PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Operating System PIMOS and Kernel Language KL1

Takashi Chikayama
Institute for New Generation Computer Technology
4-28, Mita 1-chome, Minato-ku, Tokyo 108 Japan
chikayama@icot.or.jp

Abstract

The Fifth Generation Computer Systems (FGCS) project is a national project of Japan, aiming at establishing the basic technology required for high performance knowledge information processing systems. The parallel inference system subproject is aiming at establishing parallel processing hardware technology for massive processing power and software technology for effective utilization of such hardware in the knowledge information processing field. The basic software system is responsible for providing a programming language suited for describing knowledge information processing applications software and providing a comfortable environment for program execution and software development on highly parallel computer systems.

A concurrent logic language with extensions to control program execution on parallel hardware was designed as the *kernel* language of the system. An operating system that provides a comfortable environment for parallel application software development was designed and implemented in the kernel language. This paper gives an overview of the research and development in this area in the FGCS project.

1 Introduction

The fifth generation computer systems project is a national project of Japan, aiming at establishing the basic technology required for high-performance knowledge information processing systems. The most important technologies to be provided to attain the final objective of the project are the following two.

- Problem solving methods for knowledge information processing
- Processing power for implementation of the above methods

The parallel inference system subproject is aiming at establishing both hardware and software technologies for the latter.

With the recent evolution of the hardware technology, multiprocessor systems are expected to be advantageous not only in absolute processing power but also in cost effectiveness early in the next century. There seems to be no other technology than multiprocessing to provide the computational power required for high-performance knowledge information processing systems.

The software technology for parallel processing, on the other hand, is still quite premature. In particular, the technology for building parallel software to solve complicated problems in the area of knowledge processing is far from satisfactory yet. This, we think, is at least partly due to the problems in the approach to the parallel software technology conventionally taken, that is, trying to augment already available sequential processing technologies. A new system of software technology totally redesigned for parallel processing, including algorithms, programming languages and operating systems, has to be established.

As the basis of this new technology, a concurrent logic language with extensions to control program execution on parallel hardware was designed as the *kernel* language of the system. An operating system that provides a comfortable environment for parallel application software development was also designed and implemented in the kernel language. This paper gives an overview of the research and development in this area of the FGCS project.

In the following sections, the design principles are described in section 2, the design of the kernel language in section 3, that of the operating system in section4. Experiences with the language and the operating system are described in section 5. Direction of future work is suggested in section 6, followed by concluding remarks.

2 Principles

2.1 Middle-Out Approach

When designing a computer system, two extreme approaches can be considered. One is a top-down approach, starting from problems to solve, gradually designing downwards to the level of computer architecture or even to the level of electronic devices, seeking in each level for a design most appropriate to implement higher levels. The other is a bottom-up approach, starting from

available device technologies, seeking for the best use of the lower level technology, finally finding an appropriate application area.

Neither of the approaches, however, cannot be successful by itself. In the top-down approach, design in each level requires insight into appropriate implementation of all the lower level technologies. In the bottom-up approach, design in each level requires insight into upper levels, up to application areas appropriate for the chosen design.

It is too difficult for anybody to have such insight for the broad and rather vague target of a long-term project, knowledge information processing. We thus decided to take a *middle-out* approach of designing a certain intermediate level first and conduct research and development towards two directions, upwards and downwards, simultaneously. It is not easy, of course, to find an appropriate intermediate level and to actually design that level. This, however, seemed to be the only feasible approach for a project like this one.

2.2 Kernel Language

The intermediate level we chose was the level of programming languages. Choosing this level has the following merits.

- The programming language level is not too far away from the both extreme ends of application software and hardware implementation.
- Relatively rigorous specification in the programming language level can be given more easily than in other levels.

The programming language designed to be the starting point of this middle-out approach is called the *kernel language* [Ueda and Chikayama 1990].

At the time the project started in 1982, language design and implementation technology was still premature to fix the design of the kernel language. Thus, the research started by investigating sequential systems first. In the first stage (fiscal years of 1982–84) of the project, a sequential kernel language based on Prolog, named ESP [Chikayama 1984], was designed, which formed the basis of the research and development in most of the research efforts in the first stage and early in the intermediate stage.

Design of the next version of the kernel language KL1 was started in the first stage simultaneously. Its preliminary design and implementation were done early in the intermediate stage and a fuller implementation on a experimental parallel computer system was completed within the intermediate stage (1985–88). The language has been used through the final stage (1989–) for various application research. In what follows, the kernel language means this second generation kernel language, KL1.

2.3 Logic Programming Principle

The logic programming idea gave the basis of the whole project. The image of logic programming in the original project plan seems to have been strongly influenced by a particular language Prolog. As the research proceeded from sequential systems to parallel systems, we had chosen a concurrent logic programming approach. The principle of placing "logic" as the central design principle, however, has been kept unchanged.

The principle of logic programming played a important role in selecting a particular design among many candidates. In designing the kernel language, its *soundness* in the sense of mathematical logic has been acted as a "canon", although we gave up pursuing *completeness*.[1] Many proposals to extend the kernel language with attractive features were investigated but rejected because of their *unsoundness*. On the other hand, features which do not change the meaning of the programs when interpreted as logical formulas were more freely added to the language. They have only to do with execution efficiency and nothing to do with the correctness of programs, and were clearly discriminated from the core part of the language.

These principles based on logical interpretation of programs have been quite helpful in keeping the language design coherent and, in its consequence, its implementation and its programming style coherent, as is described further in detail below.

2.4 Target Architecture

A processor with performance comparable to a full-size computer with reasonable amount of memory is now available on a single circuit board. Recent evolution of the hardware technology shows four-times increase in density of circuitry every three years. Extrapolating this, one hundred processors with reasonable amount of memory are expected to reside in one chip early in the next century. On the other hand, although the performance of single processor is steadily being improved, it might be very difficult to attain improvement by two orders of magnitude within the same time period.

With larger circuitry made practical with higher density, the design cost is beginning to dominate the total cost of processors. The design repeatability in multiprocessor systems will have great cost advantage over a complicated processor occupying one whole chip or more, even if the both systems had the same performance. Early in the next century, multiprocessor systems will thus be advantageous, not only in absolute processing power, but also in cost effectiveness even in small systems such as palm-top or wrist watch type computers.

[1] *Soundness* of a system means that any results obtained are logical consequences of the given axiom set. *Completeness*, on the other hand, means that all logical consequences can be obtained.

For application areas such as knowledge information processing that need non-uniform computation, an architecture that allows flexible resource allocation is required. For highly parallel systems, scalability of the system architecture is critical. Having these in mind, we chose a homogeneous MIMD architecture with loosely-coupled processors (or loosely-coupled *clusters* each with several tightly coupled processors) as the target architecture of the software system.

2.5 Level of the Kernel Language

An ideal programming language should allow very high level description with an implementation optimizing it to the target architecture without any human help. However, with the current technology, such a language is nothing more than a dream. It is especially so when the programs have to be optimized for execution on a large-scale loosely-coupled parallel computer systems where communication delay is not negligible. The most difficult part in the optimization will be *where* (on which processor) to execute certain parts of computation and *when* (in which order). Such a problem is known as the *mapping* problem.

As long as problem solving techniques used are relatively simple, required computation can be easily told beforehand making static mapping by compilers feasible. For knowledge information processing requiring sophisticated problem solving methods, what to compute next often depends on the result of the former steps of the computation, making static optimization of computation mapping impossible. Many research results have shown that general-purpose automatic mapping algorithm is hard to design and the selection of good mapping algorithms depends heavily on the problem solving method used.

As knowledge information processing is an area where no single universal and efficient problem solving method is known, providing one single mapping algorithm is not appropriate. Providing many mapping algorithms that cover all the known methods may still be insufficient; as research in the area is still in an early stage, many novel problem solving methods are expected to be proposed in the near future. Thus, we set the level of the kernel language so that mapping of computation can be specified in programs.

This decision of putting the responsibility of computation mapping on programmers has the drawback of making programming a more complicated task. We, however, regard this additional effort as unavoidable and essential in establishing the technology for high performance knowledge information systems. When a widely applicable mapping algorithm is established, it can be provided to the application users as a program library. With the kernel language capable of controlling program execution, writing such a library should not be difficult.

2.6 Designing a New Language

It might have been possible to take an already existing logic programming language as the basis of the kernel language and extend it with several additional features for concurrent execution. The logic programming language used most widely was (and still is) Prolog, which was the primary candidate for such extensions.

There could be two ways to tailor Prolog to a language for parallel systems. One method was to provide implicit and automatic computation mapping, which was not taken by the above-described reason. Another possible way was to make concurrent execution explicitly specified with additional language constructs. However, as the base language Prolog was designed for sequential processing, concurrency specification would add some more complexity to the language and making programs harder to understand. More importantly, if sequential execution should have made the default principle, it would have been more difficult to reorganize programs for better mapping, as different mappings require different parts of programs to run concurrently.

Another problem with such a language was pains in specifying synchronization. In programming languages in which synchronization is specified independent from conditioning, problems arise when decisions on conditional execution are made on incomplete data. On physically parallel hardware, finding such problems would become very painful because the same phenomenon is often hard to reproduce. To solve this problem, synchronization and conditioning should not be made separate.

We decided that the kernel language should be designed from scratch so that concurrent execution could be expressed in a natural way. The language should have intrinsic concurrency: language constructs imply concurrent execution in principle and sequencing is explicitly described. Synchronization should be integrated with conditioning in the language construct.

2.7 Designing a New OperatingSystem

Even though the prototype parallel inference system is an experimental system, an operating system that provides a comfortable software development environment was mandatory. One way to provide the required functionality might have been to port an already existing operating system to the parallel inference machine.

All the operating systems available then (and probably most of them even now) were designed originally for sequential systems and augmented afterwards with certain primitives for execution on parallel systems.

There were two major problems with such systems. One was that the interface of the operating system with the user programs was still based on sequencing. For example, the user program is notified of completion of requested service by the completion of execution of a pro-

cedure, supervisor call, in the user's thread of execution. This is acceptable in systems where application software is written in basically sequential languages. This, however, would not go well with software written in the kernel language with intrinsic concurrency.

Another problem was that the management policies of such operating systems were highly optimized for sequential processing. In sequential systems or small-scale parallel systems, centralization of all the management information is usually the most robust and efficient policy. This, however, is far from optimal for highly parallel systems. If the management were centralized on one processor in a highly parallel system, that processor would be responsible for too much management work and would be the bottleneck of the whole system. Moreover, every activity within the system would require communication to and from that processor, resulting in communication bottleneck.

We concluded that designing an operating system optimized for highly parallel systems was also an unavoidable and essential part of the technology for high performance knowledge information systems and decided to design and implement a new operating system from scratch. The user interface should be consistent with the design of the kernel language; sequencing should not be a part of the design of the interface. Distribution of management was essential to avoid bottlenecks, which might also affect the specification of the services provided by the operating system.

3 Kernel Language: KL1[2]

The kernel language KL1 has two layers. The basic layer is defined by Guarded Horn Clauses (GHC), which is a concurrent logic language for describing *what* computation to perform for desired result, that is, for describing *correct* programs. The description lays only those constraints on mapping of computation which are required to obtain the desired result. Based upon this layer is the full KL1 language for describing *how* such computation should actually be carried out with desired mapping of computation, that is, for describing *efficient* programs. This separation of correctness and efficiency issues or, in other words, concurrency and parallelism, seems to play an important role in bridging the gap between parallel inference systems and knowledge information processing in a coherent manner.

3.1 Concurrent Logic Language GHC

This section describes the design of a concurrent logic language Guarded Horn Clauses, which forms the basis of the kernel language KL1.

[2]This section is a rewrite of an article co-authored with Kazunori Ueda [Ueda and Chikayama 1990], except for the subsection 3.3.

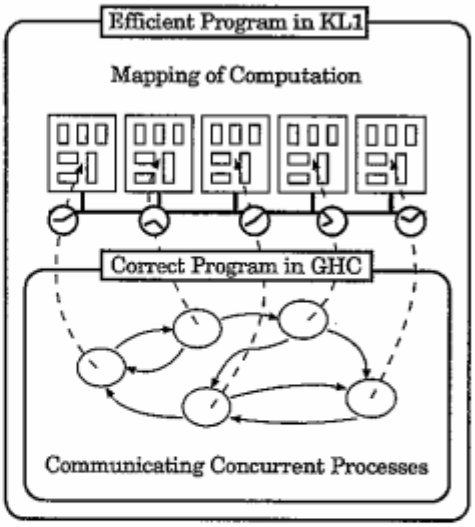

Figure 1: Two Layers of the Kernel Language

3.1.1 Concurrent Logic Languages

The design effort of the kernel language was started in 1982 with the start of the project by seeking for an appropriate framework of the language. As the concurrent logic programming framework seemed to provide the characteristics in our need, we investigated many languages in the family as the basis of the kernel language, including Relational Language [Clark and Gregory 1981], Concurrent Prolog [Shapiro 1983] and PARLOG [Clark and Gregory 1983]. This study led us to a design of a new concurrent logic language, Guarded Horn Clauses (GHC) at the end of 1984 [Ueda 1986].

GHC shares its basic framework with other concurrent logic languages. Firstly, a GHC program is a set of *guarded* clauses. Secondly, GHC features no don't-know nondeterminism (built-in search capability) but features don't-care nondeterminism, which allows description of reactive systems. Reactive systems in concurrent logic languages are based on the process interpretation of logic [van Emden and de Lucena Filho 1982], in which a goal (or a multiset of subgoals derived from it) is regarded as a process and processes communicate by generating and observing bindings (between shared logical variables and their values). Like most concurrent logic languages, all bindings are *determinate* in GHC, that is, they are never revoked once published to other processes. The determinacy of bindings is essential in reactive systems, such as an operating system, because the bindings may be used for interacting with the real outside world. The lack of built-in search capability also allows programs to specify the way of their execution in more detail, which

also matches our principle of making programs specify mapping of computation.

3.1.2 Guarded Horn Clauses

What then is the relative merit of GHC over other concurrent logic languages? In our study of various concurrent logic languages, we focused on Concurrent Prolog, which was the most expressive of them, and built its prototype implementation [Miyazaki *et al.* 1985]. The experience led us to clarify the definition of atomic operations of the language, which in turn led us to a new language with simpler atomic operations.

As explained above, one important aspect of concurrent logic languages is the determinacy of bindings. In general, the execution of a concurrent logic program proceeds using parallel input resolution [Ueda 1988a] that allows parallel execution of different goals, but under the following rules to guarantee the determinacy of bindings:

(1) The guards (including the heads) of different clauses called by a goal g can be executed concurrently, but they cannot instantiate g.

(2) The goal g *commits* to one of the clauses whose guards have succeeded.

(3) The body of a clause to which g has committed can instantiate g. The bodies of clauses to which g has *not* committed cannot instantiate g or the guards of the clauses.

(4) A goal is said to *succeed* if it commits to some clause and all its body goals succeed.

That is, before commitment, a goal can pursue two or more clauses but without generating bindings. After commitment, it can generate bindings but only one clause is left.

Another important aspect of concurrent logic languages is how synchronization is achieved. In general, synchronization is achieved by restricting information flow caused by unification. Concurrent Prolog uses read-only annotations, and PARLOG uses mode declarations which are used for compiling the unification of input arguments into a sequence of one-way unification and test unification primitives. However, in these languages, additional mechanisms are necessary to guarantee restriction (1) above.

The key idea of GHC is quite simple. It uses the restriction (1) itself as a synchronization construct. That is, any piece of unification which is invoked directly or indirectly from the guard of a clause C and which would instantiate the caller of C is suspended until it can be executed without instantiating the caller. In other words, GHC has integrated two notions: the determinacy of bindings and synchronization.

A kernel language must provide a common framework for people working on various aspects of the project including applications, implementation, and theory. Before accepting GHC as the basis of our kernel language, we had to convince ourselves that it satisfies the following conditions:

- It is expressive enough.
- It can eventually be implemented efficiently, possibly by appropriate subsetting.
- It is simple enough to be understood and used by programmers. Also, it is simple enough for theoretical treatment.

We soon made sure that GHC was expressive enough to write most concurrent algorithms that had been written in other concurrent logic languages, but that was not enough. How to program search problems was also important, because search problems are a specialty of ordinary logic languages. So we have developed a couple of methods for programming search problems [Ueda 1987], [Tamaki 1987], [Okumura and Matsumoto 1987].

For implementability, we quickly ascertained by rapid prototyping that GHC can be implemented fairly efficiently at least on sequential computers [Ueda and Chikayama 1985].

3.1.3 Flat GHC

For simplicity, we continued to study the properties of GHC and looked for a simpler explanation of the language better suited to process interpretation. Now, our interpretation is that a GHC process is an abstract entity which observes and generates information (represented in the form of bindings) and which is implemented by a multiset of body goals. The behavior of each body goal is defined by guarded clauses that can be regarded as rewrite rules.

A problem with the original definition of GHC is that guard goals do not fit well into this process interpretation. We also felt, from a practical point of view, that the expressive power of guard goals did not justify the implementation effort even if it could be implemented efficiently.

These considerations led us to reduce GHC to a subset, Flat GHC. Guard goals of Flat GHC are auxiliary conditions to be satisfied for applying the clause. The sufficient conditions to be satisfied by a guard goal as an auxiliary condition are that it is deterministic (that is, whether it succeeds or not depends only on its arguments) and that it does not produce any bindings. This restriction simplified the theoretical treatment considerably in the operational semantics [Ueda 1990] and program transformation rules [Ueda and Furukawa 1988].

To summarize, a Flat GHC program is a set of guarded clauses that can be regarded as rewrite rules of goals.

The guard of a clause specifies what information should be observed before applying the rewrite rule, and the body specifies the multiset of goals replacing the original. A body goal is either a unification goal of the form $t_1 = t_2$, whose behavior is language-defined, or a non-unification goal, whose behavior is user-defined. A unification body goal generates information by unifying t_1 and t_2, and a non-unification body goal represents the rest of the work and will be reduced further.

3.1.4 Characteristics of GHC

The semantics of Flat GHC can be understood both algebraically and logically. The algebraic one is the process interpretation mentioned above. A logical characterization of communication and synchronization was given by Maher [Maher 1987], showing that information communicated by processes can be viewed as equality constraints over terms.

Unlike Concurrent Prolog but like PARLOG, the publication of bindings is not done atomically *upon* commitment of a non-unification goal but eventually *after* commitment using a unification body goal that can run in parallel with other goals. This means that commitment in GHC is a smaller and simpler operation than in Concurrent Prolog. Moreover, in GHC, the information generated by a unification body goal is not an atomic entity but can be transmitted in smaller pieces, possibly with communication delay. We have found that this liberal computational model of (Flat) GHC is expressive enough to program cooperating concurrent processes and leaves more freedom to implementation.

Another point to note is that GHC has included control for the *correct* behavior of processes but excluded any control for *efficient* execution. GHC has left the latter to KL1 described below, in order to clearly distinguish between the two notions. This contrasts with PARLOG, which features sequential AND that can be used for suppressing parallel execution of body goals. We believe that it is important to learn that synchronization based on information flow is sufficient for writing correct concurrent programs.

Important topics on theoretical aspects of Flat GHC include the relationship with other theoretical models of concurrency such as CCS [Milner 1989] and theoretical CSP [Hoare 1985]. Although concurrent logic languages differ from CCS and CSP in their asynchronous communication and dynamically reconfigurable processes, similar mathematical techniques can be used to formalize them. We have not yet obtained a completely satisfactory formal semantics, but we are fairly confident that Flat GHC is theoretically simple enough, while it can be used for practical programming without any modification.

3.2 Practical Parallel Language KL1

As described above, we have designed a concurrent logic language Flat GHC as the basis of the kernel language. The descriptive power of the language, however, is not sufficient when efficient program execution is our concern, which was the original motivation of parallel computers.

As Flat GHC programs do not say anything about where (i.e., on which processor) the atomic operations making up a computation should be performed, there are many ways to distribute the operations over available processors. As Flat GHC programs only specify the partial ordering of atomic operations, there are many possible total orderings conforming to it. To make sure that the distribution and the ordering employed are not far from optimal, we must be able to specify physical details of execution to some extent.

We thus designed a parallel programming language based on the concurrent programming language Flat GHC, in which we can specify in certain detail *how* a program should be executed. This section describes the outline of this language, named KL1.

3.2.1 Mapping of Computation

Flat GHC programs implicitly express any potential parallelism in the sense that no ordering between atomic operations exists except for those essential for correctness. On real-world computer systems with a limited number of processors and non-negligible cost of interprocessor communication, faithful exploitation of this parallelism will almost never show optimal efficiency. To achieve reasonable efficiency, control is required on when and where each atomic operation should be performed. This control is called *mapping*.

Mapping is often implicit in sequential systems. With two possible methods to solve a problem, a good strategy on a sequential system would be trying more efficient but less reliable one first and trying less efficient but reliable one second only when the first one fails. This may not be the best for parallel systems, when the first method will not require all the computational resource (such as processors) for its execution. In such a case, the second method should be tried in parallel with the first. This computation may or may not be required depending on the result of the first method. Such computation is called *speculative* [Burton 1985]. For efficiency, computation by the second method should not interfere the execution of the first by snatching required resources. This is effected by giving priority to the first method over the second. From this viewpoint, the original sequential algorithm uses sequencing of two methods not for correctness but for efficiency to implicitly specify priority.

Sometimes more sophisticated mapping is desirable. Suppose that there are two methods to solve a problem and that, although at least one is known to find a so-

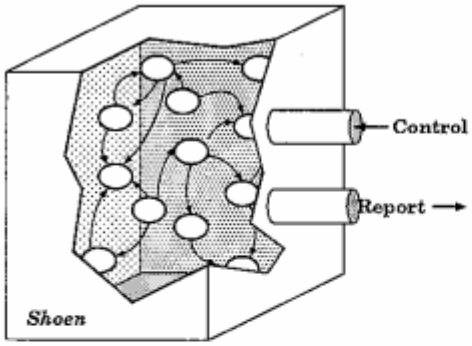

Figure 2: *Shoen* Construct

lution efficiently, we cannot tell which beforehand. In such a case, the best scheduling strategy may be to give both methods approximately the same amount of computational resource. Resource management is thus an important part of an algorithm in parallel computation.

In sequential computer systems and in parallel computer systems as extensions of conventional sequential systems, operating systems are primarily responsible for mapping. This is acceptable as far as application programs are mostly sequential and the mapping strategy is often specified by sequencing implicitly. In parallel systems where explicit mapping operations are much more frequently required, requesting each mapping operation to the operating system would incur intolerable overhead.

3.2.2 Mapping Features of KL1

To solve this problem, we have introduced into KL1 the following features, which are intended to be efficiently implemented:

Shoen: *Shoen*[3] represents a group of goals. This group is used as the unit of execution control, namely the initiation, the interruption, the resumption and the abortion of execution. Exception handling and resource consumption control mechanism are also provided through this *shoen* construct. It has two communication streams as its interface: one directs from outside of the *shoen*, called *control stream*, for sending messages to control execution in the *shoen*; the other, called *report stream*, has the reverse direction for reporting events internal to *shoen*. The *shoen* construct is an extension of the *metacall* construct proposed by Clark and Gregory [Clark and Gregory 1984].

Priority: A (body) goal of a KL1 program is the unit of priority control. Each goal has an integer priority associated with it. Each *shoen* keeps the maximum and the minimum priorities allowed for goals belonging to it, and the priority of each goal is specified relative to these. The language provides a large number of logical priority levels, which are translated to physically available priority levels provided by each implementation.

Processor specification: Each (body) goal may have a processor specification, which designates the processor (or a group of processors) on which to execute the goal.

This straightforward mechanism provides the basis of research in more sophisticated computation mapping strategies. Actually, several automatic mapping strategies have been developed for diverse problems, and relatively universal ones are provided as libraries [Furuichi *et al.* 1990].

One of the most notable characteristics of the KL1 language is that these priority and processor specifications are separated from concurrency control. We call these specifications *pragmas*. Pragmas are merely guidelines for language implementations and may not be precisely obeyed. The same is true of the controlling mechanism of *shoen;* abortion of computation, for example, may not happen immediately. This relaxation makes distributed implementation much easier.

In many parallel programming languages, the specification of parallel execution is often mixed up with other language constructs, especially with constructs for concurrency control. A major revision is often required for revising only the mapping of computation to improve efficiency, which is liable to introduce new bugs.

Although pragmas are specified within the program in KL1, they are clearly distinguished syntactically from other language constructs. Pragmas will never change the correctness of the programs,[4] though the performance may change drastically. As it is not uncommon that more than half of the effort to develop a program is devoted to the design of appropriate mapping, it is most advantageous that mapping specifications can be altered without affecting correctness of the program.

3.2.3 Keeping up with Sequential Languages

What criterion is appropriate for comparing parallel algorithms? Assume that a parallel algorithm has sequential execution time $c(n)$ (n being the size of the problem) and average potential parallelism $p(n)$. Then the total execution time by this algorithm on an ideal parallel computer is given by $c(n)/p(n)$. This means that an algorithm with more sequential execution time but with still more parallelism is considered to be a better algorithm on an ideal parallel computer.

[3] *Shoen* is a Japanese word corresponding to 'manor' in English.

[4] To be precise, the priority specification may be used for guaranteeing certain properties of diverging (i.e., autonomously nonterminating) programs.

This, however, does not hold when the potential parallelism, which may vary over time, can exceed the physically available parallelism. As physical parallelism is always limited in the real world, a parallel algorithm with sequential time complexity worse than a sequential algorithm will be beaten by that sequential algorithm for sufficiently large n, *no matter what* $p(n)$ *is*. To summarize, parallel languages must be able to express any algorithms with the same sequential time complexity as in sequential languages to be really useful.

Pure languages such as pure Lisp and pure Prolog cannot express certain kinds of efficient algorithm due to the lack of the notion of destructive assignment. GHC also is a pure language with the same inherent problem. To write efficient algorithms in these pure languages, we must be able to somehow mimic the efficiency of array operations in conventional languages.

For this reason, KL1 introduced a primitive for updating an array element in constant time without disturbing the single-assignment property of logical variables. The primitive can be used as follows:

```
set_vector_element(Vect, Index,
                   Elem, NewElem, NewVect)
```

When an array `Vect`, an index value `Index` and a new element value `NewElem` are given, the predicate binds `Elem` to the value of the `Index`'th element of `Vect`, and `NewVect` to a new array which is the same as `Vect` except that the `Index`'th element is replaced by `NewElem`.

Because some other goals may still have references to the old array `Vect`, a naive implementation might allocate a completely new array for `NewVect` and copy all but one elements. However, when it is known that no goals other than the above `set_vector_element` goal have references to `Vect`, there will be no problem in destructively updating it. In the actual implementation of KL1, a simplified, efficient version of the reference counting scheme [Chikayama and Kimura 1987] detects such a situation, in which event the new array `NewVect` is obtained in constant time.

This means that any imperative sequential algorithm can be rewritten in KL1 retaining the same computational complexity, as random access memory can always be emulated using a single-reference array. Of course, allowing only one reference to a data structure can decrease the possibility of parallel execution considerably. However, this requirement of the computational complexity becomes essential only after physically available parallelism is used up.

3.3 Higher-Level Languages

Although the kernel language KL1 allows relatively higher level description of programs than imperative languages, its description level is in the same level as Lisp, which is still too low for certain application programs in the area of knowledge information processing. This section describes research on providing higher-level language constructs upon KL1.

3.3.1 Macro Expansion

A powerful macro expansion mechanism similar to the one available in ESP [Kondoh and Chikayama 1988] is designed and implemented. This macro allows not only in-place expansions of macro invocations but also insertion of terms into the program in the levels of arguments, goals or clauses. The following are possible using these features.

- Simple in-place expansion
- Conditional compilation
- Functional notations including but not restricted to arithmetical expressions
- Implicit arguments

A goal of Flat GHC programs has very short lifetime, as it consists of only one reduction to its subgoals. To realize a process with longer lifetime, a programming style is used in which a goal recursively calls the same predicate with almost the same arguments. This programming style is used almost everywhere in the operating system and application programs. In such a programming style, the state of the process or any paths to communicate with other processes (shared variables) have to be passed as the arguments of the recursive goal. This ensures higher modularity, but always describing such arguments is too verbose, making it harder to understand or to revise programs. The implicit argument passing mechanism can be conveniently used to describe processes in a more concise manner.

The macro expansion mechanism of KL1 is so powerful that functions beyond mere syntactic sugaring can be provided using its features. However, programmers can freely choose any programming style allowed in KL1. Although this is advantageous in certain cases, restriction on the usage of the language features is profitable in making programs easier to understand and maintain. We thus started designs of higher-level languages to be compiled into KL1, which will be described in the following sections.

3.3.2 A'UM

The programming style of KL1 most frequently used is to describe a set of processes communicating through message streams [Shapiro and Takeuchi 1983]. Streams are realized by gradually instantiating a list structure consisting of binary cells. Processes are realized using tail recursion. A'UM is a programming language designed to describe such programs more directly than explicitly

writing such realization of message streams and processes [Yoshida and Chikayama 1990].

A prototype implementation of the language was a translator to KL1. As a thoroughly object-oriented language, every entity of the language A'UM, an integer value for example, appears as a process. We could find no other way than to actually implement them as processes in KL1. The choice then was whether to abandon thorough object-orientationor to implement it differently, not as a part of the parallel inference system. A'UM took the latter choice and research on its more direct implementation is ongoing [Konishi *et al.* 1992]. A prototype implementation is already operational on a system of network-connected workstations. The former approach was taken by another language with similar objectives, called AYA, which is described in the next section.

3.3.3 AYA

The design of the language AYA was initiated after we decided to let A'UM seek for pure object-orientation rather than pursue practical efficiency on the parallel inference system [Susaki and Chikayama 1991].

The design objective of AYA is the same as the initial motivation to design A'UM, namely, providing a more concise way to describe programs in object-oriented programming style of KL1. In design of AYA, a higher priority is given to practical efficiency and freedom of description than uniformity as an object-oriented languages. Not all entities are "objects": integers will not respond to "add" messages. Its design was mostly bottom-up; most of the language features were chosen based on our programming experiences in KL1.

Processes of AYA can have multiple streams to receive messages, making it impossible to interpret one single message stream to be representing an object. Communication patterns besides streams such as asynchronous interrupts are also allwoed.

A characteristic feature of AYA is the notion of *scenes*, corresponding to the macroscopic context of a process. A process can have many scenes to act in and its reaction to messages from outside will depend on in which scene it is currently acting.

Implementation effort of AYA is ongoing and a prototype translator to KL1 is already operational.

4 Operating System: PIMOS

As described above, an operating system tuned to control highly parallel programs effectively is vital for fully exploiting the power of highly parallel computer systems. The system should also be user-friendly and robust enough for practical and extensive use in parallel software research. The Parallel Inference Machine Operating System (PIMOS) was designed to fulfill the requirements and implemented in the kernel language. This section describes the overall design of PIMOS.

4.1 Prior Works

The possibility and advantages of writing a complete operating system in a concurrent logic language were suggested by Shapiro [Shapiro 1986]. Based on this principle but with much improvements in various aspects, several experimental systems such as the Logix system [Hircsh *et al.* 1987] and the Parlog Programming System (PPS) [Foster 1987] were implemented.

PIMOS resembles PPS in many aspects. This resemblance is partly due to the resemblance of the implementation languages (KL1 and PARLOG) and partly due to frequent exchange of ideas among the two groups.

A notable difference between PIMOS and the other above-mentioned systems lies in the underlying language implementations and the way the system is used. PIMOS is designed to be efficiently executed on a parallel hardware to be practically used in the research and development of application software, while other systems are built as experimental systems upon commercially available systems. In other words, PIMOS shares with other systems the objective of seeking for a novel method of constructing an operating system in concurrent logic language, but has an additional objective of providing a comfortable and efficient environment for application software development. This considerably affected various design trade-offs.

4.2 Objectives

In designing PIMOS, the following items were set as the design objectives.

Robustness: As PIMOS is to be used on a stand-alone parallel computer system, the robustness of the system is more important than in systems build upon another established system.

Internal Parallelism: The ultimate objective of PIMOS is, as stated above, to provide features for fully exploiting the power of parallel inference hardware. Various computation required in such an operating system should also be executed in parallel. Otherwise, the operating system will be the bottleneck of the whole system.

High Locality: The target architecture has loosely-coupled processors where inter-processor communication is much more costly compared with communication within one processor. Thus, the amount of communication between processors should be kept as low as possible.

Flexibility: As the hardware parameters are expected to change, the system should have enough flexibility

to be tuned to the given parameters. When tuning by changing parameters of the operating system becomes insufficient, non-trivial re-design of the system may be required. Thus, a system on whose improvement is easy is desirable.

4.3 Resource Management

Management of resources is the most fundamental and important role of an operating system. This section describes the design of the resource management mechanism of PIMOS.[5]

4.3.1 What Resources to Manage

In conventional systems, memory management and process management are the most important tasks of operating systems. As in other high-level language for symbolic manipulation, KL1 provides an automatic memory management feature including garbage collection. Thus, basic memory management is by the language implementation rather than PIMOS. As KL1 provides implicit concurrency and data-flow synchronization, context switching and scheduling are already supported by the language. Thus, PIMOS does not have to manage low-level fine-grained processes, but controls larger-grained groups of processes using the *shoen* feature of the kernel language.

On the other hand, PIMOS has full responsibility on the management of resources such as input and output devices. In the lowest level, I/O devices are provided as primitives of the kernel language to control physical device interface. Thanks to the descriptive power of the kernel language for reactive systems, such devices have a disguise of an ordinary process in the kernel language level. Their functionality, however, is at a level too low for application programs. Like any other operating systems, PIMOS virtualizes such devices, allowing application programs to control virtual devices with much higher-level functionality.

These virtual devices are actually a process that converts higher-level requests from user tasks into lower-level requests that physical devices can understand. The user tasks send their request messages to a stream connected to such a process. Thus, management of devices is management of the communication streams connected to them. Protection mechanisms are realized by inserting a filtering process to such streams, which examines messages going through the stream and rejects any illegal requests to the devices.

As mentioned above, process management by PIMOS is through the *shoen* construct. PIMOS virtualizes *shoen* also as a *task* with higher-level functionality for resource management. Tasks are a virtual device with the function of program execution with resource management facility. They can be controlled from user programs only through streams connected to it. The same protection mechanism of inserting message filtering processes is used here.

[5]More detailed description can be found in [Yashiro *et al.* 1992].

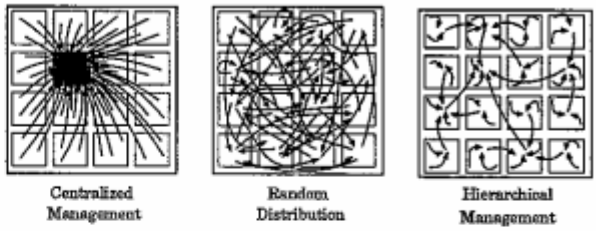

Figure 3: Distribution of Management Jobs

4.3.2 Hierarchical Resource Management

In most conventional operating systems, all the vital management information is centralized to the kernel, which is usually implemented as a single process. This centralization policy makes it easy to keep the management information consistent.

In a highly parallel system, however, such centralization of management information would become problematic. Even if the overhead of the kernel is only one percent, the processing speed of the kernel will be the bottleneck of the system in a system with only one hundred processors. Moreover, all the management requests will be targeted to the processor where the kernel process runs, resulting in a hot spot in the communication mechanism. In an operating system for highly parallel computer systems, management jobs also have to be distributed.

Random distribution of management jobs, using hashing technique for example, would relieve the bottleneck problem, but introduces a new problem of frequent communication, as the requests for operating system services arise everywhere without regard to where the service is provided.

To avoid the bottleneck and frequent communication at the same time, it is essential to distribute management jobs keeping the locality of information. PIMOS, thus, adopted hierarchical resource management policy. User tasks and resources allocated by the operating system form a hierarchical structure. As the design principle leaves computation mapping to application programs, processes of PIMOS responsible for management jobs will be allocated where requests for services arise, and those management processes also form a hierarchical structure corresponding to the structure of user tasks, called *resource tree*. This resource tree is the *kernel* of PIMOS.

No centralization of resource management information is made and no total ordering of resource allocation is

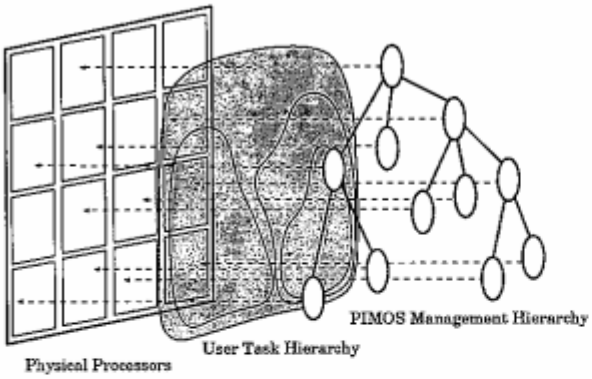

Figure 4: Task and Management Hierarchies

tried. A management process, which is a node in the resource tree, knows only of its parent and children. Allocation of a new resource is handled locally at one level in the hierarchy without reporting it to upper levels nor lower levels. When necessary, statistical summaries of management information is exchanged in the resource tree, but there is no single process that knows the state of the whole system precisely. The state of the whole system can be investigated by traversing the tree structure, but that would be costly and, because of the concurrent activities in the system, obtained information might already be obsolete when the the traversal completes. We found this loose management policy works fine without any problems.

4.3.3 Servers

All the services of PIMOS are provided by *servers*, which correspond to virtualized devices. Servers are realized as usual tasks to make the kernel compact and to enable easy addition of services.

An application program (client) requiring a service (to open a display window, for example) can ask for the service by requesting to the kernel with the name of the service. The kernel will look for the named service in a table it maintains and establishes a stream connection between the server task and the client task, inserting a filtering process for protection in the client task at the same time. Once the connection is established, the kernel will not look into messages passed through the stream; the server is protected by the inserted filter rather than a kernel process. When the service become no longer needed, the client process normally closes the communication stream. The remaining responsibility of the kernel is to notify the server of abnormal termination of the client.

4.4 File System

Earlier versions of PIMOS operating on an experimental model Multi-PSI [Takeda *et al.* 1990] left all the external input and output to its I/O front-end processor, PSI [Nakashima 1987]. This was profitable in rapidly constructing a software development environment for applications research. For massive external storage, such as disks, the imbalance of the low throughput communication with the I/O front-end and high performance processing power of the parallel hardware, however, became more apparent with PIM [Taki 1992].

We thus decided to connect disks more directly to processors of PIM for higher throughput and shorter delay. To minimize hardware development effort, we adopted SCSI (small computer standard interface) to interface disks available in the market. Although single SCSI can provide rather low throughput, PIM can have many of them, providing required total throughput.

As the interface provides only low-level block I/O to disks, we designed a file system to provide higher-level interface to application programs. In designing the file system, we took the following principles.

Distributed Cache: To lower interprocessor communication frequency, each processor should have its own cache of data in file. The cache mechanism should provide "Unix semantics": When one process writes into a file, the data should become available to other processes *immediately*. This is a constraint severer than in many distributed file systems where some delay is allowed [Levy and Silverschatz 1989], but it is mandatory in a system like PIMOS, where processes are usually cooperatively solving one problem. Thus, a distributed and coherent caching mechanism was designed, which is similar to cache coherence mechanisms provided by snoopy cache [Archibald and Bare 1986] but allows delay of communication.

Robustness: As all the system components, including the hardware, the operating system and the file system itself, are experimental and subject to damage caused by bugs, sufficient backing up mechanism is required to provide a comfortable software development environment. Logging of information vital to the file system and quick recovery mechanism using the logged information were designed.

More detailed description of the file system can be found in [Itoh *et al.* 1992].

4.5 Software Development Tools

Development of parallel software has many aspects different from development of sequential software. PIMOS provides various tools to support development of parallel software, described in this section.

4.5.1 Program Code Management

Executable programs are provided as data objects of type *module* by the kernel language and can be manipulated through language primitives by authorized software. Although the representation of executable programs differ in each hardware models, a common interface to manipulate programs is provided by PIMOS to encapsulate the differences.

Executable programs are stored in a database, which is a virtual device realized by a server task. To maintain the logical soundness of the specification, it is not desirable to introduce the notion of *modification*, not only for usual data but also for programs which are also data. Updating a program module does *not* mean modification of an already existing program, which might be running in parallel somewhere in the system; it merely means updating of the correspondence of module names and executable programs kept in the program database. The existing processes that are executing the program will not be affected by this update, except that, when the updated module is referenced by its name and the database is searched for, a new version of it will be found. Multiple versions of the same program can thus coexist in a system. This not only keeps the semantics clean but also allows efficient distributed implementation.

4.5.2 Debugging Tracer

The most frequently used tools in debugging programs are tracers that allow programmers to look into the details of program execution. PIMOS also provides a program tracer for this debugging purpose.

Execution of programs in a high level language form a hierarchical structure such as nested subroutine calls. In case of subroutines in sequential languages, substructures corresponding to subroutine invocations directly correspond to a time interval, such as "during execution of a subroutine." Tracing or not tracing that particular substructure can be effected by switching tracing on and off during that time interval. In concurrent languages, such direct correspondence does not exist as many such substructures are executed concurrently. If the number of processes is limited, providing multiple windows, one for each process, and switching tracing on each of them might be a good idea. In case of KL1 programs, the number of processes typically goes up to millions, much more than tractable this way. The tracer of PIMOS also provides a feature to direct the trace information to multiple windows, but their role is only auxiliary.

The *shoen* construct of the kernel language is used to control tracing, to obtain trace information and to control execution of traced programs. Each goal executed in a *shoen* can be marked as a *traced* goal. When the language implementation finds reduction of such a goal to its subgoals, the newly created subgoals will be reported from the report stream of the *shoen* as a message. The tracer observing the stream presents the information to the user and queries what to do with the goals, that is, whether to simply execute them or execute them with *trace* marks again. The goals can also be suspended for a while to control their execution order.

The tracer also has interface with the deadlock detection mechanism provided by the KL1 implementation [Inamura and Onishi 1990].

4.5.3 Performance Tuning

As stated above, a strong point of the kernel language KL1 is that mapping of computation, both over processors and over time, can be altered without affecting the correctness of programs. Finding a mapping which realizes efficient computation is one of the most important research topics in application software research on the parallel inference system.

However, conjecturing mapping only by statically analyzing programs is a very difficult task. In many cases, actually running the programs and gathering statistical information reveals many aspects of programs that are easily overlooked. To help such experimentation, PIMOS provides a tool for evaluating load distribution algorithms.

Profiling information of parallel programs has three axes: what, when, and where. In sequential execution, "where" is a constant and the "when" is not important, since the execution order is strictly designated. Simple profiling tools that can tell "what" (which part of the program) took how much time will thus suffice. However, all three axes are important when parallel execution is our concern. The kernel language implementation has the feature to provide three-dimensional statistics on *what* (which part of the program, or, in a lower level, whether usual computation, interprocessor communication or garbage collection) is executed *where* (on which processor) and *when*.

As it is not easy for a human to understand massive raw data from hundreds of processors, a profiling tool named *ParaGraph* is provided to analyze the data and present it to the user graphically (Figure 5). The system provides displays from several different viewpoints, making the analysis easier. The ParaGraph system is described in more detail in [Aikawa 1992 *et al.*].

4.5.4 Virtual Machine

As all the communication between user programs and PIMOS is initiated through the control and report streams of *shoens*, a user program can emulate PIMOS by running programs within a *shoen* and observing its interface streams.

The same technique also can be used to debug PIMOS itself by writing an emulator of the whole parallel computer system, a *virtual machine*. This facility provides a way to debug PIMOS under the software environment

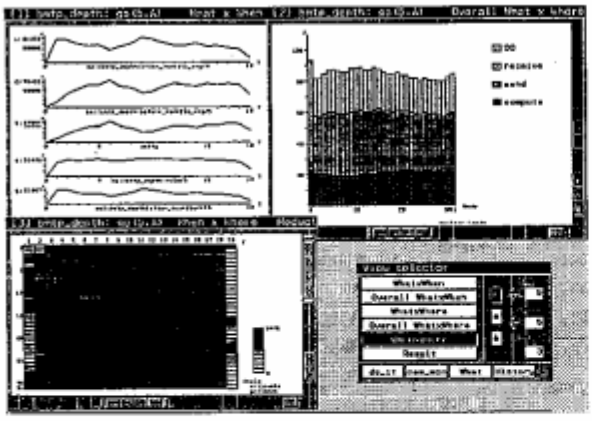

Figure 5: Sample Output of ParaGraph

provided by PIMOS itself. As the virtual machine is no more than a usual task in PIMOS, the protection mechanism of PIMOS prevents bugs of the debugged version from propagating to the real PIMOS. Also, the profiling system ParaGraph can be used for performance tuning of PIMOS. This facility has been conveniently used in debugging and tuning of the kernel of PIMOS.

5 Experiences

The first version of PIMOS was implemented on Multi-PSI [Takeda *et al.* 1990] in 1988 [Chikayama *et al.* 1988]. It has been revised with various enhancements and improvements since, through experiences with research and development of experimental software on many application areas. As the experiences with application software are reported elsewhere (see [Nitta *et al.* 1991] for example), this section mainly reports the experiences of the development of PIMOS itself in the kernel language KL1.

5.1 Automatic Synchronization

The automatic data-flow synchronization mechanism of KL1 assured portability of PIMOS to hardware systems with different architectures.

The first version of PIMOS was developed in parallel with the development of the experimental parallel inference machine Multi-PSI. During its early development phase when no physically parallel system running the kernel language was available yet, a sequential implementation was used in the development. The scheduling of goals was fixed on the implementation. We could not completely deny the possibility of any crucial synchronization problems in the system hidden by the fixed scheduling of the emulator; that was our first experience of actually writing a large-scale software in KL1.

PIMOS was ported to Multi-PSI when its KL1 implementation got ready. We found almost no synchronization problems there (except for a small number of higher-level design problems) although the scheduling on the real parallel machine is quite different from the emulator. We were certain that this should be the case, but actually experiencing this made us more confident of the great merit of writing a system in a language with automatic data-flow synchronization.

In 1991, the first model of the parallel inference machines, PIM/m and its KL1 implementation was made available for software installation. After revising the low-level I/O mechanism to fit the system to this new platform, PIMOS began working almost immediately on this system without revealing any problems. This was not surprizing as the kernel language implementation on the system used the identical scheduling policy as the Multi-PSI system.

Later in the same year, the system was ported to an emulator of PIM running on a commercially available parallel processor. The emulator was primarily for debugging the design of kernel language implementation for models consisting of loosely-coupled *clusters*, each of which has several processors sharing a memory bus. The scheduling policy of this emulator was completely different from Multi-PSI or PIM/m, as the language implementation distributes goals automatically among processors in a cluster. As we expected, and also to our surprise, PIMOS ran without any problems in itself but revealing some problems with the language implementation in stead.

Currently (February 1992), the kernel language implementation and PIMOS are being ported to other models of PIM. We are now certain that there won't be any fundamental problems in porting PIMOS to those models.

5.2 Fine-Grain Concurrency

It is true that most human algorithm designers are liable to regard computation as a sequential process and some extra effort is needed to think of many cooperating processes for a single job. This fact is sometimes regarded as against parallel processing, that designing parallel computation is unnatural for human. The implicit concurrency of the kernel language, however, resulted in interesting phenomena.

Most algorithms in fact are designed having sequential processing in mind or limited aspects of the parallelism. Once a program for the algorithm is written down in the kernel language, the program often shows much more concurrency than the designer had in mind, as the language reveals implicit fine-grain concurrency. The designer can look into the program more objectively and find different aspects of concurrency implied there. Sometimes, the concurrency so found is a good candidate for obtaining larger physical parallelism for increased ef-

ficiency. Mapping pragmas exploiting the concurrency can then be added to the program to make it run with higher parallelism and more efficiently. This should not have been possible if the language had only larger-grain concurrency.

5.3 Descriptive Power

Through the development of PIMOS, the descriptive power of KL1 for both concurrency and parallelism was proved to be sufficient.

The ability of describing reactive systems allowed the language to provide primitives to control external I/O devices in a coherent manner; external devices could be modeled as an ordinary process without introducing any extralogical features to the language. This allowed straightforward implementation of a virtual machine, which helped the development considerably.

The *shoen* construct and the priority control mechanism of the kernel language provided sufficient functionality required to control execution of various activities in the system. For example, in case a user program ran into an infinite loop, the following steps will enable interruption of such a program.

- As the device handlers are given higher priority than user processes, an interrupt from the keyboard can be sensed.
- As the command shell, which is a user task, lets jobs under its control run in a priority lower than itself, the shell can sense the interrupt.
- Using the *shoen* construct, the shell can stop the task in an infinite loop.

5.4 Ease of Programming

Many programmers seem to have felt uneasiness with the kernel language when the system first began utilized in application software development. The largest source of the problem seems to be in too much freedom of programming styles.

The bare kernel language allows multiple input/output modes of logical variables; the same process can read or write the same shared variable, depending on situations. Although this is allowed in the language, it often introduces race conditions which become problematic only with specific scheduling. Such a bug is hard to fix as tracing the execution or modifying the program to report information for debugging may change the scheduling, hiding the problem away. Gradually, a programming style has been established where I/O modes of logical variables are statically fixed. This indicated the direction of subsetting of the language (see section 6).

Another problem was how to organize numerous concurrent processes. Many styles have been tried and the object-oriented programming style [Shapiro and Takeuchi 1983] has been accepted as the *de facto* standard. Many programming idioms have been established upon this object-oriented style through experiences [Chikayama 1991], which suggested the direction of the design of higher level languages (see section 3.3).

Automatic data-flow synchronization wiped away low-level synchronization problems, allowing programmers to concentrate on higher-level issues. With the programming style established and the software development environment enhanced based on the experiences, describing parallel software in the kernel language has now become not much more difficult than programming sequential programs in other languages for symbolic processing, such as Lisp.

The largest difficulty remaining is that of designing algorithms of computation mapping for efficient execution. Separation of correctness and efficiency issues in the language design and the visual performance analysis tool facilitated experimentations of mapping algorithms considerably, but still the task is not easy. Further research in this direction seems mandatory.

6 Future Work

A problem with the current parallel inference system, consisting of parallel inference machines, KL1 implementations and PIMOS, is that the system runs only on specially devised hardware. Although the system can execute KL1 programs very efficiently, requiring special hardware is a serious obstacle in sharing the environment with researchers world-wide. A portable implementation of the kernel language working on Unix systems is available and was utilized in early stages of software development, but, as it is implemented as an abstract machine interpreter, its limited performance makes it inappropriate for serious experimental studies.

To solve the problem, research in subsetting the language to allow more concise and efficient implementations has been conducted with promising preliminary results [Ueda and Morita 1990]. A separate effort of implementing KL1 by translating to C also indicated that reasonable performance can be obtained with very high portability [Chikayama 1992]. These results indicate the possibility of implementing the language on stock hardware efficiently for use in parallel software research. In addition to such an implementation, PIMOS, especially its software development environment, should also be ported to stock hardware to provide common basis of research and development of highly parallel knowledge information processing systems.

7 Conclusion

An overview of the research and development of the basic software for the parallel inference system of the FGCS project is given.

The system aims at establishing the basis of software technology for highly parallel computer systems. The research and development adopted a middle-out approach of designing a programming language first and then continuing the design both upwards to the application software and downwards to the hardware architecture simultaneously. The kernel language KL1 and the operating system PIMOS were designed and implemented.

The systems working on experimental parallel inference hardware Multi-PSI and a model of parallel inference machine PIM have been used in the research and development of application software since 1988. Our experiences have proved that the kernel language is expressive enough for describing an operating system for parallel processing systems and various application software. The features of the language that separated correctness and efficiency issues, along with the programming environment provided by the operating system, made empirical research of parallel software much easier than in conventional environments.

Further research in computation mapping is needed in future. Development of an efficient and comfortable environment on stock hardware is another important work to be done.

Acknowledgements

The design and implementation of KL1 and PIMOS for the parallel are collaborative work of many researchers too numerous to list here. The author would like to thank Kazunori Ueda for his helpful comments on an earlier version of this paper.

References

[Aikawa 1992 *et al.*] S. Aikawa, K. Mayumi, H. Kubo, F. Matsuzawa and T. Chikayama. ParaGraph: A Graphical Tuning Tool for Multiprocessor Systems. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1992*, ICOT, 1992.

[Archibald and Bare 1986] J. Archibald and J. L. Bare. Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. In *ACM Trans. on Computer System*, Vol. 4, No. 4 (1986), pp. 273–298.

[Burton 1985] F. W. Burton. Speculative Computation, Parallelism and Functional Programming. In *IEEE Trans. Computers*, Vol. C-34, No. 12 (1985), pp. 1190–1193.

[Chikayama 1984] T. Chikayama. Unique Features of ESP. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1984*, ICOT, 1984, pp. 292–298.

[Chikayama 1991] T. Chikayama. For KL1 Programming without Tears. In *Proc. KL1 Programming Workshop '91*, ICOT, 1991, pp. 8–14. *in Japanese.*

[Chikayama 1992] T. Chikayama. A Portable and Reasonably Efficient Implementation of KL1. To appear as an ICOT Tech. Report, ICOT, 1992.

[Chikayama and Kimura 1987] T. Chikayama and Y. Kimura. Multiple Reference Management in Flat GHC. In *Proc. Fourth Int. Conf. on Logic Programming*, MIT Press, 1987, pp. 276–293.

[Chikayama *et al.* 1988] T. Chikayama, H. Sato and T. Miyazaki. Overview of the Parallel Inference Machine Operating System (PIMOS). In *Proc. Int. Conf. on Fifth Generation Computer Systems 1988*, ICOT, 1988, pp. 230–251.

[Clark and Gregory 1981] K. L. Clark and S. Gregory. A Relational Language for Parallel Programming. In *Proc. ACM Conf. on Functional Programming Languages and Computer Architecture*, ACM, 1981, pp. 171–178.

[Clark and Gregory 1983] K. L. Clark and S. Gregory. PARLOG: A Parallel Logic Programming Language. Research Report DOC 83/5, Dept. of Computing, Imperial College of Science and Technology, 1983.

[Clark and Gregory 1984] K. L. Clark and S. Gregory. Notes on Systems Programming in PARLOG. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1984*, ICOT, 1984, pp. 299–306.

[van Emden and de Lucena Filho 1982] M. H. van Emden and G. J. de Lucena Filho. Predicate Logic as a Language for Parallel Programming. In *Logic Programming*, K. L. Clark and S. -Å. Tärnlund (eds.), Academic Press, 1982, pp. 189–198.

[Foster 1987] I. Foster. Logic Operating Systems: Design Issues. In *Proc. Fourth Int. Conf. on Logic Programming*, J.-L. Lassez (ed.), MIT Press, Vol. 2, 1987, pp. 910–926.

[Furuichi *et al.* 1990] M. Furuichi, K. Taki, N. Ichiyoshi. A Multi-Level Load Balancing Scheme for OR-Parallel Exhaustive Search Programs on the Multi-PSI. In *Proc. Second ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, 1990, pp. 50–59.

[Goto *et al.* 1988] A. Goto, M. Sato, K. Nakajima, K. Taki, and A. Matsumoto. Overview of the Parallel Inference Machine Architecture (PIM). In *Proc. Int. Conf. on Fifth Generation Computer Systems 1988*, ICOT, 1988, pp. 208–229.

[Hircsh *et al.* 1987] M. Hircsh, W. Silverman and Ehud Shapiro. Computation Control and Protection in the Logic System. In *Concurrent Prolog: Collected Papers*, Ehud Shapiro (ed.), MIT Press, Vol. 2, 1984, pp. 28–45.

[Hoare 1985] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[Inamura and Onishi 1990] Y. Inamura and S. Onishi. A Detection Algorithm of Perpetual Suspension in KL1. In *Proc. Seventh Int. Conf. on Logic Programming*, MIT Press, 1990, pp. 18–30.

[Itoh *et al.* 1992] F. Itoh, T. Chikayama, T. Mori, M. Sato, T. Kato and T. Sato. The Design of the PIMOS File System. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1992*, ICOT, 1992.

[Kondoh and Chikayama 1988] S. Kondoh and T. Chikayama. Macro Processing in Prolog. In *Proc. Fifth Int. Conf. and Symp. of Logic Programming*, 1988, pp. 466–480.

[Konishi *et al.* 1992] K. Konishi, T. Maruyama, A. Konagaya, K. Yoshida, T. Chikayama. Implementing Streams on Parallel Machines with Distributed Memory. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1992*, ICOT, 1992.

[Levy and Silverschatz 1989] E. Levy and Z. Silberschatz. Distributed File Systems: Concepts and Examples. Tech. Report

TR-89-04, Dept. of Computer Science, The University of Texas at Austin, 1989.

[Maher 1987] M. J. Maher. Logic Semantics for a Class of Committed-Choice Programs. In *Proc. Fourth Int. Conf. on Logic Programming*, MIT Press, 1987, pp. 858–876.

[Milner 1989] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[Miyazaki *et al.* 1985] T. Miyazaki, A. Takeuchi and T. Chikayama. A Sequential Implementation of Concurrent Prolog Based on the Shallow Binding Scheme. In *Proc. 1985 Symp. on Logic Programming*, IEEE, 1985, pp. 110–118.

[Nakashima 1987] H. Nakashima and K. Nakajima. Hardware Architecture of the Sequential Inference Machine PSI-II. In *Proc. 1987 Symp. on Logic Programming*, IEEE, 1987.

[Nitta *et al.* 1991] K. Nitta, K. Taki and N. Ichiyoshi. Experimental Parallel Inference Software. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1992*, ICOT, 1992.

[Okumura and Matsumoto 1987] A. Okumura and Y. Matsumoto. Parallel Programming with Layered Streams. In *Proc. 1987 Symp. on Logic Programming*, IEEE, 1987, pp. 224–231.

[Shapiro 1983] E. Y. Shapiro. A Subset of Concurrent Prolog and Its Interpreter. Tech. Report TR-003, ICOT, 1983.

[Shapiro 1986] E. Y. Shapiro. Systems Programming in Concurrent Prolog, In *Logic Programming and its Applications*, M. van Canegham and D. H. D. Warren (eds.), 1986, Ablex Publishing Co., 1986, pp. 50–74.

[Shapiro and Takeuchi 1983] E. Shapiro and A. Takeuchi. Object-oriented Programming in Concurrent Prolog. In *New Generation Computing*, Vol. 1, No. 1 (1983).

[Susaki and Chikayama 1991] K. Susaki and T. Chikayama. A Process-Oriented Language AYA upon KL1. In *Proc. KL1 Programming Workshop '91*, ICOT, 1991, pp. 117–125. *in Japanese.*

[Takeda *et al.* 1990] Y. Takeda, H. Nakashima, K. Masuda, T. Chikayama and K. Taki. A Load Balancing Mechanism for Large Scale Multiprocessor Systems and its Implementation. In *New Generation Computing*, Vol. 7, No. 2 (1990), pp. 179–195.

[Taki 1992] K. Taki. Parallel Inference Machine PIM. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1992*, ICOT, 1992.

[Taki 1992] K. Taki. Parallel Inference Machine PIM. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1992*, ICOT, 1992.

[Tamaki 1987] H. Tamaki. Stream-Based Compilation of Ground I/O Prolog into Committed-choice Languages. In *Proc. Fourth Int. Conf. on Logic Programming*, MIT Press, 1987, pp. 376–393.

[Ueda 1986] K. Ueda. Guarded Horn Clauses. In *Logic Programming '85*, E. Wada (ed.), Lecture Notes in Computer Science 221, Springer-Verlag, 1986, pp. 168–179.

[Ueda 1987] K. Ueda. Making Exhaustive Search Programs Deterministic. In *New Generation Computing*, Vol. 5, No. 1 (1987), pp. 29–44.

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

[Ueda 1988b] K. Ueda. Theory and Practice of Concurrent Systems. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1988*, ICOT, 1988, pp. 165–166.

[Ueda 1990] K. Ueda. Designing a Concurrent Programming Language. In *Proc. InfoJapan'90*, Information Processing Society of Japan, 1990, pp. 87–94.

[Ueda and Chikayama 1985] K. Ueda and T. Chikayama. Concurrent Prolog Compiler on Top of Prolog. In *Proc. 1985 Symp. on Logic Programming*, IEEE, 1985, pp. 119–126.

[Ueda and Chikayama 1990] K. Ueda and T. Chikayama. Design of the Kernel Language for the Parallel Inference Machine. In *The Computer Journal*, Vol. 33, No. 6 (1990) pp. 494–500.

[Ueda and Furukawa 1988] K. Ueda and K. Furukawa. Transformation Rules for GHC Programs. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1988*, ICOT, 1988, pp. 582–591.

[Ueda and Morita 1990] K. Ueda and M. Morita. A New Implementation Technique for Flat GHC. In *Proc. Seventh Int. Conf. on Logic Programming*, MIT Press, 1990, pp. 3–17. A revised, extended version to appear in *New Generation Computing*.

[Yashiro *et al.* 1992] H. Yashiro, T. Fujise, T. Chikayama, M. Matsuo, A. Hori and K. Wada. Resource Management Mechanism of PIMOS. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1992*, ICOT, 1992.

[Yoshida and Chikayama 1990] K. Yoshida and T. Chikayama. A'UM: A Stream-Based Object-Oriented Language. In *New Generation Computing*, Vol. 7, No. 2 (1990), pp. 127–157.