*PANEL SESSION:*

# Parallelism: Are We Still Interested?


Moderator: **Mario Tokoro**
Associate Professor of Department of Electrical Engineering at Keio University.
Computer architecture, knowledge representation languages and distributed systems.

Panelists: **Charles Forgy**
Research Computer Scientist at Carnegie-Mellon University and the President of Production System Technologies.
The Co-principal Investigator with Dr. Alan Newell of the Production System Machine, PSM Project at CMU.
Developing efficient parallel interpreters for production systems.

**John Gurd**
Professor of Computer Science at the University of Manchester.
Parallel computation, data flow computers, single assignment languages, program transformation, and formal methods of hardware design.

**Randy Katz**
Associate Professor at the University of California Berkeley.
VLSI processor architecture, CAD tools, and data base systems.

**H.T. Kung**
The Shell Distinguished Chair in computer science at Carnegie-Mellon University.
Parallel architecture and computations.

**Hidehiko Tanaka**
Associate Professor of Department of Electrical Engineering at the University of Tokyo.
Computer architecture, inference machines, knowledge base systems and distributed processing.


**Moderator:** Good evening. Let us start the Panel Session titled "Parallelism: Are We Still Interested?", such a very controversial topic.

I would like to run this session so that each of the panelists will first give a position statement and then we will invite questions and comments from the audience.

First of all, I would like to clarify what the panel's title means. When we talk about parellelism, first, we have to think its granularity in a problem domain at both representation and execution levels. If we have very large parallelism in the problem domain, we need and maybe we can employ all kinds of parallelism at the execution level. To do so we have to represent such parallelism in terms of some representation languages. Otherwise, parallelism has to be automatically detected and mapped onto executing processors.

The second thing we have to think about is whether parallelism is static or dynamic. "Static" means the parallelism is predictable before execution. And we just need static mapping for this parallelism. But for the dynamic case, it is unpredictable before execution, and the parallelism could be finite or infinite. And we need dynamic mapping.

The last thing about parallelism is the order of magnitude. The panelists are requested to define parallelism on the order of 1, or 10, or 100, or 1,000 or 10,000, or one-million. And they should specify when it will be available: 3 years later, 10 years later, or 20 years later. Now, let us start.

**C.L. Forgy:** What we have been doing at Carnegie-Mellon for the last couple of years is to speed up the class of programming that we call a *production system.*

In the production system, a rule is basically laid, which says: "If a certain condition is obtained in the data at this given instant, then it is appropriate to perform this set of actions."

The parallelism in production systems is of very large grain and it is on the order of a few tens of processors. It is very dynamic. Everything has to be assigned at run time but it is predictable. We know that parallelism never gets very high though I won't be entirely pessimistic here.

I want to mention that there are some avenues open to increasing the amount of available parallelism. The most important thing at this time seems to be making parallelism explicitly available to the users of the production system.

All the work I have been talking about so far is just making an ordinary OPS-5 like production system. It means that we are presenting the standard sequential semantics to programmers. And if we actually make it apparent to the programmer that he is fronting a parallel machine, and allow him to have multiple


This manuscript was transcribed by Mario Tokoro and Yumiko Okada from a recording of a panel session held during the 13th International Symposium on Computer Architecture at the Sunshine Prince Hotel in Tokyo on June 3, 1986. Editing has reduced the content by about 40%.

threads of computation running on in his production system, then we expect we can get some useful amount of speedup.

Well, we are getting here a multiplicative effect, perhaps a factor-of-ten parallelism inside the interpreter. Of course, it's highly application dependent. That's where we had an application which permitted an average of four threads to be active at any time. Then, we get a further factor-of-four parallelism at that level. And in total, we get a factor-of-forty parallelism.

**J.R. Gurd:** Of course I am still interested in parallelism. I have my list of possible reasons. Top amongst them is that we want to do big computations in as little time as possible. Next, it may be more cost effective to build a particular powered machine out of a limited number of lower power machines that are cost effective. Also, manufacturers might be interested in the concept of extensibility of the computer architecture, i.e., the idea of having a building block computer which is extensible.

And lastly, it might be able to achieve higher reliability in systems by essentially having redundant devices in your system. I feel it will become more and more important in the long term. Because it seems clear that you can't have reliability by redundancy unless you can take parts of the system out and still have a working system. And essentially, that's the extensibility property being applied in reverse.

I think that there is a need to go towards multiple instruction stream machines which probably weren't there previously. So, it seems that the computer architecture world is arguing about the detail of machine design, not whether you actually want parallelism. I think that's an indicator that everybody is willing to go in this direction rather than just a few weird academics.

These are unresolved issues: whether we should go into fine grain or coarse grain at the machine level. What I strongly suspect is that we will find that there are different application areas that are suited to different paradigms. But, at least in the fine grain machine that we've been looking at, we are not worrying about finding parallelism; we are worrying about finding too much of it. And it is a practical problem for us to keep parallelism under control.

I think that the sort of areas we are looking at are on the order of ten to a hundred-fold hardware parallelism. And in order to keep that active, we need software parallelism of the order of about ten to a thousand fold. I think anybody who tells you that you can look at bigger parallelism than that within the next five years is kidding you. I think we have got to learn to walk before we can run. And this is about the limit of our achievable sight at the moment.

**R.H. Katz:** To the question of "parallelism: do we care?", obviously my answer is yes. The observations are that uniprocessor performance is topping out. Most of the speedups have been coming from device technology, not from new things in architecture. And we are going to reach some point, and I think it is now, where it's just going to be cheaper for you to add more processors to your system than trying to make the uniprocessor goal a lot faster. So, we are going to have multiprocessors.

For me, systems with a small number of processors are interesting, rather than having millions of millions of processors, since I have been working on such systems. So the questions are how do I harness parallelism for general purpose computation and what is the cost of such systems. I agree with the previous speaker; we should start with small scale multiprocessors since we need to learn how to walk before we can run.

Now the processor that we are working on is in the range of hopefully not one to ten but nearer ten than one. It's called "Super Symbolic Processing Using LISPs". It's a workstation sized shared memory multiprocessor. And the motivation for it is to provide a test bed for developing multiprocessor applications. The configuration is very simple. And even this machine has parallelism at many levels. The processor, of course, is pipelined, so there is some parallelism at that level. We have multiple co-processors, and functional units associated with each processor if you want. We also have multiple processors in the system. So, there are many different levels of parallelism, even in such a simple multiprocessor as this. And we have a lot of work to do to handle all that.

So, the questions I would like to leave you with are: for general purpose computation how do you harness parallelism? How do you parallelize a text editor? How are you going to do that on your connection machine? And how are you going to parallelize the compiler? Yes, there is some parallelism that can go on in a compiler, some functional partitioning in pipelining of the compiler. But it's not on the order of tens of thousands. It's maybe less than ten processes that could go into that. So, the bottom line is that even small-scale multiprocessors are interesting.

**H.T. Kung:** I was asked to talk about why are we interested in parallelism. The very fact that this question was asked surprised me. But, then I started thinking: "probably the organizer of this session had some more profound reasons behind the question." And I realize the answer is not so obvious.

First of all, we have to find out what we mean by "we". There are at least four kinds of people I can think of. First, researchers in computer architecture. They are interested in parallelism. Then, maybe, the funding agencies are not interested in parallelism. The third kind of people must be the people in computer industry. If you look at the new computers, they are all parallel computers one way or the other. You can hardly find any respectable new machines that are not parallel machines.

It is very hard to find those who do not like parallelism. How about users? Every user wants more

computer power, a factor of ten to fourth or ten to fifth speed can be given by using parallelism. So, users cannot be the ones. They love it. So, everybody likes parallelism. I don't know why we have this panel.

So, I started realizing; probably I was thinking about something wrong. Maybe the word "parallelism" has to be defined more carefully. But there is nothing wrong with it. So, I started thinking again. And I realize there must be a group of people who are not happy with it.

There indeed are a group of people who had high hopes for parallelism and have been waiting for the last fifteen years. They haven't seen much coming. I started to realize they asked the question. It's the people who are losing patience. They say, "You people have been talking about this for so many years without any real machines coming out." Then, the problem is not parallelism at all. But it's about a new kind of architecture.

Any new architecture would take a number of years of development. You just cannot get anything done very quickly. Parallel machines have not been used so rapidly in a popular way because we are building new machines. Any new machines take time to mature. I don't know any reasonable machine that can be built in less than five years. Five years is just one cycle. You probably have to take two or three cycles before the machine becomes reasonable.

Now, I'll answer the questions. One question is parallelism: why we talk about parallelism. The other one is when. We are talking about parallelism because we believe we can build the machine cost-effectively. There are people who build thousands and thousands of processors all the time. The questions are how you can use them cost-effectively and how you can build the machine cost-effectively to be competitive with other kinds of machines that people can get access to.

One area definitely, I believe, where we are going to make it is not the general purpose area. The general purpose area probably could get ten-fold speed up in 5 years, but that is not going to solve all the problems many people are interested in. We have to work on special purpose machines, where you can have the opportunities to make parallel machines to be cost effective. For example, all the signal processing machines are parallel machines. The people who deal with signal processing are all building parallel machines using thousands and thousands of processors working in parallel.

So, special purpose areas definitely will employ more and more parallelism, and we are going to move toward the more general, more flexible areas by having better building block chips for them or you can buy those chips from radio shops. We can buy nice buses on which we can hook them to make highly parallel architectures. We can have better compilers that can use those special purpose engines. So, we can move from very special purpose to less and less special purpose. We hope we are eventually going to do more in general purpose area as well.

**H. Tanaka:** My position is we should go in the direction of general purpose machines of highly parallel architecture. Why general? In case of special purpose machines, the organization methodology has been almost established. Application analysis and architecture design have been tuned to the applications.

And why highly parallel? Multiprocessor technology of small scale parallelism is practical and very nice. But it's not attractive as the research theme of a longer term. As a university person, I must research this highly parallel architecture.

Applications are transformed into programs using some algorithms and languages. And the programs are transformed into computation models. An example is dataflow or reduction or imperatives. Then the model is transformed into the execution model and after that the execution model is executed on the hardware. Hardware is implemented using some VLSI. So, many kinds of parallelism are available: task level, computation models level, logical operation level, and circuit level.

For the task level parallelism, an object model is very good. And for the computation model level parallelism, a logic model will be very nice. And at the logical operations level of parallelism, we must use pipelines, and at the circuit level, we will have high speed devices such as gallium arsenide.

My focal points are in computation models and execution models. In the computation models, the concept of granularity is very important. However, it is not enough. We must focus our attention on what kind of granule is used as well as the size of the granule.

The candidates are the functional model and the logic model. The functional model is well matched to the data driven mechanism, and the logic model has a good correspondence to the semantics. It has a nice feature, that is, easy derivation of parallelism.

Then, there is another issue: the problem of allocating activity to the hardware. Each primitive of the computation model is broken into activities which are allocated to the hardware elements such as processing elements and structure memories. There are several algorithms for allocations.

In case of static allocations, there are two kinds. One is the designation by a programmer and the other is the program analysis by compiler. Designation by the programmer is explicit but program analysis is implicit from the user's point of view.

For the dynamic allocation, we must rely on real time monitoring. Using real time monitoring, we can achieve this kind of allocation. And all three types of allocation must be used and implemented.

**Moderator:** Thank you very much. I would like to invite some comments or questions from the floor.

**Audience:** I want to say that we shouldn't rush to get answers to how much parallelism and when it will be achieved. What we should be doing is analyzing the question of what is a parallel machine. And that ques-

tion should be answered over a long period of time.

One of the things that I did not hear among the panelists and I would like to raise is the notion of space sharing. Look at your time sharing computers. There are usually forty different users that are given time using part to the machine, using time slices.

The reason that time sharing makes sense is that from time to time you can use it to solve large problems such as weather prediction, but it pays for itself because a lot of small problems are run on it for a long time.

What we need is an analogous situation in space sharing. The resources of the machine can be partitioned into small parts. Each part can support a user. And then, from time to time, you can run a real big problem on it and it is useful sharing a computer, as opposed to having a lot of personal computers. We must remember space sharing in parallel computing.

**Audience:** If I look at the past of parallelism, I see two kinds of things. I see computer architects going into very interesting architectures without really wondering so much about languages. And I will put Cm*, C.mmp, and many others in that bucket.

And I see people going after very interesting, stimulating models and taking dataflow as one of the models. I see none of these approaches are really going somewhere. And I haven't heard the panelists mention that a good, well-specified, meaningful extension to a language can make parallelism possible. And I see a lot of nice practical, meaningful and usable languages coming into play, and Multi-Lisp might be one of them.

I see computer architects doing a good job in helping these efforts, rather than going off and inventing yet another structure or trying to tackle fancy models, which are very interesting, but may be not what users would want to use.

**J. Gurd:** Well, I am inclined to agree, so I am not going to argue the case. It seems to me that the problem at the moment is one of finding our feet. And the research community is engaged in any way it can, getting a handle on dealing with parallelism more complex than the four-fold that's being offered in the marketplace at the moment.

Presumably, when they feel they understand hundred-fold parallelism, the research community can come back and say how do you want to approach this language with at least some guidelines about how you can. My feeling is that there would be restrictions to your approach to hundred-fold parallel machines. And it's a question of finding out what the nature of those restrictions are and coming back to you and telling you about how to provide these systems.

But I think an extension to C or Lisp or whatever else your favorite language is probably not the best way of approaching problem at the moment.

**H.T. Kung:** I fully agree with what you said. The biggest problem of all is the model of computation. It surprises me once every three month that there is yet another model that we think should implement. But even for small scale multiprocessors, we don't really know what we are to do in general, for even very basic things.

**R.H. Katz:** I think that's a very good point. It's only recently that we have been able to envision even small scale multiprocessors that can be generally distributed. And those machines have terrible software environments and no one really knows how to program them yet or how to harness the parallelism that is made available. Software is the major stumbling block allowing development of small scale multiprocessors as commercial products.

So, when people start working on applications, I think we will learn a lot more about how to program software for multiprocessor systems.

**Audience:** What I want to ask is about languages: is it really a matter of tiddles on new facilities on original existing languages, or should we try analyzing the bottom line out of existing languages, or really must we start from new languages? If I may express my own point of view, may be what we should be looking for this time around are languages with sound mathematical foundation.

**Audience:** I think one of the problems that we face is a chicken and egg problem and involves three actors.

First of all, we wish to build parallel computers so that we can get applications to run on them, and we design languages for the applications. Then, we design new languages for the parallel machines. And then we get applications on them, and we build better architectures and we build better languages, and we build better applications. It's a chicken and egg cycle.

Where do we start, though? My conjecture is that the weak link is the languages. The languages are way behind the hardware and in fact even behind the applications. I think that in this round of improving things we need to go through, we have to realize that languages are the weak link. It's been the tradition of computer architects to start with languages. That's how Bob Burton designed the Burroughs Family machines. He started with ALGOL and Beta machine to handle ALGOL.

But I think in parallel processing, that is the weak link, but perhaps it is the wrong place to satrt.

**Audience:** First, I would like to rephrase the chicken and egg question in a slightly different way. I will make this point, that it seems to me that quite a lot of proposals for parallel computers are solutions in search of problems.

Now I will make a real point, which was to respond to something H.T. said. He sait it's the users that want the parallelism. I don't think that's true at all. The users want high performance and low cost and that their program should continue to run. And you can sacrifice that last one only if you make great strides in one of the first two. I think that's the challenge.

**H.T. Kung:** That's the one with the difficulties here. You recommend this, however, I have not yet seen one user who is not willing to change his cult, in case you

give him one hundred-fold speed up over VAX-780. I haven't just seen one user who is not willing to change. So, the only problem we have is to give him a hundred-fold speedup. That has been tough.

**Audience:** I think programmability is absolutely essential. And in the long term, it's far more important than small factors in efficiency and speed. So, one of the things is that the programming language people and the parallel computer people have to talk to each other. And that should happen in many places.

**Audience:** What encourages me to get up and speak is that I was impressed by a lecture several years ago by a gentleman who unfortunately is not here today but should be. My ramblings here are simply to say that I believe the evolutionary approach will in fact be the path by which we go through this random walk of the search space that will eventually lead to machines ten or a hundred or a thousand times more powerful.

I do not want parallelism. What I want is performance, low cost, reliability and so forth. If parallelism can get us there, that's great. If not, then, may be we will find another way.

**Audience:** I really agree with the statements that you have made, and I think that a lot of us are working on evolutionary approaches to parallelism. And the idea is to give a guy a very high performance processor within your multiprocessor, and when he saturates on one of them, give another, and if he can't figure out how to use the other ones, then there is no hope for conquering parallelism.

**Audience:** I have to disagree with the statements. I think what we have here is a random walk which will lead us nowhere. If you are going to make your first step, you look at the plan you have and you decide where you are going, and make your step intelligently, so you won't waste your steps.

Suppose we decide to build programs for a machine with a shared memory for ten processors. Then, five years from now we have to throw those away because shared memory doesn't expand from ten processors to a hundred processors. And if we do that, we have to rewrite all our programs again.

I agree with making the first step. We have to produce machines that have ten-fold parallelism before we can go to tens of thousands. But I think we ought to do it in a way where we do not have to throw away all the codes we have written and start over again every five years. We should be able to use principles and use techniques that are expandable, at least architecturally. So that when we get better technologies, we can expand to larger and larger machines without having to discard our software.

**Audience:** I just want to ask one simple question to Professor Gurd. Could you tell us a little bit about what kind of software tools or software technology is quite effective to implement a parallel system with tens of thousands of processors?

**J.R. Gurd:** We are in a terrible shape about providing some sort of systematic way of approaching parallel systems, so that we can extract parallelism from the applications and see what's happened when we try and exploit that parallelism. But apart from the actual mechanical process of trying to extract parallelism and see what happens when you use it, I think you are asking a question about how, as an end user, do you go about getting that parallelism in first place. I think this is your question.

I think this is all tied together with the language problem. And we have already discussed the possible ways that might exist. One is to put some sort of analyzer into FORTRAN programs and explicitly to unravel the parallelism for the FORTRAN codes. And as I have indicated, I have a skeptical view of whether that's possible to the same extent as starting with a non-sequential language and using that as a means of expressing the problem.

And I should say that there are some languages in wich you don't have to express sequence. They range from the very high level languages that have been discussed here: the functional and logic languages, in which we have to note at the moment there are sequential mechanisms built in. So, for example, Prolog programs that do interesting and big jobs are full of cut statements, which actually make it very difficult to turn the codes parallel; and similarly with Lisp programs in the functional word, they are full of prog features, which essentially re-introduce sequentiality. So, even though those languages look neat from a mathematical point of view, you would still have to turn to a FORTRAN style analyzer on most of the real big programs in order to extract large amounts of parallelism.

So, it seems that on the one side the research groups are currently investigating new functional and logic languages which still maintain the higher level mathematical aspirations of the declarative languages but which allow some scope for parallel execution. And on the other side, there are people who are essentially building imperative parallel languages. Perhaps it's safe to call them "non-sequential languages", which is the sort of approach that we have taken on the dataflow project with the sizable language. I think the interesting thing is that there is a bit of convergence in that the non-sequential imperative languages look very similar to the declarative, at least the functional ones.

But there is still a lot of work to be done before those notations have worked out sufficiently well that we can guarantee:

(a) Their mathematical cleanliness in transformability and so on, and

(b) Their implementability in a parallel environment.

**Moderator:** Thank you. Does anyone like to add something to their statement?

**C.L. Forgy:** Well, one thing that has surprised me at the panel was the degree of consensus rates here. I have really expected that I was going to be the only one argu-

ing for low levels of parallelism. So, I went to some pains to explain why we had to have that in our application.

But since it didn't hapen, let me argue the other side now. I think these kinds of arguments are made before we really apply to the sort of high level cognitive functions that we are going to try to put in computers. But there are plenty of other things that people do and we want to put in computers, which are inherently highly parallel. Perceptual tasks, speech recognition to some degree, and vision of course to much higher degree are inherently highly parallel. And we will have machines that use, you know, and enormous mass of parallelism to do those kinds of tasks.

**Moderator:** Before concluding this panel, let me give a brief explanation of my view.

We have to think about the performance of parallel processing. I would like to define it by $\eta_0$ and $\eta_1$. "$a$" is process execution time, and "$b$" is communication or synchronization time. This is a very simple calculation. Performance is:

$$\eta_0=\frac{a}{a+b}=\frac{1}{1+\rho} \quad \left(\frac{b}{a}=\rho\right)$$

This is something like the efficiency of each processor, and to get the total performance of the system, it should be multiplied by $n$. This comes to be:

$$P_0=\eta_0 \cdot n=\frac{n}{1+\rho}$$

And this happens when each processor has a fixed number of neighboring processors.

But if each processor would like to communicate with all the other processors, this becomes like this:

$$\eta_1=\frac{a}{a+b\cdot n}=\frac{1}{1+\rho\cdot n} \quad \left(\frac{b}{a}=\rho\right)$$

$$P_1=\eta_1 \cdot n=\frac{n}{1+\rho\cdot n}$$

And then if "$n$" is going up; then $\eta_0$ is also going up. But for this $\eta_1$ it's questionable, not linear. If we cannot specify a problem, which means a general purpose, then we must admit some drawbacks or need some balance. And it's not so easy to obtain a big parallelism of thousand, ten-thousand, million, or something like that, with this formula.

So what I would like to say is that we need revolutionary change in the architecture and language. Now, I would like to close this session. Thank you very much.