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

THEORY AND PRACTICE OF CONCURRENT SYSTEMS
—THE ROLE OF KERNEL LANGUAGE IN THE FGCS PROJECT—

Kazunori Ueda

Institute for New Generation Computer Technology
4-28, Mita 1-chome, Minato-ku, Tokyo 108, Japan

1 INTRODUCTION

An outstanding feature of the Fifth Generation Computer Project is the idea of designing a novel kernel language that links parallel hardware and application software. KL1 (Chikayama et al. 1988), the kernel language for the Multi-PSI (Taki 1988) and the Parallel Inference Machine (PIM) (Goto et al. 1988), is based on the inherently parallel language GHC (Ueda 1988). This means that we chose to expose parallelism to software people and involve them in forming the culture of parallelism, rather than to hide parallelism from them. This paper will describe why we took this approach and will answer the questions of Ehud Shapiro in the light of our methodology.

2 LANGUAGE ISSUES IN CONCURRENT SYSTEMS

The key to the success of concurrent systems lies in how to construct and accumulate parallel software.

It is often claimed that parallel programming is difficult, but the fact is that we have never made as much effort toward creating parallel software as toward creating sequential software. We are too much accustomed to sequential programming of von Neumann computers to change our programming style. It is very important to overcome these non-technical problems and concentrate more research on parallel programming by steadily finding solutions or clues to individual technical problems.

Technical problems include language issues, with which I have been involved for years. There are two candidates for an easy-to-use parallel language: augmenting a sequential language with simple primitives (like Occam) and designing an inherently parallel language. The former might enable smoother transition from sequentiality to parallelism, but our project chose the latter approach for the following reasons:

(1) The existence of sequencing tends to make control overspecific. We wanted to distinguish between the sequentiality essential for the correctness of the algorithm and the other kinds of sequentiality.

(2) We wanted the kernel language to express any potential parallelism of a program independently of the granularity of the hardware we would design.

(3) Parallel programming will require the change of our way of programming and thinking from the von Neumann style. An inherently parallel language will better encourage it.

Another alternative might be to raise the level of the kernel language to where programmers are not bothered by control. However, we do require a parallel language with explicit control when implementing such a high-level declarative language and, more importantly, when describing the communication between a program and the outside world.

It is the attention to communication that characterizes concurrent systems both in theory and in practice. In theory, communication gives the most abstract view of a whole program and its fragments, concurrent processes. In practice, communication is the primary source of bottleneck.

The reason why control is necessary for specifying communication is that communication is a directed, irreversible activity. A language without explicit control is usually considered to be at a higher level than a language with explicit control, but the presence or absence of control is more a matter of formalism than a matter of the level of abstraction. A language without control can be used only in the fragments of a program in which communication is not made or need not be specified.

We chose to expose parallelism to software people by adopting an abstract kernel language with explicit control. It provides software people with an appropriately abstract model of parallel computation, and yet it is amenable to reasonably efficient parallel implementation. Our choice does not necessarily mean that all applications programmers must care about control issues; we could hide parallelism by implementing higher-level languages (like constraint programming languages) on top of the kernel language. The point is that *applications programmers should have explicit access to par-*

allelism if they want. The development of concurrent systems should be supported by many people at various layers from hardware to applications. Our choice allows enterprising applications programmers to consider good use of parallelism for their applications, which can be spread in the form of a programming paradigm or an embedded language whose object codes embody that paradigm.

3 FUTURE RESEARCH

Much research remains to be done on concurrent systems. Making a good parallel implementation of the kernel language will not be sufficient to motivate applications people to write parallel programs. We must show them parallel programming methodologies. We have found that although it is not very difficult to write parallel programs, it is difficult to write *good* parallel programs. We must take two more things into account: the locality of communication and load balancing.

In sequential programming, we rely so much on the flat storage structure. Large and flat memory space has made programming easy by not letting programmers think much about locality. To make full use of a parallel computer with the processing power distributed over the storage, however, we must consider storage and processing at the same time and keep the locality of communication. The notion of constant-time access is by no means scalable.

Parallel programming requires theoretical support, too. We do not yet have a practical computational model with which to argue the real efficiency of parallel algorithms running on, say, the Multi-PSI. Previous theories of parallel computation were concerned mainly with whether parallelism improves time complexity. However, the computers we are building are intended to improve *time* and not time *complexity*.

Some applications programs may have irregular structures that are too difficult to analyze statically. Such programs require a mechanism for keeping the load balance and the locality automatically. In general, a future concurrent system will be supported by a lot of techniques whose basic ideas may be discovered on the analogy of what we do in the real world as members of some community. The actual implementation of those techniques will necessitate statistical analysis.

We must also continue language and implementation research to create a more expressive and more efficient language. As for expressiveness, we must consider how to introduce meta-level operations gracefully. By meta-level operations I mean the operations that refer to and/or modify the "current" status of computation (including physical configurations and time). GHC deliberately excluded meta-level operations to reveal the essence of concurrent logic programming. It is expressive enough for ordinary programs, but is too weak for an operating system like PIMOS (Chikayama et al. 1988). Accordingly, KL1 has featured necessary meta-level operations to describe PIMOS, but we have yet to clarify their semantics by developing an appropriate model of the parallel computers running KL1 programs. Research on reflection in parallel computation (Tanaka 1988) will be helpful in the design of meta-level features.

As for efficiency, we have two directions of research: the simplification of the kernel language (without loss of expressiveness) and the development of high-level optimization techniques. The purpose of the simplification is to make processes and streams more efficient by tuning KL1 for programming with many small communicating processes. It is my consistent view that GHC is a base language from which an appropriate subset should be made. A good subset will be found through the research on sophisticated optimization that employs techniques such as abstract interpretation.

An efficient implementation of processes and streams will better support user languages such as $\mathcal{A}'\mathcal{UM}$ (Yoshida and Chikayama 1988). It will also enable us to use processes as building blocks of a database that allows concurrent access. However, such storage-intensive use of processes requires a new kind of optimization. While most of the current implementations of KL1 are tuned for computation-intensive programs that do not suspend so often, now we need optimization techniques for processes that are almost always dormant.

REFERENCES

Chikayama, T. et al. (1988) Overview of the Parallel Inference Machine Operating System (PIMOS), in this volume.

Goto, A. et al. (1988) Overview of the Parallel Inference Machine Architecture (PIM), in this volume.

Taki, K. (1988) The Parallel Software Research and Development Tool: Multi-PSI System, in *Programming of Future Generation Computers*, Fuchi, K. and Nivat, M. (eds.), North-Holland, 1988, pp. 411–426.

Tanaka, J. (1988) Meta-Interpreters and Reflective Operations in GHC, in this volume.

Ueda, K. (1988) Guarded Horn Clauses: A Parallel Logic Programming Language with the Concept of a Guard, in *Programming of Future Generation Computers*, Fuchi, K. and Nivat, M. (eds.), North-Holland, 1988, pp. 441–456.

Yoshida, K. and Chikayama, T. (1988) $\mathcal{A}'\mathcal{UM}$ — A Stream-Based Concurrent Object-Oriented Language, in this volume.