DR. ARVIND
Massachusetts Institute of Technology

Among many machine projects at MIT, including the "Connection machine" Jack Schwartz alluded to, are two major data-flow projects. I am going to review one of them, The Tagged-Token Data-Flow Machine. The goal of my project, simply stated, is to design a general-purpose parallel computer in which all processors will cooperate to solve one problem. Clearly we are interested in big problems, and the question of "What is general purpose computing?" has to be understood in that context. If an application does not have any parallelism, we are no magicians and therefore we can't invent it. However, many applications have plenty of parallelism and one can build a useful machine to exploit parallelism in a large class of applications. Table I lists some characteristics of typical applications which have massive amounts of parallelism.

Table I. Parallel Applications and their Characteristics.

Number Crunching, e.g.,
- -Scientific Computing
- -High performance
- -Simple data structure - arrays

Symbol Manipulation, e.g.,
- -AI type applications-algebraic simplifier
- -Complex data structure
- -Higher order functions
- -Structure of the program itself is important

Concurrent Real time Computing, e.g.,
- -Process control - Missile defense system
- -Number of asynchronous inputs
- -Adhoc hardware structures
- -No coherent functional view

In the area of symbol manipulation also, there are lots of programs with parallelism except that these programs are not as well understood as scientific computing. One reason is that algorithms in AI programs are not so stable. AI Programs tend to be far more complex then scientific programs. I understand Professor Wilson's concern that in scientific computing the equations are spread all over the program. In AI programs, often there are no equations: the program is the sole document of the algorithms used and the programmer's intentions.

While it may be hard to substantiate, I believe that if there is a large program which runs for long periods of time then it must have parallelism. I think it is impossible to write a 100,000 line Fortran program which runs for 2 days and which is devoid of parallelism. So I am proceeding from the assumption that if you have a large program you must have parallelism, even though you may not know about it. The third class of applications (see Table I) that I am interested in, is concurrent real-time computing, that is, complex process control. In a chemical refinery, one may find 1,000 one-board computers doing calculations in various parts of the system. Generally people don't view process control systems as application programs because they don't have a good model of parallel computing.

I have to do a little bit of preaching here. First of all, Fortran as a computer language won't do for parallel computing. This is not because the scientific programs cannot be written well in Fortran. Actually Fortran is expressive enough for these applications because most of the scientific computing involves no more than simple do-loops, and arrays and matrices as data

structures. The problem is that by the time an algorithm has been coded in Fortran, lots of parallelism has been obscured. Compiler designers have to work very hard to uncover parallelism that the Fortran programmer has obscured inadvertently. The theory of compilers for parallel machines may be well understood, but such compilers face many practical problems in optimizing a large (say, 50,000 line) code because of interprocedural and global data flow analysis. We should allow the scientific programmer to express the problem in such a way that the code retains whatever parallelism there is in the first place. The issue is not whether people "think parallel" but rather if they have tools -- languages and compilers which do not make the code unnecessarily sequential.

On the hardware side, I don't believe that multi-processing based on commercial processors can work. To employ many processors on one problem requires a fundamental change in the architecture of the processor itself regardless of what is done with the switching networks and memory structures. This change is already taking place in very high performance units. For example, in the Cray-1 one finds that the concept of Program Counter (PC) is rather fuzzy. It's not as "focused" as the PC in a Motorola 68000 microprocessor where one knows precisely which instruction is being executed. Instructions can often be executed out of order to increase performance in a high-performance system. By suitable use of interlocks a machine designer can make this shuffling of instructions transparent to the user. The negative effect of large memory latency on performance can be avoided only by changing the sequential nature

of the processors. My point is that we must accept and confront the fundamental limitations of single PC based machines so that processor designs would not appear to be a collection of "hardware hacks" implemented to achieve high performance.

I am going to propose a radical solution: change languages to functional languages and the basis of the architecture underlying the hardware to data-flow. I believe change in both language and architecture is required because that's the only way to get the best performance out of machines. Fortran is ideally suited for conventional Von Neuman computers. Nobody has been able to displace Fortran because the match is so perfect. Anytime something fancy is done to Fortran it's compilation becomes inefficient. Anytime changes are made in the architecture, changes which can't be exploited by a Fortran compiler, we either pay in terms of increased programming effort or underutilized hardware. The symbiosis of language and architecture has to be maintained, and I think this will happen with functional languages and data-flow architectures.

Here is a thirty-second explanation of functional languages and data-flow (see Fig. 1). Functional languages are really much closer to the way scientists and engineers think about problems. I have a harder time with computer scientists because they already know programming. If someone dosn't know programming they are much better off starting with functional languages, because basically one has to know only primitive or base functions like plus, minus, test-for-zero, and rules for combining functions. Rules for combining functions are simple function composition, conditional composition and recursion.

Composition of functions is something that engineers and scientists understand very well. To take the trivial example shown in Fig. 1, the program, f(g(a,b),h(a)) may be written as

```
Let x = g(a,b);
    y = h (a);
in f(x,y).
```

It almost looks like an imperative program where first x is computed, then y is computed, and then x and y are substituted in f. However, note that if one thinks in terms of functions, one doesn't ask absurdly simple questions like can g and h be done in parallel. Of course they can be done in parallel since they are functions, and functions don't effect each other. The value of sin(x) does not get affected by the evaluation of cos (x)! Those are the kind of beautiful properties functional languages have. They are also easier to program in and eventually they will be more efficient to execute than imperative languages. Today, functional languages are compiled on sequential machines and the compiled code is inefficient because the underlying architecture is not well suited to the task. It should be noted that this problem is analogous to the problem of Fortran compilers which generate very inefficient code for data-flow machines.

Figure 1 shows the connection between functional languages and data-flow graphs. It is easy to view the composition of functions in terms of data-flow graphs. Each box in the graph represents a function which can be a plus, minus, fast Fourier transform or even a linear equation solver. Boxes are connected by lines which represent data-dependencies among functions. The excution of these programs can be thought of in terms of arrival of data along these lines at a box, the box being enabled and

then "firing" or executing. Finally data is produced as results and is forwarded to other boxes. The natural consequence of viewing things in this manner is that any operator that is enabled can be fired. So the default is parallelism here, the execution is constrained only by the data dependencies. Note in Fig. 1, f cannot fire until h has finished execution; however, after g has output something, it can accept the next round of data and start computing with it. So given a stream of data, g, h and f may all fire simultaneously.

Next, lets consider the possibility of queuing tokens on the arcs of a data-flow growth. Let's label each token with its destination instruction address and its position in the queue. As shown in Fig. 2, the ith token as well as the i + 1st token may be in the queue at the same time. Why am I doing all this? Because I would also like to exploit, what I call, temporal parallelism in programs. If there are enough processors and several sets of tokens on input arcs, I should be able to perform several firings of the same function simultaneously. This is the kind of parallelism my machine would exploit. The basic rule in the abstract machine is that whenever two tokens have the same label they get together, the instruction specified in the label is fetched, and the operation specified in the instruction is performed. Thus, as stated earlier, you should think of a token as carrying a name (a tag) and some data.

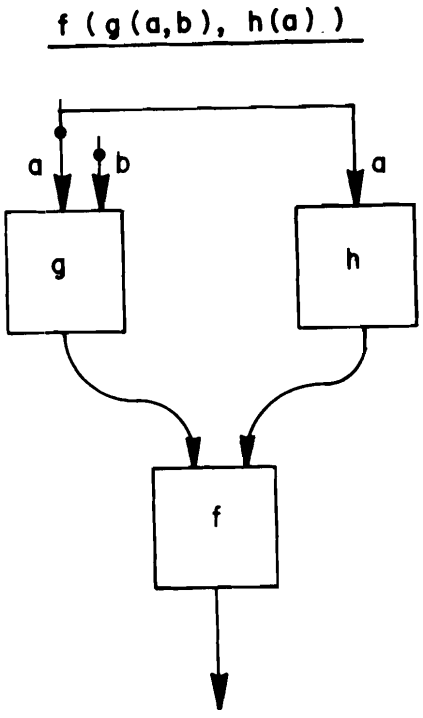

Fig. 1. Functional languages and data-flow. Here g and h can be executed in parallel; execution of f and g may also overlap.

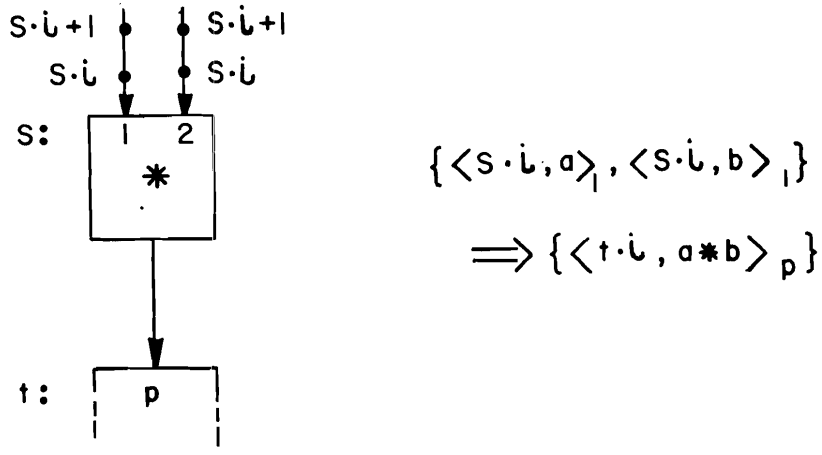

Fig. 2. The U-interpreter. A scheme for tagging tokens. Each distinct execution of an operator is given a unique (activity) name, each token carries a destination activity name.

What kind of machine will execute in this manner? Figure 3 shows an architecture consisting of N identical Processing Elements (PE's). It doesn't matter, as far as the functionality of the machine is concerned, how the processors are connected. The interconnection network may affect the performance but is not reflected in the programming model. We assume that every processing element is capable of sending tokens to any other processing element. Figure 4 shows the internal structure of a processing element and is important to understand because it's very different from a conventional Von Neumann computer. A token carrying a tag and data arrives at the processing element. The first thing the token encounters is the Waiting-Matching section which is initially empty. Remember our abstract machine has the very simple rule that when two tokens have the same label they must get together. If the token finds its partner in the Waiting-Matching Section it goes to the Instruction Fetch section, otherwise it "waits" in the Waiting-Matching Section. The Instruction-Fetch Section has a program memory associated with it.

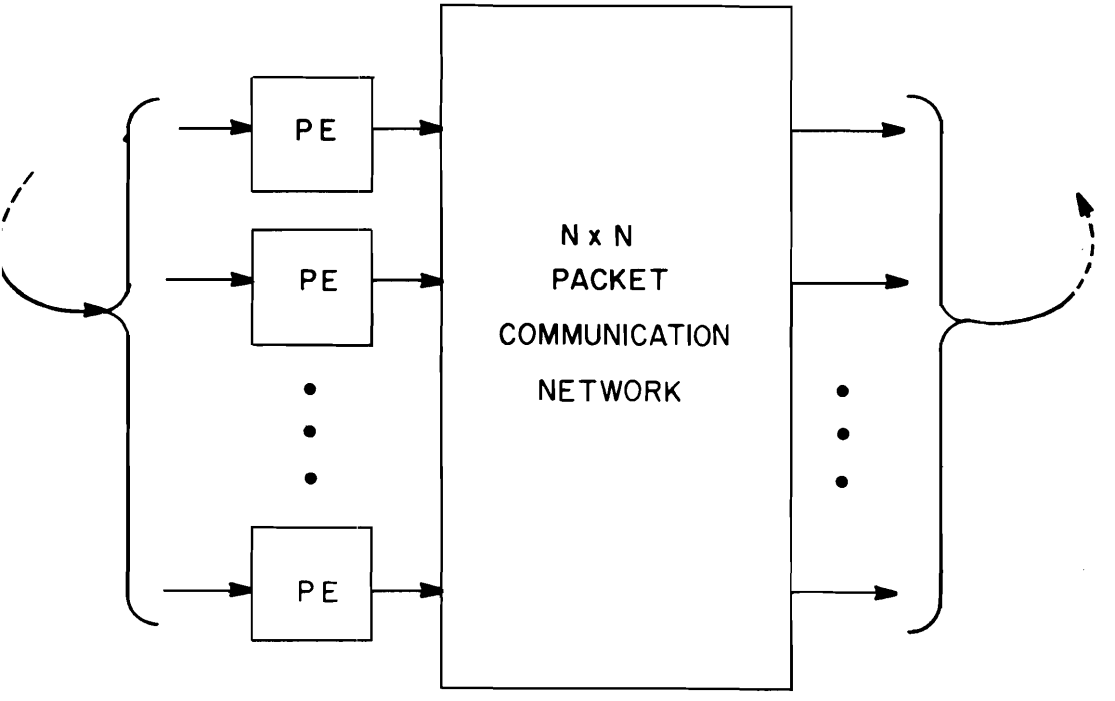

Fig. 3. An overview of the proposed architecture.

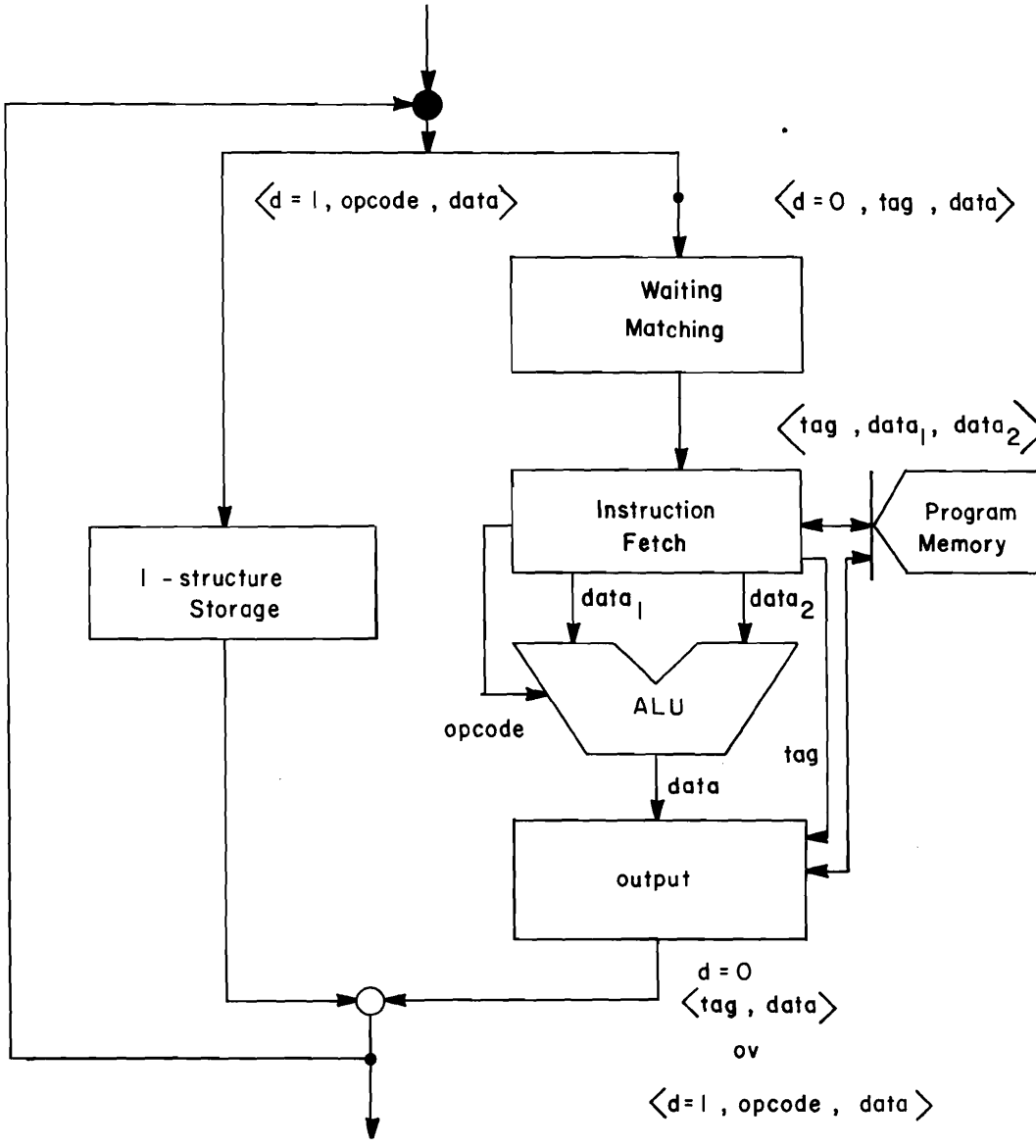

Fig. 4. One processing element.

The instruction at the address indicated by the tag in the packet is fetched. The fetched instruction says for instance, a-ha I am an addition operation. After this the operator and the operands are passed to the ALU. Notice the difference from a conventional computer where after the instruction has been fetched, the operands indicated by the instructions are fetched from the memory. Only then something is done with the operands in the ALU. In our processor, instruction fetch is done after the operands have arrived to find out what is to be done with the operands. You can have any type of Arithmetic Logic Unit here. The ALU produces data as well as tags for the data. Finally the processor outputs the results packet. This is how one Processing Element which is a complete computer in itself, works.

Now, if two such devices are available how will we make use of them? Well, a very simple strategy can be followed. One can say all the tokens with even tags remain on the left-hand processor and all the tokens which have odd tags should go to the right-hand processor. (Of course, more sophisticated schemes than this can be imagined.) This will automatically divide the work, roughly equally, among two PE's. Many different strategies for distributing work are supported by our machine. The important point is that no central authority is involved in distributing work. The Output Section only deals with the input tag and data, and a copy of the program to generate a new tag, and hence, the number of the number of the destination processor.

The data structure storage in this machine has something similar to the HEP computer; there are extra bits assiciated with each word of the memory. As shown in Fig. 5, these bits indicate

whether a word is empty or full. If a "read" is attempted on an empty word, the I-structure storage controller remembers the destination (i.e., the tag) where the data should be forwarded whenever it is stored in the word. The "store" operation causes the status of the word to be changed to "full," and in case there are deferred reads, the data to be sent to the destination of the deferred read operations. This type of storage, I think, is essential for high performance multiprocessor machines to avoid the so-called "read-before-write" problem.

Now, I will describe the communication system. Every PE is provided a 4x4 or 8x8 switching element and switching elements are connected to each other in any reasonable topology (see Fig. 6). A switching element receives a token (a packet) with a destination address on any of its input parts. Packets arrive asynchronously at the input parts and, hence, several packets may arrive simultaneously at a switching element. The switch looks up the destination address in a table which is kept inside the switch. The table essentially tells which output ports will take the packet closer to its final destination. If any of these output ports is free the packet is forwarded, otherwise it is held in a buffer in the switch. Basically the communication system is a store and forward packet communication network of very flexible topology.

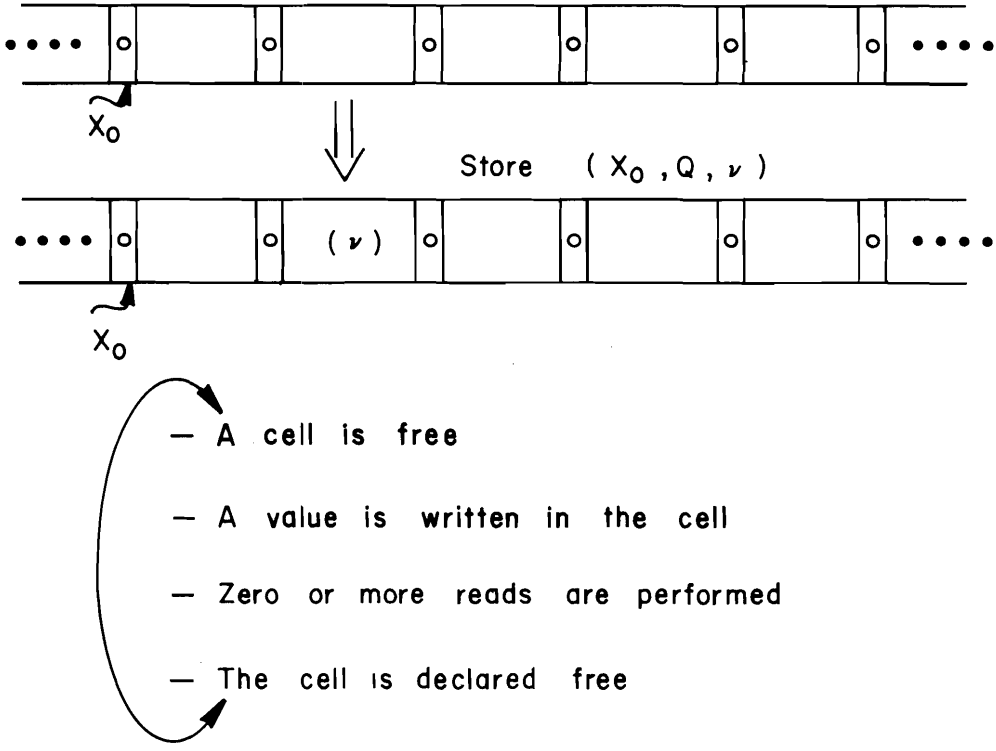

Fig. 5. I-structure storage.

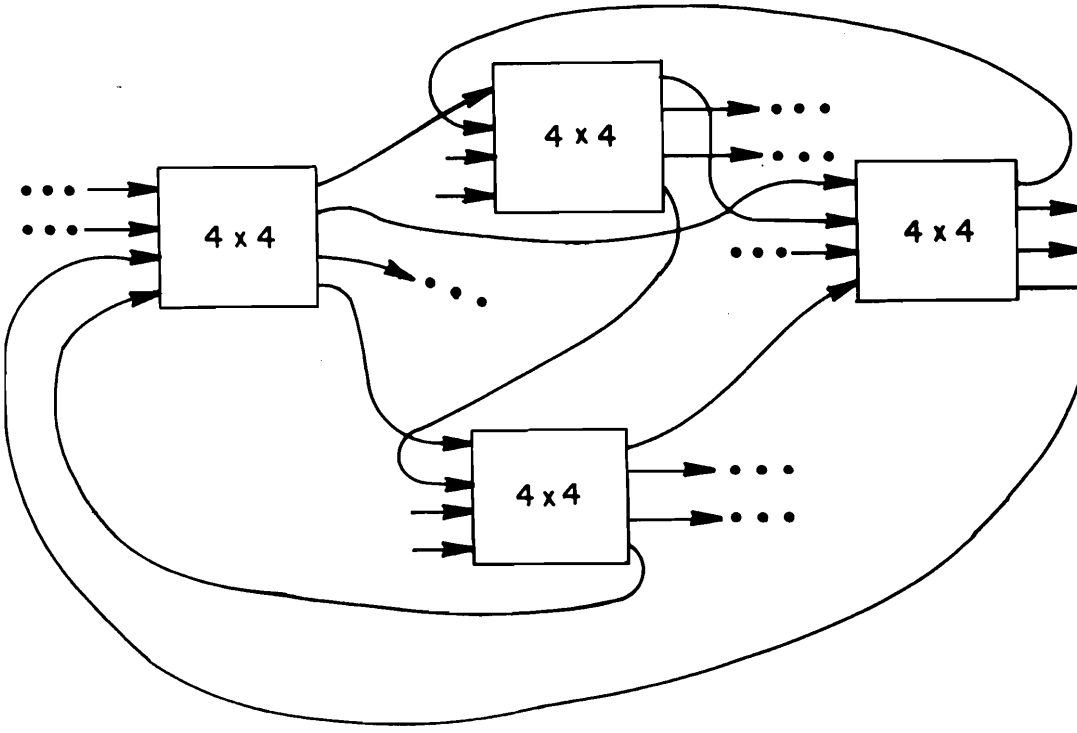

Fig. 6. Communication system. An interconnection of switching elements.

Next, I want to describe what we are building. We started out four years ago thinking, rather naively, in terms of custom VLSI chips. We hoped that our PE would fit in a single chip. This dream did not last long but we still hoped that the PE would fit at least on one board. It took another year to realize that the amount of custom hardware we would have to build to fit the PE on one board will involve seven custom chips of M68000 complexity. A hardware project of that magnitude is just too risky. That is to say, we would never have been able to find out if our architecture was defective or if the hardware was flaky. It's clear to us that we are in the business of testing an architectural idea and therefore it is necessary to take a fairly conservative approach to constructing hardware. Even the ultimate speed of the machine is not of real importance to us except to the extent that it should be fast enough to run some real user code. It doesn't have to be as fast as a Cray-1, but it has to be fast enough so that an application programmer who spends time programming the machine does not feel that his time has been wasted. In this way a programmer will get a taste of the future, at least as far as programming is concerned, and can take comfort in the fact that the next version of the machine may be faster than any sequential computer.

Thus, at some point we gave up the idea of building a real machine and decided to simulate as well as emulate the architecture. Since we had simulated an earlier version of the machine and were aware of the effort required, we were not thrilled about simulation initially. The push towards simulation came from IBM people. They said look if you guys really want us

to believe the potential of your architecture, you have to simulate the machine in a fair amount of detail. A cooperative effort with IBM Yorktown is underway now. We have received the gift of an IBM 4341 with 16 megabytes of physical storage for simulation experiments. We are running the same simulation program at Yorktown and MIT, and we hope to start running experiments on the simulator in the fall. This system is already about 250 pages of Pascal Code and it may grow by another 100 or 150 pages when the code to monitor the performance of the data-flow machine is included. My guess is that it will take about 24 CPU hours on the IBM 4341 to execute about 20 million data-flow instruction. Twenty million instructions do not represent a large time on a supercomputer, but properly designed simulation experiments should increase our understanding of the dynamic behavior of data-flow programs.

In order to execute even more instructions per experiment we are building a Multiprocessor Emulation Facility (MEF). The facility, funded by DARPA, will consist of 64 Lisp Machines connected together by a high bandwith packet communication network. The Lisp machines are of the Symbolic 3600 variety. Most of you are probably not familiar with these machines. Well, a Symbolic 3600 is a single user machine costing about $90,000! The minimum configuration consists of 2 megabytes of storage per processor. The interconnection network which is being designed by us will provide a bandwith of 4 megabytes per second per port. We think a 3600 will not be able to generate more than this much traffic if it is doing any useful computation.

We will make the MEF behave like the data-flow machine by making each 3600 emulate a Processing Element. Thus, 3600's won't look like Lisp machines, and the Lisp run time environment won't play any role in the emulated data-flow machine. However, Lisp machines provide a sophisticated programming environment and we are doing all our program development in Lisp. It should be noted that the internal parallelism of a PE would be emulated on a 3600 (which is a sequential machine) by multitasking or virtual concurrent processes.

Figure 7 shows the complete Multiprocessor Emulation Facility. Because this facility is going to be very expensive, external users will also have access to it. Only 8 of the 64 machines in MEF will be full machines while the rest will be without disks and displays. The terminals on full machines may be thought of as operator consoles on a main frame. Of course the system will be connected to local networks, so that remote program development can be done. It will be possible to do the development of an interpreter for a novel architecture on a local Lisp machine, and then ship the interpreter to the facility. In a sense this emulation facility is an analog of a big accelerator laboratory, where people would come to do experiments after having designed their experiments at home.

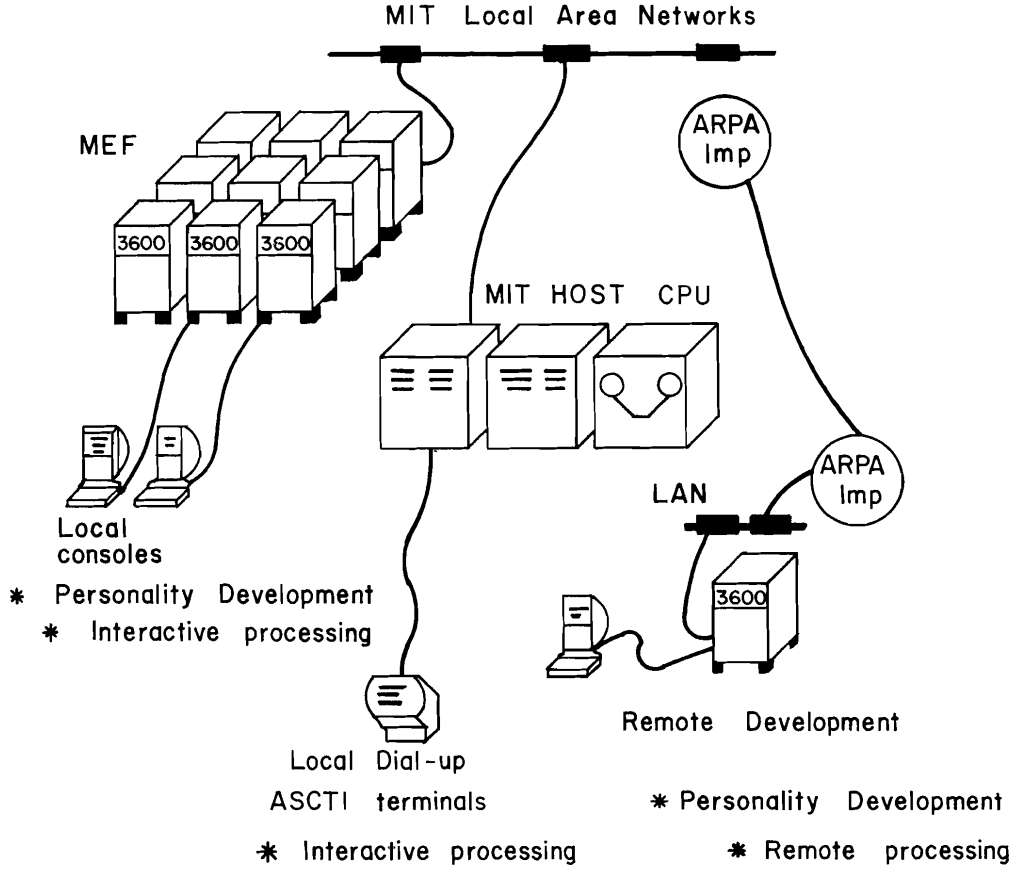

Fig. 7. Multiprocessor Emulation Facility (MEF).