Technical Panel Discussion

Parallelism in New Generation Computing

INTERNATIONAL CONFERENCE ON
FIFTH GENERATION COMPUTER SYSTEMS 1984
第五世代コンピュータ国際会議・東京

Participants

Chairman:

H. AISO	Keio Univ., Tokyo

Panelists:

M. AMAMIYA	NTT Musashino ECL, Japan
K. FURUKAWA	ICOT, Japan
D. MAY	Inmos, U.K.
E. SHAPIRO	Weizmann Institute of Science, Israel
S. J. STOLFO	Columbia Univ., U.S.A.

Chairman: Good afternoon ladies and gentlemen. Welcome to the final session. This is the panel discussion entitled ''Parallelism'', in future generation computing. My name is Hideo Aiso. I am associated with the Department of Electrical Engineering of Keio University, Yokohama, Japan. It is an honor indeed to chair this session, and also to serve as the program chairman for this conference.

The session has been organized to discuss one of the most important and probably most interesting topics, that is, parallelism. From various points of view, it seems to involve innovative potentialities for future generation computer systems.

Everyone agrees that parallelism using advanced technologies such as VLSI, innovative architectures, or novel programming languages should be the way to get ever increasing processing capability. However, we have recognized that some very important but serious problems have to be resolved to achieve that goal. Therefore, the main objective of this session is to review the important technological issues on parallelisms with emphasis on those encountered in logic programmings and innovative computer architectures.

We will endeavour to explore potential approaches to solutions. I'm very happy to introduce to you five distinguished panelists who have been very active in research on parallel processing, and in new research areas such as logic programming languages and innovative computer architectures.

From the left, the first speaker will be Dr. Makoto Amamiya. He received the Ph.D. degree in electrical engineering from the University of Kyushu and is now associated with the Musashino Electrical Communication Laboratory of Nippon Telephone and Telegram Public Corporation. He will approach the issue from the standpoint of data flow architecture.

The second speaker will be Mr. David May. He graduated from the University of Cambridge and he now works for Inmoss, of the United Kingdom. He will present his position from the standpoint of the architecture of VLSI systems and parallel processing languages.

The third speaker will be Dr. Ehud Shapiro. He was awarded the Ph.D. degree in Computer Science from Yale University and is now working at the Wiseman Institute of Science, Israel. He will give us the outlook from the viewpoint of the relationship between sequential and parallel processing.

The forth speaker will be Dr. Koichi Furukawa. He received the Ph.D. degree in Information Science from the University of Tokyo. Dr. Furukawa has worked for the Electro-technical Laboratory. Now he is a researcher at ICOT. He will give his position from the viewpoint of novel programming languages.

The last speaker will be Prof. Salvatore Stolfo. He received the Ph.D. degree in Computer Science from the Courant Institute of New York University. He is Associate Professor at Columbia University, and he will approach the issue from the viewpoint of large scale parallelism for Expert Systems.

Before moving on to the panel discussion, let me explain briefly how this session will proceed. Each panelist will speak on the issue of parallelism from the stand point of his expertise and his research field for 15 minutes. After that, the panelists will respond to questions from the floor All questions and comments will be welcomed after the presentation of the positions.

These may be followed by 30 to 40 minutes of general discussion. Now I'd like to start this session with Dr. Amamiya.

Dr. Amamiya: Thak you. First, I want to talk about parallelism and parallel machine architectures from the viewpoint of the data flow concept. But, I will discuss the issue from a rather abstract position than that of concrete machine architecture. There are two types of approach to parallel machine architecture. One is the task-oriented or algorithm-oriented approach and the other is the generic approach. The task-oriented approach is used in for example, image processing or partial differencial equation calculation. This has an array structure, or mesh structure algorithm. This may be an implementation engineering dependent aspect. Here, I'll speak the second type of approach. That is the generic approach, derived on computer science. This characteristic type of problem has a non-fixed or flexible, dynamic structure to its algorithm. Examples are AI programs or simulation of artificial phenomenon like our social activities.

A difficult but important problem in this second type of approach is how to map the structure of the program domain to the structure of the machine. This is the principal issue in our work on parallel machine architectures.

When we think of the structure of the program domain, there exists various kinds of parallelism. The characteristics of paralleism in the program domain are classified two types. The programs are mixtures of the two types. When we think of information processing or problem solving, we think in terms of the concept of functional and logic programming. The first type is parallel computation based on the divide and conquer algorithm. For example, the familiar algorithms are quick sort and merge sort algorithms. We can exploit the linear time execution of the program by using linear list data structures. And, in the Logic, or inference, program domains, OR-parallelism is one type of this kind of parallelism. The second kind is concurrent computation, i.e., stream-or pipe-line processing. This type of parallelism is achieved through linear recursion with data construction. The typical examples are set-operations or prime number generation by the seive method. In the logic program paradigm, AND-parallelism is this type. The third kind of parallelism is a mixture of these tow types.

Now let us consider the three main characteristics of information processing. The first is operation on data. That is, mapping from input data to output data. This is straight forward functional program wing. The data flow, is pre-determind. The second type is a data retrieval, such as in relational data bases. This is specified by the relations among data. Here, data flow is not predetermind. That is, data flow is bi-directional. This is also a feature of logic programing. The third is time dependent processes. Both functional and logic programs can't nicely deal with time dependent processes. However, sensitivity to history is the important issues in the practical information processing.

The concepts of status and sequential operation are necessary in practical information processing. The concurrent programming feature is required to deal with time dependent processes in execution. Modularization is a very important concept. In summary, the machine and programming language must incorporate all of three these features of information processing. That is, functional programming and logic programming should be amalgamated using the modularization concept.

Further, Coordinative processing or the message flow system is also a very important concept.

An other important point concerning parallelism is the concept of set. In AI programs, the basic concept is generate-and-test. At first, a set of data is generated by some predicate or gathering process which satisfies a given predicate. Then, the following procedure is used to test the generated data which satisfies given predicates. In this process, those predicates are specified as relations among data. The next important concepts are coordinated problem solving, and modular and hierarchical construction of programs. This is the abstruction technique. This brings in the message flow concept, which is the same concept as intermodule concurrent execution. It is, in other word, the object-oriented approach.

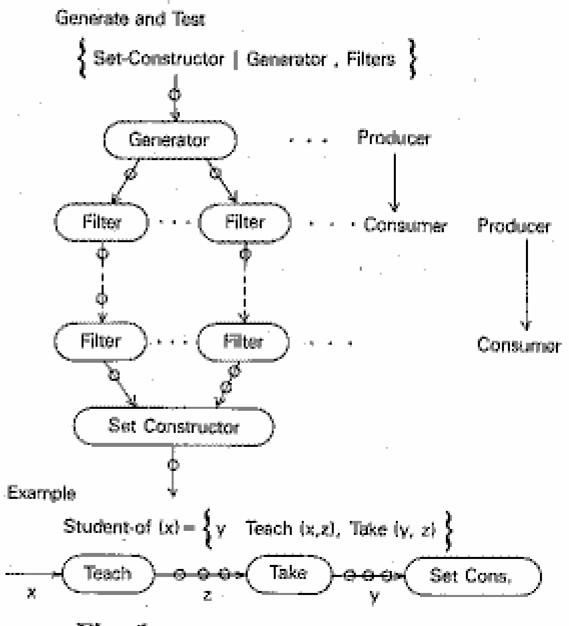

Fig. 1 IMPORTANCE OF SET CONCEPT

This is a very simple example showing the importance of set concepts. There is a test procedure which is a set expression. At first the generator produces some data set and then the filters, filter out the generated data. Fianlly, the set constructer, the data constructs set. This processing feature is a steram-oriented calculation technique. A quite simple example is the concept of a student of X, where X is a teacher. The student of X is such that X teaches some curiculum Z and student Y takes the curriculum Z.

Now, I want to talk about how to implement this kind of parallelism. From my point of view, it is clear that data flow architecture is a basic concept for the control of such parallel computations. Second, by structure data manipulation is also an important problem. For example, list sturcture data is very convenient for representing very complex data structures. On the othr hand, structure data manipulation consumes a lot of time, since complicated memory operations are necessary. Therefore, high level memory operations, such as pattern matching, should be supported by hardware. As the data flow architecture is based on packet communication, the pipeline operation is possible between execution control on processing units and memory operations on memory units. Thus, data flow architecture resolves the memory latency problem by using a pipeline control technique. There is also the problem of communications between processors. In parallel processing machines a large number of computing modules are connected by a network. As the communication overhead is serious, high-speed packet switching networking is the inevitable technique. Packet switching network and packet switching communication techniques can resolve the communciations overhead. The pipeling operation between execution and communication control is neatly effected by the packet communication technique. The data flow architecture resolves the gaps between inter-processor communication and intra-processor execution.

As pointed out above, the data flow architecture has the excellent features of the parallel machine architectures. But at the moment, there do not exist any practical data flow machines. What are the problems to be solved here? Research problems concerning parallel machine architectures are listed here. One is an appropriate granularity needs to be set in the data flow that which is the best: fine grain, medium grain or coarse grain? Well, it depends on the application area. However, in general, programming language features should support the most suitable granularity levels for the data flow architecture. For example, the data flow concept combined with a reduction concept and semi-dynamic scheduling of the execution sequence will be one of the solutions, and then program transformation technique would be suggested. Next, there's the problem of the task schedule for load balancing. The load should be evenly distributed between many processors in the system. The third problem is finding the right hardware implementation technique. How to implement the data flow processor cheaply, how to implement a structure memory to support higher level operations, and how to realize a packet switching newtwork.

The problem of huge wires arising in the network implementation has to be solved. The VLSI technique can provide some solutions to the hardware implementation problem. The fourth difficulty is, implementation of the parallel inference mechanism, which supports such a generate-and-test scheme in AI programs. A specific kind of high-level data flow control is necessary to support parallel inference mechanisms. The last problem is implementation of a coordinated problem-solving system. That is, the realization of the module concept on the basis of the functional and logic programming paradigm.In conclusion, what I have been trying to show is that we have to move on from the data flow concept to the message flow concept. This is my standpoint. Thank you.

Chairman: Next. Dr. May, please.

Dr. May: Thank you. I shall talk first about languages and second about the architecture of highly concurrent systems. I belive that to express many of the real world problems that we are dealing with, we need a language which contains the following:, sequence and states, concurrency and communication, hierarchy and locality. These are all things that you can observe in the real world. They are needed for applications and they are also needed for describing the structure and behavior of implementation. In other words, a language which contains such features allows one to map real world problems directly onto VLSI implementations.

I believe these ideas should be represented quite explicitly in the language and should in some sense directly correspond to constructs in the language. I'll give you an example which is of course a language I'm very familiar with OCCAM. Sequence and state translate into sequential constructs and assignments to variables. Concurrency is represented directly by a parallel construct, communication by input and output operations. Hierarchy is represented by block structure, a technique that's been known for many years, locality by declarations, also known for many years. Once we have a language like this a programmer has the tools to explicitly describe the structure and behavior either of collections of programmable computers or of special purpose VLSI hard-ware.

One of the important features of a language for concurrent systems is whether it has formal porperties. We will want to be able to verify and prove properties of concurrent programs because it will be very difficult to establish their correctness by simulation and testing. We also want to be able to use program transformations on such programs because we will often want to be able to move between the sequential version of a program and the parallel version of a program. We want to be able to make these transformations in the knowledge, that were not introducing errors and that the two programs both have the same behavior. We can do this by regarding a program as a predicate which describes all observations of a machine executing the program. This allows a logic to be constructed which can be used to reason about the program. So we can have logical programming without logic programming!

While on the subject of logic programming and declartive languages, much has been said about the implicit parallelism of declarative languages. It seems to me that this either gives rise to tóo little parallelism, and doesn't use all the capability available, or gives rise to too much parallelism and causes problems of allocating processors to tasks. I'm not sure that we're yet in the position to give these declarative languages a good implementation.

I've already raised the issue of locality which seems to me to be important. If we want concurrency of processing and communication we have to try to make the processing operations as local as possible.

That is presumably best done by combining the processor with some local memory to which it has very rapid access. This can be achieved ideally by putting them both on the same silicon chip.

Secondly, we need locality of communication and the best way to achieve that is to communicate between only pairs of processes.

This way we can avoid sharing things, in particular, sharing processors, memories, or communication systems.

Locality is nothing new in computer architecture; indeed many advances in the past have resulted from exploiting it. Examples are base registers, stack pointers, caches, and so on. These all use locality to improve the performance of computer systems.

This kind of architecture using essentially local operations with no global buses, or global clocks, or global synchronizing agents and gives rise to some interesting problems. I'd just like to make a plea to language designers. It's terribly easy to introduce into languages convenient programming features which are nearly impossible to implement in a distributed system because they implicitly require global coordination or synchronizaiton. I recommend from my own bitter experience that languages should be designed first to make sure that the concurrent implementation is as efficient as possible. It will then be very easy to provide an implementation on a sequential processor.

A sequential processor can implement

almost anything by simulation!

One very open question at the moment is what is a good ratio in a system between processing, memory and communications. In our current state of VLSI technology we can implement a processor of, say, ten million instructions per second on about the same amount of area as two kilobytes of memory, or a communication system to connect it to the rest of the system.

This means we have a rather interesting choice, in that we can either use own silicon for building a ten MIPS processor with four megabytes of memory—this would be a conventional type of computer, or we can have a computer have which will do 10,000 million instructions per second and only two megabytes of memory. These two computers are approximately the same size. They both take about a thousand VLSI devices. So they are both quite small computers. The problem, of course, is how we would actually construct a system containing many many processing elements, with small amounts of memory dispersed through the system, in such a way that we can apply it to practical problems.

How can we construct such a system? Obviously, such systems will have to be extremely regular. If it's going to have thousands of processors, we can hardly connect them in a tangle. There are a number of possible choices. There aren't really many of them. We can make structures like pipelines, one, two, three dimentional arrays, possibly more, but after about the sixth dimension, wiring would become so difficult to just make it completely impossible.

There are some slightly more exotic structures-computing surfaces—where, for example, we take a two dimensional array and fold it to make a torus.

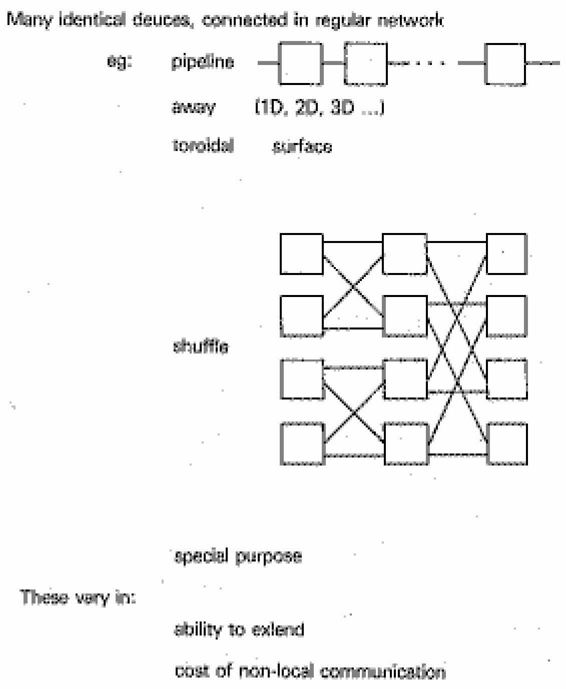

Fig. 2 SYSTEM STRUCTURE

Then there are various kinds of shuffle exchanges. There are a large number of them, all of which have slightly different connection structures. They will extend of course through many many more stages than I've shown here.

Then there are many possible special purpose structures which are of course tailored to specific applications.

These structures all vary in their ability to extend. Obviously a pipeline can be extended indefinitely. A shuffle becomes progressively more difficult to extend as it grows, because of this wiring crossover here which gets worse and worse the more stages are added. But, up to a thousand or two thousand or so processors, it's quite feasible.

The structures also vary in another property—the cost of non-local communication, by which I mean a processor communicating with another processor which is not an immediate neighbour. Of course the pipeline is particulary bad because it may involve communication thourgh a very large number of intermediate points. The shuffle is the best. In that it in general requires only a logrithmic number of steps to communicate through the network. So one might hypothesise that a shuffle would be quite a good structure to use as a fairly general purpose engine where the mapping problem is fairly difficult to solve, because it essentially allows one to ignore many of the mapping problems.

I don't really think there are any solutions as to exactly which one of these structures one should choose yet. Its a topic for futures research. Where are we with concurrent algorithms for these kinds of machines? Well, some already exist, particularly in the signal processing, numeric processing, and simulation areas. These are regular problems which are very easily mapped onto regular processor arrays. The optimal results in such cases are usually achieved by combining sequential and parallel algorithms. This is one of the reasons why I feel it's particularly important to have programming languages containing both sequential and parallel programming facilities. It's particularly important of course to choose the appropriate ratio between processing and communication. That is, how much internal operation is done by a proccesor each time it receives a message. At one end of the spectrum one has effectively a data flow architecture in which one inputs a message, performs one single operation and then outputs a result. At the other end of the spectrum one has sequential programming where one inputs a large amount of data, performs a lot of processing sequentially and then outputs the result. In practice, however, it seems to me possible to adjust the algorithm to make this ratio such that the communication overheads disappear in relationship to the processing cost. So I suppose my overall conclusion is that concurrent programming is different, not difficult! Thank you.

Chairman: Thank you.

Dr. Shapiro: Well it seems that without much coordination my talk begins exactly where Dr. May's talk ends. The first slide reads that the development of parallel computers will not be easier than the development of sequential computers. But it also means that it shouldn't be much more difficult than the development of sequential computer. I'd like to offer a few simple statements which some of you may find obvious, and some of you may find absurd, and we'll see how this results in the discussion.

That is a main thrust of my statement. I don't mean it'll take us another forty years to reach stage with parallel computers that we are at with sequential computers, but I mean that at least we should go through all the transitions which sequential computers have gone through.

So, in the same way that there is a theory that a child, before he's born is going very rapidly through all the stages of evolution of the human race from the beginning of history. I believe that the new born child of parallel computers will go through the same stages of evolution as the sequential computer, although perhaps in a much shorter time scale. More concretely, first we'll have our computers and they will have some machine language. Some machine language that perhaps only a few experts will know how to use, like in the old days when computers come around. After we gain some experience, using these new computers with this new machine language, perhaps we will understand better how to design high level languages for these computers.

After we have a lot more extensive experience with these machine languages, we'll then know how to compile high level languages into the machine language. Remember that it took us a long time to understand how to compile high level languages into machine language, for sequential computers to get a lot of experience in programming in machine language before we knew how to construct programs that actually generate good machine language code. There is no reason to believe that something different will happen with the parallel computers. Another point of similarity is algorithms. The number of algorithms started to grow very rapidly (and I think that's the point, that will be made by Koichi Furukawa as well.) Only after we had working sequential computers that we could actually get our hands on. I think that the same thing will happen with parallel computers. Even though work on parallel algorithms has already been progressing for quite a long time, the development will be much much more rapid as soon as we have a working parallel computer.

So, what do we need in this phase of development? I think these are three components which are essential as the starting point. We need a good abstract machine. I think we can't ignore the fact that we are going to program a parallel computer. We can't close our eyes and write porgrams as we wish and hope that they will quickly on a parallel computer. We must have a model of the parallel computer in our minds, but it shouldn't be too low level and shouldn't be too complex. It should be abstract enough, so we can have a clear concept of it in our mind. But it should be close enough to the concrete architecture so there won't be a big gap between our concepts and the actual running program.

The next thing we need is a good machine language. As I said, I believe we will have to program parallel computers in machine language for a while, perhaps for two, three or, four years to get a lot of experience with them before we are able to compile a higher level of language on them. So it's very important that the machine language be very good, be close enough to the computer so it can still be executed fast, but still managable by human beings, perhaps only by expert programmers who do the research for a while, but still managable. And the third thing we need is good algorithms and I mentioned what I think about them before.

What we cannot afford to ignore, at least in the very first steps in this development, is explicit control of locality of communication in the abstract machine. I think David May made a very strong point for the locality and regularity of the system design of concrete machines and we cannot ignore this aspect in the abstract machine that we want to progam as well. Otherwise, the gap between the concrete machine and the abstract machine will be large and we will lose a lot of efficiency when actually trying to simulate the abstract machine. So we cannot ignore the locality in the abstract machine definition.

We cannot ignore explicit parallelism in a machine language. As David May said before, I do not believe that today we know enough to start programming as if we don't know anything about the machine and rely on someone else, a smart compiler, the runtime system, or whatever, to discover the parallelism for us. We're still in an embryonic stage in parallel processing, and we cannot afford to ignore this—the main aspect of parallel processing. The last and most important thing, I think we cannot ignore both locality and parallelism in algorithm design, which I think is very obvious.

The implication of these points is that the machine language that we use to control and program the abstract machine should have explicit control both of locality of communication and parallelism. David May said it in other words. So I agree with that, but he concluded that the declarative languages don't have this ability, and here I disagree. Just to make a more concrete point, for example, I believe that concurrent Prolog augmented with an explicit mapping notation does have explicit control both of locality and of parallelism. So I agree with David's assumption and with the argument. But I don't agree with his conclusion that he made about declarative languages.

I think it's premature at this stage, where this stage may last one, two, three, or five years, to try to exploit parallelism automatically, because we don't know enough right now. And I think it's also premature for the same reasons to compile high level languages into the machine language. Perhaps here, my notions of a high level language and a machine language are quite different from the conventional ideas. I think that a certain restricted subset or certain restricted classes of logic of programming languages can be viewed as a machine language. When I say high level languages, I mean even higher than that. I think the key distinction between a machine language for a parallel computer and a high level language for a parallel computer is the ability to explicitly control parallelism and locality. The machine language should be characterized not by syntax not even by type such as functional, logic, CSP, or whatever. It should be characterized by its ability to control locality of communication explicitly and to control parallelism explicitly. As long as it can control these two things explicitly, it should be viewed from this standpoint as a machine language, a usable machine language.

Now the questions are more fine tuned. Is it too high level to exploit the actual power of the machine? Is it too low level for systems programmers to use? That's really the question right now in terms of tuning the machine language. But as long as the machine has explicit control of locality and explicit control of parallelism, then under this framework, it should be viewed as machine language. I think that's where we should start. After we know a lot about how to program in an assembly, language or machine language for parallel computers, where an implementation of concurrent Prolog or some subset of it is an example, then maybe it'll be time to go to higher level languages without explicit control of these aspects.

I take a rather optimistic stand on these issues and believe that programming styles, higher level languages, and algorithms will emerge as soon as such a good machine model is available. I also believe that during this course of development all or most of the important concepts of conventional computer science will have to be rediscovered. It's not that the new computers will have to be completely different. We suddenly will not need algorithms or structured programming, or other major important concepts of conventional computer science. It's not that logic programming will put computer science in the waste basket. I don't believe this will be the course of development. I think the course of development will be that almost all important machine independent concepts of computer science should remain from this generation to the next one, but perhaps rediscovered within a different framework. One such example, which I came across through my experience in concurrent programming, is the duality of data structures and process structures. Data structures play an essential role in the design and implementation of sequential algorithms, and I think there is a similar concept in concurrent programming: that is process structures. Process structure is a new concept, but it's very similar, almost isomorophic to the concept of data structure in sequential programming. Here's an example of an old concept rediscovered within a new context. Thank you.

Dr. Furukawa: I'd like to talk about part of more application side and parallelism in knowledge information processing in general. I think it's a very difficult research subject. There's no promising way to achieve this goal yet, and to attack this problem, we need clear up four basic issues right away.

One is to find key problems to be attacked. We need proper tools for this research, we need a systematic approach, and we need lots of people to join in the research.

These four problems are interconnected as follows: that is, "Key problems will be found through systematic research activities using adequate tools by many researchers." What I claim or what I used to claim, is that logic programming plays a very important role as a kind of glue for integration. I will come back this later again.

We view logic programming as a good approach to building a bridge between knowledge information processing and highly parallel computer architecture. And between logic programming and highly parallel architecture, I think the computation model is very important.

The next thing for the systematic approach is overall research covering from hardware to application. We need a complete set of all activities. In some cases, it is possible to achieve parallel computation just looking for only lower level or only higher level parallelism. But to cover the entire system is very difficult. You have to have a good way to extract many parallelisms from an application, you need an efficient language to express such algorithms and a parallel computer which can execute them very efficiently.

I think we are now ready to go on to the research of this very difficult problem. We have now a set of good tools to deal with it.

In hardware, we have VLSIs and even if we don't have yet parallel inference machines they are certainly possible. In software, we have parallel languages, for symbolic computation. In the applications field, we have several knowledge programming languages for distributed problem solving. These are examples of some available tools. In software, as parallel languages, we have Concurrent Prolog and we are designing KL-1. In applications, Actor and Contract Net have been proposed and we are proposing a language called Mandala. Actually I think we need lots of researchers to work on problem.

We should compare our work with the research effort devoteed to developing the sequential computer culture up until now. We have to notice the rapid growth of research on algorithm, I mean sequential algorithms right after the emergence of the sequential computer. So I want to conclude my talk by saying that if we are prepared with proper tools and a correct attitude, I think we can rapidly develop a culture of parallel computing. Thank you.

Dr. Stolfo: This has been pretty TAME so far. So I'll help to liven things up a bit, I hope. First let me start by giving a context I'll be coming from. I'll be playing devil's advocate for the most part. Everything I say is on the philosophical level in this little minitalk. I don't necessarily it all believe entirely, but in any event maybe we can get some discussion.

I should tell you a little bit about my activities over the past three years or so. I have implemented, along with a members of the technical staff of AT & T Bell Laboratories an expert system that has become a commercial product and is currently being marketed and sold. By AT & A, brought it from the initial stages of study and investigation up to product development, but I do not work for the AT & T. I'm also a principal architect of a parallel computer designed to run expert systems at high speed and at low cost. We have had a working prototype of this machine since April, 1983, and we are very close, relatively speaking, to completing a larger prototype of the machine.

I certainly do believe that panels should be controversial. The first slight bit of controversy is that I come from the world that believes the existing AI technology of LISP and production system technology still has a long way to go, and has just barely been applied. It's a bit premature in my view to jump into Prolog, without first applying the existing technology to its fullest extent. So I'll talk about production system languages embedded within LISP. The ACE expert systems referred to earlier is an example of such a program. What I wish to do with many other people in the United States who use these programming formalisms is to speed them up.

There's good reason to speed them up. I don't know of a reason to speed up a fifth generaiton computer yet, but I do know why we should speed up production systems. What is a production system, for those who may not familiar with this terminology. You may have heard of rule-based systems. The kind of a formalism I have in mind is the OPS formalism.

A production system has three major components, a working memory of facts, a production memory which is essentially a large collection of if-then rules, and an interpreter which has a very simple process for recognizing all the rules whose left-hand sides match the state of the working memory, select one of those rules, and then applies its right-hand side which typically is additions and deletions of the working memory. It's a very simple problem solving formalism. It goes a long way, however. In this formalism, one perform a very limited form of unification pattern matching against data structures. There are many examples of this kind of formalism, OPS from CMN, Emycin from Stanford, and the list goes on. There are now several AI companies in the United States selling programming formalisms similar to these as products for people to develop experts programs. So one issue we have been considering, and I'll tell you why in just a few minutes, is how can we speed up production system programs? Well, I'll give you just a little bit of technical discussion about this and then I want to quickly switch to more 'philosophical issues.

Let' first consider the three phases of the interpreter where the work is done, recognition, selection, and action. Recognize the rules, which match, select one rule and act on the right-hand side of that rule. What is the parallelism in the recognition phase?

If you take two seconds to look at the formalism, the first idea that comes to mind is to match the rules in parallel, rather than as on a serial machine, sequentially. So, each rule will be assigned a distinct processor which will match it independently of other rules assigned to their own processors. But there are other sources of parallelism in the recognition phase. The left-hand side of each rule is a collection of pattern elements.

We may wish to match a signle pattern element against working memory where we distribute the working memory to a series of processors as well.

Yet, when you have a large number of rules, a large number of patterns, and large amounts of working memory, you need a large number of processors, quite right. Thus if you need a large number of processors, you need a very efficient implementation. What is the importance of parallelism in the recognition phase? Study of one class of production system's, the OPS formalism has shown that 90 percent of the time's spent in recognition. So if we can speed that process up, we're doing pretty good. The time in this process in general depends upon the size of the working memory and the production memory.

The anticipated effects that should really be stated as a future long-term goals i.e., to reduce that time to a constant. The time to match the rules we hope will be also in dependent of the size of the production memory and the working memory. That's a goal.

Implementation: We have machine called the DADO machine that we are building to realize algorithms to run these programs. The essence is a large number of moderate to small processing elements. Moderate meaning 8-bit, perhaps 16, parhaps 32-bit processors within the order of a few thousand bytes of memory and a high speed interconnection network embedded with in a binary tree.

Future problems? The granularity problem, which can only be determined by studying application programs. How big should the processing element be? We don't know. How about parallelism and the select phase. That turns out to be not as important as recognition. You can solve it by logrithmic time operations especially on a binary tree structures. But one doesn't really get substantial ipmrovement of speed. You really win in recognition and you win in the action phase.

There are two sources of parallelism in the action phase. One is action on multiple rules concurrently. That's somewhat like multiple selection. But again to select the rules that you wouldn't in fact execute a simple operation. The hard part is if you have a set of rules that you would like to apply them concurrently, that's a tricky problem. But it is a source of parallelism. Another source of parallelism is getting away from the traditional OPS-style of temporally redundant production systems and to allow massive changes to working memory on the righthand side of a rule. If one can implement that in parallel, one gets tremendous performance improvements. So two sources are: firing multiple rules, and massive changes to working memory.

Anticipated effects: Something interesting that happened this past year, we had a visitor named Toru Ishida from NTT, who worked with us to look as this very issue for OPS-style production system programs and he found that by suitably relaxing the way one writes production system rules, he achieved a factor of 6 to 10. It means you can fire 6 to 10 rules on average on each recycle. That's not bad.

The way we implement the DADO machine is to partition the rule set to minimize synchronization. A prblem for future research is how does one really adequately restructure the rulebase to remove the sequencialities imposed by the formalism. I'm going quickly because I have a lots of slides. The last parallelism issues is how to manipulate working memory in parallel. In this case, if we treat the working memory as a large data base and permit massive changes we can get quite a bit of performance improvement. There are many examples of systems that could use such an ability for large data base applications like the ACE system. The anticipated effects? ACE presently runs on a VAX 780. When it runs nobody else can get near the task.

It is a true AI program. Our projections from our studies indicate it'll run in minutes on a the DADO machine which is smaller than the 780.

Implementation, on the DADO machine requires distributin of working memory and processing of working memory in parallel. A problems is again the granularity issue for the PE. Also how does one distribute, I, partition, working memory.

That ends at the technical portion of this presentation. We go on to philosophy. For what we've learned from dealing with production systems exclusively over the past five years is that the present formalisms that are available have, of course, been invented within the serial programming environment. Much serialization is imposed on programming, when you deal with these formalisms. They do not encourage parallel thinking although, you would think a purely declarative formalism like production systems has so much inherent parallelism. There's a lesson here. When serial programmers who've been programming serial machines for so many years are given a formalism that you would think is parallel, they still think serially and program serially. Parallel processors must be made available as quickly as possible to experiment with more expressive formalisms. I'll give you a quick status report.

We have an architecture called DADO. DADO is a 15 processing element machine that has been running since April of 1983. DADO2 will soon exist with 1023 processors, and in fact we'll be implementing ACE and hopefully Digital Equipment Corporation will let us use the program as well. We are implementing a more expressive productive system of formalism that goes beyond OPS and of course we are still dealing with logic programming through the LPS system that was talked about earlier in the week. Eventually we hope to have a machine which would have many more PE's than DADO 2 and would have a VHSIC or VLSI implementation.

Question; why is this important to you? What I want to say now is more of a philosophical warning, but certainly not an indictment.

Question; how many people in this

room have implemented a real world expert system that meets the following conditions? It performs an important or useful task, makes someone's job easier and thus increases productivity, has been demonstrated to produce cost savings in its implementation identified the market, and made a profit.

If we can't meet all those constraints let's try to meet two of them. To put it another way, is the fifth generation computer a solution in search of a problem? What are true problems? Why do you want fifth generation computers to run fast? Has anyone demonstrated that the programs which run so slow that present machines are useless? Has anyone demonstrated that an expert system that is so slow is not practical? I don't know of any. To make an the inevitable grand leap to the fifth generation, we must first train, as is going on now, the existing and upcoming generation of programmers and researchers.

But we must also train users, because they're the ones who are going to buy it. They are the ones who are going to have to live with it, not program it. The users have no idea what AI is, how it works, what it means. Now, we must start tapping and defining the market, before we build fast machines to execute something when we don't even know what it is. How?

You have to first build applications define the markets, before you begin building hardware and looking at high-speed processors for particular formalisms without ever having built a substantial program in that formalism. This has to be done to produce first working expert systems which can be sold at high volume, which can become part of the mainstream of the computer uses in this world. The mainstream of computer use are in the data processing industry, running big IBM mainframes for data base programs.

How are they going to use an expert system? What will an expert system do for them? Then how do you identify and define the requirements to make those programs better, better in speed and better in terms of how you use them, then you can invent a fifth generation computer.

Necessity is still the mother of invention. I don't know what the need is. So thus we must define the necessity.
Thank you.

Chairman: Well, the presentation of the positions has finished. So let's proceed to the question and answer period. Are there any questions or comments?

If you have any, please come to the microphone nearest to you, and please tell us your name and affiliation first.

Prof. Iann Barron, Inmos: I have been listening to the conference and I have some slightly general comments to make. One thing must impress everyone, I think, is how difficult it is to implement Prolog in a sequential environment. By comparison with a straightforward programming language, the implementation is extremely complex and extremely difficult. If we then add to those problems the question of concurrency in systems, perhaps we are multiplying the difficulties of the systems we are trying to explore. I'd like to suggest that really there are two aspects of intellectual endeavour which need to be addressed to meet the fifth generation objectives. The first of these is that, let me go back slightly, I think we can characterize the computing elements that we have used so far as being first order computing, first order sequential computing elements.

We're now asking the question of can we make concurrent computing sytems, and that is one aspect. The second aspect is the question of can we make higher order computing elements.

I take a first order computing element as being program taking data values as its arguments and a higher order computer as a program taking other programs as its arguments. I wonder whether we should separate these two questions and investigate them separately, rather than confusing them as we are all tending to do at present. If I just look it to the panelists and try and see where they are on this hierarchy, Dr. Amamiya is looking at data flow. He's looking at it in higher order context. It seems to me we haven't addressed the question of whether data flow is a valid way of exploiting concurrency in the first order context. Indeed, I would have consideable reservations for myself as has been pointed out down the table in that data flow tends to fail to exploit the locality in programs.

To David May, I would say he has addressed explicitly the problem of concurrency in first order systems. He has created the system and in fact we have not observed its capability. But then he has to address the question of higher order programming. To Ehud Shapiro: his position is that he's adopting a relatively simple method of exploiting concurrency in a higher order computing system, and that is simplifying the problem to some extent. I would be concerned that his starting point,in terms of a Prolog-like language, may constrain his ability to explore those questions. Indeed, I should have observed a bit earlier on but I forgot I'm afraid, one of the characteristics again of fifth generation systems is the arbitrary starting point of Prolog. Prolog has some very attractive properties and it has the serendipity that some of the implementations of Porlog can exploit concurrency. But maybe it is just serendipity.

To Mr. Stolfo, my impression, and I don't know his work, is that he has been addressing the question of a higher order machine and is putting that into a simple concurrent environment. It does seem to me that there's not been sufficient investigation of higher order machines to really move on to the question of their concurrency.

And now I'll come back to Dr. Furukawa, and to ICOT and where is ICOT in all this? It's starting off with an arbitrary language environment. It is trying to put in concurrency in a form which tends to be not well explored, and that is the data flow type environment. it is trying to explore the possibility of higher order computation.

ICOT is addressing these three problems together, and I would ask, is that too much as a single step?

Chairman: Could you summarize your question?

Prof. Barron: I think that ICOT has taken an arbitrary point to start investigating these problems. I think it is investigating first the quesition of nature of higher order computation and the question of concurrency together and taking both those in a very difficult form. And I wonder whether this is wise.

Chairman: So please respond to his questions, Dr. Shapiro first?

Dr. Shapiro: I thought I was presenting a rather conservative view when I was standing there, trying to discourage all sorts of high hopes and large steps. You seem to be presenting probably an even more conservative position than mine.

So I guess at least we both agree on the conservative side. That's a starting point, and then the second and the real question is how conservative do you want to be? The proof will be really in the actual implementations. You claim from the beginning, or your initial position is that languages like OCCAM are implementable and can exploit concurrency and I hope that we'll see that in a few weeks, month's. No, its just a joke. I think the gap between implementing OCCAM and implementing Prolog-like languages is not as big as you think. I think for Prolog, this has been demonstrated to some extent, as there are more implementations of Prolog in C than I can count. It's more or less clear that doing a mediocre Prolog system is very, very easy. Two or three years from now, it'll not be counted even as an honors thesis for the undergraduate. I have an undergraduate who didn't receive an honor for that. So it's just a matter of development of technology. It's true that doing a high performance Prolog is still harder, and the only real answer to your skepticism is just in delivering the results. There isn't that much to talk about in assessing how easy or hard it is. We just have to do it. See whether it's as hard as you believe or less.

And another comment is conceruning Prolog the so-called arbitrary strating point of ICOT.

Perhaps you Dr. Furukawa will respond?

Dr. Furukawa: Yes, to pursue these two targets simultaneously is very difficult, I know. But, what I think is important is that we at least want to make a model of an entire system. In order to get such a model of information processing in parallel in a computer architecture, we need to investigate many aspects simultaneously. And once you succeed in making a model, maybe very vague, then you can start to polish that model. I think we need such a model for this type of longterm research. Is that an answer to your question?

Dr. Amamiya: In very fine grain data flow the one advantage of the data flow scheme is that very high parallelism can be exploited by using the fine grain data flow concept.

But on the other hand, the dataflow scheme has an confining or exploiting locality. As I pointed out, the coarse grain, or medium grain data flow level should be set in the data flow architecture in order to improve the communication overhead using locality. For example, one function or one process should not be distributed to several processors, but be allocated to one porcessor.

Within a processor, the computation will be done one at a time due to the constraints of the hardware structure. From this point, the fine grain data-driven control is not appropriate. Therefore, as I pointed out, semi-static scheduling based on the data dependency analysis is necessary and this coarse grain data flow control feature shouled be adopted in the hardware implementation. On the other hand the software concept should keep the data flow and functional semantics.

Prof. Edward Feigenbaum, Stanford University, USA: I'd like to just make one comment and then answer Professor Stolfo's question. The comment is that I think all the people in this room are heroes in a sense in this struggle. I hope someone is taperecording this panel for replaying in 1994. And we will find every, all of these discussions very quaint and amusing after all the dust settles and the real paradigms emerge from this confusion. We'll be amazed that we were so bewildered at the time.

Now, to answer Professor Stolfo's question. Maybe I'm one of the few people who qualify on the list and I certainly believe that designs should be applications-driven rather than the result of a combination of interesting gadgets, whether the gadgets come from electrical engineers or mathemaical logicians.

Now, it's certainly true that there's nothing the housewife needs that requires the second stage of ICOT work or anybody's work on parallel computation because the time taken to do simple advisory functions for the housewife is trivial even on personal computers. But there are important applications where we are orders of magnitude off in the ordinary AI routines of generate and test and chaining of various kinds.

For example, the earliest experts system DENDRA which is used now by the chemical industry, requires for interesting problems using LISP or BCPL anywhere from, say, ten to twenty hours on a conventional computer like DEC-2060. Applications to certain defence problems that fall into a class which I call signal to symbol transformations or signal understanding require immense amounts of computing. I won't speak directly about the applications, but the understanding of massive amounts of signal data has been prototyped in LISP machines and has been measured to be between three or four orders of magnitude off real time, and real time is necessary. That's already existing prototype systems that we can measure on LISP machine. Incidentally, the base for the measurement is the XEROX-1108.

Looking into the near future, we see an application unfolding at Stanford involving collaboration between my laboratory's heuristic programming project and the chemistry department for analysis of the structure of proteins in solution, not crystals, from NMR data, and that is another problem where we appear to be several orders of magnitude off of the ability to realize the current ideas. The computational speeds are not fast enough for significant applications to science and engineering. The application that was mentioned in the ICOT presentations was the expert systems of AI applied to VLSI CAD. Early work on that essentially failed not for lack of ideas, but because we were about ten to the third off usable human speeds in these systems. Where will we get this speed? The economic imperative drives us to parallelism because improvements in serial computation will be incremental. We don't need 15 percent more, or 50 percent more. We need three orders of magnitude. The chipguys have given us a new technology. It's the difference between printing and manuscript writing in the middle ages. Building an ordinary computer is like manuscript writing compared to printing computers on silicon. So the only hope for giving me my three of four orders of magnitude is to investigate parallel processing, which of course, is what motivates you, myself, all the ICOT people, and everyone else to do this.

One last comment. I've only talked about today and tomorrow, but let's look at the real future. In the real future, there are some of us in the room that want to create remarkably intelligent machines, not simply good problem solvers to help chemists but intelligence of the highest order in computers. That will require certain kinds of learning techniques for which the early experiments like the URISCO experiments of Lenat have shown enormous amounts of computing power are necessary to combine knowledge stored in a knowledge base into new knowledge structures that are useful in building up even further knowledge structures in the machine.

Lenat himself for example uses networks of XEROX LISP machine at XEROX park. He uses all available LISP machines at night by stringing them together in big networks. So for the truly intelligent, the very intelligent machines of the far future, we are going to have to master those parallel computation mathods or the machines will simply be too expensive or unrealizable.

Chairman: Thank you for your interesting nice comments. Would you like to make additional comments, Dr. Stolfo?

Dr. Stolfo: Oh, yes. First of all, I'm quite well aware of computation-intensive tasks. That's one of the reasons why I'm building a parallel processor. There's absolutely no disagreement. The issue is though how do you get those three or four orders of magnitude, how do you know you need them. You know you need them not from studying a specific expressive formalism. You find out by studying problems and trying to tackle those problems and getting applications to work. Then you see what computation resources you really need. In constructing programs it's not just simple notions of search and bit patterns in a computer memory which people call a knowledge representation.

The real issue is: What is the knowledge you need to put into the program, and the only way you can learn about those sorts of issues is by getting your fingernails dirty and implementing such programs. That was the warning I was sending. Now, the parallel processing in some senses is a suffering from chicken and egg problem. I think focussing on speeding up a particular formalism may be misguided without application first.

Prof. Feigenbaum: Just one last comment. The fact that the housewife may not need it is not totally relevant. The VLSI CAD designer of 1992 may need it when that person is designing 500 transistors per chip.

Chairman: Next question?

***Dr. Krutar,* U.S. Naval Research Laboratory:** I want to challenge Professor Stolfo's claim that necessity is still the mother of invention. It maybe a mother of invention, but if you look around at most of the inventions on the market, they're not necessary. They're there because they want them, not because they need them. And you analyse the market in terms of what people will buy, not in terms of what they need. Any bartender knows that.

I disagree with you.

And relating to the experts systems for the housewives, I think that one of the largest markets we would have would be, especially since I'm a proud father of a very active three-year-old, well it would be very nice to have an expert system for raising children. I'd like to know when we can get one on the market with prices we can afford, and I'd like to point out that there's all kinds of motherhood statements we could make about a system like that.

Dr. Stolfo: How does one respond to that?

It doesnt' much matter whether it's a difference of want and need. It still must be done. The fifth generation is quite clear what it's saying about the next generation of computers and what those computers might be or might do. But still I don't know what they really will do. It's not just the point of views of the masses buying it. I just don't think it's quite clear for many people working in this field: what am expert system is in the first palce, and that's the real issue. This program must be constructed first on available hardware, and then evaluated. I'm being repetitive. But one of the things that occurs in computer science, is that computer scientists in general never really fully explore what they have already invented. They're aleady looking for something that's better than what already is and is not sort to state-of-the-art. But I say the opposite. I think what exists today in computer technology and the computer languages without parallel processeing are still fairly competent devices. The VAX 780 and ACE is a single example.

Dr. May: Can I add the microelectronics industry, however, is hardly ever application-driven. The INTEL 4004 was designed to control cash registers not to be used as a general purpose microcomputer. Certainly from our experience, I don't think we had much idea as to exactly what kind of application areas people might have in mind, for transputers when we started designing them. It's now very clear that there are a large number of applications particularly in areas of computer aided design. Which we failed to anticipate, for which the device appears highly suitable. These include things like finite element analysis, molecular simulation, fundamental physics graphics animations currently used for flight simulators, but that will be new year's arcade game. Finally circuit simulation, which we really have to have or else how do we build the sixth generation computers?

Dr. Shapiro: I'd like also to respond to this issue, and let me start by saying that Prof. Stolfo's criticism of the presentation of ICOT is based on what perhaps can be called the American fallacy, and the American follacy is that the fifth generation project is about building expert systems. Well it is not. I can't really speak for ICOT, but I can try to present my view. You may mont to call it the Israeli fallacy about the fifth generation project. I think the fifth generation project is about building a new computer technology. And why do we need a new computer technology? We need new computer technology because existing computer technology has two very rare problems. The first problem is that computers are too slow. The second problem is that programmers are too slow. These are the two problems of current computer technology. In this case, these are the problems that the new technology attempts to solve. Perhaps the insight that the fifth generation project offers, and this could be also an answer to Iann Barron, is that these two problems can't be solved independently. You can't solve one of them without solving the other. That's why you need this overturn leap of solving the two of them together. Why can't you solve each one of them separately? You can't stay with existing von Neumann languages or low level von Neumann languages and offer more powerful computers in an attempt to solve only the first problem that computers are too slow. This is because as we know programming is painful as it is. If we try to add concurrency & communication to low level conventional programming languages, it will be really a setback for programming. So if we try to solve only the parallel processing problem or the slowness problem without addressing the programmer productivity problem, we won't succeed. On the other hand, we can't solve the second problem independently. We can't increase programmer productivity on existing machines. Why? Because even with the best minds and best tools, higher level languages still run too slowly on conventional computers. That's why we must resest to low level long vuges to solve computation-intensive programs, which means a lot of hard programming. That is why, in order to produce this new technology which solves both problems of slowness of computers and slowness of programmers, we must advance in parallel, or together an both problems. We must produce a parallel machine that can execute a language which is not inferior to the best languages we have today on sequential computers. And I guess that's the key insight of the fifth generation project and the main thrust of their approach as I see it.

Prof. van de Riet, Vrije Univ., The Netherlands: I have a more technical question. I would leave the matter of philosophy of housewives. As I understood, the fifth generation is not only meant for expert systems as has been said now, but in particular for solving problems in software engineering. In the fifth generation, the first language KL-0 is based on Prolog, on sequential Prolog, and KL-1 is based on concurrent Prolog, there's always a problem of how much parallelism can really be exploited. We have seen some studies about the four queens problem, about the six queens problem. Another problem I didn't hear mentioned is the eight queens problem, but maybe that's for the future.

There is a very good example now available of a software engineering problem that is the SIMPOS operating system which, as I understand, consists of 30K of Prolog lines of KL-O lines or SIMPOS lines. I don't know precisely. Now, I have two technical questions. The first is this: Current Prolog is sequential, and for many things in operating systems you need sequentiality. You gave to say first this and then that. If you go to concurrent Prolog to KL-I, you gave to change that. You can't use the sequentiality of Prolog, but you have to add things in (gourds). Maybe even flags. The flags are of the kind, this statement first, then that statement.

My first question. Is that being looked at? Is that studied with the SIMPOS operating system at hand? Is someone looking at this porblem? How much does it cost to change the sequentiality into non-sequentiality?

The second question is: If you look at the SIMPOS program, and you look at parallelism, you really can see now how much and-parallelism is possible, the same for all parallelism. Is that already being undertaken? If so, what are the results?

Dr. Furukawa: I have to make your question clear. Is your first question, do we have any idea how to convert sequential into parallel control?

Prof. van de Riet: Yes, how much of the sequentiality of Current Prolog, are you using in this software engineering program which is called SIMPOS?

Dr. Furukawa: And the second question?

How much do you expect to exploit from the inherent parallelism which is available within that SIMPOS operating system?

Dr. Furukawa: I think we need to rethink, rebuild in concurrent languages. There's lots of common features if we think of the specification. If we extract the specification part of SIMPOS, then it is possible to convert that part into a concurrent logic programming language. It might be difficult ot achieve the direct translation. However, still I think it's much easier than the case in convention languages. And is the next question about how to extract parallelism in the SIMPOS system?

Dr. Shapiro: Have you ever tried to evaluate how much parallelism is involved in SIMPOS?

Dr. Furukawa: No. SIMPOS is a very new operating system and you know, at ICOT, we have had only two and half years. There has not been enough time.

But I think Dr. Shapiro will comment on this.

Dr. Shapiro: I said before I don't believe in discovering parallelism en passant. You have to plan for it and you have to program it explicitly, at least today and for the next few months or years. So that's why I don't believe in taking SIMPOS and converting it directly into concurrent Prolog. A lot of the complexities of SIMPOS result from the lack of concurrency in Prolog. So the designers of SIMPOS were forced to include constructs to Prolog to support message passing and object-oriented programming, because it was not in the basic mechanism of sequential Prolog. It is my hope, my belief, that if an operating system is written directly in Concurrent Prolog, you would need fewer extensions or even no extensions whatsoever to the language to implement an entire operating system in it and hopefully the result would be much smaller and much cleaner. What you do have to provide is some stream like interface to the different I/O devices. But it seems, at this point of our research, that this is all you need in order to write an entire operating system in Concurrent Prolog. I believe, that an operating system written in Concurrent Porlog in the right way. It will be completely different than SIMPOS.

Prof. Arvind, MIT, USA: I have some comments, questions actually. One point that was made by both Mr. May and Dr. Shapiro is that it's premature to exploit parallelism automatically. I think I disagree with that, because, first of all, we know how to code algorithms so that all the parallelism in the algorithm can be made obvious, all the inherent parallelism in the algorithm. So I consider the problem to be already solved.

We also have fairly good models for machines. The only question in what Dr. Furukawa was saying, we need some tools to actually evaluate and tune, you know, so that we can observe what happens when we take a realistic application, write it in the these languages, and run if on these new machines to figure out whether the parallelism is exploited on the machine or not. That's a very hard problem, because the amount of effort that is involved in building a machine on which you can run a large program is really massive, because if we could simulate that sort of thing on conventional machines we won't have to do these Mickeymouse......

I guess I'm making just two points. One is that we know how to code algorithms so that the parallelism that was present in the application code is obvious.

We have representations, data flow graphs are a very good representation for that.

The second point is that we do have very good abstract models for machines which can exploit parallelism. What we do not have yet is, I mean we haven't taken a large applications coded it in one of these languages, generated data flow graphs, and run it on one of the these architectures to find out if we actually exploited the parallelism. In order to do that, we have to build lots of tools. We have to build very good compilers which are fairly robust. We have to build machines which will stay up, you know, for a week or two and don't go down every ten minutes. These machines can't be small machines because if they are very small machines we could have perhaps simulated the idea on conventional machines. And that's where the state-of-the-art is in this game.

We are in the period of experimentation, and that's the only way we're going to find out whether we're on the right track or not. That's the only point I wanted to take.

Dr. May: I don't think I disagree with that? I don't think I actually said it was necessarilly premature to look for automatic ways of doing parallelism. I just think that there are not automatic ways of doing parallelism. I just think that this stage a large number of applications that can be tackled directly without automatic ways of using parallelism. I certainly agree that it's worth while to be able to try doing some experiments with automatic parallelism and that large scale simulation will probably be necessary.

Presumably one would want to design large scale simulator in a rather more straightforward rotation which doesn't itself make use of large scale parallelism.

...Sorry, which doesn't make use of any automatic analysis of parallelism. Which brings me to another point I've been wanting to make for half an hour or so. One of the things which puzzled me about the use of Prolog and Prolog-like languages is why one should feel it's necessary to use Prolog-like languages at the all levels of the implementation. It seems to me that at the micro-code level, I'm not convinced that a Prolog-like language is the best thing to use. At the level immediately above the micro-code level the languages of the level of C or OCCAM are fine. At the level above that maybe one could use these languages to implement Prolog. This kind of layering has been common for many years. It's a highly effective technique.

Certainly one presumably wouldn't use Prolog for going even through the micro-code level down to the transistors. So the thing that puzzles me particularly about the idea of using Prolog-like languages all the way through is, why? We already hve some good tools at the bottom levels.

Dr. Stolfo: Can I just make one last quick comment that I wanted to make less than 20 minutes ago, which is a response to Ehud Shapiro.

What is the term, American fallacy 3, that the fifth generation computer is not an expert system machine? Is that a good parphrase of what you said, yes?

Dr. Shapiro: You said the American fallacy is that the fifth generation project is about building expert system machines.

Dr. Stolfo: Fine, then, correct everything I said over the past two hours and replace every occurente of expert system by fifth generation program. Who has written a fifth generation program what is fifth generation porgram? In my estimation, the best example that exists today, is an expert system program.

That's the point I want to make. And also how does one develop new technology? You said the fifth generation computer project is trying to develop new computer technology? That's great. How does one develop technology? That's not science, that's technology. You develop a technology by good science. And what is good science? One develops a theory to explain phenomena, construct experiments to test the theories out?

Perform the experiments and evaluate the results. What are the theories? How can one ever develop a theory. You can't develop a theory without the phenomena. So what are the phenomena? Fifth generation programs. That's my point, thank you.

Dr. Furukawa: I want to say someting about the invention of LISP. LISP used to be a not very well-known language. The purpose of LISP at the start was not to develop an expert system, I think. But now it's a main tool used to develop expert systems, and what I'd like to say is, it is a nice language in itself. It has great expressive power and a new culture will grow up to top of such a nice base. That is, I think, included in this project.

Chairman: Well, I understand that we still have many questions and comments on parallelism, but it's almost time to close this session. I believe we've had a very informative panel discussion and I'd like to express my thanks to all the panelists for their valuable presentations and discussion, and also to all of you for your cooperation. Thank you very much indeed.