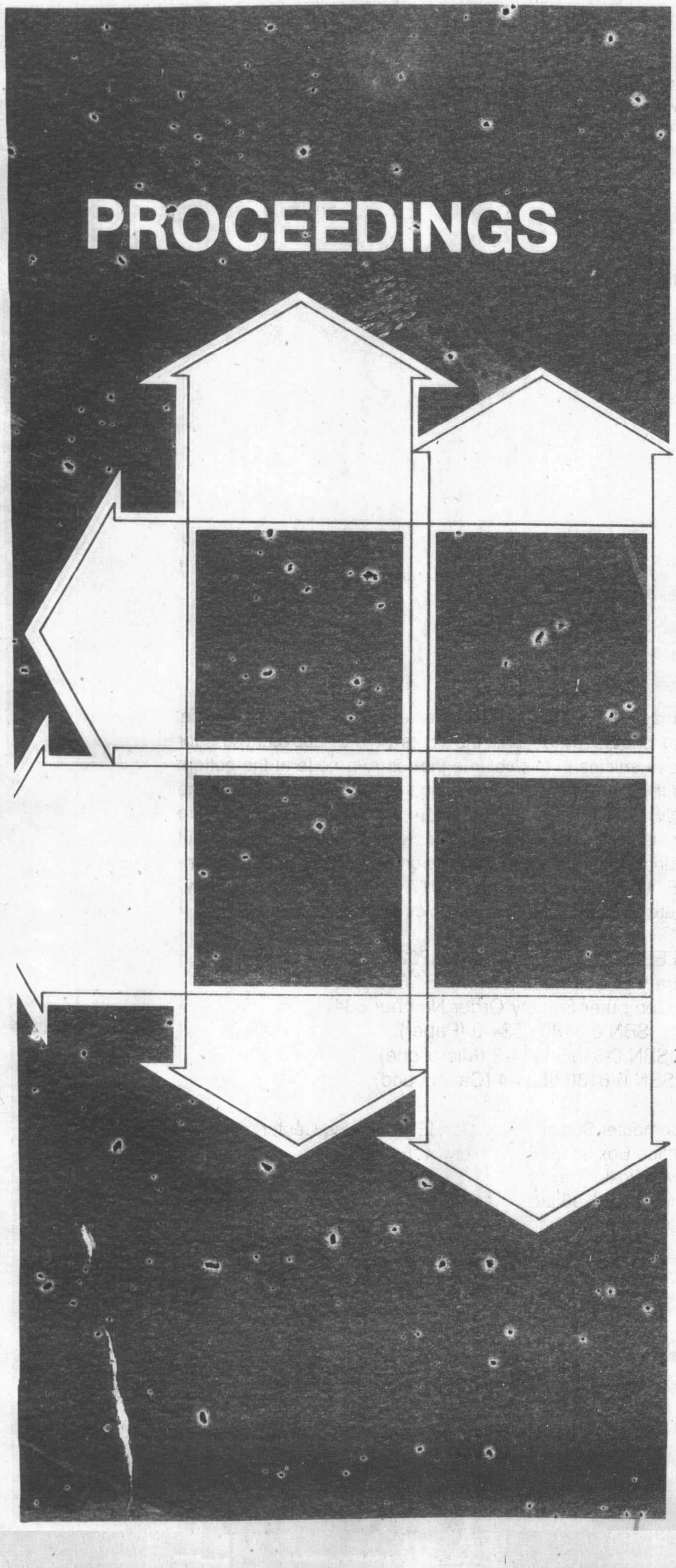
21686

The 4th International Conference on

DISTRIBUTED COMPUTING SYSTEMS



PROCEEDINGS

The 4th International Conference on

DISTRIBUTED COMPUTING SYSTEMS

San Francisco, California
May 14–18, 1984

SPONSORED BY

IEEE COMPUTER SOCIETY

1884 1984 A CENTURY OF ELECTRICAL PROGRESS
THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.

In cooperation with
Association for Computing Machinery (ACM)
Information Processing Society of Japan (IPSJ)
Institut National de Recherche en Informatique et en Automatique (INRIA)

IEEE CATALOG NUMBER 84CH2021-4
LIBRARY OF CONGRESS NUMBER 84-80276
IEEE COMPUTER SOCIETY ORDER NUMBER 534
ISBN 0-8186-0534-0

COMPUTER SOCIETY PRESS

Published by IEEE Computer Society Press
1109 Spring Street
Suite 300
Silver Spring, MD 20910

IEEE Catalog Number 84CH2021-4
Library of Congress Number 84-80276
IEEE Computer Society Order Number 534
ISBN 0-8186-0534-0 (Paper)
ISBN 0-8186-4534-2 (Microfiche)
ISBN 0-8186-8534-4 (Casebound)

Order from: IEEE Computer Society
Post Office Box 80452
Worldway Postal Center
Los Angeles, CA 90080

IEEE Service Center
445 Hoes Lane
Piscataway, NJ 08854

1884 1984 A CENTURY OF ELECTRICAL PROGRESS The Institute of Electrical and Electronics Engineers, Inc.

GENERAL CHAIRMAN'S MESSAGE

We have seen a strong movement in the past few years towards distributed computing systems. The cause of this movement is not obvious. Some would argue it is driven by technology. I would argue it is a natural part of evolution, fueled by technology, but driven by real needs that can't be satisfied by centralized computer systems. But whatever the reason, it is happening, and it has caused many of us to reevaluate our views about algorithms, operating systems, languages, and the traditional roles played by CPU's, memory, and communication structures. I believe this series of conferences has contributed to our understanding of this movement and has helped to guide it. It has been a great pleasure for me to serve as General Chairman for this, the Fourth International Conference on Distributed Computing Systems. It has been especially rewarding to work with so many dedicated volunteers and to be involved in an area of rapidly growing interest and importance to the computer field.

The primary purpose of this conference is to facilitate dissemination of recent results and progress, and with the exception of the authors and attendees, it is the Program Chairman and his committee who make the major contribution. Earl Swartzlander, Jr., together with the International Program Committee, and his six Vice Chairmen and their program committees, have done an outstanding job of selecting papers for presentation at this conference. I am also happy to announce that Earl has been named as the General Chairman for the next conference, and that Ming T. Liu has agreed to serve as Program Chairman.

We have always taken pride in the international participation in this conference. This is made possible by the efforts of the International Committee which includes Associate Chairmen from nine countries and was coordinated this year by H. J. Siegel.

As General Chairman, my only job is to ask others to do the work. I was fortunate to find so many willing and talented volunteers. Managing the publicity for a conference is a difficult task. It is the publicity deadlines that determine the preconference schedule, and it is the Publicity Chairman who keeps us on this schedule. Special thanks go to Bill Buckles who has ably performed this task for both the third and fourth conferences. Leah Jamieson Siegel, as Treasurer, has had the difficult job of ensuring we don't become a financial burden on the Computer Society while at the same time delivering a conference at a reasonable price. For many attendees, an important feature of this conference has been the pre- and post-conference tutorial program. Doug DeGroot worked hard to put together an outstanding set of four tutorials. Local arrangements are being handled by S. Diane Smith and James McGraw who have worked hard to make sure the conference runs smoothly. Special thanks go to Sid Fernbach whose quick action allowed us (and several other Computer Society conferences) to obtain alternate space at the Hotel Meridien after the fire at the Cathedral Hill Hotel. Larry Wittie coordinated our efforts with other professional societies, technical committees and special interest groups. And as always, we have depended heavily on the staff of the Computer Society's Conferences and Tutorials Office, directed by Harry Hayman. Finally, I owe a great debt to Vivian Alsip, my secretary, without whose help I would be lost.

Many others have given freely of their time and talents. Their names are listed in this proceedings, but their real reward should be knowing that their efforts have helped make this conference a success.

D. H. Lawrie
Urbana, Illinois
February, 1984

Fourth International Conference on Distributed Computing Systems

May 15-17, 1984
Hotel Meridien
San Francisco, California

THIRD INTERNATIONAL CONFERENCE ON
DISTRIBUTED COMPUTING SYSTEMS

Hollywood, Florida
October, 1982

BEST PAPER AWARD
Presented to

Marco Ajmone Marsan
Istituto di Elettronica e Telecomunicazioni
Politecnico di Torino
Italy

For His Paper Entitled

Bounds on Bus and Memory Interference
in a Class of
Multiple Bus Multiprocessor Systems

Table of Contents

Session 1A
Data Flow Systems

Chairman

G. LeLann

PERFORMANCE ANALYSIS OF A DATA-FLOW COMPUTER WITH VARIABLE RESOLUTION ACTORS

J. L. Gaudiot and M. D. Ercegovac*

Department of Electrical Engineering-Systems,
University of Southern California,
Los Angeles, CA 90007

* UCLA Computer Science Department,
University of California,
Los Angeles, CA 90024

Abstract

Most data-flow systems assume basic data-flow actors at a machine language level. In this paper, we propose to increase the level of the basic data-flow actors, i.e., decrease the "resolution" with which the compiler looks at the higher level language. This approach provides a mechanism to control the communication overhead and improve the overall efficiency of the machine. The paper presents several of the effects of the variation of the size of the actors. The observations made using the deterministic simulation of a data-flow machine are presented and compared with the results of a simple analytical model. A trade-off is observed in all cases, and the optimal size of the data-flow actor varies with such parameters as the number of PEs in the multiprocessor and communications costs.

I. Introduction

In data-flow languages [1,2,3,5,6,9,10] instructions are scheduled for execution as soon as all their operands have arrived. A data-flow program is viewed as a graph made of actors interconnected by arcs. The arcs can carry tokens bearing value. An actor is enabled when all its input arcs carry tokens. When loops are involved, it is necessary to distinguish between the data destined for different iterations. This can be achieved by labeling the tokens with their iteration level [4]. In this case, an actor becomes executable when a complete set of tokens carrying identical tags is present on its input arcs. To study the effects of variable resolution actors, we use the architecture, proposed in [3].

It appears that simplicity alone dictated the choice of machine language instructions as the basic level of computation in most proposed data-flow machines: it is the most natural instruction set since it has an equivalent in conventional computers. The operations envisioned deal with simple concepts (addition, subtraction, etc.) and have few operands.

Due to the explicit sequencing model of data-flow languages, all instructions cause overhead in order to link them with both their predecessors and successors. In the case when the resources for processing of instructions and data communications between processors are limited, this uncontrolled overhead may degrade the expected performance. An appropriate grouping of instructions will lower this overhead while permitting parallelism to be exploited at a higher level.

A model in which actors are not necessarily at the same level of resolution is a *variable-resolution* data-flow model. In this paper, we will therefore deal with *data-flow actors* (sometimes also referred to as *data-flow instructions)* These data-flow actors (or more simply *actors* when no confusion is possible) may contain more than one *machine instruction*. A machine instruction is an elementary operation similar to those encountered in the instruction set of a conventional computer (add, subtract, etc.). Note that when a data-flow actor contains only one machine instruction, we are considering the highest possible level of resolution.

II. Variable Resolution Model

We assume that the compiler can detect program structures that are amenable to clustering and group together the necessary instructions into macro-actors. (The criteria used are outside the scope of this study). At run-time, the macro-actors can fire when all the input arguments are available. The embedded code is executed inside a PE without external communications, and, as

CH2021-4/84/0000/0002$01.00©1984 IEEE

before, the results are formated into output tokens. Note that no assumption on the size of the actors is made, and that, in the same program, several actors of different size could exist.

Abstraction of the nodes to a higher level can lead to a more efficient computation when resources are limited. However, since large macro-actors (with regard to the size of the program) may lower the available parallelism, a trade-off exists. The level of resolution is therefore a critical parameter, and its influence on the performance of a system with limited resources is now discussed.

The variable resolution approach presents the following advantages and disadvantages:

- A better locality of execution can be obtained since logically close expressions are grouped together. The resulting performance is improved due to the sequential execution, without extra processing, of inherently sequential segments of code. While in a "conventional" data-flow system, a result must go through the entire Matching Store cycle before being available to the next operator, it is here immediately ready. In this case, no result needs to be cycled back to the Matching Store and, therefore, the associated queueing delays are avoided. A computation isolated inside a macro-actor can proceed without interruption once it has been initiated.

- The ratio of the number of arguments (input to or output from actors) over the total number of variables that circulate through the machine is lower. This in turn leads to a lower number of Matching Store cycles necessary for a given amount of processing demanded by the program. Such a reduction on the critical path will mean shorter execution time.

- A lower number of Matching/Store cycles is obviously brought about by the smaller number of arguments that must now transit through the memory. This means that, on the average, a smaller memory is needed. This in turn translates in a savings in terms of cost or, more importantly, in terms of speed. Since no truly associative memory of consequent size exists to this day, hashing algorithms must be employed in order to simulate the associative function. Once hashed, the key is placed in the appropriate memory location. If the location is already occupied, a secondary function must be used. This means that multiple memory accesses are necessary when collisions of this sort occur. When the hashing algorithm and the arrival pattern give a random assignment of memory locations, the size of the store with regard to the number of elements to access becomes the critical factor in determining the probability of a collision. Therefore, when a smaller amount of packets are expected to require storage, the size of the memory can be reduced or the access time can be improved for the same cost.

- A lower amount of actual processing (besides token handling) need to be performed when resolution decreases. This has already been discussed in another paper [7].

- Less data movement is involved. The execution of the program in the macro-instructions can be performed by using internal scratch-pad registers. This does not violate data-flow assumptions because the behavior of the actors is still functional when viewed from the outside. No corruption of these registers can occur because it has been assumed from the beginning that once a macro-actor has been initiated, its processing would be carried out without interruption.

- A hierarchy will be very appropriate for the code memory. When a macro-instruction is taken out of the instruction queue and its execution initiated inside the processor, its instructions are fetched from the code memory and sequentially processed. This provides us with a very good indication of the code that can be prefetched in a multi-level memory system.

- A better performance under low parallelism can be obtained. It has been observed [6] that, in their basic form, data-flow machines do not perform well when the program has little inherent concurrency. It appears that a conventional design is a better choice for sequential (or quasi-sequential) programs. This is due to the fact that each data-flow instruction has a high intrinsic control part while von Neumann instructions are simply sequenced by their relative position in the listing of the program. This is where a data-flow system with a variable resolution scheme brings a new advantage. In effect, the system becomes a hybrid data-flow/von Neumann system, using the advantages of data-flow languages when parallel execution is possible, rejecting the scheme and adopting von Neumann concepts when a sequential execution is warranted.

- Compatibility: The basic assumption of functionality is maintained at the highest level. This means that data-flow concepts are preserved at the system level where the control requirements are most complex.
Only at the lowest level, where the overall complexity is quite manageable, is there a possibility to use other models of computation such as conventional von Neumann. Of course, the macro-actors can internally use the data-flow model. That is, a system with several levels of resolution can use independent data-flow models at each level. We feel that this property is important for efficient implementation of hierarchical systems. In comparison, traditional data-flow models emphasize "one-level" system structure. Note however that, due to the limited amount of available resources, *non-functionality must*

exist even if it is at the hardware level of physical adders in the ALU and scratch-pad registers.

While variable resolution may offer these many interesting features, notably in terms of overhead reduction, it conversely could also bring about several drawbacks:

- More wait than in a low resolution system may be incurred in some instances. This is due to the fact that the data-flow semantics do not permit an actor to be fired before all operands have arrived on its input arcs. Consequently, some outer instructions which would have otherwise been marked for execution must wait for the operands of other outer instructions to become available.

- Similarly, lumping together several data-flow actors leads to losses in parallelism. This is illustrated clearly in Fig.1a which shows an excellent candidate for lumping. The results from the first multiplier are only used by the other one and therefore should not be made to go through the entire cycle. However, if this situation is not present, we will have to group instructions as shown in Fig.1b. This means that the two adders can no longer be used in parallel.

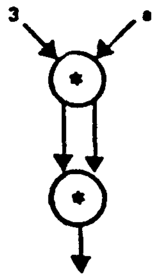


Fig. 1.a: A Good Candidate for Lumping

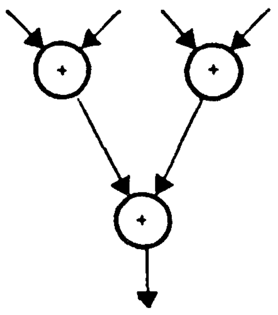


Fig. 1.b: Loss of Parallelism with Lower Resolution

- The decision of partitioning is made more difficult by the use of macro-actors. This introduces the need for a bigger and smarter compiler that is able to detect opportunities for macro-actor lumping and will produce the appropriate code.

- Since the macro-actor contains sequential machine language for execution, the data-flow assumptions are violated. This makes it difficult to call a routine that is not local to the macro-actor. The program structure must be kept intact.

III. The Analytical Model

The model of data-flow machine which we chose for performance analysis with variable resolution is discussed in detail in [11]. The execution graph is the unfolded graph after all loops and conditional expressions have been executed. A different execution graph could be obtained for a different set of input data for the same program. This graph can be regarded as the execution trace of the program. Note that the execution graph by essence can contain no loops or conditional instructions. It is a pure Directed Acyclic Graph in itself.

From the start of the program to its end, it is possible to distinguish a path through the execution graph which is the longest in terms of time. I.e., this path is drawn from the first executed instruction to the next, and so on, until the last executed instruction is reached.

In this first model, each Processing Element contains only one data-flow actor. In this context, the processing time of each actor is given by the sum of its associated communication time and its execution time. The communication time is the duration it takes for the result to reach the next actor on the longest path. On the average, the total execution time T of the data-flow program is given by:

$$T = L(C+E) \quad (1)$$

where L is the length of the longest path, C is the average communication cost and E is the average execution time.

When the number of Processing Elements is much lower than the number of executed actors, the notion of longest path is replaced by the notion of "critical path". This critical path is obtained by threading through the unfolded Directed Acyclic Graph from its root node (last actor to be executed) to one of its leaf nodes (first actor to be executed) by using only the links where the last token to the instruction arrived [11]. Note that this path would exactly correspond to the longest path as previously defined if each Processing Element was only allocated to one actor.

Let N be the number of Processing Elements, L the length of the critical path as defined earlier, M the average time spent in the Matching Store, A the average execution time in the PE, and c the average communication penalty.

The critical path L can be approximated by a decreasing function of the number of processors: factor λ/N in the equation below), and a constant δ which corresponds to the basic, unpartitionable task in the program and represents the number of those instructions in the critical path that will have to be executed sequentially due to the structure of the program:

$$L = (\lambda/N) + \delta \qquad (2)$$

where N is the number of Processing Elements, δ is the minimal task, and $\lambda + \delta$ is the total number of instructions (i.e., length of the critical path when only one processor is used). Note that λ can be viewed as the difference between L, the total number of instructions, and δ, the number of instructions in the unpartitionable part of the program. Note that the communication cost C is a linear function of the number N of PEs. This is due to the fact that, as the same program is allocated to a larger number of PEs, the average distance that any token must travel increases proportionately.

The execution time T of the program is therefore given by:

$$\begin{aligned} T &= L(A+M+cN) \qquad (3) \\ &= (\lambda/N+\delta)(M+A+cN) \\ &= [\lambda(M+A)/N]+(\delta(M+A)+c\lambda)+\delta cN \end{aligned}$$

Let $\alpha = \lambda(M+A)$, $\beta = \delta(M+A)+c\lambda$ and $\gamma = \delta c$. Then

$$T = (\alpha/N)+\beta+\gamma N \qquad (4)$$

The term γN corresponds to the performance fall-off that will be observed after the optimal number of processors has been exceeded. The asymptotic line has a slope $\gamma = \delta c$. The term δ corresponds to the longest path through the tree (if an infinite number of processors was available); it is a comparatively small number. This means that the term associated with the slope of the performance fall-off line is small compared to the α and β parameters. These variables indicate the speedup obtainable with an increase in the number of processors. In other words, the performance fall-off after the optimal number of processors has been reached is small compared to the performance increase observed before this optimal has been reached.

When resolution is variable and the number of PEs is fixed, the calculation remains similar. As before, the total execution time is given by

$$T = L(A+M+C) \qquad (5)$$

Since the number of instructions is increased by the resolution factor r, the pure execution time is

$$A = a_0 r \qquad (6)$$

where a_0 is the atomic execution time.

The number of input arguments to the new (lumped) actors is a function of the average fan-out f of the basic instruction. The problem is similar to finding the ratio of the surface of a circle over its perimeter. This is due to the fact that, as the size of the data-flow actor increases, the number of instructions inside it (surface of the circle) will increase faster than the number of data arcs leading to it (perimeter). For f=2 (binary tree), the multiplicative factor is $1/(r+1)$. It will be applied to M, the time in the associative memory, as well as c, the communication penalty. Note that an extra factor has been incorporated in the calculation of L in order to account for the loss of parallelism which will translate into more waiting for new operands. This is the term ϵr. It increases as a function of r, the number of instructions per actor. This is because the critical path is decreased by the decrease in the total number of instructions (term λ_1/r), but it is at the same time increased as more wait on some data arcs will mean other instructions must be executed. These instructions that are executed during the waiting state therefore become part of the critical path:

$$M = m_0/(r+1) \qquad (7)$$

where m_0 is the associative memory time for atomic data-flow actors;

$$c = c_0/(r+1)$$

where c_0 is the average communication penalty for atomic data-flow actors, and

$$L = \lambda_1/(r+\epsilon r)$$

where $\lambda_1+\epsilon$ corresponds to the previous $\lambda+\delta$ (λ_1 and ϵ are functions of N, the number of PEs). Now

$$T = (\frac{\lambda_1}{r}+\epsilon r)[a_0 r+(m_0+c_0)/(r+1)] \qquad (8)$$

After approximating $r+1$ to r for large values of r, and after a simplification of the coefficients we obtain the new result:

$$T = (a_1/r^2)+a_2+a_3 r^2 \qquad (9)$$

where

$$a_1=\lambda_1(m_0+c_0)$$
$$a_2=\epsilon(m_0+c_0)+\lambda_1 a_0$$
$$a_3=\epsilon a_0$$

There is again a trade-off curve. Note that the improvement is a function of $1/r^2$, while the degradation is also a quadratic function. This is in contrast with the case of the varying number of PEs where only a proportional variation was observed.

After taking the derivative of (8), it can be shown that the optimal level of resolution is given by:

$$r_0 = \left[\lambda_1(m_0 + c_0)/\epsilon a_0\right]^{1/4} \tag{19}$$

This demonstrates clearly that the optimal number of instructions per macro-actor will increase along with the overhead ration (ratio of communication and matching time over execution time itself). This can be seen in the ratio of m_0 (associative memory) + c_0 (communication costs) over a_0 (execution time per se). It can be noted, however, that the increase in resolution is not proportional to this ratio, but to its 4th root. It therefore varies rapidly for lower values of the overhead ratio. Note also that this optimal value is also a direct function of λ_1 which is itself proportionally related to $1/N$. This means that the optimal number of instructions per actor decreases with the number of PEs.

These observations lead to the following conclusions: the number of instructions per data-flow actor must be increased along with the overhead ratio of the machine. This means that in the case of a poorly designed processor (such as a non-dedicated processing element), the level of resolution should be adjusted in order to provide optimal performance. For example, if an off-the shelf microprocessor was to be used as the basic building block of the machine, the associative memory time m_0 would be tremendously increased (about 10-fold) for a comparatively constant execution time a_0. The level of resolution would then have to be modified. Likewise, if the communication network was slow or overloaded (term c_0), the level of resolution would vary in the same ratio.

Note, however, that this model is a mean-value model and that the results obtained are not an accurate predictor of the execution of a particular program but rather will describe the overall performance of the execution of a class of programs

IV. Simulation And Analysis

In order to confirm the above comments, a deterministic simulation of a hypothetical data-flow machine similar to [8] was undertaken. The results of the simulation were compared with the prediction of a simple analytical model. Two cases, discussed below, are considered for performance evaluation.

Case 1: Varying the number of Processing Elements

During the simulation, we observed the behavior of the machine with regard to a variable number of PEs. One benchmark was a binary tree with 9 levels. The rationale for the binary tree are several-fold. First, the number of levels was so chosen because it provided a sizable number of instructions (1023) for a range of PEs(1 to 192). The very regularity of the graph pattern allowed us to automatically create the data-flow program and easily modify parameters such as allocation and size of the basic data-flow actor. Then, although the shape of the computation is very geometric by choice, it should be noted that on the overall, the obtainment of any single result from a program will go through a similar pattern. This means that when several results are expected, we would observe several of these trees intertwined. Fig. 2. represents the execution time of the program as a function of the number of Processing Elements in the system. The following can be observed from the simulation results:

- In a first part (from 1 to 30 Processing Elements), a virtually hyperbolic improvement in performance exists. This means that communication costs have practically no effect at all on the performance at this point. The tokens have a comparatively smaller transit penalty to incur and, since the number of Processing Elements is still small, a large part of the tokens are still local. Note the drastic speed-up obtained: a ratio of 23 between the execution with 1 PE and the execution with 32 PEs.

- A plateau in the total execution time happens when the machine has between 30 and 60 processors. This is due to the fact that the increase in communication costs now offsets at each step the improvement in performance. An important conclusion can be drawn: this plateau means that the choice of the number of elements in the machine does not have a drastic influence on the final performance, and that it is better, at allocation time, to include more processors than estimated necessary.

- A slow fall-off, almost linear after 60 PEs is due to the increasing importance of the communication penalty: the same number of instructions is now spread across a higher number of processors. The locality ratio is diminished and delays proportionally increase. However, the degradation is very slow. This confirms an earlier statement about the choice of the number of processors. Note that this experiment does not mean that data-flow machines cannot comprise more than 60 PEs if they are to be efficient at all. In fact, it indicates that the allocation process must be optimized in its choice of the number of PEs, possibly by not using all the processors at its disposal in the physical machine.

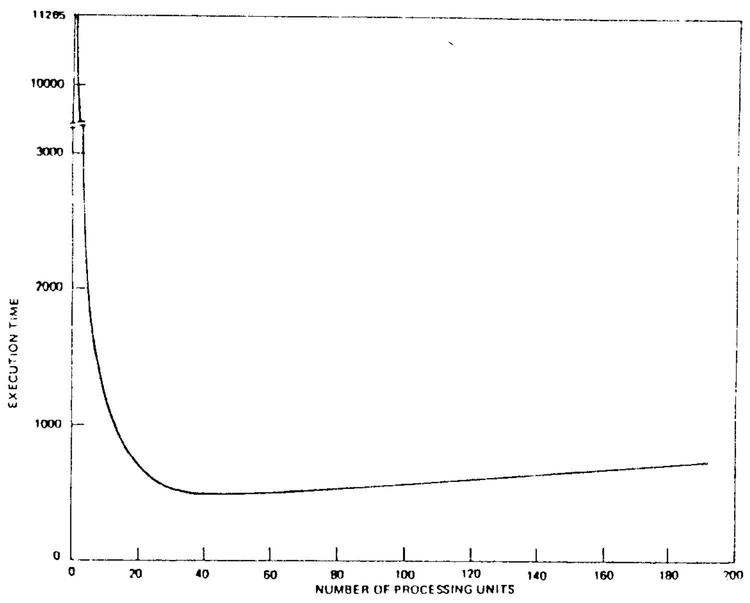


Fig. 2: Execution of a 9-Level Binary Tree

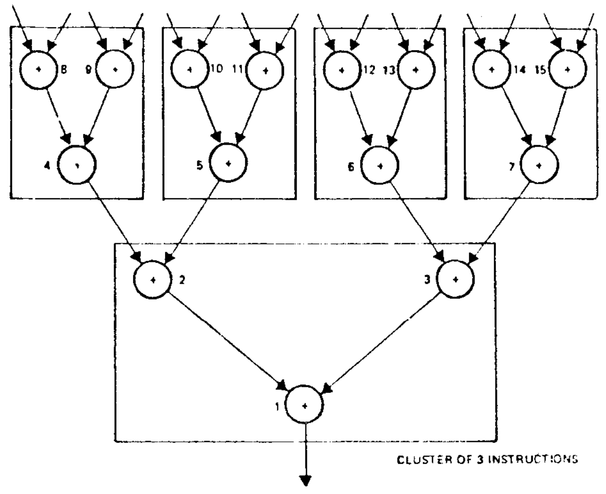


Fig. 3: A Tree with Resolution 1/3

It should be noted that these observations are comparable to the results of [8] which confirms the appropriateness of the choice of a binary tree for a benchmark.

Case 2: Varying the level of resolution

The next experiment performed consisted in lumping together several instructions. The same binary tree benchmark was kept. In order to conserve some regularity, groupings were done with clusters of 3, 31, and 63 instructions. Fig. 3. shows a 3-level binary tree superimposed on the corresponding 4-tree of clusters of 3 instructions.

At the simulator level, each cluster of 3 instructions is assigned a special instruction code. In an actual machine, it would correspond to a pointer to the actual code to be executed. This code could not be standardized and thus would be created by the compiler. Note that the instruction packet contains 4 operands in this example.

For consistency purposes, the same amount of calculation was kept independently from the resolution. This means that, for example, a binary tree with 9 levels becomes a 4-tree with 4 levels at resolution 1/3.