IDENTIFYING AND SELECTING SEQUENTIAL THREADS IN DATA FLOW PROGRAMS

Daniel R. Greening
Milos D. Ercegovac

February 1988
CSD-880008

Identifying and Selecting Sequential Threads in Data Flow Programs

Dan R. Greening
Miloš D. Ercegovac

University of California, Los Angeles

January 13, 1988

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. This paper presents a method for partitioning a cyclic data flow graph into a collection of sequential threads, reducing overall execution time in appropriately designed machines.

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. However, through simulation of a "typical" case, we can select a reasonable partitioning.

In simulating two example programs, 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 time. We can reduce execution time by identifying naturally sequential threads in a data flow program which require no matching. Generally, one can find several ways to partition a data flow program, each with a different execution time.

We present an algorithm for enumerating all maximal sequential thread partitionings of a data flow program. We show its application to two example data flow programs, INTEGRATE and RECURSIVE_AQ. Using simulation, we compare the execution times of different partitionings, and discuss the implications of our results.

1.1 Definitions

We use the term "actor" to refer to a primitive data flow operation, and the term "thread" to refer to a collection of actors that naturally execute in sequence. The data flow machines we discuss here bypass matching and unnecessary communication for tokens passed between actors in the same thread. 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 passing of data between two actors within the same thread. Intra-thread communication does not require the use of a matching store. "Inter-thread communication" refers to the passing of data from one thread to another. This will either require the use of a matching store or the startup of another thread, both of which additional time.

We group edges into "enabling groups" and "production groups." Enabling groups are sets of incoming edges. In one enabling group, all edges share a common sink actor. When tokens rest on all edges in an enabling group, that actor can consuming the tokens and begin execution. Several enabling groups can be associated with a given actor.

Production groups are sets of outgoing edges. In one production group, all edges share a common source actor. When an actor completes execution, it sends tokens out on only one production group, one per edge.

We use thick arcs to connect members of enabling groups and production groups. Figure 1 shows an example of this diagrammatic convention.

1.2 The Problem

Data transmission from one actor to another incurs a large time cost when operands must pass through a matching unit. 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, even though fine-grained programs can introduce greater parallelism. Machines with long transmission or matching delays exacerbate the effect [Chan81,Leco81], as shown by both simulations [Gaud85,Huda85] and mathematical constructions [Gaud84].

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

The problem of identifying "unary-output followed by unary-input" constitutes a special case of the problem 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 base our work on a hypothetical tagged-token data flow machine which can execute actors in a thread without sending intermediate tokens through a matching store or generating unnecessary communication delay.

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 Gaudiot), while we attempt to reduce overall communication and matching delays to a minimum. We show how to obtain the set of *all* maximal sequential partitionings of a data flow graph. We often find several alternatives to choose from, each with a different execution time.

Choosing the partitioning with the least execution time is incomputable. Simulation of a "typical" terminating case for the data flow graph provides an execution-time heuristic, measuring the "badness" of different partitionings.

We provide two example programs, then partition and simulate them. We find that even choosing the worst-case partitioning for each example program improves execution

time by a substantial margin.

2 Identifying Sequential Threads

One can partition data flow graphs in several ways. We restrict ourselves to partitioning in a way that preserves all parallelism, but which incorporates as many actors into each thread as possible. To ensure this, 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.

Since input tokens must be available before a thread starts, inherent parallelism in the system could be destroyed by interior actors that require inter-thread data. Therefore, 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.

However, we do allow any actor (i.e., not necessarily the last actor in a thread) to produce output tokens which are sent to another thread. This is easy to implement in hardware. When an actor produces a result, it could be immediately sent to the matching store, for example, rather than wait for the entire thread to complete.

Algorithm PSB partitions a program graph into the largest possible sequential threads according to the above rules. It uses some notation from our discussion of probabilistic data flow graphs [Gree87].

Specifically, 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. $\overline{P}_s^v$ is the set of all enabling groups for vertex v. Therefore, the term $|\overline{P}_s^v| > 1$ 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

1	$S \leftarrow \{v \in V \mid (\exists e \in E, \delta_0(e) \neq -1 \wedge sink(e) = v) \vee \delta_0(v) \neq -1\}$	;	Establish starting nodes.
2	$S \leftarrow S \cup \{v \in V \mid \lvert\overline{P}_v^v\rvert > 1\}$	;	If a pred(v) may not start v, include v.
3	$M \leftarrow S, j \leftarrow 0$	;	Initialize.
4	while $S \neq \emptyset$ do	;	As long as there remains a starting node,
5	select any $s \in S$	;	... select one as this partition's start.
6	$S \leftarrow S - \{s\}, j \leftarrow j+1, k \leftarrow 0, Q \leftarrow \emptyset, Found \leftarrow true$	;	Update S, get next P, initialize.
7	while $Found$ do	;	If we found a good successor, continue.
8	$Found \leftarrow false, k \leftarrow k+1, P_{(j,k)} \leftarrow s, Q \leftarrow Q \cup \{s\}$	;	Go to next vertex, add vertex to P and Q.
9	for all $v \in V$ such that $v \in succ(s) \wedge v \notin M$ do	;	Check each successor node for "goodness".
10	if $pred(v) \subseteq M$	;	Have predecessors been placed?
11	$M \leftarrow M \cup \{v\}$	;	Yes, this one will be too.
12	if $pred(v) \subseteq Q \wedge \neg Found$	;	Are all predecessors in partition?
13	$Found \leftarrow true, n \leftarrow v$	;	Yes. Put v in partition.
14	else	;	
15	$S \leftarrow S \cup \{v\}$	;	No. Put v in start nodes.
16	end if	;	
17	end if	;	
18	end for	;	
19	$s \leftarrow n$	;	Set up to put our successor in P.
20	end while	;	
21	$P_{(j,k+1)} \leftarrow e$	;	Mark end of this partition.
22	end while	;	

Algorithm PSB (Partition Sequential Blocks)

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.

Theorem 1 *Upon completion of Algorithm PSB, two-dimensional array P contains all vertices in V, except unexecutable vertices.*

PROOF. First, we assume that $\forall v \in V, [c \in C \wedge v = end(c)] \Rightarrow [\exists e \in E, end(e) = v]$. By this assumption, we disallow vertices executable solely by input constants. Because vertices driven solely by constants have no clear interpretation, and in some machines they could generate an infinite number of output tokens, we exclude them.

By statement 1, we know that the labels of all vertices that might execute immediately after initiation of the program are in S. Statement 7 executes at least once for every $s \in S$, by statements 4 to 6. No statement in Algorithm PSB removes an element of S, except statement 6. Thus, if at any time during Algorithm PSB's operation, $s \in S$, then by the algorithm's termination $\exists i \in \{1, \ldots, k\}, P_{(i,1)} = s$.

Assume that $a \in V$, and a is executable. We want to show that for any $b \in succ(a)$, where b is executable, that $\exists i, j \in \mathbf{Z}^+, P_{(i,j)} = b$.

Suppose $b \in succ(a)$. Then $\exists e \in E, start(e) = a \wedge end(e) = b$. By statements 3 and 11, M contains all vertices that at some point were members of S, or those included in some previously constructed thread. We know by the above argument that all such elements of S *will* be in a thread. Therefore, if $b \in M$, b is in a thread.

Now consider statement 7. If a can receive a token, at some point the program's s equals a at statement 9. If $b \notin M$, we execute the body of the loop. If $pred(b) \subseteq M$, that is, all predecessors of b are executable and have been placed in P, then by statement 11, b will be added to M, and thus b will be placed in a thread (either immediately at statement 19, or later through statement 15).

We know that all vertices that can execute immediately upon program initiation are present in P. We know that if a vertex is present in P, and successor vertices will also be present in P. So by induction, the labels of all executable vertices are in P. ■

Theorem 2 *Algorithm PSB preserves the ordering of directed graph G.*

PROOF. To prove this, we have to show that no thread P_i in P internally violates the ordering of G. Then we must show that externally, no ordering violations can occur.

By statement 12, a vertex v can only be included in P_i if all $\bar{v} \in pred(v)$ precede it in P_i. Therefore, internally all threads P_i preserve the ordering of G.

Externally, according to the machine's operation, all incoming tokens must be present before a thread begins. Therefore, P_i cannot begin until the external ordering is satisfied. ■

Theorem 3 *Algorithm PSB preserves all parallelism present in the original data flow graph.*

PROOF. Assume otherwise. Then $\exists i, j, k \in \mathbf{Z}^+, j < k$, such that $P_{(i,k)}$ precedes $P_{(i,j)}$ in graph G or no ordering exists between $P_{(i,k)}$ and $P_{(i,j)}$.

Theorem 2 implies that no ordering exists between $P_{(i,k)}$ and $P_{(i,j)}$. But by line (9) of PSB, the *succ* relation orders all vertices in the string P_i. With this contradiction, we proved our theorem. ■

2.1 Execution Times for Different Partitionings

While the use of Algorithm PSB will reduce communication delays the indeterminancy of statement 9 allows several different partitionings of most graphs. In some cases, selecting one partitioning over another will accelerate an algorithm.

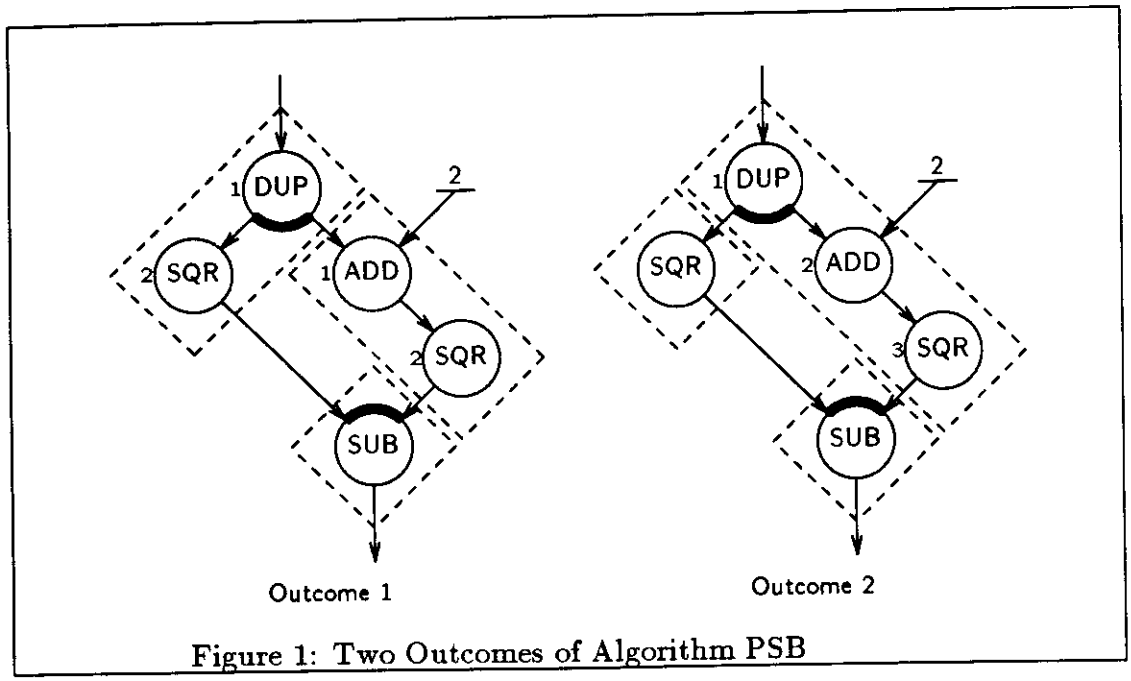

Figure 1: Two Outcomes of Algorithm PSB

In Figure 1, we see two possible outcomes, depending on which $v \in succ(s)$ statement 9 choses when s = DUP. 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 more nearly the same time.

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 and $\mathbf{R}$ is the set of real numbers. If $c \in C$, then $t(c)$ is the execution time of that operation. t_1 is the average time to process intra-thread communications. t_2 is the average time to process inter-thread communications. $t_1 \leq t_2$ because the latter must go through the matching store, while the former need not [Wats82].

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

Ideally, we would generalize this concept, generating an algorithm to choose the optimal set of sequential threads. We shall discover this task difficult.

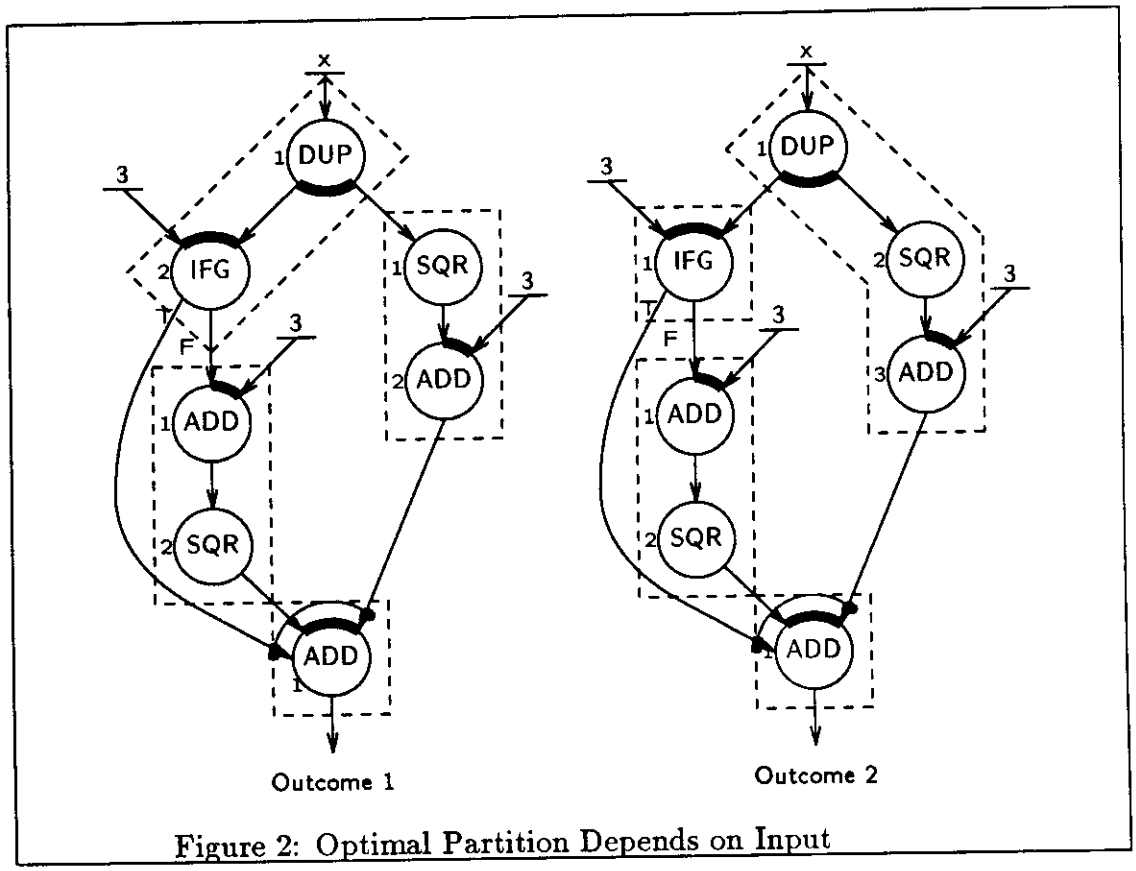

Figure 2: Optimal Partition Depends on Input

Theorem 4 *Partitioning an unevaluated data flow graph into optimal sequential blocks is incomputable.*

PROOF. We prove this theorem by example. Observe the data flow program fragment in Figure 2. There are two possible partitionings under algorithm PSB. Assume the incoming value $x > 3$, and the execution time of each vertex is the same. Then the leftmost partitioning completes earlier. Conversely, with $x \leq 3$, the rightmost partitioning completes earlier.

Thus, the optimal graph partitioning depends solely on the value of incoming value x. But the algorithm used to calculate x can be any data flow algorithm. By [Turi36], predicting the value of x is incomputable. Therefore, partitioning a data flow graph into optimal sequential threads is incomputable. ■

Since partitioning is incomputable by Theorem 4, we must rely on approximations. Applying non-deterministic Algorithm PSB to a data flow graph will result in a set of different sequential partitionings. We must choose one of these.

Partition execution times will differ between partitionings because intra-thread communication time will be less than inter-thread communication times. Using simulation, we can obtain an estimated completion time for each partitioning, and choose the partitioning which completes in the least amount of time.

3 Experiences with Partitioning

To evaluate the utility of sequential thread partitioning, we applied Algorithm PSB to two example programs, obtained sets of sequential threads (partitionings), and executed them using an idealized data flow simulator. In this section, we briefly describe the simulator and present our results.

3.1 The Simulator

Our machine corresponds roughly to the Manchester machine [Gurd85], except in the following ways:

1. Communication plus matching time is given a fixed value for each edge. On a real machine, this time will vary depending on the machine load, and the size of the data tokens.
2. The matching unit can handle operations which require more than two input operands. The Manchester machine matching unit imposes a maximum on the number of input operands: two.
3. The matching unit handles arbitrary multiple "enabling groups." The Manchester machine allows only one enabling group per instruction.
4. Our system does not allow two edges to share the same sink vertex and sink position. We provide an equivalent construct in the MERG instruction.
5. An infinite number of processors are provided. There is no processor contention.
6. No matching-unit or communication contention occurs.

3.2 Instructions Provided

The simulator has an easily extensible instruction set. We implemented only those instructions necessary to simulate our example data flow programs. The instructions provided include:

MERG This instruction allows any number of input edges, each in its own enabling set. There is only one output edge. An input edge will be randomly selected from those which contain tokens, the token absorbed and reproduced as an output token. This operation executes in zero time.

DUP The DUP instruction has one input edge. It duplicates an input token, and outputs it on two output edges.

SUBR The SUBR instruction takes no input edges and allows any number of output edges. It serves simply as a label for a subroutine. The output edges correspond to the parameters of the subroutine. CALL instructions which refer to this subroutine must have the same number of input edges (not including the subroutine name) as the corresponding SUBR instruction has output edges.

CALL The first input edge to a CALL instruction should be a string constant naming the subroutine to be called. The remaining input edges correspond to parameters. There should be the same number of input parameter edges as there are output edges to the corresponding SUBR instruction.

The CALL instruction allows a varying number of output edges. The number of output edges for a particular CALL instruction should equal the number of input edges to the appropriate RET instruction in the called subroutine.

The CALL instruction first obtains a new unique "invocation ID" for the subroutine being called. It saves the invocation ID of its input tokens, the label of the CALL instruction, and the new invocation ID in an "invocation memory." It changes the invocation IDs of its input tokens to the new invocation ID. The CALL instruction then send the tokens out on the output edges of the named SUBR instruction.

RET This instruction takes any number of input edges and produces no output edges. When it receives input tokens, it looks up their invocation ID in the invocation memory. It sets the invocation IDs of its input tokens to the *old* invocation ID. It then produces output tokens on the appropriate CALL instruction.

PLUS, MINUS, TIMES These instructions all take two input operands and produce one output token. Their functions are obvious. If inputs are integers, they produce an integer output. If inputs are real, they produce a real output.

ABS This produces the absolute value of its single input token. If the input is an integer, it produces an integer output. If the input is real, it produces a real output.

OR, AND, NOT These instructions perform corresponding logical operations on their boolean inputs.

BRR This branch instruction takes a number of inputs greater than 1. The first input edge must carry a boolean value. If there are n input edges, the BRR instruction must

have $2(n-1)$ output edges. If the first input edge carries a false value, the tokens on input edges 2 through n will be copied to output edges 1 through $(n-1)$. If the first input edge carries a true value, the tokens on input edges 2 through n will be copied to output edges n through $2(n-1)$.

CGR This "compare greater" instruction compares its two input tokens. If the left is greater than the right, it outputs a true value. Otherwise it outputs a false value.

ADL This "add to iteration level" instruction increments the iteration level number of the left input token by the integer value of the right input token.

SIL This "set iteration level" instruction sets the iteration level number of the left input token to the integer value of the right input token.

STUB This instruction merely absorbs an input value, and produces nothing.

ISERROR This instruction produces a true output value if the input token is an error value. It produces a false output value if the input token has any other value.

The exact input format is described in [Gree87]. Some data in the input language are superflous to the simulator, and are used by a statistical analysis package we have developed.

3.3 Sample Programs

Both example programs were originally written in SISAL, a stream-oriented, Pascal-like applicative language. We used the retargetable SISAL compiler developed by Lawrence Livermore National Laboratory [McGr85] to generate data flow program object, and converted the object to our input format.

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.

3.3.1 Sample Program 1: INTEGRATE

The INTEGRATE program is derived from an example discussed in [Gurd85], and converted to SISAL version 1.2. 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 input for our analysis and simulation programs follows

```
(edge =a 1 0 0.0)
(edge =b 1 0 0.0)
(edge =c 1 0 0.02)
(edge =d 1 0 0.02)
(edge =e 1 -1)
(edge =f 1 -1)
(edge =g 1 -1)
(edge =h 1 -1)
(edge =i 1 -1)
(edge =j 1 -1)
(edge =k 1 -1)
(edge =l 1 -1)
(edge =m 1 -1)
(edge =n 1 -1)
(edge =o 1 -1)
(edge =p 1 -1)
(edge =q 1 -1)
(edge =r 1 -1)
(edge =s 1 -1)
(edge =t 1 -1)
(edge =u 1 -1)
(edge =v 1 -1)
(edge =w 1 -1)
(edge =x 1 -1)
(edge =y 1 -1)
(edge =z 1 -1)
(edge =aa 1 -1)
(edge =ab 1 -1)
(edge =ac 1 -1)
(edge =ad 1 -1)
```

```
(edge =ae 1 -1)
(edge =af 1 -1)
(edge =c0 0 -1)
(edge =c1 0 -1)
(edge =c2 0 -1)
(edge =c3 0 -1)
(edge =c4 0 -1)
(edge =c5 0 -1)
(edge =c6 0 -1)
(edge =s0 0 -1)
(finalvertex *F ((1 =af)))
(vertex *32 NOP 0 -1 () ((1 =a)))
(vertex *33 NOP 0 -1 () ((1 =b)))
(vertex *34 NOP 0 -1 () ((1 =c)))
(vertex *35 NOP 0 -1 () ((1 =d)))
(constantvertex *C0 1.0 ((1 =c0)))
(vertex *0 CGR 1 -1 ((1 =c0 =g)) ((1 =h)))
(vertex *1 DUP 1 -1 ((1 =h)) ((1 =m =o)))
(vertex *2 BRR 1 -1 ((1 =n =f)) ((0.02 =s0)(0.98 =l)))
(vertex *3 BRR 1 -1 ((1 =m =e)) ((0.02 =j)(0.98 =k)))
(vertex *4 DUP 1 -1 ((1 =o)) ((1 =n =p)))
(vertex *5 BRRdt 1 -1 ((1 =p =i)) ((0.02 )(0.98 =q =r)))
(vertex *6 DUP 1 -1 ((1 =q)) ((1 =s =t)))
(constantvertex *C1 0.02 ((1 =c1)))
(vertex *7 ADR 1 -1 ((1 =r =c1)) ((1 =u)))
(vertex *8 MLRd 1 -1 ((1 =s =t)) ((1 =v =w)))
(constantvertex *C2 1 ((1 =c2)))
(vertex *9 ADL 1 -1 ((1 =u =c2)) ((1 =x)))
(vertex *10 ADR 1 -1 ((1 =l =v)) ((1 =y)))
(vertex *11 DUP 1 -1 ((1 =x)) ((1 =ad =ac)))
(constantvertex *C3 0.01 ((1 =c3)))
(vertex *12 MLR 1 -1 ((1 =y =c3)) ((1 =z)))
(constantvertex *C6 1 ((1 =c6)))
(vertex *13 ADL 1 -1 ((1 =w =c6)) ((1 =ab)))
(vertex *14 ADR 1 -1 ((1 =k =z)) ((1 =ae)))
(constantvertex *C4 1 ((1 =c4)))
(vertex *15 ADL 1 -1 ((1 =ae =c4)) ((1 =aa)))
(constantvertex *C5 0 ((1 =c5)))
(vertex *16 SIL 1 -1 ((1 =j =c5)) ((1 =af)))
(vertex *20 MERG 0 -1 ((1 =a)(1 =aa)) ((1 =e)))
(vertex *17 MERG 0 -1 ((1 =b)(1 =ab)) ((1 =f)))
(vertex *18 MERG 0 -1 ((1 =c)(1 =ac)) ((1 =g)))
(vertex *19 MERG 0 -1 ((1 =d)(1 =ad)) ((1 =i)))
(vertex *S0 STUB 0 -1 ((1 =s0)) ())
end
```

Version	*Simulation*
No partitioning	761 cycles
Partitioning # 1	459 cycles
Partitioning # 2	507 cycles

Table 1: INTEGRATE: Simulation Results

Algorithm PSB identified two partitionings for the INTEGRATE program:

```
Partition 1
     (*19) (*6 *8 *13) (*10 *12) (*14 *15) (*5 *7 *9 *11)
     (*18 *0 *1 *4) (*2 *S0) (*3 *16 *F) (*17) (*20)
   Zeroed edges:
     =W =T =S =Y =AE =X =U =R =0 =H =G =S0 =AF =J

Partition 2
     (*19) (*7 *9 *11) (*5 *6 *8 *13) (*10 *12) (*14 *15)
     (*18 *0 *1 *4) (*2 *S0) (*3 *16 *F) (*17) (*20)
   Zeroed edges:
     =X =U =W =T =S =Q =Y =AE =0 =H =G =S0 =AF =J
```

Results of the simulation of INTEGRATE are shown in Table 1.

3.3.2 Sample Program 2: RECURSIVE_AQ

This program is taken from an example shown in [McGr85], a recursive adaptive quadrature program which integrates an arbitrary function, using a supplied stop condition.

We inserted the subroutines necessary to integrate the function $x^2 + 3x - 8$ 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
```

We provide a graphic description for the resulting data flow program in Figure 3.

The results of simulating the RECURSIVE_AQ program appear in Table 2. A detailed listing of the partitions is supplied in [Gree87].

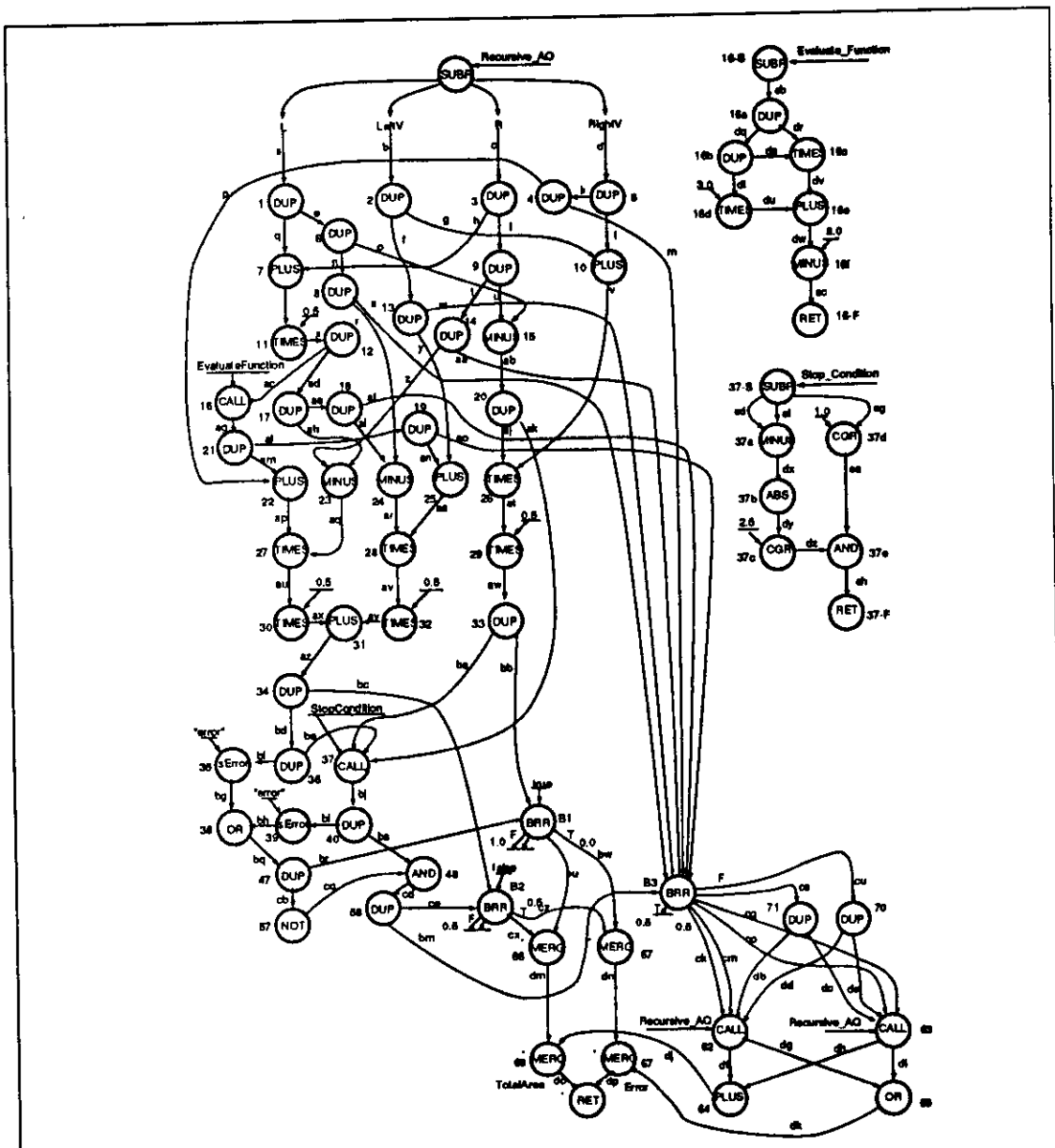

Figure 3: RECURSIVE_AQ: Probabilistic Data Flow Graph

Version	*Simulation*	*Version*	*Simulation*
No partitioning	324 cycles	Partitioning # 33	259 cycles
Partitioning # 01	259 cycles	Partitioning # 34	264 cycles
Partitioning # 02	264 cycles	Partitioning # 35	259 cycles
Partitioning # 03	259 cycles	Partitioning # 36	264 cycles
Partitioning # 04	264 cycles	Partitioning # 37	259 cycles
Partitioning # 05	259 cycles	Partitioning # 38	264 cycles
Partitioning # 06	264 cycles	Partitioning # 39	259 cycles
Partitioning # 07	259 cycles	Partitioning # 40	264 cycles
Partitioning # 08	264 cycles	Partitioning # 41	259 cycles
Partitioning # 09	259 cycles	Partitioning # 42	264 cycles
Partitioning # 10	264 cycles	Partitioning # 43	259 cycles
Partitioning # 11	259 cycles	Partitioning # 44	264 cycles
Partitioning # 12	264 cycles	Partitioning # 45	259 cycles
Partitioning # 13	259 cycles	Partitioning # 46	264 cycles
Partitioning # 14	264 cycles	Partitioning # 47	259 cycles
Partitioning # 15	259 cycles	Partitioning # 48	264 cycles
Partitioning # 16	264 cycles	Partitioning # 49	259 cycles
Partitioning # 17	259 cycles	Partitioning # 50	264 cycles
Partitioning # 18	264 cycles	Partitioning # 51	259 cycles
Partitioning # 19	259 cycles	Partitioning # 52	264 cycles
Partitioning # 20	264 cycles	Partitioning # 53	259 cycles
Partitioning # 21	259 cycles	Partitioning # 54	264 cycles
Partitioning # 22	264 cycles	Partitioning # 55	259 cycles
Partitioning # 23	259 cycles	Partitioning # 56	264 cycles
Partitioning # 24	264 cycles	Partitioning # 57	259 cycles
Partitioning # 25	259 cycles	Partitioning # 58	264 cycles
Partitioning # 26	264 cycles	Partitioning # 59	259 cycles
Partitioning # 27	259 cycles	Partitioning # 60	264 cycles
Partitioning # 28	264 cycles	Partitioning # 61	259 cycles
Partitioning # 29	259 cycles	Partitioning # 62	264 cycles
Partitioning # 30	264 cycles	Partitioning # 63	259 cycles
Partitioning # 31	259 cycles	Partitioning # 64	264 cycles
Partitioning # 32	264 cycles		

Table 2: RECURSIVE_AQ: Simulation

4 Conclusion

A cursory glance at Tables 1 and 2 makes it clear that, at least for these examples, selecting a particular partitioning makes only a small difference in the execution time. For examples 1 and 2, the best-case partitioning execution time beats the worst-case partitioning by 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 grows exponentially with the number of multiple-successor nodes. The decomposition methods of [Kape87,Kape88] can address this problem to some extent. However, our experience seems to indicate that (since worst-case and best case times do not differ substantially) using a cheap heuristic to reduce the number of partitions generated by Algorithm PSB would provide an efficient data flow optimization scheme. We are currently looking into alternative heuristics and search-space reduction schemes for Algorithm PSB.

References

[Chan81] T.L Chang and P.D. Fisher, A Block-Driven Data-Flow Processor, pp. 151–155, in *Proceedings of the 1981 International Conference on Parallel Processing*, Columbus, Ohio (1981).

[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).

[Gree87] D.R. Greening, *Modeling Granularity in Data Flow Programs*, Master's thesis, UCLA, Los Angeles, California (1987), 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).

[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).

[Kape87] A. Kapelnikov, R.R. Muntz, and M.D. Ercegovac, A Methodology for the Performance Evaluation of Distributed Computations, in *Proceedings of the IFIP Conference on Distributed Processing* (October 1987).

[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).

[Leco81] M. Lecouffe, Architecture of a Multiprocessor Using Data Flow at a Program Block Level, pp. 160–161, in *Proceedings of the International Conference on Parallel Processing*, Columbus, Ohio (1981).

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

[Turi36] A.M. Turing, On Computable Numbers, with an Application to the Entscheidungsproblem, pp. 241–265, in *Proceedings of the London Mathematical Society, Series II* (December 1936).

[Wats82] I. Watson and J.R. Gurd, A Practical Dataflow Computer, *Computer*, 15(2):51–57 (February 1982).