PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1988,
edited by ICOT.

THEORY AND PRACTICE OF CONCURRENT SYSTEMS –A POSITION PAPER

David H. D. Warren
Department of Computer Science
University of Bristol
Bristol BS8 1TR, U.K.

What do we mean by concurrent systems?

We have to be careful what we mean by a concurrent system. Computer systems are built in levels one on top of another, for example an application on top of a high-level language emulator on top of microcode on top of hardware. Concurrency at one level does not necessarily imply concurrency at another. For example, a pipelined processor has concurrency that manifests itself at the microcode level, but is irrelevant and invisible to the higher levels. Equally, there may be concurrency at a higher level but no concurrency at a lower level. For example, an operating system running on a sequential machine supports concurrent activities at a high level without there being any parallelism at the machine level. In this case, the concurrency at the high level is apparent rather than actual. There is no true parallelism in the sense that operations are in fact performed simultaneously leading to an increase in speed. In this paper, I shall reserve the word "parallelism" for concurrency in this more restricted sense, and use "concurrency" as the more general term. The two concepts are often confused.

I shall call a system a concurrent system only if there is concurrent activity apparent at the highest level. Thus an application should not be regarded as a concurrent system simply because it is running on a parallel computer.

Is concurrency irrelevant to the real problems we wish to solve with computers?

Some applications are intrinsically concurrent (e.g. an operating system or an airport simulation), and are best expressed in a concurrent programming language. However most problems that we want to solve on a computer are not intrinsically concurrent, and do not require a concurrent programming language. Thus concurrency is indeed irrelevant to most (but by no means all) real problems.

Of course, we would like our applications to run faster, and parallel computers are one very promising way to achieve this. However we should not confuse parallelism with concurrency and feel obliged to reprogram our application in a concurrent programming language. Parallelism is best exploited at a lower level, and concurrency should then only be of concern to the implementor of that lower level.

What are the fundamental differences between sequential and parallel systems? Should these differences be exposed or hidden?

The only fundamental difference between a sequential and a parallel system should be that the parallel system runs faster! Any other differences should be hidden. If I ask a builder to build me a house, it shouldn't concern me whether he uses one workman or many. Equally, if I want a computation performed, it shouldn't concern me whether the computer has one processor or many.

Do you envisage a transition in mainstream computing from sequential to parallel systems? Can you specify preconditions and milestones for such a transition?

I believe parallel computers will only gain widespread use when parallelism can be exploited invisibly to the normal programmer(or user). Computers are hard enough to use, and applications are difficult enough to program, without introducing a further dimension of complexity. Parallel computers will only supplant sequential ones when they can be treated as "black boxes" that happen to run faster.

This is very difficult to achieve with conventional programming languages. Conventional languages have a notion of time and change of state built into them, and depend on assignment as the basic operation. They can be classified into sequential languages (e.g. Fortran) and parallel languages (e.g. Occam). It is difficult for the language implementor to extract parallelism from a sequential language because the semantics of the language is so much bound up with a particular order of execution. This has led to the development of parallel languages. In these languages parallelism can be exploited, but only at the expense of making it very visible to the programmer.

To exploit parallelism invisibly, I believe the most promising approach is to switch our attention to declarative languages (e.g. Prolog and other logic programming languages). Declarative languages define a computation through a declarative description of the problem, plus some control information which serves to shape the computation of a solution. Declarative languages have

two big advantages. Because the language is declarative, it is easier to produce correct programs to solve complex problems. Because the language is not based on assignment, and doesn't force any particular execution order, it is easier to exploit parallelism.

Prolog is often viewed–wrongly in my opinion–as a sequential language, probably because the original implemntations were sequential, and because the language's operational meaning is generally explained in sequential terms. However, I would argue that the Prolog control information (goal ordering, clause ordering and cut) serves only to define the size and shape of the computation that is to be carried out, and leaves largely unspecified the order of operations. Thus in the Aurora system which we have implemented (and is described in this Proceedings), the computation tree is constructed in or-parallel fashion while supporting the full Prolog language. This idea can be extended to encompass and-parallelism as well as or-parallelism while preserving the same language semantics and abstract view of a computation. This we have called the Andorra model (partially described as part of a paper in this Proceedings by my colleague Seif Haridi, and implemented in prototype form by my colleague Rong Yang).

Thus declarative languages are in general neither sequential nor parallel, but should be viewed as neutral towards parallelism. Control information is also ideally largely neutral towards parallelism, although certain language features tend to force a sequential view (e.g. side effect predicates in Prolog), and certain language features tend to force a parallel view (e.g. read-only variable annotations in Concurrent Prolog).

Can you envisage the structure of future general purpose parallel computing systems?

I believe parallel computers of the future must be truly general purpose, and must allow multiple processors to treat all data as shared and uniformly accessible. This implies shared virtual memory but does not necessarily imply shared physical memory. Our proposal for a scalable multiprocessor with these properties, called the data diffusion machine, is described elsewhere in these Proceedings. The machine is completely general purpose in that it can potentially support any kind of application in any kind of language. However it was motivated by the desire to exploit parallelism transparently through declarative language systems such as Aurora and Andorra.

Is concurrency a nuisance inflicted upon us by hardware capabilities? Or is it a blessing that will lead us to better ways of thinking about problems?

Parallelism is in some sense a nuisance that we must endure if we want our applications to run faster. However, hopefully it is a nuisance that need only concern the implementors of the lower levels of a computer system.

Concurrency, as I have mentioned, is an essential feature of certain kinds of applications, and demands new kinds of programming language. It has led to the development of an important new family of declarative languages, the committed choice languages (Parlog, Concurrent Prolog, GHC). These languages are better able to express applications where concurrency is intrinsic. However, in other respects they are more restrictive than Prolog and not so widely applicable. The Andorra model gives Prolog much of the capability of committed choice languages, and it is my belief that the advantages of Prolog and committed choice languages can be combined in a single language, which I will call Andorra Prolog. Ideas in this direction are still emerging; Seif Haridi presents one approach in this proceedings.