$. Further recursive invocations would hang up at producers for producers for $G''$, etc., until the graph eventually deadlocks or the wavefront of a recursive invocation is routed around the blockage and reaches the outputs of $G$. At that point, the calling wavefront can proceed, if it is not blocked by one
of its recursion predecessors. But, it is unlikely that it would be on a route different from its successor and all of its predecessors, except in special cases, for instance, if \( G'' \) had no external input other than the \( \text{apply}^{-1} \) node.

To support recursion, we make \( G'' \) a separate piece, thus, equipping it with a monitor that can stack up wavefronts at its boundary. The rest of \( G \) is put into \( G' \). See Figure 4. Wavefronts corresponding to recursive calls to \( G \) stackup at the monitor for \( G'' \), waiting for the corresponding firing of the \( \text{apply}^{-1} \) node. Eventually one of them is routed around the \( \text{apply} \) node and reaches the outputs of \( G \). The \( \text{apply}^{-1} \) then fires, allowing the calling wavefront to proceed. When that wavefront completes, it fires the \( \text{apply}^{-1} \), allowing its predecessor to proceed, etc. These stacked-up wavefronts complete in LIFO order, just the way that recursive invocations of procedures return in a von Neumann architecture.

The \( \text{apply}^{-1} \) node itself cannot be placed in \( G'' \) since it receives its inputs with tags that have been modified by the \( \text{apply} \) node and that differ from those of the calling wavefront, which will be the context of the rest of the inputs to \( G'' \).

![Figure 5. Graph for \( \sum_{i=1}^{m} F(i) \).](image)

Not only do wavefronts corresponding to recursive calls hangup at the \( \text{apply}^{-1} \) node, other wavefronts tend to be delayed there as well, since execution of the invoked graph tends to be a slow operation. The structure just described allows wavefronts that are routed around the \( \text{apply} \) node to complete without getting slowed down behind by previous wavefronts that are waiting at the \( \text{apply}^{-1} \) node. So this structure enhances pipeline performance, as well as supporting recursion.

### 2.3 Partitioning Anomalies

Partitioning can lead to various anomalies including deadlock, parallelism reduction, and performance degradation. Deadlock can occur in a partitioned acyclic graph. Consider, for example, a graph consisting of a chain of four single-input-single-output nodes. One partition of this graph consists of one piece containing the first and last nodes of the chain and the other containing the two middle nodes. A deadlock will ensue with the resulting graph, since each piece gets one input from the other. Any partition of an acyclic graph which leads to a cyclic graph of "pure" pieces is a deadlock. One easy way to avoid deadlocks, followed in [10-13], is to insist that every piece have either a single input or a single output. This prevents circular dependencies from arising in acyclic graphs.

Reduction of parallelism can occur in partitioning loops. Consider the graph to compute \( \sum_{i=1}^{m} F(i) \) shown in Figure 5. The loop in this graph computes in parallel the recurrence equations:

\[
i_{i+1} = i_i + 1
\]

\[
\sum_{i=1}^{n} F(i)
\]

The \( i \) circulates and initiates \( n \) distinct activations of \( F \). The \( \sum \) circulates, accumulating partial sums.

In Figure 5, if the entire graph except the feedback arcs is one piece, there is only one activation of \( F \) per iteration, since a new iteration of the entire piece cannot be initiated before the previous one has concluded. Such partitioning is simple but not very attractive as far as parallelism is concerned.

If \( F \) takes a long time to compute, \( i \) will circulate much faster than \( \sum \) thus invoking many concurrent activations of \( F \). To avoid losing concurrency through partitioning, one may partition the acyclic portion of the loop into two pieces, one for the acyclic portion of each recurrence equation, with the predicate subgraph being part of the piece associated with \( i \). In this case, many distinct activations of \( F \) will execute in a pipeline fashion. The level of concurrency is determined by the number of activations that can pipeline through \( F \) at a time. Such a partitioning, however, requires carefully specified firing rules for the resulting pieces. A study of such firing rules can be found in Landry [18].

Other anomalies can occur in terms of performance degradation. Consider a piece consisting of two parallel chains of single-input-single-output nodes. Suppose that one chain is much longer than the other and that the shorter one is on the critical path. Such a piece would effectively lengthen the critical path by the difference in the lengths of the two chains -- performance could degrade significantly.

### 3. ARCHITECTURAL DESIGN

This hybrid model requires architectural support for determining when a piece is activated and for executing...
activated pieces. This section describes such a multiprocessor architecture. It consists of a set of processors connected through a communication network. Each processor has a single memory element (ME) and a single processing element (PE). See Figure 6. The graph of pieces is mapped onto this architecture. The monitor for a given piece is mapped to an ME, and a subset of that piece is mapped to the corresponding PE.

Pieces of a program are activated by MEs. Once activated, a set of tokens is sent in packet form to the corresponding PE for execution. We store the nodes for the ME, and the rest of that piece is mapped to the corresponding PE.

Intermediates results within a piece stay in the piece's assigned PE, thereby decreasing traffic from the ME to the PE. The monitor for a given piece is mapped to an assigned PE, thereby decreasing traffic form the ME to the PE.

As in the dataflow multiprocessor proposed by Arvind [4], we also provide a short-path from each PE back to its corresponding ME -- short" in terms of communication delay. This provision eliminates the need to use the communication network for computed results destined for pieces assigned to their generating processors. This feature is particularly useful in promoting efficient execution of loops.

![Figure 6. A dataflow architecture for hybrid graphs.](image)

Intermediate results within a piece stay in the piece's assigned PE, where they are used by their consumer nodes. Intermediate results incur less communication since intra-PE communication tends to be much faster than inter-PE communication.

Since these intermediate results do not use the communication network, larger pieces tend to diminish network traffic, as well as to shorten average delays. However, independent nodes within a piece must contend for execution on the PE to which the piece is assigned. Hence, larger pieces tend to increase execution conflict. To maximize system performance, one must choose a partition granularity (i.e., the size of the individual pieces) that balances the reduction in communication delay against overhead incurred in terms of processor contention.

In studying this performance tradeoff, we use the execution graph of a program rather than the program graph itself. This is the program graph with all loops unrolled and all conditional firings eliminated or left, depending on the course of the computation -- for a given program graph, different input data can lead to different execution graphs. By the critical path in such a graph we mean the longest path in terms of processor time -- communication time is not taken into consideration in determining this path.

One limiting case in terms of processor loading is that of extremely heavy loading with all PEs fully utilized. In this case, the time to fire the graph is \( N/n \), where \( N \) is the total amount of processor time required by the graph and \( n \) is the number of PEs in the system. Reducing communication overhead via partitioning is of no consequence since communication is fully overlapped with computation.

One simplification made here is the assumption that the delay in the communication network is \( d \) units of time, independent of load. Also, we assume that the delay between the arrival of the last input token for a piece and the sending of the packet of input tokens for the execution of the piece is zero.

The other limiting case in terms of processor loading is that of extremely light load where an excess of processors are available. In this case, the time needed to complete the firing of the graph is the time needed to fire its critical path. We assume that the pieces along the critical path are a random sample and that each piece's portion of the critical path is the critical path of that piece, then the length of the critical path in the partition graph is \( m/u \), where \( m \) is the number of nodes on the critical path and \( u \) is the average length of the critical path of a piece of the partition. (Poor partitioning can greatly expand this value.) Thus, the time to complete the firing of the entire graph is \( (k\epsilon+d)m/u \), where \( k \) is the average number of nodes per piece and \( \epsilon \) is the average processor time per node. This is the time required to fire the critical path. \( m/u \) pieces times the sum of the time \( k\epsilon \) to fire each piece plus the time \( d \) required to finish communicating results to the next piece on the critical path.

Essentially, we have assumed an architecture with sufficient communication bandwidth, i.e., where there is no queuing delay in the communication network. As load increases, queuing delay increases within the PEs, so communication time represents a smaller portion of the total service time for each node. Since partitioning is a technique to reduce communication overhead, it is in the lightly loaded case that partitioning yields the greatest performance benefits.

But, it should be noted that partitioning reaches the point of diminishing returns as partition size increases. The communication overhead decreases by a factor of \( u \) and rapidly becomes a small fraction of the total service time with increasing values of \( u \). Meanwhile, the computation time on the critical path increases by a factor of \( k/u \). So, for example, if computation time and communication time are equal, then the case where \( u=2 \) and \( k=4 \) doubles the processing time along the critical path and cuts the communication time in half for a net loss of 25% in performance compared to the unpartitioned case (i.e., the case where
The case of moderate processor loading involves a certain amount of queueing delay within the PEs. Since the rate at which work gets done on the graph is the speed of the individual processors times the number of PEs times the utilization factor, our goal should be to maximize the utilization factor -- the other two factors are usually fixed by external considerations. In general, one can view the effect of communication delay on performance as that of withholding (from the ready queue of the PEs) instructions that are ready for execution, thereby increasing the probability of a PE being idle.

Figure 7. A two-dimensional grid structure.

In [14], we developed a queueing model that exhibits the fundamental tradeoffs involved in partitioning in conditions of moderate load: improvement in processor utilization due to the fact that fewer results are tied up in the communication network vs. decrease in processor utilization due to the fact that more enabled nodes are in pieces that are only partially enabled (therefore unfirable) and due to increased processor contention (decreased parallelism) since all nodes of an enabled piece must be fired by the same processing element.

The model shows that for architectures in which PEs have a low degree of internal execution concurrency, the optimal granularity is generally small, even when the communication delay is several times the average node firing time. But smaller partitions lead to smaller reduction in communication delay. Measures may be taken to enhance the degree of internal concurrency of individual PEs, if larger partitions are to be exploited.

Figure 8. (a) A graph for computing the dot product of a with b. (b) Resulting grid for the graph in (a).

4. THE DATA-DRIVEN CELL ARRAY

A variety of processor arrays, such as systolic and wavefront, have recently been proposed and many computational algorithms for these arrays have been devised [15-17]. Both systolic arrays and wavefront arrays have proven to be effective for executing highly regular computations. Dataflow computations do not exhibit such regularity and, therefore, are less suitable for such arrays. Still, many computations, e.g., array operations, have an inherent parallelism,
and it should be possible to exploit this parallelism through properly structured processor arrays.

To exploit this parallelism, we structure each dataflow piece derived above as a two-dimensional grid of nodes, as shown in Figure 7. A grid piece receives its input data through the nodes of the top and left edges, and emits its results through the nodes of the bottom and right edges. Each interior node in the grid communicates unidirectionally with its four adjacent neighbors. It receives input data from the two upper neighbors, its predecessors, and emits results to the two lower neighbors, its successors. The choice of this grid structure is based on the study in [11-13], which shows that many array computations can efficiently be embedded into such a structure.

Figure 8 shows a possible structuring of a dataflow piece that computes the dot product of two four-component vectors, into a 5-by-5 grid structure. As input, the piece needs two vector tokens, one carrying vector a and the other vector b. Each s node is to split a vector token it receives into two halves. In the grid, arcs receiving no data tokens (not shown in the figure) are assumed to carry null tokens. Data tokens carry a value, whereas null tokens carry nothing but a signal. Null tokens are used in node enabling, but not in firing. Nodes receiving no data tokens as input are assumed to perform null operations. We use a to denote such nodes in Figure 8. A null operation performs nothing, but it emits a null token as output. Nodes for communicating tokens are introduced during transformation, whenever necessary. In Figure 8, all nodes at levels 3, 5, and 8 are introduced for this purpose. A structuring heuristic that attempts to minimize the number of additional levels of nodes required for the structuring is presented in [14].

4.1 Structure

To support parallel movement of a single wavefront of tokens within a grid piece, we structure each PE as a linear array of cells. See Figure 9. We assign a grid column of nodes to each cell for execution. The number of cells required for a grid is determined by the number of grid columns. Thus, five cells are needed for the grid shown in Figure 8.

Each cell array receives input data from its input section (IS), and emits computed results to its output section (OS). Each cell in the array communicates unidirectionally with its two adjacent neighbors. Each cell receives two operands; the first operand from the left cell, its predecessor, and the second from the IS or itself. It emits two results; the first result to the OS or itself, and the second to the right cell, its successor. The leftmost cell receives the first operand from the IS, while the rightmost cell emits the second result to the OS. Hence, the leftmost cell has the IS as its predecessor, whereas the rightmost cell has the OS as its successor.

Each cell is provided with three dedicated buffers: two for holding the operands and the other one for storing the computed result. To facilitate the firing check, each buffer has a flag bit to keep track of its availability. Thus, the firing check for a cell can be done by simply examining these flag bits for the appropriate values. In addition to buffers for the values of tokens, each cell is given a tag buffer for holding the tags of the tokens received by the cell.

Each cell has its own control unit and an ALU capable of performing logical and integer operations, including integer multiply. Floating-point and complex operations are performed through the cooperative action of several cells.

Each cell array is associated with a program memory (PM) that stores instructions corresponding to the nodes of grid pieces assigned to this cell array. Concurrent access to program instructions is essential for efficient computation. We, therefore, divide the program memory into dual-port modules, each of which can be accessed through one port by a single cell. The host computer loads instructions for these pieces through the other port.

![Figure 9. A one-dimensional cell array with four cells.](image-url)
4.2 Principles of Operation

A token carries a value, a tag, and a port number. All tokens destined for piece $P$ carry a tag of the form $<C/P/I>$, where the context $(C)$ specifies a particular invocation, the portion of program graph and the iteration number $(I)$ specifies a particular loop iteration within this invocation. The port number specifies the input arc of a grid piece to which the token is sent. We refer to such a reference as a logical reference.

It has to be transformed into a physical reference before execution. In the proposed architecture, grid pieces are enabled by MEs and executed by their corresponding cell arrays. Therefore a token referencing a grid piece must carry a physical tag specifying which ME enables the piece. Since a piece is stored in a program memory attached to its assigned cell array, rather than in the ME where it is enabled, it is necessary for such a token to carry its destination instruction address in the memory. We assume that all the grid columns of a piece are stored at the same starting address in each module. Thus a physical tag has the form $<C/m/b/I>$, where $m$ and $b$ refer to an ME id and a starting memory address for the block of instructions corresponding to a grid column of nodes, respectively.

When a matching set of tokens for a piece are received by an ME, they will be sent as a packet to the corresponding cell array for execution. The packet has the following format:

$$<C/b/i>, <v, J_{1,0}>, <v, J_{2,0}>, \ldots, <v, J_{m,0}>,<v, J_{0,1}>, <v, J_{0,2}>, \ldots, <v, J_{0,n}>$$

where $v$'s are the token values, $J_{i,0}$ ($i=1, 2, \ldots, m$) are the input ports for the rows of the piece, and $J_{0,j}$ ($j=1, 2, \ldots, n$) are the input ports of the columns. When the IS of a cell array receives a token packet, it will load these tokens to the cells in the array, according to their associated port numbers. Ports receiving no tokens are filled with null tokens. Tokens (data and null) destined for ports $J_{0,j}$ are loaded to the corresponding cell, whereas tokens destined for other ports are loaded to the left-most cell. More specifically, the values of tokens destined for ports $J_{0,j}$ are stored into the second input buffer of the corresponding cell, and the tags $<C/b/i>$ are stored into the individuals' tag buffers. The values of tokens for ports $J_{i,0}$ are sequentially loaded to the first input buffer of the left-most cell, but their tags are no longer needed.

Any token received by a cell will trigger the cell to use the value of the second field $(b)$ in the tag buffer to fetch the corresponding instruction from its memory module. The cell also checks whether the token carries "data.” If so, and if the output buffer of the cell is empty, then the cell will either execute the instruction just fetched or wait for the other “data” to arrive, and then proceed with execution. If the cell expects only one argument, it will not emit its result until the “null” value has arrived. If the first received token is null, the cell absorbs it and nothing happens. If a second token received from the other input is again null, the cell emits a null value to the output buffer.

Depending on where this computed result is needed, the cell will do one of the following:

(i) If it is needed by both this cell and its successor, then a copy is sent to this cell's second input buffer, and the result is left unchanged.

(ii) If it is needed by this cell only, then a copy of it is sent to this cell's second input buffer, and the result is replaced by a null value.

(iii) If it is needed by the successor only, then a null value is inserted into this cell's second input buffer, and the result is left unchanged.

On completion of firing, old values will be overloaded. The cell sends an acknowledge signal to its predecessor, and increments the value of the second field in the tag buffer by one. The acknowledge signal informs the predecessor that it may gate its newly produced output from its output buffer to the first input buffer of the cell.

Null tokens are used to ensure that the wavefronts of tokens corresponding to different pieces, or distinct invocations of the same piece, proceed in a pipeline fashion. The array can accept a new activation when the left-most cell has finished its execution on the current activation. It is not necessary for the array to wait until all the cells of the array have produced output. The array, thus, permits up to two activations to execute concurrently.

Concurrency in the array can be improved by inserting a buffer queue between adjacent cells to buffer results waiting to be consumed by the cell's successor. As soon as the computed result is queued, the cell can start firing its next input immediately, thereby enhancing concurrency.

In the execution mechanism described above, the cell might sit idle during instruction fetches. Conventional processors employ a variety of techniques, such as instruction prefetch, etc., to avoid this latency. Since the nodes in each grid column will all be fetched and executed sequentially by the corresponding cell, conventional instruction-prefetch is well suited for eliminating this latency. To support this, an instruction buffer is provided for each cell to store prefetched instructions.

5. SUMMARY AND CONCLUSION

We have presented a hybrid graph model for dataflow computation. It integrates the concepts of the tagged-token model with those of the static model. This hybrid model supports recursion and iteration without requiring full token-matching -- a limited amount of matching of tokens suffices. This implies savings in computing resources including the computational time needed for matching. The model was
presented from its formulation, through a discussion of the problems of pipelining merging operators and function invokers in a dataflow graph, to a description of a data-driven cell array architecture that supports this hybrid form of dataflow computation.

The maximum number of independent operations in a piece is bounded so that the assigned cell array can concurrently execute independent operations within that piece without processor contention. Immediate results stay in registers local to the individual processing elements of the assigned array, thereby short-circuiting the inter-PE communication network. The cell array architecture exploits parallelism and pipelining within a piece.

6. REFERENCES


