(4)

CCS Research Report No. 630

Communication-Efficient Arbitration Models for Low-Resolution Data Flow Computing

by

Camille C. Price*
A. Charnes

December 1988

*Visiting Scholar
Permanent address: Department of Computer Science
Stephen F. Austin State University
Nacogdoches, Texas 75962-3063

This research was partly supported by ONR Contracts N00014-81-C-0236 and N00014-82-K-0295, and National Science Foundation Grants SES-8408134 and SES-8520806 with the Center for Cyber-netic Studies, The University of Texas at Austin. Reproduction in whole or in part is permitted for any purpose of the United States Government.

CENTER FOR CYBERNETIC STUDIES

A. Charnes, Director

College of Business Administration, 5.202
The University of Texas at Austin
Austin, Texas 78712-1177
(512) 471-1821

COMMUNICATION-EFFICIENT ARBITRATION MODELS FOR LOW-RESOLUTION DATA FLOW COMPUTING

Abstract

Low-resolution data flow computing offers a practical compromise between conventional control-flow computing models and the specialized architectures required for fine-grain data flow processing. We give a formal specification of an arbitration facility that simultaneously partitions and statically assigns operations to processors. This general model is based on differences in the processors, diversity of data links in the network, size of tokens flowing between nodes in the data flow graph, memory limitations on the processors, and considerations to promote parallelism. A network model solves the static problem for bipartite and tree structured data flow graphs. Based on this centralized static allocation scheme, data tokens are automatically routed to processors, and the run-time scheduling process is distributed among the processors. Dynamic arbitration implemented as a centralized facility takes inadequate advantage of network capabilities. A general decentralized (distributed) dynamic arbitration scheme maps tasks to processors at run-time, with the association of tasks to processors based on intertask communication and network data link characteristics. Task migration is supported by treating both data and code as tokens. No centralized control or mass storage are required in this communication-efficient arbitration model.

Keywords: Data flow, Large-grain data flow, Scheduling, and Task allocation.

1. Introduction

The concept of data flow computing has received considerable attention in recent years inasmuch as it provides an alternative to the traditional control flow model of computing. In the data flow model, a computation is scheduled to occur as soon as all of its necessary inputs are available, thus there is the potential for the exploitation of concurrency (parallel processing) even in a program designed for the more conventional, sequential control flow environment.

Data flow considerations have attracted the attention of researchers at various levels of hardware and software design. Data flow systems may be characterized by the granularity of their operations and data. Specialized computer architectures have been developed to implement high-resolution (fine-grain) data flow. At this level, individual machine instructions are executed as their operands become available. The assumption underlying efforts at this level is that *sufficient improvement in overall execution time of a program can be made* to outweigh the overhead required by the unconventional hardware supporting such a data-driven system.

At the other extreme, the concept of executing higher-level functions or procedures in response to the availability of data is referred to as low-resolution (coarse-grain) data flow computing. At this level, specialized hardware is not a critical issue; rather, operating systems must be designed to schedule the execution of procedures and to manage the flow of data to serve as inputs to the appropriate procedures at the proper time.

Lower resolution implies greater operation complexity (e.g., operations may be routines such as sorting or matrix inversion); and the data to be routed from one node to another may be arbitrarily large data structures (such as lists or matrices). Low resolution extracts less potential concurrency from a computation.

Nevertheless, many of the advantages of data flow still accrue and can be achieved with minimal architectural support and scheduling overhead.

The arbitration facility within a data flow system is that capability which matches data inputs to the computation requiring them, and places inputs and instructions on a processor which can perform the computation. Effective arbitration should include a consideration of the communication pattern among the tasks that comprise a program and the topology and link characteristics of the underlying processor network. In this paper, we develop models for efficient arbitration in a low-resolution data-driven computational environment in which system performance is sensitive to interprocessor communication overhead.

2. The Development of the Data Flow Concept of Computing

Although recent interest in data flow computing has centered around the use of multiple processors to effect parallel computation, the most rudimentary data flow designs were based on single processor architectures in which parallelism was achieved through instruction pipelining. Such a data flow processor is designed to recognize which of the instructions in its program memory are "enabled" (or "ready to fire" or "have their inputs available"), and to select and dispatch an enabled instruction to an execution unit. Instead of a single locus of control (maintained as a Program Counter), there is a concurrency (or arbitrary sequencing) of certain instructions that may be exploited, subject to availability of resources.

In such an environment, it is clear that "concurrent" execution can be extended to truly "simultaneous" parallel execution of operations if there are multiple processors in the system. The resulting parallelism has the potential then to yield greatly reduced program completion times, which is the primary goal of all

investigation into data flow computing. Consequently, most of the current data flow work is for data flow multiprocessors.

The program being executed in a data flow computer is represented not as a traditional sequential computation but rather in the form of a **data flow graph** DFG (V, E). Each node in V represents an operator or a computation specified by one or more instructions; and arcs in E represent data dependencies (and therefore also precedence constraints) between nodes. Tokens are dealt with purely as data **values** rather than as **addresses**; thus, the model permits no shared memory, and implementation is most easily accomplished in a message-passing network of processors. This form of representation has the advantage that it does not impose sequencing constraints other than those dictated by data dependencies in the algorithm; thus, all possible parallelism in the algorithm is exposed.

High level programming languages, such as LUCID [Rasmussen et al], VAL [Ackerman and Dennis], and FPL [Ercegovac et al, Backus], have been created to facilitate the development of software for data flow computing. The compilers for these languages generate a data flow graph which is then executed (or interpreted) by the data flow machine. The concept of a language which deals with **values**, rather than variable names and **addresses**, is the foundation of **functional**, or applicative, languages; and the absence of shared memory in this class of programming languages makes data flow programs free of the "side effects" which are the source of major difficulties in conventional concurrent systems. Each data flow operator uses only its own private operand tokens and produces new operands to be used by other operators.

In a multiprocessor data flow computer, performance is strongly affected by the way in which operations (tasks) are allocated among the processors. If the association between operations and processors is made before run-time (as part of

the compilation of the program), this is referred to as **static allocation**. Static allocation avoids the time-consuming assignment process during run-time and therefore can speed up the execution of the program. If sufficient information is not available in advance, or if it is desired to take advantage of information about the program that is revealed only at run-time, then **dynamic allocation** of operations takes place when each operation is ready to execute.

General Data Flow

Comprehensive historical perspectives on the development of data flow computing over the past decade may be found in [Agerwala and Arvind], [Denning], and [Veen]. We cite here the work which seems to be the most relevant to our efforts. High-resolution data flow has been thoroughly investigated by [Dennis] at MIT and at the University of Manchester [Watson and Gurd] using a tagged token architecture. Low-resolution data flow principles are less clearly defined, but nevertheless apparently hold an attraction for practical reasons to many of the researchers mentioned below.

A pure low-resolution data flow system with identical CYBERPLUS processing nodes [Babb et al] avoids the allocation problem by placing the entire application program in each processor and broadcasting all results to all processors. Sequencing is done by a distributed scheduling mechanism at run-time. The duplication of code and result packet transmission represents an overhead that would not be tolerated in most data flow systems.

Because the advantages of data flow seem obvious, but universally optimal methods for managing data flow systems have clearly not emerged, several researchers have developed hybrid systems with data flow constituents [Broy]. A combined data flow and control flow model is described in [Carlson and Fortes]. Dynamic allocation in a hybrid data-driven and demand-driven environment is

described by [Jagannathan and Ashcroft]. Combined models to be used for real-time high-speed data gathering are given by [Barkhordarian], but the data flow elements of this system are somewhat weakly developed, and static allocation of operations to processors is not done automatically but manually by the programmer. A combination vector processor and low-resolution data flow processor as described in [Requa and McGraw] is designed to allow arbitrary forms of concurrency for a variety of applications on heterogeneous processors.

Some data flow systems have been developed for special applications. A pragmatic approach for balancing and optimizing large-scale regularly-structured scientific programs, expressed as data flow programs, in systolic systems of identical processors is given by [Rong]. Low-resolution static data flow is applied to signal processing problems in which parameters were known *a priori*, as described in [Lee and Messerschmitt].

Performance analysis of static data flow for large array operations is done by [Levin]; and a model of the maximum degree of parallelism in a static data flow computer is given in [Gui-zhong et al].

Communication Considerations

Interprocessor communication overhead is an inevitable consequence of the use of multiprocessors to execute related computations. [Gaudiot and Ercegovac] make the observation that minimization of communication is still on unsolved issue in data flow computing just as it is in general multiprocessor systems. In the absence of analytical tools to measure this performance cost, they use simulation studies of data flow to calculate the effects of communication in a specialized (ring-structured) network.

[Dennis, 1979] noted the problem of partitioning instruction sets to avoid the high cost of routing results, relative to the lower cost of forwarding results locally

within a processor. The MIT architecture for fine-grain processing treats this problem at the hardware level so that each instruction is equally accessible to result packets.

A study of the cost-effectiveness of critical path scheduling [Lloyd] in the MIT data flow computer is given by [Granski et al]. Since in practice there may not be enough idle processors to execute all fireable instructions, priorities are assigned to instructions according to a modified critical path algorithm. Allocation is determined by a centralized scheduler at run-time, and instructions and data packets are sent in response to requests from idle processors. The algorithm simplistically assumes a zero-delay in the communication network, and simulations are based on a fixed delay. (It is concluded that critical path scheduling does not pay off except in highly regular programs having few conditionals.)

[Ercegovac et al] deal with the communication problem through software in a static data flow system. DFG partitioning and allocation occurs in three phrases: 1) coalesce basic blocks of code containing only sequential code; 2) when communication time delays exceed the gain from parallelism, coalesce adjacent blocks (precluding parallelism); and 3) assign nodes in the new DFG to a partition based on critical paths. This is representative of the very few bonafide efforts to manage communication overhead, but still no consideration of network data link characteristics is made.

In order to reduce routing delays in a fine-grain data flow multiprocessor, [Hong et al] perform a static allocation by partitioning the DFG into tree-like structures, then map communicating computations onto adjacent processors where possible.

The Hughes Data Flow Machine [Campbell] is a fine-grain machine intended as an embedded signal and data processor. Automatic static allocation of nodes is based on a breadth-first ordering of DFG nodes, then graph partitioning

taking into account communication with upstream nodes. The composite heuristic allocation scheme aims to locally minimize communication cost and maximize parallelism, and can be adapted to specific network topologies.

A comprehensive discussion of static allocation criteria and assumptions (and in particular interprocessor communication issues) is given by [Ho and Irani]; however, their graph partitioning strategy assumes a uniformly connected processor network and infinite memory at each processor, and therefore, cannot be implemented in real systems.

The MAX data flow architecture [Rasmussen et al] is designed to be implemented in radiation-hard, space qualified technology, to support applications ranging from simple instrument controllers to signal processors and complex robotics systems. The technique chosen for these requirements is low-resolution data flow with traditional von Neumann processing elements as the building blocks. Not only data but also the code which implements operations are treated as tokens, thus the migration of code and data can be dealt with by the same mechanism. Dynamic allocation of large-grain tasks to processors is to be performed in a distributed manner, using a "token locality" goal, with each processor having knowledge of the DFG structure. Further details of the MAX system have not been specified; however, the realism in the requirements and the feasibility of implementing the hardware with current technology have provided much of the motivation for the models to be presented in Sections 3 and 4.

Typical assumptions upon which many of the referenced allocation and scheduling schemes are based include:

- uniform execution times for all operations
- identical processors
- constant number and size of operands (tokens)
- unlimited processor memories

• fully-connected (uniform) processor interconnection network

Clearly these are inappropriate assumptions for general multiprocessor networks, and particularly so for low-resolution data flow systems in which instruction code and data tokens are likely to be of arbitrary size, reflecting the nature of the application. We will develop a more general model that is appropriate for arbitration in low-resolution data flow multiprocessor systems.

3. Communication-Efficient Arbitration: Static Model

We have seen several static allocation strategies [Ercegovac et al, Hong et al, Campbell] which first partition a DFG into subsets of nodes, then map the subsets onto processors which will execute the operations corresponding to the nodes. When partitioning the DFG, the goal is typically to cluster nodes so as to minimize communication among nodes belonging to different clusters, giving also some con-sideration to critical paths in the DFG that govern the degree of parallelism.

It is indeed a difficult problem to simultaneously minimize interprocessor communication costs and maximize parallelism. In fact, these are conflicting goals since total execution length is minimized by dispersing operations among the processors while total communication costs are minimized by clustering operations on as few processors as possible.

Furthermore, the simple partitioning schemes seen in the literature do not consider the disparate distances between pairs of processors in an arbitrary network. The partition is made on the basis of the inter-node communication in the DFG. But the fact is that it is not possible to know the real cost of a graph partition and allocation until we also know the data link cost that applies to each cut in the partition. By separating the processes of partitioning and allocation, all hope of achieving an optimal placement of operations on processors is abandoned and

ignored. We present a mathematical programming model that addresses all of the goals which should govern a static allocation. The following notation will be used:

m = number of operation nodes in the DFG

s_i = size of operation i (amount of memory required)

e_i = expected execution length of operation i (number of instructions)

c_{ik} = amount of communication units from operation i to operation k

= weight on arc from i to k in the DFG

n = number of processors, $n < m$

b_j = capacity of memory (buffer) at processor j

R_j = speed of processor j (average time per instruction)

d_{jr} = distance (cost per data unit) for data link from processor j to processor r

The decision variable $x_{ij} = 1$ means operation i is assigned to processor j, and is zero otherwise. The problem is to find a partition of DFG and a mapping of the operation subsets onto the processors to find

$$\min \sum_{i=1}^{m} \sum_{j=1}^{n} e_i R_j x_{ij} + \sum_{i=1}^{m} \sum_{k=1}^{m} \sum_{j=1}^{n} \sum_{r=1}^{n} c_{ik} d_{jr} x_{ij} x_{kr}$$

subject to

$$1) \sum_{i=1}^{m} s_i x_{ij} \leq b_j \qquad \text{for } j = 1, \ldots, n$$

$$2) \sum_{j=1}^{n} x_{ij} = 1 \qquad \text{for } i = 1, \ldots, m$$

$$3) \; x_{ij} = 0, 1$$

The linear terms in the objective serve to reduce the finish time of the program by placing the longer tasks on faster processors. The quadratic terms serve to minimize the cost of the partition of the DFG but more particularly to do so with regard to the data link costs corresponding to the intercluster communication amounts.

The first constraint realistically recognizes memory limitations at each processor, and includes in the size of each operation the amount of buffer space that will be required for its output tokens. This is essential in low-resolution data flow since there is not necessarily a centralized token queue or shared result memory.

The second constraint assures that each operation is assigned to only one processor. (If this equality were relaxed to a $\geq$ inequality, the resulting multiple copies may have the effect of reducing communication costs, but now additional control mechanisms would be necessitated to select **which** copy to execute at the required time. In addition, the linear objective terms would no longer accurately reflect the execution cost.)

The static arbitration model stated above as a zero-one quadratic programming problem is NP-complete [Garey and Johnson], and therefore, has no apparent efficient solution. Methods can be devised to solve small instances of the problem or special cases optimally, but heuristics will have to be relied upon for solving general problems of practical size in a reasonable amount of time.

Some observations are in order concerning practical approaches for dealing with the conflicting goals of minimizing execution and communication costs and achieving a high degree of parallelism. Several data flow system designers [Campbell], [Ercegovac, Chan and Ravi], [Ho and Irani], deal with the arbitration problem in two phases: 1) first partitioning the DFG to minimize communication

among the nodes (operations), then 2) allocating subsets of nodes to processors. The first phase can be formally described as follows:

Graph Partitioning Problem

NP-complete: (Garey & Johnson)

Given graph G = (V, E), weights w (v) for each $v \in V$ and L (e) for each $e \in E$, and positive integers B and J, find a partition of V into disjoint sets $V_1, V_2, \ldots, V_n$ such that $\sum_{v \in V_i} w(v) \leq B$ for $1 \leq i \leq n$ and such that if E' contained in E is the set of edges that have their two endpoints in two different sets V_i, then $\sum_{e \in E'} L(e) \leq J$.

This two-phase approach is valid with respect to our objective function, only under certain conditions. We must assume identical processors and unit tasks; that is, $s_i = s$ and $e_i = e$ for all operations i, and $b_j = b$ and $R_j = R$ for all processors j. We must furthermore be willing to specify *a priori* the number n of subsets for the partitioning. The value of n may be as large as the number of available processors, or as small as $\lceil m / \lfloor b/s \rfloor \rceil$, filling a few processors to capacity. Under these conditions, if we optimally solve the Graph Partitioning Problem and optimally map the reduced DFG onto the processor network (which can be done since these two graphs are isomorphic), then we have minimized $\sum_i \sum_k \sum_j \sum_r c_{ik}\, d_{jr}\, x_{ij}\, x_{kr}$.

During the second phase, allocating subsets to processors, as we observe the data link cost values d_{jr}, we realize that we might have lower total costs if we had chosen a different number of subsets or less well balanced subsets. Thus, partitioning the DFG into a pre-determined number of subsets, without knowledge of the cost d_{jr} of an intercluster communication, may lock us into a sub-optimal

result before the mapping phase even begins, as is shown in the following example.

Consider the DFG in Figure 1 with six unit operations and a network of three identical processors. In an optimal partition into three subsets, we find that the cost of the partition is

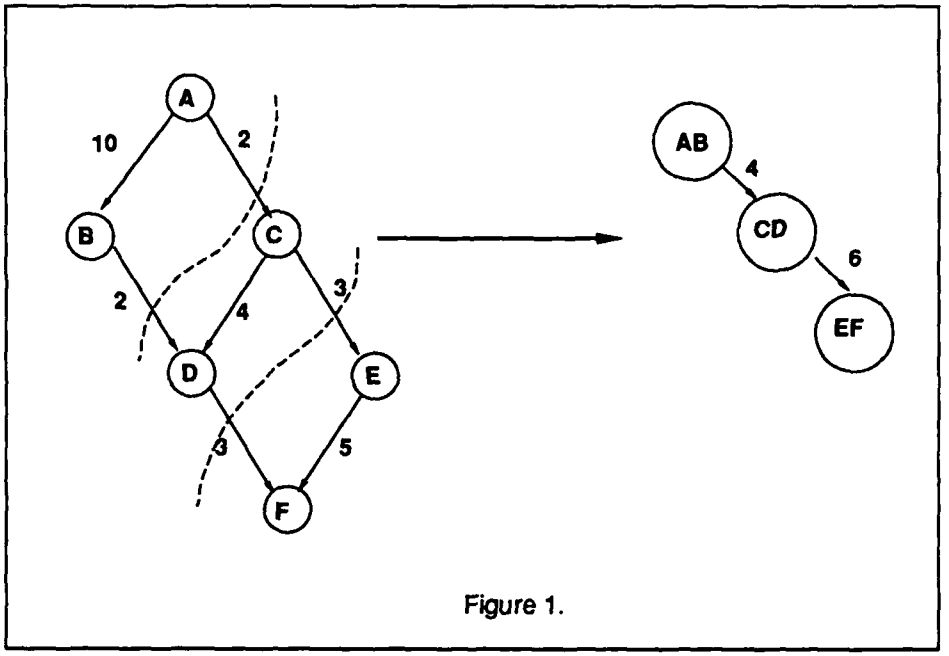
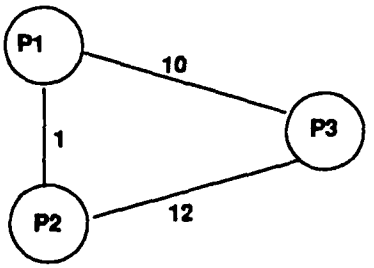
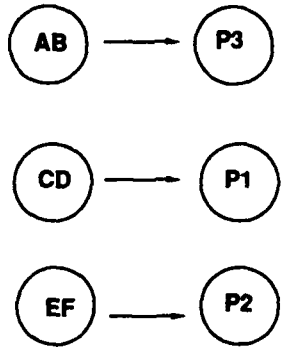


Figure 1.

2 + 2 + 3 + 3 = 10. If this reduced graph is mapped onto a network such as the following:

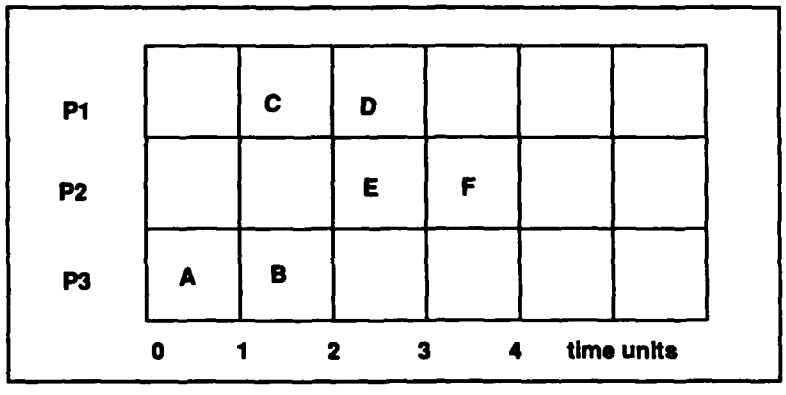


the best we can do is to map the graph edge with weight 6 onto the data link with cost 1, and the graph edge with weight 4 onto the data link having cost 10. Thus



and the cost of this mapping is $6 \cdot 1 + 4 \cdot 10 = 46$.

The execution length for this configuration is 4, as shown in the Gantt chart:



However, since processor 3 is separated from processor 1 and 2 by high-cost data links, it may be beneficial to use only processors 1 and 2. Indeed, partitioning the DFG into two subsets, we obtain the reduced graph shown in Figure 2, and the cost of the partition is

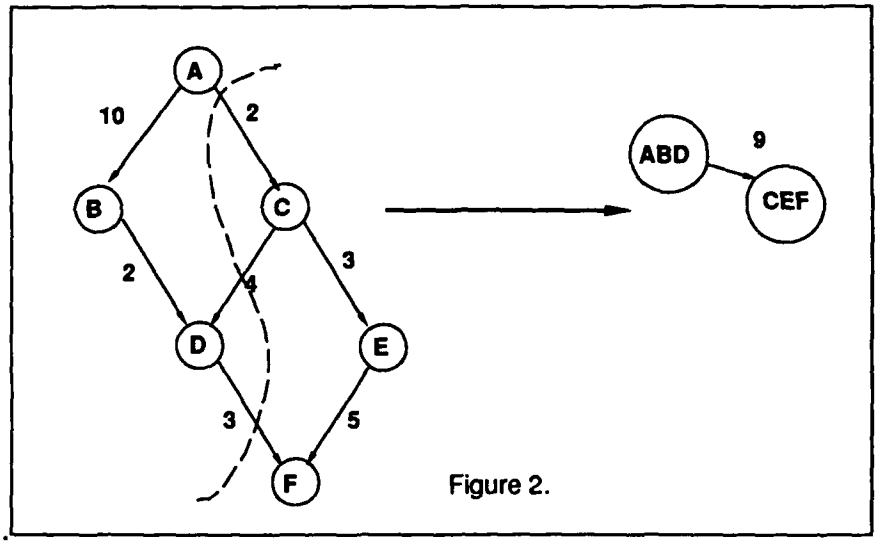
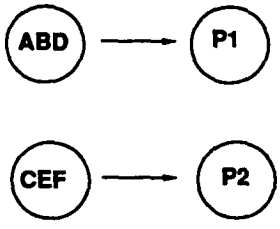
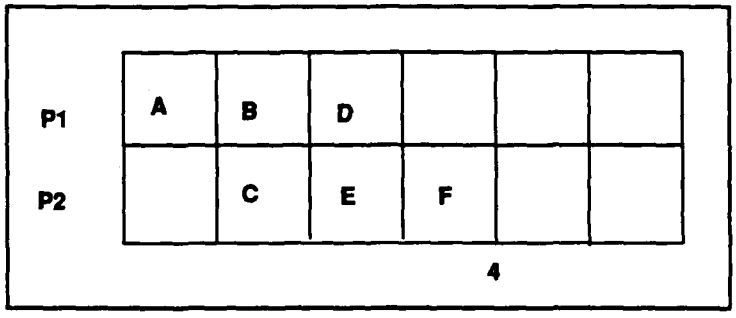


Figure 2.

2 + 4 + 3 = 9. Mapping these two subsets onto a subnetwork:



we have a mapping whose cost is 9 · 1 = 9. Furthermore, the execution length 4 does not suffer from the reduction in the number of processors:



Due to the computational complexity of both the graph partitioning and the mapping problems, data flow system designers have relied on heuristic solutions for these two problems. [Ercegovac et al] use a partitioning heuristic based solely on DFG structural characteristics, followed independently by allocation heuristics which take network structure into account. [Campbell and Ho & Irain] treat partitioning and allocation simultaneously, but do so by making only one pass through the DFG and considering only the communication with predecessor nodes. These designers do not address the optimal software allocation issue in the full sense expressed by our model stated earlier.

The static allocation problem can be represented as a Generalized Network Model, and this model permits efficient solution to the problem when the DFG is either a bipartite graph or a tree. Suppose DFG is a bipartite graph, $V = V_1 \cup V_2$, and $V_1 \cap V_2 = \varnothing$. The primary nodes are of the form (ij) representing the assignment of operation i to processor j. Nodes corresponding to operations in subset V_1 are banked at the left, and nodes for operations in V_2 are banked at the right. If $i \in V_1$ and $k \in V_2$, then an arc connecting corresponding nodes has a cost (reflecting linear and quadratic costs), and a flow across this arc means operation i is assigned to processor j **and** operation k is assigned to processor r:

Nodes and arcs between the source $\mathcal{S}$ and nodes in the left bank, and between nodes in the right bank and the sink $\mathcal{T}$, serve to enforce the constraints requiring that each operation be assigned to exactly one processor.

Since each operation in the left bank may, in a given problem instance, communicate with **any number** of operations on the right, there must be sufficient flow exiting from a left bank node to correctly measure the cost of such communication. For this we introduce nodes and arcs, as suggested in [Charnes, . . . , Lovegren, . . . , Wolfe et al]. For any left node (ij), we attach a "hose arc" with a multiplier or network gain value of g_i, where g_i = the number of operations with which operation i communicates, leading to a "gain node" (gij). From gain node, g_{ij}, we have exactly g_i "communication control arcs", to carry exactly one unit of flow to each of the operations k with which operation i communicates. "Nozzle nodes" then lead to "spray arcs" which discriminate among the n possible assignments of operation k. This sub-assembly is shown in Figure 3.

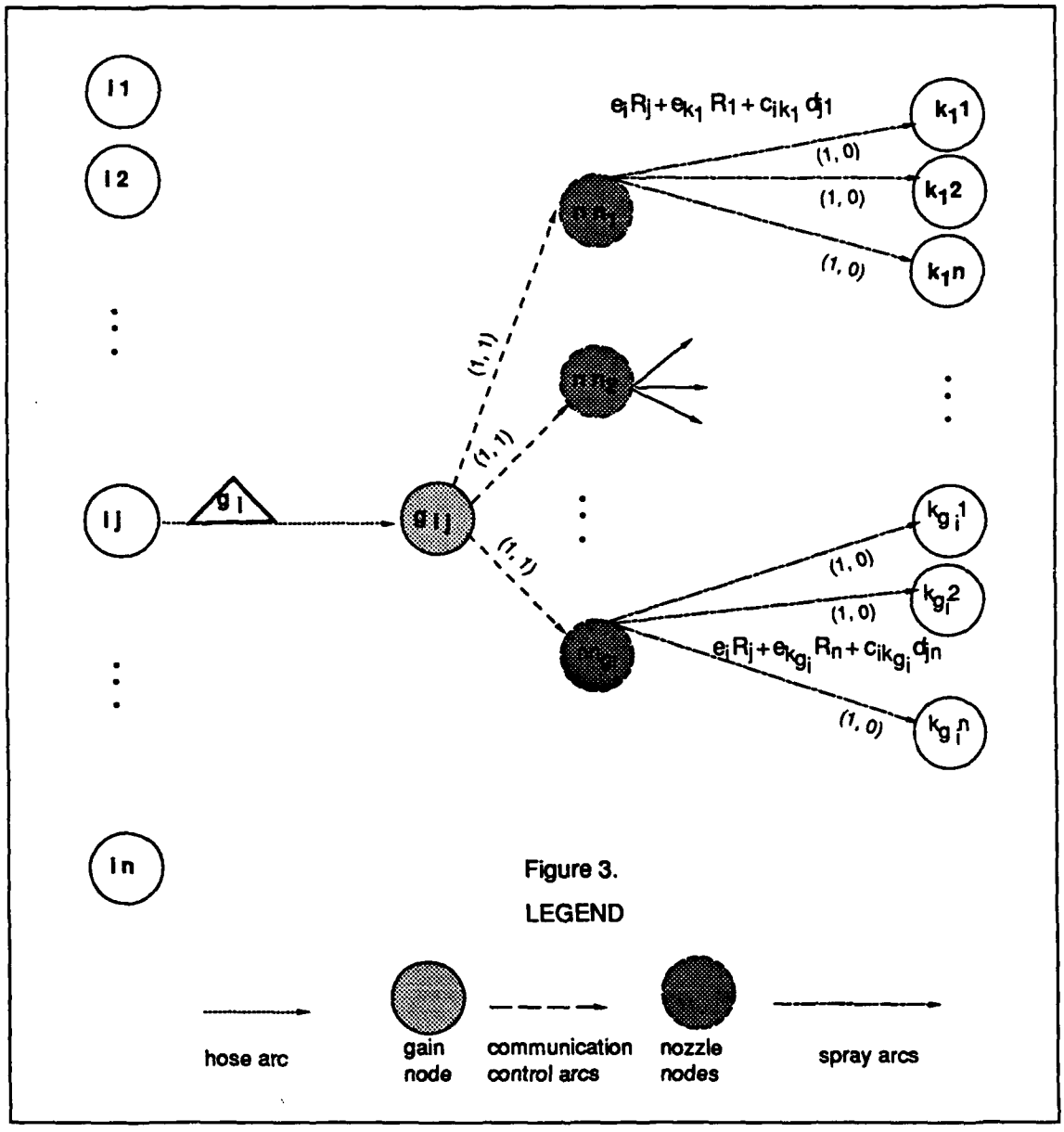


Figure 3.

LEGEND

The full network is shown in Figure 4, with enough detail to allow an understanding of the model. This network model for a complete bipartite graph, where $|V_1| = |V_2| = m/2$, has

$(\frac{m^2n^2}{4} + \frac{m^2n}{4} + \frac{3mn}{2} + m)$ arcs and $(\frac{m^2n}{4} + \frac{3mn}{2} + m)$ nodes.

A similar network model can be developed for a DFG having tree structure. Instead of two banks of primary nodes, there is a bank for each level in the tree. Because of the strict tree structure, flows of one are preserved for each operation, and network structures on the right to limit the number of assignments for each operation are not necessary.

The network model does not extend to arbitrary DFG structures. The difficulty arises in operations which receive communication from more than one other operation. Such nodes require an assignment limit specifying that the operation be assigned to exactly one processor, but as seen in Figure 5, once the limit is imposed, the unique assignment is lost and cannot be recovered to measure the correct quadratic cost.

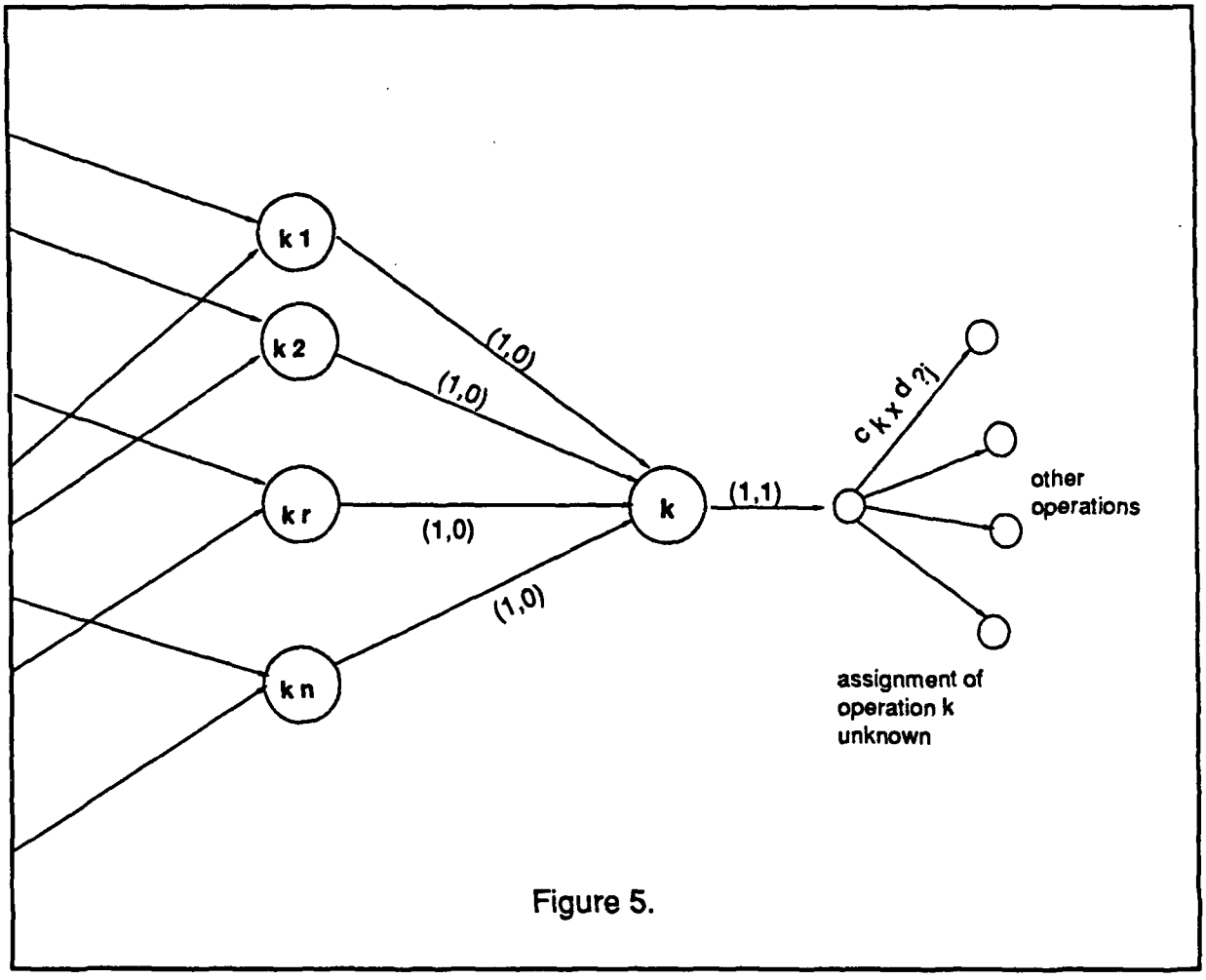

k 1
k 2
k r
k n
k
(1,0)
(1,0)
(1,0)
(1,0)
(1,1)
c k x d ?j
other
operations
assignment of
operation k
unknown

Figure 5.

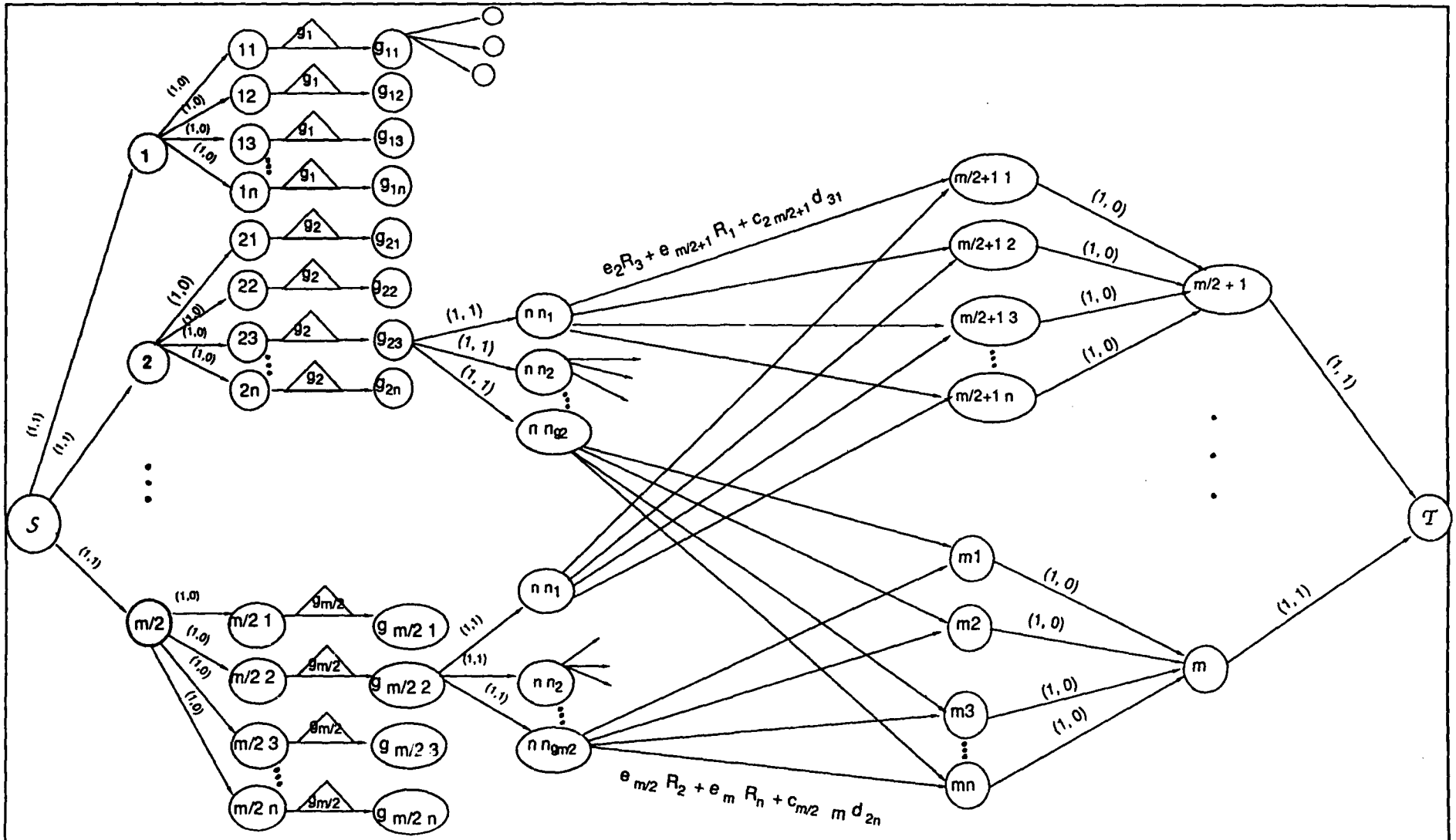
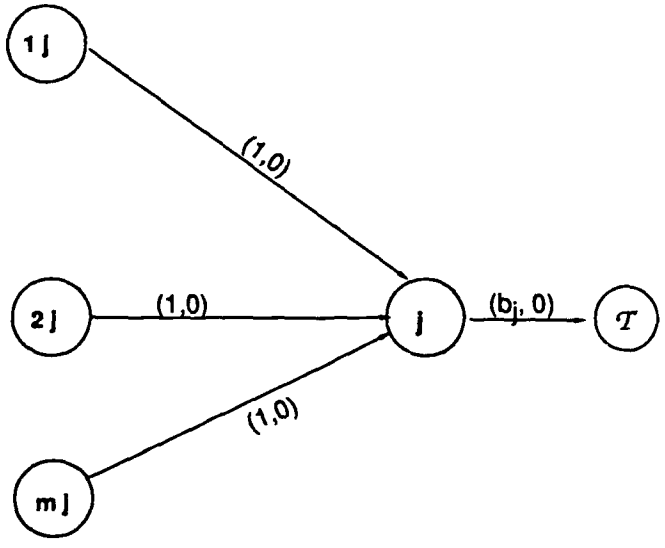


Figure 4. Generalized network model for static allocation of a bipartite data flow graph

We are able to handle multiple-receiver nodes only because the bipartite structure provides a means of placing assignment limits on both banks of primary nodes. (In the case of trees, there are no multiple-receiver nodes.)

In order to impose the constraints on the number of operations which can be assigned to each processor, additional network structure is applied on top of that shown in Figure 4. Suppose $s_i = 1$ for all operations i. Then for each processor j, j = 1, . . . , n, add the following:



Any flow through nodes 1j , . . . , m_j accumulates and is subject to the upper bound of b_j on processor j. Although this is not a standard network structure, a computationally efficient preprocessing of the network allows for the creation of "supplies" of operation weights and the imposition of the bounds b_j [Wolfe].

Such a static allocation is made as a part of the compilation process, and the code comprising each function is loaded into the appropriate processor before run-time. Since the mapping process is complete before any operation executes, the actual physical

destinations of the outputs produced by all operations are known to the operations. During execution time, data tokens are automatically routed by the communication network to the (pre-established) destination processor as soon as they are produced. Ready tokens then reside in local queues within each processor.

Operation scheduling is distributed (decentralized) among all the processors. Whenever a processor becomes idle, the local arbitration facility must determine for which (if any) resident operation all required input tokens are in the token queue. Any such operation is eligible (enabled), but in case there is more than one enabled operation, ties are broken by choosing the most "critical" operation.

Let us now define the notion of a "critical" operation. Recall that parallelism is an important goal in data flow computing. Some systems deal with this goal by balancing (leveling) the computational load among the processors. However, we note that load balancing may be counterproductive since it increases communication costs between operations. In any case, the finish time for the entire program cannot be less than the length of the critical path (described below) in the DFG; therefore, it is not cost-effective to do any more than to spread the load only to the extent necessary to allow the most heavily loaded processor to finish in an amount of time not exceeding the length of the critical path. Completely uniform loads would not necessarily serve this purpose.

The length L of the critical path in an (acyclic) DFG is determined according to the following algorithm.

Critical Path Algorithm

Create (if necessary) a single initial node and a single final node in the DFG. An artificially added node i has $e_i = 0$.

Label the final node with its execution length e_i. Any node i is eligible for a label if all its successors are labelled, and a label is computed as

$$\text{label}(i) = \max_{\substack{\text{all successors } j \\ \text{of node } i}} \{\text{label}(j)\} + e_i \}$$

Let L be the label on the initial node. The label on each node is a lower bound on the amount of time required for the DFG program to complete once this operation is ready to execute. (The lower bound is tight if there are no communication costs.) Nodes with larger labels are the *most urgent* with respect to program completion time. The DFG may be pre-processed with the Critical Path Algorithm before any static or dynamic arbitration algorithm is applied.

Returning to the context of operation scheduling, nodes with larger labels are executed first. Thus the critical path labels give rise to a priority list, and the resulting scheduling scheme belongs to the class of **list scheduling algorithms,** whose characteristics are described in [Coffman and Denning]. Schedules produced in this way are of no more than twice the optimal length with respect to execution times [Coffman, Polychronopoulous].

The static arbitration model is completely general and applies to any processor network configuration and arbitrary operations and output tokens. It is the only model that has been proposed that simultaneously considers node

communication amounts **and** the data link costs that apply to these communications.

4. Communication-Efficient Arbitration: Dynamic Model

Dynamic allocation and scheduling avoids the pre-processing required for static arbitration, but incurs the expense of substantial decision-making at run-time. Clearly it is not feasible to optimally solve a large quadratic assignment problem during application program execution. Instead we resort to heuristic procedures that make reasonably good decisions quickly. In a sense, the minimization of scheduling overhead becomes a primary objective if the application program is to execute efficiently.

We assume that nodes in the DFG are not clustered into subsets; each node (operation) may be individually assigned to any processor.

Centralized Model*Since operations are not distributed among processors* initially, a **centralized instruction memory** must be provided in order to store the code corresponding to all the DFG nodes. There is also a **centralized token queue** for data tokens. The **centralized arbitration facility** matches tokens to operations and, when all tokens for a particular operation are found in the token queue, an instruction packet containing data and code is created and routed to an available processor. (A processor is deemed "available" if it is not currently executing an operation and its output buffer is empty, indicating the completed transmission of the data results produced by the previous computation.)

Three cases arise:

1) If there are multiple instruction packets ready for one available processor, the selection of an operation is made according to critical path labels.

2) If there are multiple processors idle and available for a single instruction packet, the selection of a processor is made according to a combination of the following criteria, which can be independently weighted:

a) Choose that processor r with minimum data link distance d_{cr} from the centralized instruction memory processor c, to locally optimize communication costs.

b) Choose the fastest processor, i.e., with the smallest R_j (to minimize the linear terms in the objective function).

3) If there are multiple operations **and** processors, select the most "critical" operation, and apply case 2).

This arbitration scheme is reminiscent of arbitration networks in high-resolution data flow systems where diverse data link characteristics are not considered. Unfortunately, this centralized arbitration model does not directly take advantage of the interconnections that exist between pairs of processors, but rather only those links from the centralized facility to the individual processors. (This under-utilization of system capabilities could be corrected by allowing nodes to function in a store-and-forward manner in an indirect routing scheme, which would also improve fault-tolerance in the event a direct centralized link fails.)

Greater advantage of network capabilities is obtained through the following decentralized dynamic arbitration model.

Decentralized Model

The decentralized model does not require a central memory for code and data nor a centralized control point for arbitration. Instead the responsibility for control and storage is distributed among the processors.

Initially all operation code is distributed randomly and uniformly among all the processors. Each processor possesses a central processing unit for execution of operations in the application program, and an optional co-processor to handle management functions. The local memory of each processor is organized into an **instruction buffer** and a **token buffer**; however, the boundary separating these areas need not be fixed. The memory capacity b_j at processor j must be adequate to contain the code for its resident operations plus any output tokens produced as a result of executing these operations. That is, in this case, the size s_i of an operation must include the size of output data structures as well as the code for operation i.

Centralized memory is required only for the following information which must be available to all processors:

1) Table to indicate status of processors: idle/busy
2) Matrix D to describe the current network configuration and time-varying link characteristics
3) Table to indicate for each processor the currently available instruction buffer and token buffer space
4) Table to indicate the processor on which each operation is currently resident, a flag indicating whether "enabled", and a list of the tokens required by each operation (Let P_i denote the processor containing operation i)
5) Table indicating the location and size of all ready tokens. Let t_{ik} denote the k^{th} token for the i^{th} operation, $s_{t_{ik}}$ denote the size (memory

space required) for token t_{ik}, and $P_{t_{ik}}$ denote the processor on which the token t_{ik} currently resides

The above data structures are to be maintained in fast assoc-iative memory with controlled shared access and may be referred to respectively as the:

1) Processor directory
2) Network configuration table
3) Buffer directory
4) Operation directory
5) Token directory

The determination of which processor is to execute an enabled operation is made according to a bidding system, an idea originally suggested by [Salama]. Idle processors bid on enabled operations, and the processor making the low bid performs the execution of the operation. The value of the bid includes a consideration of processor speeds, as well as communication costs. Because of communication costs, the arbitration facility may elect to migrate the operation to the processor containing the tokens, port the tokens to the site of the operation, or move data tokens **and** code to the most suitable processor.

The bidding process is initiated within a processor j whenever the processor directory entry for processor j indicates an idle status. Idle processor j consults the operation directory and performs the following bidding algorithm for every enabled operation i:

Decentralized Bidding Algorithm

if $s_i \geq$ local available buffer space, then no bid, else

1) $bid_i = e_i R_j + s_i d_{p_{ij}} + \sum_{k=1}^{T_i} (s_{t_{ik}} \cdot d_{p_{t_{ik}}j})$

where T_i denotes the number of tokes needed for operation i

2) transmit bid to P_i

Notice that if $P_i = j$ then $d_{p_{ij}} = 0$, and if $P_{t_{ik}} = j$ then $d_{p_{t_{ik}}\,j} = 0$. Therefore, the computation of bid $_i$ is valid even if operation i or any of its tokens already reside locally on processor j, and in that case, the reduced communication costs associated with a local execution are reflected in a low bid value. Notice also that even if an operation and all its tokens reside locally on processor j, if some other processor can route code and data to itself **and** perform the operation cheaper than processor j can do locally, then that other processor will submit the lower bid.

Bid arbitration is performed periodically by all processors j containing an enabled operation i.

Decentralized Arbitration Algorithm

Select minimum bid for operation i.

Send code for operation i to the processor submitting the successful bid.

The low-bidding processor now contains the code for operation i, and must send requests to all processors containing tokens t_{ik}. When all tokens arrive, the operation is executed and results tokens are placed in the local token buffer for use by other operation(s).

The decentralized arbitration system is active whenever there are available processors and enabled operations; therefore, progress in the application program is guaranteed.

5. Summary

With low-resolution data flow, we sacrifice some of the potential for low level concurrency that would be offered by high-resolution data flow. But in this compromise we require no specialized hardware to create the data-driven environment, and a low-resolution data flow graph can be created by a conventional compiler or even by the application software designer. Our arbitration models can be implemented completely in software for arbitrary programs on general multiprocessor systems.

We present models for static and dynamic arbitration. Static models require pre-processing but avoid run-time delays for decision-making. Dynamic models take advantage of current software and network configurations during run-time. Our formulations and network models are unique in that they address the problem of communication overhead in a multiprocessor system as well as the usual data flow execution issues.

Future work should be done to develop solutions for the quadratic zero-one problem which can be used to establish a static allocation. Experiments should be performed to determine the value of minimizing the products $c_{ik}\, d_{jr}$ as opposed to solving the partitioning and mapping problems separately. In the case of dynamic decentralized arbitration, computational experiments should be done to determine whether a "good" initial distribution of operations gives better system performance than a random initial allocation with subsequent code migration.

References

Ackerman, W.B., 1979, "Data Flow Languages", *AFIPS Conference Proceedings*, 1979 National Computer Conference, Vol. 48, pp. 1087–1095.

Ackerman, W. B. and J. B. Dennis, June 1979, "VAL–A Value Oriented Algorithmic Language: Preliminary Reference Manual", MIT Laboratory for Computer Science Technical Report, MIT/LCS/TR-218, Cambridge, Mass.

Agerwala, Tilak, February 1982, "Data Flow Systems", *Computer*, pp. 10–13.

Babb, Robert G., July 1984, "Parallel Processing with Large-Grain Data Flow Techniques," *IEEE Computer* 17, 7, pp. 55–61.

Babb, Robert G., II, Lise Storc, and William C. Ragsdale, 1986, "A Large-Grain Data Flow Scheduler for Parallel Processing on CYBERPLUS", *Proceedings*, IEEE International Conference on Parallel Processing, pp. 845–848.

Backus, J., August 1978, "Can Programming be Liberated from the Von Neumann Style? A Functional Style and its Algebra of Programs", *Comm. ACM 21*, pp. 613–641.

Barkhordarian, Shahram, 1987, "RAMPS: A Real Time Structured Small-Scale Data Flow System for Parallel Processing", *Proceedings*, IEEE International Conference on Parallel Processing, pp. 610–613.

Broy, Manfred (ed.), 1985, "Control Flow and Data Flow: Concepts of Distributed Programming, *Proceedings*, NATO ASI, Springer-Verlag, Berlin.

Campbell, Michael L., 1985, "Static Allocation for a Data Flow Multiprocessor", *Proceedings*, International Conference Parallel Processing, pp. 511–517.

Carlson, William, W. and J.A.B. Fortes, 1987, "On the Performance of Combined Data Flow and Control Flow Systems: Experiments Using Two Iterative Algorithms", *Proceedings*, IEEE International Conference on Parallel Processing, pp. 671–679.

Charnes, A., W. W. Cooper, B. Golany, V. Lovegren, W. T. Mayfield, and M. Wolfe, 1985, "A Goal Programming System for the Management of the U.S. Navy's Sea-Shore Rotation Program", *Human Resource Policy Analysis: Organizational Applications*, R. J. Niehaus (editor), Praeger Publishing Company, pp. 145–172.

Coffman, E. G., Jr. (editor), 1976, *Computer and Job Shop Scheduling Theory*, Wiley, New York.

Coffman, E. G.,Jr. and P. J. Denning, 1976, *Operating Systems Theory*, Prentice-Hall, Englewood Cliff, New Jersey.

Denning, Peter J., July 1978, "Operating Systems Principles for Data Flow Networks, *Computer*, pp. 86–96.

Dennis, Jack B., 1979, "The Varieties of Data Flow Computer", *Proceedings*, IEEE 1979 International Conference Distributed Systems, pp. 430–439.

Dennis, Jack B., 1985, "Models of Data Flow Computation", *Control Flow and Data Flow: Concepts of Distributed Programming*, Manfred Broy (editor) Springer-Verlag, Germany.

Dennis, J. B., G. R. Gao, and K. W. Todd, July 1984, "Modeling the Weather with a Data Flow Supercomputer", *IEEE Trans. Computers* C-33, pp. 592–603.

Ercegovac, M.D., P. K. Chan, and T. M. Ravi, 1984, "A Data Flow Multimicroprocessor Architecture for High-Speed Simulation of Continuous Systems", *Proceedings*, International Workshop on High-Level Architectures, Los Angeles.

Gajski, D. D., D. A. Padua, D. J. Kuck, and R. H. Kuhn, February 1982, "A Second Opinion on Data Flow Machines and Languages", *Computer* 15, 2, pp. 58–69.

Garey, M. R. and D. S. Johnson, 1979, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, San Francisco.

Gaudiot, J. L. and M. D. Ercegovac,1984, "Evaluation of Ring Communication Networks in a Data Flow Computer", *Proceedings* , 3rd Annual Phoenix Conference on Computers and Communications.

Granski, Michael, Israel Koren, and Gabriel M. Silberman, September 1987, "The Effect of Operation Scheduling on the Performance of a Data Flow Computer", *IEEE Trans. Computers* C-36, 9, pp. 1019–1029.

Gui-zhong, Liu and Ci Yun-gui, 1986, "A Model of Quantitative Analysis Performance Evaluation of Static Data Flow Computers", *Proceedings*, IEEE International Conference on Parallel Processing, pp. 611–615.

Ho, Lawrence Y. and Keki B. Irani, 1983, "An Algorithm for Processor Allocation in a Data Flow Multiprocessing Environment", *Proceedings*, IEEE International Conference on Parallel Processing, pp. 338–340.

Hong, Yang-Chang, Thomas H. Payne, and LeBaron O. Ferguson, 1985, "An Architecture for a Data Flow Multiprocessor", *Proceedings*, IEEE International Conference on Parallel Processing, pp. 349–355.

Jagannathan, R. and E. A. Ashcroft, 1984, "Eazyflow: A Hybrid Model for Parallel Processing", *Proceedings*, IEEE International Conference on Parallel Processing, pp. 514–523.

Johnson, D., February 1980, "Automatic Partitioning of Programs in Multiprocessor Systems", *Proceedings*, IEEE CompCon, pp. 175–178.

Kavi, K. M., B. P. Buckles, and U. N. Bhat, 1986, "A Formal Definition of Data Flow Graph Models", *Proceedings,*, IEEE Trans. Computers, C-35, 11, pp. 940–948.

Koren, Israel, Bilha Mendelson, Irit Peled, and Gabriel Silberman, October 1988, "A Data-Driven VLSI Array for Arbitrary Algorithms", *Computer*, 21, 10, pp. 30–43.

Lee, Edward Ashford and David G. Messerschmitt, January 1987, "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing", *IEEE Transactions Computers*, C-36, pp. 24–35.

Levin, E., 1985, Suitability of a Data Flow Architecture for Problems Involving Simple Operations on Large Arrays, *Proceedings*, IEEE International Conference on Parallel Processing, pp. 518–520.

Lloyd, Errol L., July 1982, "Critical Path Scheduling with Resource and Processor Constraints", *Journal of the ACM*, 29, 3, pp. 781–811.

Polychronopoulos, C. D., August 1986, "On Program Re-structuring Scheduling and Communication for Parallel Processor Systems", *Report 595*, Center for Supercomputing Research and Development, Ph.D. Dissertation, University of Illinois.

Rasmussen, Robert D., Robert M. Manning, and Richard S. Ward, 1987, Technical Report, Jet Propulsion Laboratory, California Institute of Technology.

Requa, Joseph A.E. and James R. McGraw, May 1983, "The Piecewise Data Flow Architecture: Architectural Concepts", *IEEE Transactions Computers*, C-32, 5, pp. 425–438.

Rong, Gao Guang, 1984, "Pipelined Mapping of Homogeneous Data Flow Programs", *Proceedings*, IEEE International Conference of Parallel Processing, pp. 532–534.

Salama, M.A., 1987, *Dynamic Task Allocation in Multiprocessors*, Unpublished correspondence.

Sharp, J. A., 1985, *Data Flow Computing*, Ellis Horwood, Ltd.

Veen, Arthur H., December 1986, "Data Flow Machine Architecture", *ACM Computing Surveys*, 18, 4, pp. 365–396.

Watson, Ian and John Gurd, February 1982, "A Practical Data Flow Computer", *Computer*, pp. 51–57.

Wolfe, Michael, 1988, Unpublished communication.

Zhao, W., K. Ramamritham, and J. Stankovic, May 1987, "Scheduling Tasks with Resource Requirements in Hard Real Time Systems", *IEEE Transactions Software Engineering*, SE-13, No. 5, pp. 564–577.

SECURITY CLASSIFICATION OF THIS PAGE (When Date Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER CCS 626	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Communication-Efficient Arbitration Models for Low-Resolution Data Flow Computing		5. TYPE OF REPORT & PERIOD COVERED Technical 6. PERFORMING ORG. REPORT NUMBER CCS 626
7. AUTHOR(s) Dr. Camille C. Price Dr. A. Charnes		8. CONTRACT OR GRANT NUMBER(s) N00014-81-C0236 N00014-82-K-0295
9. PERFORMING ORGANIZATION NAME AND ADDRESS Center for Cybernetic Studies The University of Texas at Austin Austin, Texas 78713		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS		12. REPORT DATE December 1988 13. NUMBER OF PAGES 37
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

This document has been approved for public release and sale; its distribution is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Keywords: Data flow, Large-grain data flow, Scheduling, and Task allocation.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Low resolution data flow computing offers a practical compromise between conventional control-flow computing models and the specialized architectures required for fine-grain data flow processing. We give a formal specification of an arbitration facility that simultaneously partitions and statically assigns operations to processors. This general model is based on differences in the processors, diversity of data links in the network, size of tokens flowing between nodes in the data flow graph, memory limitations on the processors, and considerations to promote parallelism. A network models solves the static problem for bipartite and tree structured data flow graphs. Based on this

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601 |

SECURITY CLASSIFICATION OF THIS PAGE (When Date Entered)

Abstract continued

allocation scheme, data tokens are automatically routed to processors, and the run-time scheduling process is distributed among the processors. Dynamic arbitration implemented as a centralized facility takes inadequate advantage of network capabilities. A general decentralized (distributed) dynamic arbitration scheme maps tasks to processors at run-time, with the association of tasks to processors based on intertask communication and network data link characteristics. Task migration is supported by treating both data and code as tokens. No centralized control or mass storage are required in this communication-efficient arbitration model.