A Parallel Algorithm for Correlating Event Streams

Daniel M. Zimmerman and K. Mani Chandy
Computer Science 256-80
California Institute of Technology
Pasadena, California 91125 USA
dmz@cs.caltech.edu mani@cs.caltech.edu

Abstract

This paper describes a parallel algorithm for correlating or "fusing" streams of data from sensors and other sources of information. The algorithm is useful for applications where composite conditions over multiple data streams must be detected rapidly, such as intrusion detection or crisis management. The implementation of this algorithm on a multithreaded system and the performance of this implementation are also briefly described.

1. Introduction

Many applications require near real-time correlation of noisy data generated in large numbers of event streams. The detection of potential bioterror incidents requires integration of information from emails and other texts, time-varying incidence rates of diseases across the country, threat analyses, and movements of possible terrorists. Dealing with hurricanes requires tracking the hurricanes, tracking ships and planes, monitoring the capacities of shelters and hospitals, monitoring flood levels and road conditions, and even tracking individuals using cell phones and RFID tags. Other applications, such as money laundering detection and intrusion detection, also require correlation of noisy heterogeneous data generated by large numbers of sources.

The increasing availability of events generated by RFID readers and sensors of a multitude of types, event services on the Web such as news feeds, events generated by ERP applications, and data provided by the public allows organizations to sense threats and opportunities and respond appropriately. Integrating multiple heterogeneous streams of information in a timely fashion is a challenge that can be handled by employing multiprocessor machines.

People in different roles in an organization may be concerned about different threats and opportunities. In the aftermath of a hurricane, public health workers are concerned about issues such as hospital occupancy and blood supply; electric utilities, on the other hand, are concerned about how best to deploy their repair crews to restore power. The number of conditions that must be monitored can be very large. Data fusion algorithms such as the one described in this work can help integrate data streams and identify complex conditions rapidly to enable timely responses.

Critical conditions—threats or opportunities—are specified as predicates over event stream histories. A predicate defines a pattern; for example, a predicate could be that the one-week moving point average rate of incidence of a disease in any county is two standard deviations away from a regression model developed using data from a one-month window in neighboring counties. The conditions are typically complex functions of event histories, and these functions often use models such as statistical regressions, time series analyses, clustering of points in multidimensional spaces, and simulations of systems as diverse as boilers and financial markets.

Each model or computational unit makes assumptions about its environment and expects to be notified if these assumptions are violated. As an example, consider a system for pricing electrical energy. The system is comprised of many models, such as models forecasting temperature variation in the coming day, load on the power grid and future prices. The model for power demand may assume that temperature will vary in some fashion—for instance, 15°C at midnight, 20°C in the early morning and 30°C at noon. The power-demand model expects to receive an event if data from a sensor or some other model indicates that its assumptions about future temperatures are wrong. If the temperature sensor measures temperature at midnight to be 10°C when the power-demand model expects it to be 15°C, then the sensor sends a message to the power-demand model and the model adjusts its assumptions about temperature appropriately. The critical aspect of model composition is this: information is conveyed by the *absence* of events as well as the presence of events.

Data fusion may be carried out by large networks of complex models, but rates of events flowing between mod-

Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05)
1530-2075/05 $ 20.00 **IEEE**

els could be small if assumptions made by models are violated infrequently. Consider, for example, an application to detect money laundering. One of the steps in the application may be to detect anomalies in banking transactions, where anomalies are defined as outlier points in a statistical regression model. We can construct the anomaly detector module in two ways: (1) the module outputs a message for each input message (banking transaction) that it receives, where the output message is either that the transaction is anomalous or that it is acceptable; or (2) the module outputs a message only when it receives an anomalous transaction. If one in a million transactions is anomalous then the rate of events generated using the second option is only a millionth of that generated using the first option. We will see later that the second option causes race conditions in parallel implementations, and our challenge is to deal with such race conditions effectively. A central aspect of our implementation is that efficiency is obtained by exploiting the key fact that *information can be conveyed by the absence of messages*.

Event correlation is carried out by a network of computational modules. Modules may execute models such as simulations of boilers or analyses of stochastic differential equations representing financial systems. Since modules and events can be quite complex, the system is implemented using object-oriented technology (Java) rather than relational database technology and SQL.

This paper describes an algorithm for correlating multiple data streams that can be implemented on a single multiprocessor machine. The algorithm is implemented in Java 1.5 and exploits the fact that compositions of complex models can compute efficiently with relatively low rates of inter-model communication.

2. Problem Definition

The system that integrates and correlates multiple data streams to detect threats and opportunities is called a *data fusion system* or an *event correlation system*. We assume in this paper that there is no delay between the instant at which an event is generated and the instant at which it arrives at the data fusion system. We also assume that each event has a timestamp indicating the instant at which the event was generated, and that timestamps are accurate. Thus, we assume that all events with timestamp t reach the data fusion system at time t. In reality, events may be delayed between the sensor and data fusion system, and clocks used to determine timestamps may drift from each other resulting in errors; we do not analyze errors in this paper.

Consider the sequence of events arriving at the data fusion system. Assume that events arrive at times t_1, t_2, t_3, ... All events that arrive at the same time are considered part of the same *phase*. Phases are indexed sequentially; all events that arrive at time t_k belong to phase k. The collection of events that arrive at phase k represents a snapshot of the system and its environment at time t_k. The data fusion engine treats all the events in a given phase as being representative of the same instant. One solution is to require the data fusion engine to complete execution of one phase before initiating execution of the next phase. We describe a more efficient solution, in which multiple phases are executed concurrently but with the same logical effect as if they were executed sequentially.

The computation in a data fusion system is represented by an acyclic directed graph in which vertices represent computational modules and edges represent information flow between modules. Vertices without incoming edges are called *source vertices*. Messages from information sources, such as sensors, arrive at source vertices. Vertices without output edges are *sink vertices*. Sink vertices are read by input/output units outside the data fusion system. The computation of the data fusion system must be serializable; though modules are executed concurrently, the logical effect must be the same as executing only one phase at a time in serial order all the way from the sources to the sinks.

3. A Parallel Algorithm

We wish to develop an efficient parallel algorithm, for a shared-memory multiprocessor, to execute the computation graphs described in the previous section. The algorithm should be able to execute multiple vertices concurrently, and should also allow multiple phases of the computation to execute concurrently. Essentially, the execution of the computation graph should be pipelined as much as possible. Figure 1 depicts a 10-node graph in which 5 phases are being executed concurrently, where nodes near the top of the graph are executing earlier phases of the computation than nodes near the bottom. In this section, we first discuss how this pipelining can be accomplished; we then present a parallel algorithm that executes the computation graphs with pipelining; finally, we present a correctness argument.

3.1. Pipelining

In order to pipeline the execution efficiently, we must determine the parts of the computation that can execute concurrently. Since the output of a particular vertex for a phase p depends only on its inputs for phase p and its internal state, any vertex for which all the inputs for phase p are known can execute concurrently with other vertices for which all the inputs for phase p are known. Thus, a large part of the problem is to determine when the inputs for phase p for a particular vertex are known.

The obvious solution to this problem is to ensure that every vertex receives a message on every one of its inputs during every phase; then, whenever a vertex has messages with

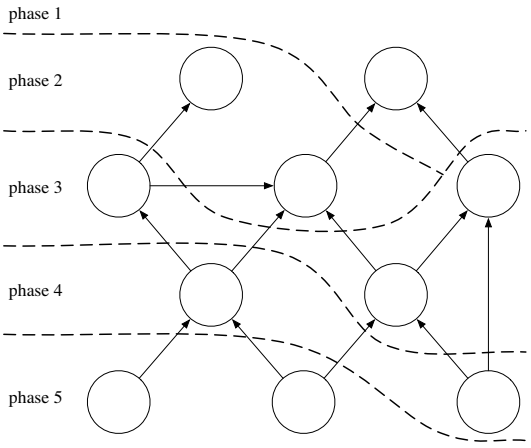


Figure 1. A 10-node graph in which 5 phases are being executed concurrently.

phase p waiting on all its inputs, it can execute phase p. Unfortunately, this obvious solution is inefficient, because it requires every vertex to both carry out a computation for every phase and send a message on every one of its outputs for every phase. It seems likely that, in the majority of computation graphs, only a small number of vertices will change their outputs during any given phase of a computation; if this is not the case, the idea of structuring the computation as a graph may not be appropriate in the first place. It would therefore be preferable to minimize the amount of computation and the number of messages sent.

The amount of computation can be minimized by making vertices compute only when their inputs change. This also reduces the number of messages sent, because a vertex that doesn't compute also doesn't send messages. However, it makes detection of when all the inputs for a particular vertex are known for phase p more difficult, because there is no guarantee that the vertex will receive messages on all its inputs for phase p, and no reliable way to determine that it will *not* receive such messages. Assume vertex v has two inputs, and receives a message for phase p on one of them and nothing on the other. There is no way for v to determine that it can execute phase p until it receives a message for a later phase on its other input, but because v only receives messages on its inputs when the outputs of other vertices change, there is no guarantee that it will ever receive such a message. Thus, we need a way to determine when v can execute phase p.

3.1.1. Vertex Numbering. To help determine when vertices can safely execute, we assign indices to the vertices in the graph such that they are topologically sorted; that is, all edges in the graph are directed from lower-indexed vertices to higher-indexed vertices. In an N-node graph, these indices range from 1 to N. In addition to being topologically sorted, we require the indices to satisfy an additional restriction. We first define this restriction and then show how the restriction helps us determine when it is safe for a given vertex to execute a given phase. The notation we use for quantification throughout this paper is $\langle$*op boundvars* | *ranges* $\triangleright$ *expression*$\rangle$, where *op* is an associative and commutative operator with an identity element, *boundvars* is the set of bound variables, *ranges* is a predicate restricting the ranges of the bound variables, and *expression* is the expression to be quantified.

Assume that, in an N-node graph, the vertices are given integer indices v, $1 \leq v \leq N$. Let $E(x, y)$ be **true** if there is an edge directed from node x to node y, and **false** otherwise, and let $S(v)$ be defined as follows, for $0 \leq v \leq N$:

$$S(v) = \langle \cup w | \langle \forall u \mid E(u, w) \triangleright u \leq v \rangle \triangleright \{w\} \rangle \quad (1)$$

$S(v)$ is the set of vertices all of whose predecessors are indexed v or lower, and $S(0)$ is (by inspection) the set of vertices with no input edges; the vertices in $S(0)$ are the source vertices of the graph. Figure 2(a) shows a graph with topologically sorted indices and its corresponding S values.

The additional restriction on vertex numbers is that, for every v, the vertices in $S(v)$ must be indexed sequentially from 1 to $|S(v)|$. For example, in Figure 2(a), $S(1)$ is $\{1, 2, 3\}$ and is indexed sequentially from 1 to 3. However, $S(2)$ is $\{1, 2, 3, 5\}$ and is not indexed sequentially because 4 is missing. Therefore, the numbering of the graph in Figure 2(a), though topologically sorted, does not satisfy the additional restriction.

Now, consider the graph in Figure 2(b), where vertices 4 and 5 are transposed. This numbering, which is also topologically sorted, satisfies the additional restriction because all the $S(v)$ are indexed sequentially.

Let $m(v)$ be the cardinality of $S(v)$; then for graphs that satisfy the additional restriction, $S(v) = \{1, 2, 3, ..., m(v)\}$. In Figure 2(b), the sequence of values of $m(v)$ from $v = 0$ to $v = 7$ is [3, 3, 4, 5, 5, 6, 7, 7].

From the definition of m, we get the following properties:

$$\langle \forall u, v \mid 1 \leq u < v \leq N \triangleright m(u) \leq m(v) \rangle \quad (2)$$
$$\langle \forall v \mid 1 \leq v < N \triangleright v < m(v) \rangle \quad (3)$$
$$m(N) = N \quad (4)$$

In the next section, we describe how to use m to determine when we can execute vertices concurrently.

3.1.2. Concurrent Execution. The behavior of the function $m(v)$ on a graph numbered according to our restrictions allows us to determine when a particular vertex has all the inputs it needs to execute a particular phase p, and therefore to determine the set of vertices and their corresponding

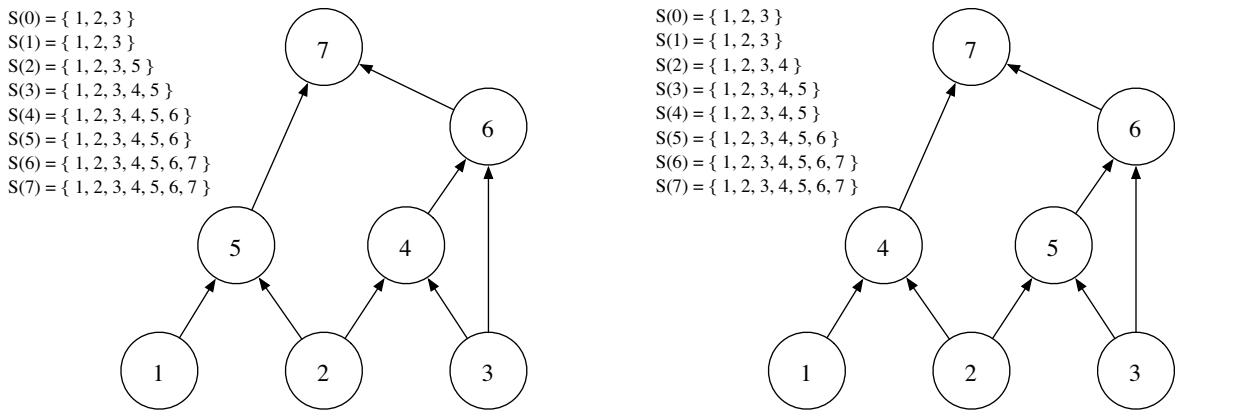


Figure 2. Two topologically sorted graphs and their corresponding S(v) values.

phases that may execute concurrently. When we say that a vertex "executes" a phase p, we mean either that it consumes inputs that are associated with phase p and performs some computation, or that its computation for phase p is determined to be unnecessary because none of its inputs have changed for phase p.

The function $m(v)$ is used to determine which vertices may execute, as follows: when all vertices indexed v and lower have finished executing phase p, all vertices indexed $m(v)$ and lower have sufficient information (changes, or the absence thereof, for all inputs) to execute phase p. This is the case because the predecessors of all vertices indexed $m(v)$ and lower are indexed v and lower, by definition of $m(v)$.

In order to determine the vertices and phases that may execute concurrently, we consider each pair (v, p), consisting of the vertex indexed v and phase p, separately. Execution of a vertex-phase pair (v, p) is the execution of phase p by the vertex indexed v, using any inputs v has received for phase p and using previous values for any inputs it has not received for phase p.

A value x_p for each phase p indicates how much of the phase has been executed. We define x_0 to be N, and x_p for $0 < p$ to be the highest index such that $x_p \leq x_{p-1}$ and all vertices indexed x_p and lower have finished executing phase p. The restriction $x_p \leq x_{p-1}$ prevents higher-numbered phases from "overtaking" lower-numbered ones and violating the serializability of the outputs.

Clearly, two distinct pairs (v, p) and (v, q) must not execute concurrently, because a single vertex can only execute one phase at a time. In addition, for $p < q$, (v, p) must execute before (v, q) because phases must execute in order. At any moment, subject to the preceding two restrictions, all pairs (v, p) such that $x_p < v \leq m(x_p)$ may execute phase p concurrently because all their inputs are known.

We determine the set of vertex-phase pairs that may execute concurrently as follows. Assume that vertices are indexed $1, 2, ..., N$, phases are numbered $1, 2, ...$, and no phases are skipped. Further assume that, for each phase p, x_p is the maximum index such that all vertices indexed x_p and lower have already executed (x_p is 0 if phase p has not yet started). Then the set of vertex-phase pairs that have sufficient information to execute at any given time—a full set of inputs—is given by:

$$\mathit{full}_\infty = \langle \cup v, p \mid 1 \leq p \wedge (x_p < v \leq m(x_p)) \rhd \{(v, p)\} \rangle \quad (5)$$

Note that there are an infinite number of vertex-phase pairs in full_∞ corresponding to each source vertex, because of the way m is defined. Since full_∞ contains multiple vertex-phase pairs with the same vertex index, the set of vertex-phase pairs that may execute concurrently given our previously-stated restrictions on concurrent execution is the subset of full_∞ consisting of the vertex-phase pairs with the minimum phase for each vertex, given by:

$$\mathit{ready}_\infty = \langle \cup v, p \mid (v, p) \in \mathit{full}_\infty \wedge \langle \forall q \mid (v, q) \in \mathit{full}_\infty \rhd p \leq q \rangle \rhd \{(v, p)\} \rangle \quad (6)$$

The sets full_∞ and ready_∞ determine exactly what vertices may execute concurrently given any state of the system described by the set of x_p for $1 \leq p$. However, by themselves they are not enough to allow us to implement a feasible algorithm for concurrent execution of a computation graph. The set definitions must be restricted in order to create a feasible algorithm that uses them.

First, we note that full_∞ encompasses all possible phases, starting from phase 1. We can restrict this by in-

troducing the idea of the maximum phase that has started execution. We define p_{max} such that the phase number p of every phase that has started execution falls within the range $1 \leq p \leq p_{max}$. We also introduce the Boolean $msg_{(v,p)}$, which is **true** if there is at least one message with phase p waiting on an input of vertex v, and **false** if there are no messages with phase p on inputs of vertex v. A message for phase p waiting on a vertex input is considered to remain on that input until the vertex has finished executing phase p, at which point it is consumed. Using these new terms, new full and ready sets may be defined as follows:

$$full = \langle \cup v, p \mid 1 \leq p \leq p_{max} \wedge msg_{(v,p)} \wedge x_p < v \leq m(x_p) \rhd \{(v,p)\} \rangle \quad (7)$$

$$ready = \langle \cup v, p \mid (v,p) \in full \wedge \langle \forall q \mid (v,q) \in full \rhd p \leq q \rangle \rhd \{(v,p)\} \rangle \quad (8)$$

Source vertices present a problem with this formulation, because they have no inputs and can therefore never have messages waiting on their inputs. We assume for the sake of this formulation that every source vertex receives a phase signal from an external source for every phase p. This phase signal is treated as an input, allowing the source vertex to be part of the full set. After a source vertex is executed for phase p, it consumes the phase p signal just as a non-source vertex would consume its phase p input messages.

Next, we introduce a new set, the partial set, containing all the vertex-phase pairs that have at least one new input but do not have sufficient information to execute (that is, they have a partial set of inputs). The partial set is defined as follows:

$$partial = \langle \cup v, p \mid 1 \leq p \leq p_{max} \wedge msg_{(v,p)} \wedge m(x_p) < v \rhd \{(v,p)\} \rangle \quad (9)$$

The partial, full, and ready sets tell us what vertex-phase pairs need to be executed, as well as when they are eligible for execution. We can construct an algorithm that maintains data structures corresponding to these set definitions, the *msg* values, and the values of p_{max} and x_p. If this algorithm executes vertex-phase pairs only when they are in the ready set, and guarantees that every vertex-phase pair placed in that set gets executed exactly once, then the set definitions guarantee that the algorithm will correctly execute the computation graph. Figure 3 depicts part of a correct execution of a computation graph, showing the set memberships of the vertex-phase pairs for two phases of the execution.

3.2. Algorithm Description

The algorithm, which can run on a shared-memory symmetric multiprocessor with an arbitrary number of processors, consists of computation processes (Listing 1) executed concurrently by an arbitrary number of threads. Each process is structured as an infinite loop; each execution of the loop consists of taking the next unexecuted vertex-phase pair from a run queue, executing it, and updating the data structures accordingly.

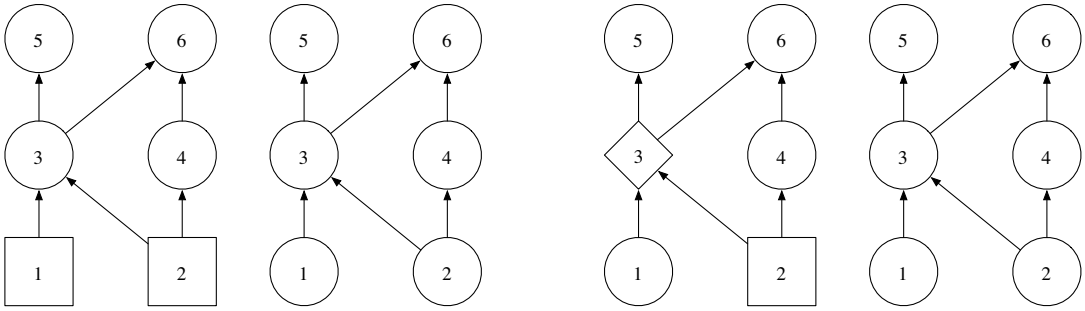
The run queue is assumed to be a thread-safe queue: any thread executing a dequeue operation suspends until an item is available for dequeuing, and the dequeue operation atomically removes an item from the queue such that each item on the queue is dequeued at most once. It is also assumed to be empty at system initialization time. A lock is used to guarantee that each thread has exclusive access to the data structures while updating them; we denote the lock and unlock operations by **lock** and **unlock** statements. We assume that there is an additional process, called the *environment* and shown in Listing 2, that also has access to these data structures and the same lock. The environment is responsible for initializing the data structures (other than the run queue) and for starting new computational phases. In a real-world implementation of this algorithm, the environment process would determine when phases begin and end based on the arrival of data from external sources such as sensors; in this simple form, however, the environment merely starts new phases repeatedly. It starts a phase p by putting all the vertex-phase pairs (v,p), where v is a source vertex, into the full set and then updating the ready set and run queue appropriately.

In Listings 1 and 2, the lines referring to the *msg* values are enclosed in [ ] braces; this reflects the fact that the *msg* values are ghost variables, used in the correctness argument (because they are part of the set definitions) but not necessary for the correct functioning of the algorithm.

3.3. Correctness Argument

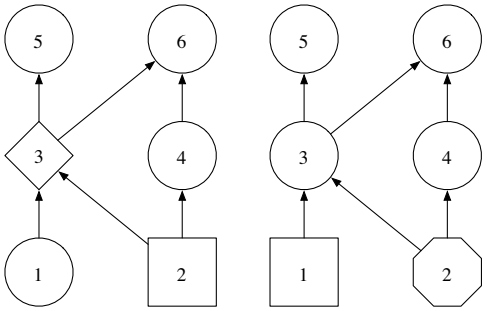
Space constraints prevent us from including a formal correctness proof of the algorithm; instead, we present an informal correctness argument. The algorithm is considered to be correct if it maintains the partial, full, and ready sets according to definitions (7)–(9), executes concurrently only those vertex-phase pairs that are concurrently in the ready set, and executes every vertex-phase pair that is present in the ready set (at any time) exactly once. If the system fulfills these conditions, then because of the way we have defined the three sets it will generate correct results.

We must demonstrate that the algorithm correctly maintains the partial, full, and ready sets according to their definitions. These are represented in the algorithm listings as the *partial*, *full*, and *ready* variables. Their definitions depend on the values of p_{max} and msg. We must first show that these two status variables are maintained correctly—that is, they are consistent with their definitions—before moving on to demonstrate the correctness of the set ma-

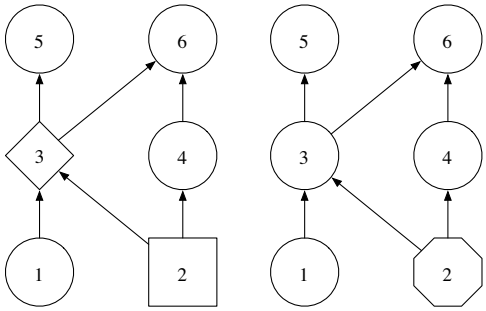


(a) Phase 1 initiated

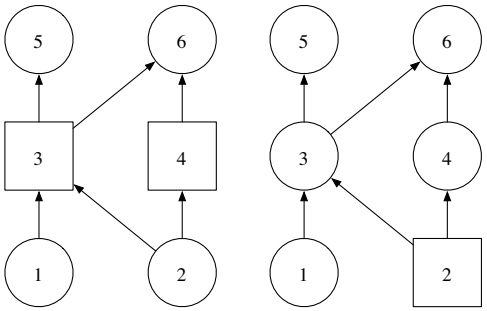
(b) (1, 1) executed, generated output



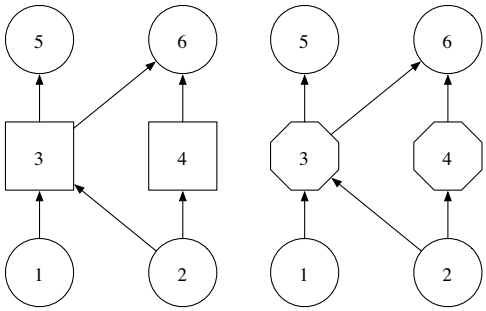
(c) Phase 2 initiated



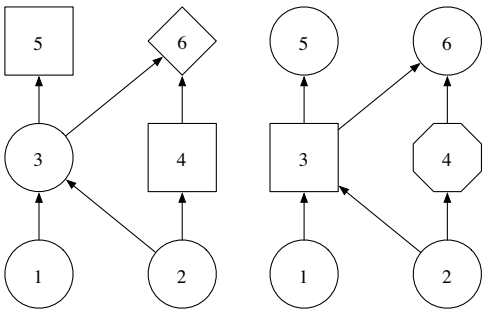
(d) (1, 2) executed, generated no output



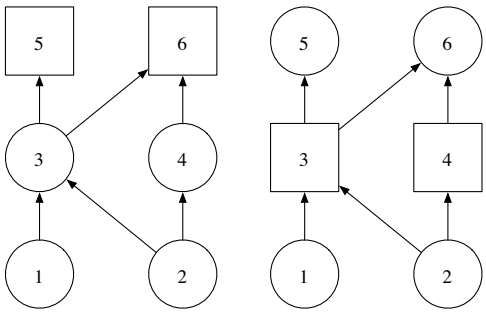
(e) (2, 1) executed, generated output



(f) (2, 2) executed, generated output



(g) (3, 1) executed, generated output



(h) (4, 1) executed, generated output

Figure 3. Eight steps in the execution of a computation graph. The two graphs in each subfigure represent execution phases 1 and 2, with phase 1 on the left. Vertices depicted as circles (○), diamonds (◇), octagons (⯃) and squares (□) are in no set, only the partial set, only the full set, and both the full and the ready set, respectively.

Listing 1 A computation process.

```
1:  loop
2:      (v, p) := run-queue.dequeue
3:      execute the computation for (v, p)
4:      lock
        {remove (v, p) from sets}
5:      full := full \ {(v, p)}
6:      ready := ready \ {(v, p)}
7:      [msg_(v,p) := false]
8:      for all (w, p) such that v generated an output for w
        during its execution of phase p do
9:          partial := partial ∪ {(w, p)}
10:         [msg_(w,p) := true]
11:     end for
        {update x_p to reflect the current state of the
        computation}
12:     i := p
13:     repeat
14:         if ⟨∃w ▷ (w, i) ∈ (partial ∪ full)⟩ then
15:             x_i := ⟨min w | (w, i) ∈ (partial ∪ full) ▷ w⟩ −
                1
16:         else
17:             x_i := N
18:         end if
19:         if x_i > x_(i−1) then
20:             x_i := x_(i−1)
21:         end if
22:         i := i + 1
23:     until p_max < i
        {move newly full pairs from partial set to full set}
24:     newly-full := ⟨∪w, q | (w, q) ∈ partial ∧ w ≤
        m(x_q) ▷ {(w, q)}⟩
25:     partial := partial \ newly-full
26:     full := full ∪ newly-full
        {add newly ready pairs to ready set and run queue}
27:     for all (w, q) such that (w, q) ∈ full ∧ (w, q) ∉
        ready ∧ ⟨∀r | (w, r) ∈ full ▷ q ≤ r⟩ do
28:         ready := ready ∪ {(w, q)}
29:         run-queue.enqueue (w, q)
30:     end for
31:     unlock
32: end loop
```

nipulations. We assume that the system is started with a single thread running the environment process and some number k of threads running computation processes. The exclusive access guaranteed by the use of the **lock** and **unlock** statements around data structure accesses makes reasoning about the data structure manipulations straightforward.

Within the correctness argument, we refer to program statements by concatenating their listing and statement numbers with a "."; for example, statement 6 of listing 2 is referred to as statement 2.6.

Listing 2 The environment process.

```
1:  lock
2:  partial, full, ready := ∅, ∅, ∅
3:  p_max := 0
4:  next := 1
5:  x_0 := N
6:  ⟨‖ i | 0 < i ▷ x_i := 0⟩
7:  [⟨‖ i, v | 0 < i ∧ 1 ≤ v ≤ N ▷ msg_(v,next) := false⟩]
8:  unlock
9:  loop
10:     lock
        {start a new phase and add its source vertex pairs to
        full set}
11:     p_max := next
12:     for all v such that v is a source vertex do
13:         full := full ∪ {(v, next)}
14:         [msg_(v,next) := true]
15:     end for
        {add newly ready pairs to ready set and run queue}
16:     for all (w, q) such that (w, q) ∈ full ∧ (w, q) ∉
        ready ∧ ⟨∀r | (w, r) ∈ full ▷ q ≤ r⟩ do
17:         ready := ready ∪ {(w, q)}
18:         run-queue.enqueue (w, q)
19:     end for
        {update next}
20:     next := next + 1
21:     unlock
22:     sleep for some amount of time
23: end loop
```

3.3.1. Status Variables. The initialization statements for the environment—statements 2.2–2.7—are guaranteed to execute before any manipulation of variables by the computation processes. This is the case because the run queue is initially empty, and therefore each computation process must suspend on statement 1.2 waiting to dequeue a vertex-phase pair. We examine p_{max} first, then msg.

The p_{max} value is modified only by the environment process. Initially, p_{max} is set to 0 (statement 2.3). This is consistent with its definition, since no phases have started execution. The only statement that changes p_{max} is statement 2.11, the second statement of the environment process loop, which sets it to be equal to *next*. By inspection, the environment process loop is also the only part of the system that is capable of inserting, into any of the three sets, a vertex-phase pair having a phase number that was not already present in one of the sets; it repeatedly starts a phase numbered *next* and increments *next*. Since *next* is initially 1, giving p_{max} the value of the phase that is about to be started each time through the loop ensures that all phases that have started execution fall within the phase number range $1 \leq p \leq p_{max}$. Therefore, p_{max} is consistent with its definition.

The msg values are manipulated by both the environment process and the computation processes. They are initialized for a phase p when phase p is started in the environment process loop; it is clear by inspection that no vertex-phase pair with phase p can be executed by a computation thread without phase p having been started by the environment process, and computation threads modify only msg values for the phases they are currently computing. Initially, all $msg_{(v,p)}$ are set to **false** (statement 2.7), and then the $msg_{(s,p)}$ for all source vertices s are set to **true** (statement 2.14). This is consistent with the definition of msg because, as discussed previously, we consider sources starting a new phase to have an incoming message on a "phase signal" channel.

When a computation thread finishes computation for a vertex-phase pair (v, p), it sets $msg_{(v,p)}$ to **false** (statement 1.7), and sets $msg_{(w,p)}$ to **true** for all w such that v generated an output for w (statement 1.10). These statements keep the msg values consistent with their definition, as follows. Once the computation for (v, p) is complete, the inputs for v with phase p are considered consumed (and therefore $msg_{(v,p)}$ must be **false**), while the generation of new messages for w with phase p means that $msg_{(w,p)}$ must be **true**.

3.3.2. x. The x_p values must be shown to be accurately maintained in order to demonstrate the correctness of the set manipulations. For each phase p, x_p is initialized to 0 by the environment process (statement 2.6); as no vertex-phase pairs with phase p can have executed by the time this initialization occurs, this is consistent with the definition of x_p. The only time x_p is subsequently changed for a phase p is after a vertex-phase pair with phase p or lower finishes execution (statements 1.12–1.23).

When each phase is examined, there are two possible cases. Either there are still other vertex-phase pairs with phase p in the partial or full set, or there are not. If there are not, then phase p is complete, because no more messages can be sent with phase p if no more vertex-phase pairs with phase p execute, so x_p is set to N (statement 1.17). If there are, then statement 1.15 sets x_p to $\langle \min w \mid (w, p) \in (partial \cup full) \rhd w \rangle - 1$ (in the interest of brevity, we refer to this quantity as v_{min}). In either case, the value is checked to ensure that it does not exceed x_{p-1} (statements 1.19–1.21). This update of x_p is consistent with the definition, for the following reason: messages can only be sent from vertices with lower indices to vertices with higher indices, and only vertices that are already in the sets or that subsequently receive messages can be executed. Therefore, at the time x_p is set to v_{min}, any vertex indexed v_{min} or lower must have either already executed, or must not need to execute (because its inputs have not changed), for phase p, while vertices numbered higher than v_{min} are either still executing or have not yet executed. This shows that setting x_p to v_{min} is consistent with the definition of x_p as the highest index such that all outputs from all vertices indexed x_p or lower for phase p are known.

3.3.3. Set Manipulations. Now that x_p, p_{max}, and msg have been shown to be consistent with their definitions, the set manipulations can be examined. We examine first the set manipulations by the environment process, then the set manipulations by the computation processes.

When the environment process starts, it initializes all the sets in a way such that their definitions are satisfied (statements 2.2–2.3): all the sets are empty, and the range from 1 to p_{max} is an empty range. Assume that the sets are consistent with their definitions when the lock is acquired at the beginning of each loop iteration (after statement 2.9). Statement 2.11 sets p_{max} to *next*. After statement 2.11 the set contents have not changed, but they are still consistent with their definitions because there are no vertices (v, p_{max}) such that $msg_{(v,p_{max})}$ holds. Statements 2.12–2.14 insert $(v, next)$ for all source vertices v into the full set and set $msg_{(v,next)}$ to **true** for those vertices; this makes the full set consistent with its definition, because $m(0)$ is the index of the highest-numbered source vertex. Finally, statements 2.17–2.18 are executed once for each vertex-phase pair (v, p), such that p is the minimum phase in the full set for vertex v, that is in the full set but not in the ready set. This makes the ready set consistent with its definition. No further set manipulations take place in the environment process loop, so at the **unlock** statement (2.21), the invariant that the sets remain consistent with their definitions has been preserved.

Next, assume that the sets are consistent with their definitions when a computation process acquires the lock (statement 1.4). Upon acquiring the lock, it has completed the execution of (v, p); by removing (v, p) from the full and ready sets and setting $msg_{(v,p)}$ to **false** (statements 1.5–1.7), the full and ready sets are kept consistent with their definitions. Next, every (w, p) such that v generated an output for w while executing phase p is added to the partial set and has $msg_{(w,p)}$ set to **true** (statements 1.8–1.11). This keeps the partial set consistent with its definition at this point in the execution, because x_p has not yet changed. Next, x_p and the x_i values for subsequent phases are updated (statements 1.12–1.23). Statements 1.24–1.26 then move any vertices that are within the appropriate range from the partial set to the full set, keeping those sets consistent with their definitions given the new value of x_p. Finally, statements 1.28–1.29 are executed once for each vertex-phase pair (v, p) contained in the full set but not in the ready set such that p is the minimum phase in the full set for vertex v. This makes the ready set consistent with its definition. No further set manipulations take place in the computation process loop, so at the **unlock** statement (1.31), the invariant that the sets remain consistent with their definitions has been preserved.

3.3.4. Run Queue. All that remains to demonstrate the correctness of the algorithm is to show that each vertex-phase pair placed in the ready set is executed exactly once. First, note that vertex-phase pairs may move through the sets in a very limited number of ways. Either they are placed directly into the full set (by statement 2.13), or they are initially placed in the partial set (by statement 1.9). They can then be moved from the partial set to the full set (by statements 1.24–1.26), placed into the ready set exactly once (by statement 1.28 or statement 2.17), or removed from both the full and ready sets (by statements 1.5–1.6). The key observation is that it is impossible for a vertex-phase pair to be put into the ready set twice. Given that observation, the definition of the queue data structure, and the fact that each vertex-phase pair is enqueued once at the time it is placed in the ready set (by statement 1.28 or statement 2.17), it is clear that each vertex-phase pair is executed exactly once. Therefore, any execution of the graph using this algorithm generates correct results.

4. Implementation and Measurements

We have developed a prototype implementation of the algorithm described in Section 3 using Java 1.5. We chose Java 1.5 over earlier versions of the Java platform because of its robust concurrency support and its generic typing constructs, both of which significantly reduced the work required to develop the prototype. In particular, we made use of the `java.util.concurrency` classes `Lock`, `Condition`, `BlockingQueue` and `ThreadPoolExecutor` to implement the concurrency control for manipulation of the variables, the run queue, and the pool of computation threads. In addition to the pool of computation threads, there is always an additional thread that runs the implementation's equivalent to the environment process; thus, there are always at least two threads contending for exclusive access to the data structures.

The prototype implementation takes as input an XML specification file for a computation, which includes a specification of the computation graph with vertices as instances of Java classes conforming to well-defined guidelines. The specification file also contains simulation parameters, such as the number of timesteps to run and random seeds to use for the generation of random values by source vertices. The implementation makes use of several optimizations and custom data structures to make the operations described in Listings 1 and 2 efficient.

We have so far been able to perform only limited performance testing of our prototype implementation. On a dual-processor machine running Solaris, we have found that identical computations see a speedup of approximately 50% when two computation threads are running, compared to the speed when a single computation thread is running. As mentioned previously, there is always a thread running for the environment process; thus, the 50% speedup is a reasonable result (because the number of threads contending for the data structures is increased from 2 to 3). We have not yet been able to test on machines with more than two processors; however, we predict that as long as the computations performed by the vertices take significantly more time than the computations performed to maintain the data structures, the speedup will be close to linear in the number of processors when we use a thread pool containing one computation thread for each processor.

5. Related Work

A great deal of work has been carried out on dataflow networks since the 1970s, with Kahn-McQueen networks [12] and subsequent research by Dennis [9] and by Arvind and Culler [1] among others. The model of computation we use is a variant on dataflow that is called Δ-dataflow [13]. Both dataflow and Δ-dataflow networks are represented as directed graphs in which vertices represent computations and edges represent message-passing channels. A computation in a dataflow graph executes when messages are present on each of its input channels. By contrast, in our model, a computation is executed when all the information required for the computation is available, even though some information may be conveyed by the *absence* of messages. This is the key difference between our work and earlier work on parallel computations using dataflow graph models.

Serializability has been studied in the context of databases for decades [15]. Our problem is also to develop concurrent algorithms that are serializable—they have the same effect as executing one phase at a time. The way we achieve serializability is very different from the way it is achieved in databases. We don't use start and end transaction primitives, and we don't use rollbacks.

An immense amount of work has been carried out on algorithms for multithreaded systems. Earlier work has not, however, considered serializable Δ-dataflow networks.

A recent overview of work on sensor networks is provided by Culler [8]. Algorithms for data fusion directly within networks of sensor nodes—as opposed to within data fusion engines—have been studied extensively [4]. Some of these algorithms also use the insight upon which the algorithm discussed in this paper is based: the absence of messages can encode valuable information.

Data stream management systems (DSMS) also deal with the problem space studied in this paper, continuous detection of predicates on data streams [2, 3, 5, 6, 7]; their primary focus is on extending database management systems and SQL to deal with events. Event servers [11] deal

with the same problem as well; their focus is on extending object-oriented approaches, rather than relational database approaches, to deal with events. The algorithms described in this paper can be employed in DSMS and event server systems that run on multithreaded architectures.

Distributed simulation algorithms [14] predict the future, up to some point called the "look-ahead time", by using models. The idea of prediction using models and proofs employed in our system is essentially the same as that used in distributed simulation. Kinetic data structures (KDS) [10] detect conditions on moving objects. KDS also predict the future, up to some point, by using properties of mobile objects. This idea in KDS is similar to the idea in distributed simulation and in our algorithm. Most of the work on KDS does not deal with execution on multiprocessor machines.

6. Future Work

The algorithm we have presented only uses a single shared-memory multiprocessor to perform data fusion. We are investigating various ways of using networks of multiprocessor machines to improve performance and efficiency, including methods for partitioning the computation graph across multiple machines and replication of event streams to multiple distinct computation graphs.

In constructing our algorithm, we have assumed that timestamps are perfect and that there are no transmission delays between sensors and the data fusion engine. This allows us to treat all messages that arrive at the fusion engine at time t as a snapshot of the environment at t. In reality, clocks in sensors are noisy and message delays may be significant and random. The fusion engine must wait long enough after time t to ensure that sensor data taken at time t arrives with high probability. Incorporating more accurate notions of concurrency and the passage of time are necessary for analyzing error: the probability of false positives (claiming that conditions occur when they don't) and false negatives (not detecting conditions that do occur).

7. Summary

In this paper, we have described a multithreaded algorithm for correlating event streams. The algorithm is a serializable version of Δ-dataflow. The results of preliminary experiments with an implementation in Java 1.5 are promising, and we intend to conduct experiments with larger symmetric multiprocessors. The algorithms given here are useful for applications with large volumes of asynchronous events, such as crisis management dealing with natural or man-made disasters.

Acknowledgements

The research described in this paper has been supported in part by the National Science Foundation under grant CCR-0312778, ITR: Information Infrastructures for Crisis Management.

References

[1] Arvind and D. Culler. Dataflow architectures. LCS Technical Memo MIT/LCS/TM-294, Massachusetts Institute of Technology, Feb. 1996.

[2] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *2000 ACM SIGMOD International Conference on Management of Data*, May 2000.

[3] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, June 2002.

[4] A. Boulis, S. Ganeriwal, and M. B. Srivastava. Aggregation in sensor networks: An energy-accuracy trade-off. In *First IEEE International Workshop on Sensor Network Protocols and Applications (SNPA)*, May 2003.

[5] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. B. Zdonik. Monitoring streams - a new class of data management applications. In *28th International Conference on Very Large Data Bases (VLDB)*, Aug. 2002.

[6] D. Carney, U. Çetintemel, A. Rasin, S. B. Zdonik, M. Cherniack, and M. Stonebraker. Operator scheduling in a data stream manager. In *29th International Conference on Very Large Data Bases (VLDB)*, Sept. 2003.

[7] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *First Biennial Conference on Innovative Data Systems Research (CIDR)*, Jan. 2003.

[8] D. Culler, D. Estrin, and M. Srivastava. Overview of sensor networks. *IEEE Computer*, 37:41–49, Aug. 2004.

[9] J. Dennis. Dataflow supercomputers. *IEEE Computer*, 13(11):48–56, Nov. 1980.

[10] L. J. Guibas. Kinetic data structures: A state of the art report. In *Third International Workshop on Algorithmic Foundations of Robotics*, Aug. 1998.

[11] iSpheres Corporation. iSpheres Halo event server. `http://www.ispheres.com/`.

[12] G. Kahn. The semantics of a simple language for parallel programming. In *IFIP Congress 74*, Aug. 1974.

[13] R. Manohar and K. M. Chandy. Δ-Dataflow networks for event stream processing. In *IASTED International Conference on Parallel and Distributed Computing and Systems*, Nov. 2004.

[14] J. Misra. Distributed discrete event simulation. *ACM Computing Surveys*, 18(1):39–65, Mar. 1986.

[15] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill, 2001.