# Using Simulation and Markov Modeling to Select Data Flow Threads

Dan R. Greening

Miloš D. Ercegovac

University of California, Los Angeles

**Abstract**

Communication and matching delays between actors in a data flow graph present a significant performance degradation factor. We can reduce these delays by partitioning actors into large sequential threads, and bypassing matching and queueing operations in communications that occur between actors in the same thread.

We provide an algorithm to give the set of all maximal sequential partitionings for a data flow graph. Selecting an optimal partitioning from this set is incomputable. We explore two heuristics for selecting a partitioning: typical-instance simulation and Markov analysis.

In the two programs we tried, the Markov heuristic selected the same partitionings as typical-instance simulation. Selecting a different partitioning had little impact on the execution time. However, using the worst-case maximal partitionings improved execution time by 33% and 18% over the non-partitioned programs. Using the best-case partitionings improved execution time by 39% and 20%.

## 1 Introduction

Communication and matching delays often dominate the execution time of data flow programs. This may be the major reason data flow machines have not replaced control flow machines: the gains reaped from increased parallelism in data flow programs have been offset by increased communication and synchronization time. We can reduce this overhead by identifying naturally sequential threads in a data flow program which require no matching. One can find several ways to partition a data flow program, each with a different execution time.

We present an algorithm for enumerating maximal sequential thread partitionings of a data flow program. We present two heuristics for choosing among different partitionings: typical-instance simulation and Markov analysis. We apply these heuristics to two data flow programs, compare the execution times of different partitionings, and discuss the implications of our results.

### 1.1 Definitions

We use "actor" to refer to a primitive data flow operation, and "thread" to refer to a collection of actors that naturally execute in sequence. If a program has been divided into threads, we call it a "partitioning."

When a program consists entirely of unpartitioned actors, we call it a "fine-grained" data flow program. When a program's actors have been partitioned in any fashion (not necessarily into threads), we call it "coarse-grained."

"Intra-thread communication" refers to the (inexpensive) passing of data between two actors within the same thread. "Inter-thread communication" refers to the (expensive) passing of data from one thread to another.

### 1.2 The Problem

Data transmission from one actor to another incurs a large time cost when operands must pass through a matching unit or through slow communication channels. Thus, coarse-grained data flow programs—where many intermediate operands do not pass through a matching unit—often run faster than equivalent fine-grained programs. Machines with long transmission or matching delays exacerbate the effect, as shown by both simulations [Gaud85,Huda85] and mathematical constructions [Gaud84].

In experiments on 29 numerical analysis programs, Manchester machine researchers discovered that unary instructions comprised between 56% and 70% of the total instructions executed [Gurd85]. They bypassed the matching store when an instruction with a unary output followed an instruction with a unary input, to reduce overall execution time.

Identifying "unary-output followed by unary-input" constitutes a special case of identifying "natural sequential threads in a data flow program." Precedence relations can force data flow program fragments that include $n$-ary (not just unary) operations to run sequentially.

### 1.3 Our Contribution

We extend Gurd and Watson's work to identify and combine naturally sequential $n$-ary actors into large threads. In doing this, we preserve inherent parallelism (unlike [Gaud85]), while reducing overall communication and matching delays to a minimum. We show how to obtain the set of maximal sequential partitionings of a data flow graph. We often find several alternatives, each with a different execution time.

Choosing an optimal partitioning is incomputable. Simulation of a typical terminating instance provides a good, but expensive, heuristic for estimating execution time. Markov modeling provides an approximate, but less expensive, heuristic.

Markov chains allow us to compare the performance of different partitionings without executing the program. We show that we must remove closed subsets from the Markov chains, an unfortunate consequence of the stochastic nature of the model.

To evaluate the two heuristics, we wrote a suite of programs:

1. A program which reads a probabilistic data flow graph, obtains the set of all maximal sequential partitions, converts each to a Markov chain, removes closed subsets, and produces the resulting Markov transition probability matrix.
2. A Markov chain solving program, which produces stationary probabilities and mean return times.
3. A tagged-token data flow simulator, based loosely on the Manchester machine.

We will show two example programs, then partition and apply both typical-case simulation and Markov modeling heuristics.

This work has been supported in part by the UCLA Academic Senate under Grant No. 4361.

```
1   S ← {v ∈ V|(∃e ∈ E, δ0(e) ≠ −1 ∧ sink(e) = v) ∨ δ0(v) ≠ −1}
2   S ← S ∪ {v ∈ V| |P_s^v| > 1}
3   M ← S, j ← 0
4   while S ≠ ∅ do
5       select any s ∈ S
6       S ← S − {s}, j ← j + 1, k ← 0, Q ← ∅, Found ← true
7       while Found do
8           Found ← false, k ← k + 1, P_(j,k) ← s, Q ← Q ∪ {s}
9           for all v ∈ V such that v ∈ succ(s) ∧ v ∉ M do
10              if pred(v) ⊆ M
11                      M ← M ∪ {v}
12                      if pred(v) ⊆ Q ∧ ¬Found
13                              Found ← true, n ← v
14                      else
15                          S ← S ∪ {v}
16                      end if
17              end if
18          end for
19          s ← n
20      end while
21      P_(j,k+1) ← ϵ
22  end while
```

**Algorithm PSB (Partition Sequential Blocks)**

# 2 Identifying Sequential Threads

One can partition data flow graphs in several ways. We choose to preserve all parallelism, but incorporate as many actors into each thread as possible. We adopt the following set of rules:

> All input tokens coming into a sequential thread must be ready before the thread starts. Otherwise, if a sequential thread could partially complete and then wait for a token, deadlock can occur. We will not incorporate an actor into a thread if it requires tokens from another thread and it is not the first actor in the thread. We allow any actor (i.e., not necessarily the last actor in a thread) to send intermediate output tokens to other threads.

Algorithm PSB partitions a program graph into the largest possible sequential threads according to the above rules. It uses the following notation: $V$ is the set of vertices, or actors, in a data flow graph. $E$ is the set of edges in a data flow graph. $\delta_0(e) = -1$ means that no token rests on edge $e$ in its initial state. $\delta_0(v) = -1$ means that vertex $v$ is not processing data in its initial state. $P_s^v$ is the set of all enabling groups for vertex $v$. Therefore, the term $|P_s^v| > 1$ in statement 2 of Algorithm PSB asks whether a vertex may be started by more than one set of vertices. The functions $pred(v)$ and $succ(v)$ return the set of predecessor and the set of successor vertices for vertex $v$, respectively.

Algorithm PSB first places all vertices that must begin sequential blocks in set $S$ (statement 1). It uses this criteria: If an *initial* token rests on a vertex's incoming edge (i.e., if residual time $\delta_0(e) \neq -1$) or the vertex itself is processing a token ($\delta_0(v) \neq -1$), that vertex is placed in $S$. Statement 2 adds those vertices which have more than one enabling group.

Threads then grow from these "starting vertices" in the loop beginning at statement 7. If an immediate successor to the current vertex, $s$, *requires* a token from $s$ to begin execution, and if that successor accepts tokens *solely* from the vertices preceding it in the current thread (set $Q$), it is selected as a successor within the thread.

Any other successor vertices, whose predecessors are entirely in $Q \cup M$, will be added to the set $S$. These will serve as additional thread-starting vertices.

Upon completion of Algorithm PSB, each $P_i$, where $1 \leq i \leq j$, comprises one sequential block.

## 2.1 Execution Times for Different Partitionings

The indeterminancy of statement 9 in Algorithm PSB allows several different partitionings of most graphs. In Figure 1, we see two possible outcomes. In Outcome 1, a result token will appear on the left input of SUB earlier than on the right input, because the left path requires fewer vertices (machine operations) and fewer exposed edges (communication delays) than the right path. In Outcome 2, the two result tokens will appear at SUB at approximately the same time.

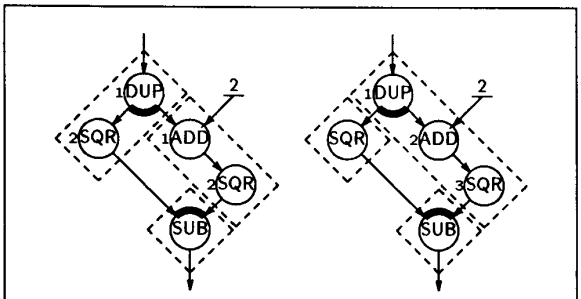

Figure 1: Two Outcomes of Algorithm PSB

**Remark 1** *Outcome 2 of Figure 1 is faster than Outcome 1.*

PROOF. Let $t: C \rightarrow \mathbf{R}$, where $C$ is the set of all operation codes, $\mathbf{R}$ is the set of real numbers, and $t(c)$ is the execution time of $c$. Let $t_1$ be the average time to process intra-thread communications, and let $t_2$ be the average time to process inter-thread communications. $t_1 \leq t_2$ because $t_2$ includes matching store time, while $t_1$ does not.

The Outcome 1 program takes $T_1 = t(\text{DUP}) + max(t(\text{SQR}), t_1 + t(\text{ADD}) + t(\text{SQR})) + t_2 + t(\text{SUB})$ time units. The Outcome 2 program takes $T_2 = t(\text{DUP}) + max(t_1 + t(\text{SQR}), t(\text{ADD}) + t(\text{SQR})) + t_2 + t(\text{SUB})$ time units. $T_2 = T_1 - t_1 - t(\text{ADD}) + max(t_1, t(\text{ADD}))$. Since $t_1 > 0 \wedge t(\text{ADD}) > 0$, we see that $T_2 < T_1$. ■

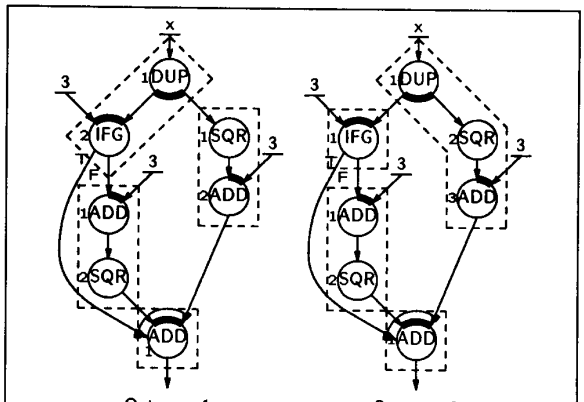

Figure 2: Optimal Partition Depends on Input

**Theorem 1** *Partitioning an unevaluated data flow graph into optimal sequential blocks is incomputable.*

PROOF. Two partitionings result from Figure 2 under Algorithm PSB. With $x > 3$, the left partitioning completes earlier, otherwise the right partitioning completes earlier.

The optimal graph partitioning depends on the value of incoming value $x$. Since the value of $x$ (computed by a preceding data flow graph) is not known, partitioning the data flow graph into optimal sequential threads is incomputable. ■

Applying non-deterministic Algorithm PSB to a data flow graph will result in a set of different sequential partitionings. Theorem 1 implies that heuristics must be used to select a partitioning from this set.

# 3 Simulation

Our simulator corresponds to the Manchester machine [Gurd85], except that in our system

1. communication plus matching time is fixed for each edge,
2. the matching unit handles more than two input edges per operation,
3. The matching unit handles arbitrary enabling groups,
4. an infinite number of processors are provided, and
5. no matching-unit or communication contention occurs.

# 4 Markov Analysis and PDFGs

As an alternative to a deterministic approach (simulation), we can represent a data flow program by its statistical behavior. Here, we describe Probabilistic Data Flow Graphs, their conversion to Markov chains, and the computation of expected execution times.

Probabilistic data flow graphs remove the notion of executing data flow operators, and substitute transition probabilities. Typical arithmetic operators retain their previous control-flow under the PDFG model, but conditional branches and loops change to purely stochastic operators with fixed transition probabilities.

## 4.1 PDFG Construction

Each vertex corresponds to a machine operation, and each edge corresponds to the communication of a value from its source vertex to its sink vertex.

Each edge or vertex holds a set of *tokens*. Like Petri nets [Pete77], probabilistic data flow graphs can be *safe* or *unsafe*. We restrict our analysis to *safe PDFGs*, where an edge contains at most one token. This greatly reduces the resulting Markov state-space, but introduces some restrictions.

A PDFG executes as follows: Before a vertex begins executing, one token must be available on each of a set of *enabling edges*, called an *enabling set*. A vertex may have several enabling sets. The vertex consumes a token from each edge in one enabling set.

After a delay it puts a token on each edge in a set of *production edges*, called a *production set*. One vertex may have several production sets, but it may place tokens on only one production set at a time. A production set is called *ready* if none of its edges carry tokens. When a vertex wants to place tokens on a not-ready production set, the vertex halts until the production set is ready.

Each enabling set and production set carries a rational weight value. The probability that a ready enabling set will fire is the ratio of its weight over the total weight of all *ready* enabling sets on the same vertex. The probability that tokens will be produced on a production set after its source vertex fires is the ratio of its weight over the total weight of all production sets (including not-ready sets) on the same vertex. We require neither enabling sets nor production sets to be disjoint.

For example, suppose vertex $v$ has three enabling sets $\overline{E}_1 = \{a, b, c\}, \overline{E}_2 = \{c, d\}$, and $\overline{E}_3 = \{e, f\}$ with weights $w(\overline{E}_1) = 0.3, w(\overline{E}_2) = 0.5$, and $w(\overline{E}_3) = 0.2$. Suppose tokens rest on edges $a, b, c, e$ and $f$. Then enabling sets $\overline{E}_1$ and $\overline{E}_3$ enable vertex $v$. We compute the probability that vertex $v$ will consume tokens from $\overline{E}_1$ in Equation 1.

$$p(\overline{E}_1) = \frac{w(\overline{E}_1)}{w(\overline{E}_1) + w(\overline{E}_3)} \tag{1}$$

Unlike enabling set probabilities, production set probabilities are independent of which sets are ready. Thus, if $\overline{E}_1, \ldots, \overline{E}_n$ are production sets for vertex $v$, the probability that production set $\overline{E}_i$ will receive tokens after vertex $v$ fires is given by Equation 2.

$$p(\overline{E}_i) = \frac{w(\overline{E}_i)}{\sum_{j=1}^{n} \overline{E}_j} \tag{2}$$

Conversion of a PDFG to a Markov chain follows by establishing a set of PDFG states. Tokens present on the initial graph are "executed" by vertices, and are transmitted along edges. A Markov chain is generated in the typical method by connecting states with their successor states, and assigning computed probabilities to each edge. Since we use safe PDFGs, the resulting Markov chain is finite.

## 4.2 Obtaining Probability Estimates

A critical issue in using a stochastic model is the determination of enabling and production weights. Two methods appear fruitful: extrapolated simulation and programmer estimates.

First, using simulations of low execution-time sample data sets, one can obtain transition frequencies. Extrapolating these transition frequencies to normal execution-time data sets, one can obtain enabling and production weights.

Second, the programmer can often produce reasonable transition frequency estimates, and include them as pragmatic remarks in program source.

## 4.3 Closed Subsets in Markov Chains

We define the *maximal closed subset* of a Markov chain as the largest closed proper subset of the Markov chain (there is only one). A maximal closed subset includes those states which become "trapped" and which cannot return to the start state. Algorithm FMCS finds the maximal closed subset.

| | | |
|---|---|---|
| 1 | $S \leftarrow \{s_i\}, F \leftarrow \emptyset$ | ; $S$ is unvisited nodes not in MCS. |
| 2 | while $S \neq \emptyset$ do | ; Find vertices that reach any $x \in S$. |
| 3 | select any $x \in S$ | ; Visit one vertex. |
| 4 | $F \leftarrow F \cup \{x\}$ | ; Add $x$ to list of found vertices. |
| 5 | $S \leftarrow S \setminus \{x\}$ | ; Remove $x$ from list of unvisited. |
| 6 | $S \leftarrow S \cup \{\{v \in V_i \mid < y, x > \in E_i\} \setminus F\}$ | ; Get vertices that can start this one. |
| 7 | end while | ; |
| 8 | $M \leftarrow V_i \setminus F$ | ; Set $M$ to Maximal Closed Subset. |

**Algorithm FMCS (Find Maximal Closed Subset)**

We assume that the programmer will not intentionally program an infinite loop. We remove maximal closed subsets and normalize transition probabilities appropriately. Otherwise, our expected execution time would be infinite in some cases.

Figure 3 demonstrates this. We remove the state transition with probability $c$ from the Markov chain, and set its probability to zero. We normalize the other state transition probabilities associated with this vertex, multiplying them by $1/(1-c)$.

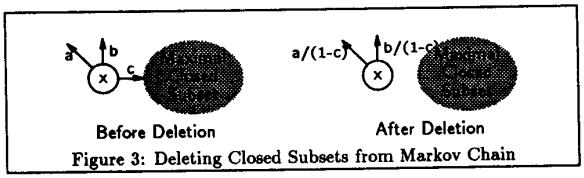


Figure 3: Deleting Closed Subsets from Markov Chain

## 4.4 Obtaining Expected Execution Time

Each Markov chain is characterized by a transition probability matrix T. Several algorithms have been produced which find the stationary probability vector $\mu = \lim_{i \to \infty} \mu^{(i)}$ in order $O(t^3)$ time [Heym87], where $t$ is the number of columns in T. If $\mu_1$ is the stationary transition probability for the start state, then $1/\mu_1$ is the mean return time for the start state and the expected execution time for the data flow program.

### 4.5 Limitations of the Stochastic Model

Three problems arise from our stochastic model. First, finding enabling and production set weights is an incomputable problem. At best, our methods for obtaining those weights produce reasonably good approximations.

Second, data flow operations do not operate stochastically. As a result, programs which include interdependent branches cause errors in the model.

Third, many present machines allow several tokens to rest on an edge. Likewise, in some machines, many copies of the same actor or vertex can be operating on different sets of tokens simultaneously. Safe PDFGs do not correctly model the multiple token case.

## 5 Experimental Results

Our two trial programs were written in SISAL, a stream-oriented, Pascal-like applicative language [McGr85]. All data flow instructions in these examples were simulated in one cycle. Inter-thread communication and matching took one cycle. Intra-thread communication took zero cycles.

| *Version* | *Analysis* | *Simulation* |
|---|---|---|
| No partitioning | 602.1 cycles | 761 cycles |
| Partitioning # 1 | 286.0 cycles | 459 cycles |
| Partitioning # 2 | 301.3 cycles | 507 cycles |

Table 1: INTEGRATE: Analysis vs. Simulation

### 5.1 Sample Program 1: INTEGRATE

The INTEGRATE program is derived from an example discussed in [Gurd85]. The source follows:

```
define Integrate
function Integrate (returns real)
    for initial
        int := 0.0;
        y   := 0.0;
        x   := 0.02
    while
        x < 1.0
    repeat
        int := 0.01 * (old y + old y);
        y   := old x * old x;
        x   := old x + 0.02
    returns
        value of sum int
    end for
end function
```

The resulting data flow program appears in [Gree88]. Algorithm PSB identified two partitionings for the INTEGRATE program. Each partitioning reduced the 23 actors to 10 threads. Results of Markov analysis and simulation are shown in Table 1.

### 5.2 Sample Program 2: RECURSIVE_AQ

The RECURSIVE_AQ program is taken from [McGr85], a recursive adaptive quadrature program. We supplied routines to integrate $x^2 + 3x - 8$ from $x = 0$ to $x = 10$.

```
define Recursive_AQ

type Interval =
     record [ X_Low, Fx_Low,
              X_High, Fx_High : real ];
type Interval_List = array[ Interval ]

function Evaluate_Function( X: real returns real )
  (X * X) + (3.0 * X) - 8.0
end function

function Stop_Condition( Area_1, Area_2,
        Interval_Width: real returns boolean )
  (abs(Area_1 - Area_2) < 2.5)
     & (Interval_Width < 1.0)
end function

function Recursive_AQ( L, Leftv, R, Rightv: real
        returns real, boolean )
  let
    Mid := (L + R) * 0.5;
    Midv := Evaluate_Function(Mid);
    Prev_area := (R - L) * (Rightv + Leftv) * 0.5;
    New_Area := (R - Mid) * (Rightv + Midv) * 0.5
        + (Mid - L) * (Midv + Leftv) * 0.5;
    Done := Stop_Condition(Prev_Area,
        New_Area, R-L );
    Abort := is error(New_Area) | is error(Done)
  in
    if Abort then Prev_Area, true
    elseif Done then New_Area, false
    else
      let
        Left_Area, Abt_Left
          := Recursive_AQ(L, Leftv, Mid, Midv);
        Rgt_Area, Abt_Rgt
          := Recursive_AQ(Mid, Midv, R, Rightv);
      in
        Left_Area + Rgt_Area , Abt_Left | Abt_Rgt
      end let
    end if
  end let
end function
```

A graphic description of Recursive_AQ appears in Figure 4. Partitioning reduced 68 to 40 threads. Markov analysis and simulation divided the graph into two classes of partitionings. These results appear in Table 2.

| *Version* | *Analysis* | *Simulation* |
|---|---|---|
| No partitioning | 65.3 cycles | 324 cycles |
| Partitioning Class # 1 | 52.3 cycles | 259 cycles |
| Partitioning Class # 2 | 53.3 cycles | 264 cycles |

Table 2: RECURSIVE_AQ: Analysis vs. Simulation

## 6 Conclusion

Since Markov analysis chose the same partitionings as simulation in our trials, it seems to be a promising heuristic for choosing thread partitionings.

Selecting a particular partitioning made only a small difference in the execution time. For examples 1 and 2, the best-case partitioning execution times beat the worst-case partitionings by only 9.4% and 1.8% respectively. However, even the worst-case partitioning beats no partitioning by a substantial margin, 33.3% and 18.5% for examples 1 and 2. The best-case partition beats no partitioning by 39.6% and 20.0%.

We note that the number of partitionings generated by Algorithm PSB can be large. The decomposition methods of [Kape88] can address this problem to some extent. However since worst-case and best-case times do not differ substantially, using a cheaper heuristic to reduce the number of partitions generated by Algorithm PSB might provide a more efficient scheme. We are currently looking into other heuristics for Algorithm PSB.

## References


[Gaud84] J.L. Gaudiot and M.D. Ercegovac, Performance Analysis of a Data-Flow Computer with Variable Resolution Actors, in *IEEE Proceedings of the International Conference on Distributed Computing Systems* (1984).

[Gaud85] J.L. Gaudiot, R.W. Vedder, G.K. Tucker, D. Finn, and M.L. Campbell, A Distributed VLSI Architecture for Efficient Signal and Data Processing, *IEEE Transactions on Computers*, C-34(12):1072–1087 (December 1985).

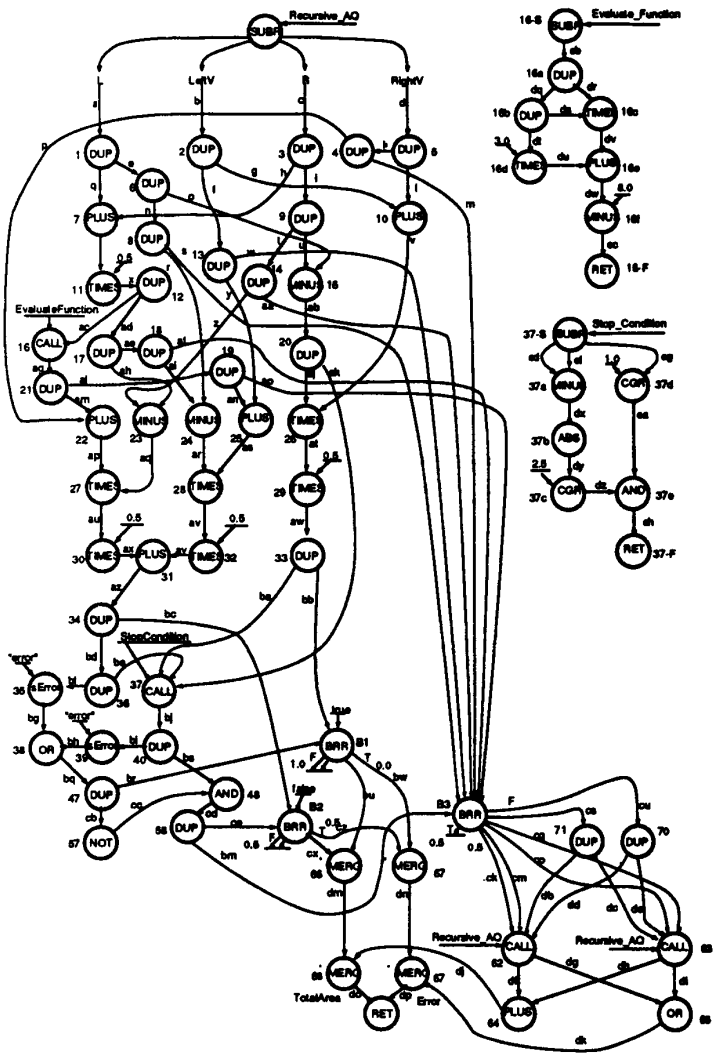


Figure 4: RECURSIVE_AQ: Probabilistic Data Flow Graph


[Gree88] D.R. Greening, *Modeling Granularity in Data Flow Programs*, Master's thesis, UCLA, Los Angeles, California (1988), Computer Science Department.

[Gurd85] J.R. Gurd, C.C. Kirkham, and I. Watson, The Manchester Prototype Dataflow Computer, *Communications of the ACM*, 28(1):34–52 (January 1985).

[Heym87] D. Heyman, Further Comparisons of Direct Methods for Computing Stationary Distributions of Markov Chains, *SIAM Journal on Algebraic and Discrete Systems*, 8(2):226–232 (April 1987).

[Huda85] P. Hudak and B. Goldberg, Distributed Execution of Functional Programs Using Serial Combinators, *IEEE Transactions on Computers*, C-34(10):881–891 (October 1985).

[Kape88] A. Kapelnikov, R.R. Muntz, and M.D. Ercegovac, A Modelling Methodology for the Analysis of Concurrent Systems and Computations, to appear in *Journal of Parallel and Distributed Computing* (1988).

[McGr85] J.R. McGraw and S.K. Skedzielewski, *SISAL: Language Reference Manual, version 1.2*, Technical Report M-146, Lawrence Livermore National Laboratory, Livermore, California (1985).

[Pete77] J.L. Peterson, Petri Nets, *ACM Computing Surveys*, 9(3):223–252 (September 1977).