ICOT Technical Report: TR-439

TR-439

Distributed Implementation of KL1 on the Multi-PSI/V2

by
K. Nakajima, Y. Inamura, N. Ichiyoshi,
K. Rokusawa and T. Chikayama

December, 1988

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191 ~ 5
Telex ICOT J32964

Institute for New Generation Computer Technology

Distributed Implementation of KL1 on the Multi-PSI/V2

Katsuto Nakajima Yū Inamura Nobuyuki Ichiyoshi
Kazuaki Rokusawa Takashi Chikayama
Institute for New Generation Computer Technology (ICOT)
4-28, Mita-1, Minato-ku, Tokyo, 108, JAPAN

Abstract

KL1 is a stream AND-parallel logic programming language based on Flat GHC. This paper describes the implementation issues of a parallel KL1 system. The target machine of this system is a non-shared memory multi-processor, Multi-PSI/V2, in which up to 64 processing elements (PEs) are connected by a message passing network. The key issues are (1) how to achieve efficient intra-PE and inter-PE garbage collection, (2) how to reduce the amount of inter-PE communication, and (3) how to avoid making redundant copies over many processors.

The well-defined semantics of KL1 allowed incremental intra-PE garbage collection by the Multiple Reference Bit (MRB) technique and incremental inter-PE incremental garbage collection by the Weighted Export Counting (WEC) technique. The communication required for inter-PE process control is minimized by the Weighted Throw Counting (WTC) scheme. We introduced a global structure management technique to avoid making duplicate copies of the same large data.

The implementation is completed, and the system is being used to research parallel software, and to evaluate for refining it and also for designing the future system, the parallel inference machine, PIM.

1 Introduction

The Japanese Fifth Generation Computer Project has a target to build a parallel inference machine, PIM [Goto 88], for running large-scale logic based programs. It aims to achieve hundreds of times the performance of the present computer systems by using a parallel processing mechanism. In the initial stage of the project, we realized that research in parallel software is important for designing a highly parallel inference architecture, because the architecture is influenced by the nature of application programs and little was known about it. The problem was that no machine existed that had the capability of running parallel application programs of realistic size. The Multi-PSI machine was developed to fill this gap [Taki 88]. It also served as a workbench for evaluating various new implementation techniques.

The Multi-PSI is a non-shared memory multi-processor, whose processing elements (PEs) are the CPUs of the personal sequential inference (PSI) machine, which are readily available. Up to 64 PEs are connected to each other to form a two-dimensional mesh network with a message switching and automatic routing capability. The Multi-PSI/V1 (up to 6 PEs) used the first version of the PSI as PEs, and the new Multi-PSI/V2 uses the more compact and faster version, PSI-II [Nakashima 87] as PEs.

Non-shared memory architecture was chosen for the Multi-PSI, since the machine was to serve as a prototype of a scalable PIM, the number of whose processors rules out strict shared-memory architecture.

After we developed a distributed implementation of the KL1 (Kernel Language version 1), which is an AND-parallel logic programming language, on the Multi-PSI/V1 [Ichiyoshi 87], we continued with our research in efficient implementation, and came up with an improved implementation on the Multi-PSI/V2.

This paper describes the problem we faced in designing our KL1 implementation, presents the way we (partially) solved these problems, and reports on the current status and the remaining problems.

2 KL1 Features and Implementation Issues

2.1 KL1 (Kernel Language Version 1)

KL1 (Kernel Language Version 1) is a stream AND-parallel logic programming language based on Flat GHC. The KL1 program is made up of a collection of Guarded Horn Clauses, whose form is:

$$\underbrace{H :- G_1, \ldots, G_m}_{\text{guard}} \mid \underbrace{B_1, \ldots, B_n.}_{\text{body}} \; (m > 0,\ n > 0)$$

where H is called the *head*, G_i the *guard goals*, and B_i the *body goals*. The vertical bar (|) is called the *commitment operator*. The logical reading of the clauses is the same as GHC [Ueda 86].

KL1 is provided with the following meta-programming functions so that it becomes a practical and efficient parallel language not only for describing application programs but for an operating system.

(1) Shōen mechanism: A shōen is a meta-logical unit to control or monitor the KL1 goals in it. It has a pair of streams, named *control stream* and *report stream*. The control stream is used to start, stop or abort the goals from outside shōen. Termination of all goals or events that occurred inside a shōen, such as a failure or an exception, are reported on the report stream from inside the shōen. Shōen can be nested to form a tree-like structure (shōen tree) whose leaves are KL1 goals.

(2) Resource management: The system should be safe from a wastefully running user goal such as an erroneous infinite loop. Shōen should be given some resources for the execution inside it through the control stream. The resource shortage is reported on the report stream.

(3) Priority pragma: Scheduling by using *priority* contributes to efficient problem solving. The shōen has a priority range and each goal inside it can have individual priority within this range. The priority is specified by a priority pragma with a relative value in the allowed range in the source program $(\ldots, B@priority(Prio), \ldots)$.

(4) Throw goal pragma: A throw goal pragma $(\ldots, B@processor(PE), \ldots)$ in the source program denotes the static load distribution. It also contributes to efficient execution on a multi-processor machine.

2.2 Implementation Issues for KL1

To execute KL1 on the network connected parallel machine, the following points should be kept in mind in designing the system.

2.2.1 Garbage Collection

As goals are not executed in the depth-first manner, the stack mechanism used in most Prolog implementations is not suitable for KL1 implementations. Therefore, heap based memory management must be used for flexible memory use, although memory reclamation is generally inefficient with this scheme. The time spent in garbage collection (GC) may seriously affect the system performance. On a non-shared memory multi-processor, the degradation by GC should be considered more seriously, because naive inter-PE GC might take time proportional to the length of the reference chains over many processors. Implementing incremental GC with low costs for inter-PE and intra-PE data is one of our major issues because it is better than non-incremental GC in terms of the accessing locality, which leads to, for example, a good cache hit ratio.

2.2.2 Data Management

KL1 has a property where data objects that have been instantiated once can be copied, while keeping the program logic. To allow local accesses, data shared by PEs should be copied. However, indiscreet copying leads to needless data transfer.

2.2.3 Message Communication

Message passing communication is more expensive than communication on a shared memory. The communication delay is also large. We have to pay attention to reduce the amount of inter-PE communication and to maintain quick responses.

3 KL1-B and Its Implementation

3.1 Execution Model

KL1 programs are compiled into the sequence of a WAM-like abstract machine instruction set, KL1-B [Kimura 87]. It is a register based instruction set and serves as an efficient interface between language and machine architecture.

Goals in a PE are categorized as: (1) *ready goals* which are waiting for execution, (2) *current goal* which is being executed, or (3) *suspended goals* which are hooked on variables to be instantiated (Figure 1).

A reduction cycle is described as follows. When a current goal calls a predicate, the guard parts of the clauses for the predicate are tried one by one. If no clause commits and at least one clause is suspended, the goal is hooked to the causal variable of the suspension. If no clause commits and there is no suspended clause, a *failure* is reported on the report stream of the parent shōen with the goal information. In both cases, a new goal is popped from the goal stack to evaluate. If one of the clauses commits, all the body goals except for the leftmost one in the source code are pushed to the goal stack. The leftmost goal is chosen as the next one to evaluate. When a clause without body goals commits, another goal is popped from the goal stack.

Every reduction cycle, request for (non-incremental) GC and message arrivals from the network are checked, and they are processed if necessary. This timing, called the *slit check*, is most suitable for switching the process because the PE is free from goal contexts.

3.2 Executing KL1-B

The KL1-B instructions including some tens of instructions for built-in predicates are directly interpreted by the microcode to attain a reasonable execution speed as a practical tool for software research.

The microcode of the PE can perform various functions in parallel, such as tag insertion, two-way or multi-way branching on a tag, specially prepared counter and flag operation, ALU operation and memory access operation. The arguments for the unification are put on the registers at every reduction cycles. The control information such as the priority and the shōen resource is manipulated on the registers as long as the execution context can remain unchanged.

3.3 Goal Stack

Strict scheduling with priority can be managed by having only one prioritized goal stack in the system. However, the rush to such a global resource causes a serious bottleneck. To avoid this, every processor has a prioritized goal stack, at the sacrifice of scheduling strictness. In our experience, local goal stack management is realistic and efficient enough to control the execution in the system in most cases.

3.4 Memory Management

As stated in 2.2.1, efficient GC is vital in a KL1 implementation. We have developed the MRB technique [Chikayama 87] for intra-PE incremental GC. In this scheme, the pointer has one-bit information to tell whether it is the only pointer to the referenced data. Even with one-bit counter, most, but not all, garbage cells are collected because data objects rarely have multiple referencers in KL1 programs. Collected cells are linked in the free lists to be reused. We have several individual free lists for records of various sizes.[1] When records in a free list are exhausted, a pre-determined number of new records is created on the heap top and linked to the list.

[1] In the current implementation, the sizes are from 1 to 8, 16, 32, 64, 128 and 256. Record over 256 w is allocated on the heap top.

4 Inter-PE Processing

Goals with throw goal pragmas in a shōen are distributed by `throw_goal` messages over many processors. Distributed goals communicate with each other through the shared variables by `read` or `unify` messages. This section discusses how the shared data is accessed from outside the PE, and how the distributed goals are monitored or controlled in these situations.

4.1 Inter-PE Data Management

4.1.1 Copying Shared Data

When a goal is thrown to another PE, its arguments are also carried with it. If the argument is an atomic value, the value itself is attached to the goal. If it is an unbound variable, a pointer to the variable is created and carried. For a structure argument, there are three choices. One is to create and carry a pointer to the structure (zero level). The contents are read when they are actually used in a unification. The second is to copy all the elements of the structure including all nested substructures (infinite level). The third is to copy all the elements at the surface level (one level).

In a distributed system like the Multi-PSI where the cost of the inter-PE reference is relatively high, it is better to copy for later accesses in many cases. However, copying at an infinite level may cause unnecessary duplication because the passive or active unification for the structure might fail at any level. It seems better to limit the copying level according to the situation. In our current implementation, copying at zero level is done at goal throw timing and copying at one level at unification timing. However, if it is known that the element of a structure will be read early or late (such as the tail of a stream), it is better to copy the elements at one time as long as they are bound to a value. This is left as a future optimization.

4.1.2 Export/Import Table

When a PE exhausts its heap memory, garbage must be collected. If the PE does not know whether a cell is referenced from other PEs, or whether it is garbage, the PE cannot perform GC for its memory without cooperation by all PEs. The global GC, where all PEs performs GC at one time with exchanging messages of marking and telling movements of object cells, is a solution, but it is very time consuming.

In a large scale non-shared memory multi-processor, local (non-incremental) GC, where the PE performs GC alone for its memory when exhausted, is desirable in terms of the system performance.

It is possible if the object cells referenced from outside are represented by a kind of global ID for the external PEs. The garbage collecting PE only has to maintain the translation table according to the local object movements at a local GC. The table is called the *export table* (Figure 2). The object cells referenced from the external PEs are said to be *exported*, and on the referencing side, they are said to be *imported*. The global ID is represented in the form $< pe, entry >$, where *pe* is a PE number and *entry* is the entry position of the export table. The global ID is called the *external ID*.

4.1.3 Incremental Inter-PE GC by WEC

In order to reclaim the garbage cells pointed to by the export table, the entries of the table have to be collected when they become garbage. We employed the weighted export counting (WEC) method [Ichiyoshi 88] to perform inter-PE incremental GC. This scheme is based on the weighed reference counting (WRC) principle [Watson 87] and has the following novel features.

- When an importing pointer is divided in two, its weight is split and no message is sent to the exporting PE to maintain the reference count.
- No racing occurs in terms of count (weight) zero checking at the exporting PE.

An integer representing some WEC value is attached to an exported pointer and is stored in the *import table*. Each entry for the imported pointers accumulates a WEC when the same data is imported again. (See 4.1.4) The number of the import, the *import count*, is also counted. When the contents of the imported pointer become unnecessary, `release` message is sent to the exporting PE to return the amount of the WEC. `Release` messages are sent when: (1) an instantiated value is returned by an `answer_value` message as the response of `read` message, (2) the import count reaches zero by incremental GC by using the MRB mechanism, or (3) the imported pointer becomes garbage by local GC in the imported PE. (See 4.1.6)

When an export entry receives a `release` message, the WEC in the entry is maintained. If it becomes zero, the entry is reclaimed. In this case, the exported object cell itself may also be reclaimed when the export table entry is known to be the single reference to the cell by the MRB mechanism.

4.1.4 Re-exporting

The same variable may be exported to the same PE. If the re-exported reference to a variable is given with a different external ID, it cannot be determined as an variable that was originally the same. Therefore, the importing PE may send **read** messages twice and if the object is a structure data, it is brought twice by **answer_value**. This can be avoided by reusing the same export/import table entry. For this purpose, the *export hash table* and the *import hash table* are provided on each side. The export hash table associates the exported object addresses with their external IDs, and the import hash table associates the imported external IDs and the import table entry.

4.1.5 White or Black Export by Using MRB

Our external reference management with WEC has overhead in terms of maintaining both WEC and import count, and of looking up the hash table to check the re-exporting. Fortunately, the MRB mechanism can be used to optimize this. In order to export a single reference pointer at a low cost, a simplified pair of export and import tables, called *white export* and *import table*, are used. The original table is called *black export/import table.* From our observation, once duplicated pointers will often be copied again later. In contrast, a single reference pointer will not be duplicated after being exported somewhere else. Thus, the white export and import table do not have the hash tables because the exported pointers are rarely exported again for the same reason.

The white import table can be considered as an import table for the pointers whose WEC and import count equals one, and its entries are immediately released when the imported pointers are collected by the MRB GC.[2] The white export entries for them are also released by only receiving **release** message. The effectiveness of this optimization depends heavily on the programs and the evaluation is one of our future areas of research.

4.1.6 Local Garbage Collection

The current implementation of local GC is based on the conventional copying GC scheme. The garbage collector moves all data cells reachable from the prioritized goal stacks and the export table to a new heap area. After copying, valid entries in the import table are swept. If unmarked entries are found, **release** messages are sent to the exporting PEs to return their WECs.

The local GC can be a big factor in the total performance because PEs communicating with

[2] If an imported pointer is split, the MRB of both pointers is turned on so that the import entry is not released when one of the pointers becomes garbage.

the garbage collecting PE have to wait for its termination. One solution to improve the GC time would be the *generation GC* [Lieberman 83][Nakajima 88], which avoids moving long life objects at every GC. This is an on-going study.

4.2 Global Structure Management

In our export system, the external ID originates from the exporting PE. For example, if PE_B has a copy of the structure in PE_A, and PE_C has external references to both the copy in PE_B and the original in PE_A, their external IDs are not the same. PE_C will have two copies after reading them. If the structure is big and will live long like the program code, it is inefficient in terms of both the memory space and the data transfer overhead. In the worst case, as many copies as the number of PEs in the system can be created in a PE.

To solve this problem, we introduced the *structure ID* for such structures, which is a global ID attached to an instantiated structure. By using this, the same structure is duplicated only once in a PE even if the external data to be read is in a different PE. Currently, we manage the structure ID for only program code objects.

When a `read` message is sent to a PE for a structure with structure ID, only the ID is returned in the `answer_value` message. If the PE requesting read receives only the ID, it looks up the *structure ID hash table* with the returned ID to search for the structure address if the PE already has the structure. If it is not found, the `read` message is sent again to copy it. The other hash table, the *structure address hash table*, is used when the PE returns the structure ID instead of the structure itself in the `answer_value` message.

The problem in this scheme is the collection of the garbage ID, which needs a kind of global GC scheme, and is left for future research.

4.3 Goal Control by Shōen and Foster Parent

A shōen and the goals in it (*child goals*)f have to communicate for the execution control from the shōen such as stopping and abortion, and for the report from the child goals such as the goal termination and the resource shortage. In order to reduce the message traffic towards a shōen, we can employ a cache technique. When a goal is moved from the shoen PE (PE in which the shōen exists) to another, a *foster parent* is created on the PE to which the goal migrated [Ichiyoshi 87]. Only one foster parent is created for the shōen on each PE which has goals belonging to the shōen (Figure 3).

The foster parents have the *shōen status*, that is *running, stopped, aborted,* etc., and the *child count*, which is the number of child goals created on the PE and the cached resource information for it. Goal termination is checked at the shōen only when one of the foster parents sends a `terminated` message to report that the child count in it has reached zero.

The detection of the goal termination is one of the difficult problems in parallel language systems, especially, when messages may be in transit on the network as in the Multi-PSI. Even if all the foster parents report termination, the shōen is not necessarily terminated because there are goals in transit.

Our solution for it is the weighted throw counting (WTC), which is also an application of the WRC scheme. In this scheme, each shōen manages the amount of the count. WTC is attached to a thrown goal or a unify message, and the foster parents accumulate WTCs on receiving the messages. Foster parents can split it when they throw child goals. When all the goals in a foster parent terminate, the amount of WTC kept by the foster parent is returned to the shōen. The shōen can determine the true termination of the child goals when all WTC are returned back.

For a full description of the WTC scheme, see [Rokusawa 88].

5 Conclusions and Future Works

This paper discussed issues of implementing KL1 on a loosely-coupled multiprocessor, and described the way we have solved them in our implementation on the Multi-PSI/V2. The parallel operating system, *PIMOS* [Chikayama 88], and several application programs are now running on the Multi-PSI/V2. The performance on a single PE is 150 K reductions/sec for KL1 append. The system performance with large programs will be measured in the near future. We also have plans to evaluate the effectiveness of the various ideas for optimization, such as white export/import table, global structure management, and incremental intra- and inter-PE garbage collection by MRB and WEC.

We want to extend this research to *load balancing*, one of the major topics in parallel processing. Currently, we are interested in: (1) defining the load, taking account of the priorities as well as the number of executable goals in a PE, and (2) coordinating the load locally by using only local information. The P^3 (processing power plane) scheme [Takeda 88] is a hopeful candidate of the load balancing mechanism on a scalable multiprocessor system like the Multi-PSI.

Acknowledgments

We would like to thank the ICOT Director, Dr. K. Fuchi, and the chief of the fourth research laboratory, Dr. S. Uchida, for giving us the opportunity to pursue this research. We would also like to thank the researchers of ICOT and the cooperating companies, who have worked with us in designing and implementing the KL1 system on the Multi-PSI/V2.

References

[Chikayama 87] T. Chikayama and Y. Kimura. Multiple reference management in Flat GHC. In *Proceedings of the Fourth International Conference on Logic Programming*, 1987.

[Chikayama 88] T. Chikayama, H. Sato and T. Miyazaki. Overview of the Parallel Inference Machine Operating System (PIMOS). In *Proceedings of the International Conference on Fifth Generation Computer Systems*, ICOT, Tokyo, 1988.

[Goto 88] A. Goto, M. Sato, K. Nakajima, K. Taki, A. Matsumoto. Overview of the Parallel Inference Machine Architecture (PIM). In *Proceedings of the International Conference on Fifth Generation Computer Systems*, ICOT, Tokyo, 1988.

[Ichiyoshi 87] N. Ichiyoshi, T. Miyazaki, and K. Taki. A distributed implementation of Flat GHC on the Multi-PSI. In *Proceedings of the Fourth International Conference on Logic Programming*, 1987.

[Ichiyoshi 88] N. Ichiyoshi, K. Rokusawa, K. Nakajima and Y. Inamura. A New External Reference Management and Distributed Unification for KL1. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, ICOT, Tokyo, 1988.

[Kimura 87] Y. Kimura and T. Chikayama. An Abstract KL1 Machine and Its Instruction Set. In *Proceedings of 1987 Symposium on Logic Programming*, Sep. 1987.

[Lieberman 83] H. Lieberman and C. Hewitt. A Real-Time Garbage Collector Based on the Lifetimes of Objects. Commun. ACM, 26(6): 1983.

[Nakajima 88] K. Nakajima. Piling GC — Efficient Garbage Collection for AI Languages. In *Proceeding of the IFIP WG 10.3 Working Conference on Parallel Processing*, 1988.

[Nakashima 87] H. Nakashima and K. Nakajima. Hardware architecture of the sequential inference machine : PSI-II. In *Proceedings of 1987 Symposium on Logic Programming*, Sep. 1987.

[Rokusawa 88] K. Rokusawa, N. Ichiyoshi, T. Chikayama, and H. Nakashima. An efficient termination detection and abortion algorithm for distributed processing systems. In *Proceedings of the 1988 International Conference on Parallel Processing*, Vol. I, 1988.

[Takeda 88] Y. Takeda, H. Nakashima, K. Masuda, T. Chikayama and K. Taki. A Load Balancing Mechanism for Large Scale Multiprocessor Systems and its Implementation. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, ICOT, Tokyo, 1988.

[Taki 88] K. Taki. The parallel software research and development tool: Multi-PSI system. Programming of Future Generation Computers, Elsevier Science Publishers B.V. (North-Holland), 1988.

[Ueda 86] K. Ueda. Guarded Horn Clauses: A Parallel Logic Programming Language with the Concept of a Guard. Technical Report TR-208, ICOT, 1986.

[Watson 87] P. Watson and I. Watson. An efficient garbage collection scheme for parallel computer architectures. In *Proceedings of Parallel Architectures and Languages Europe*, June 1987.

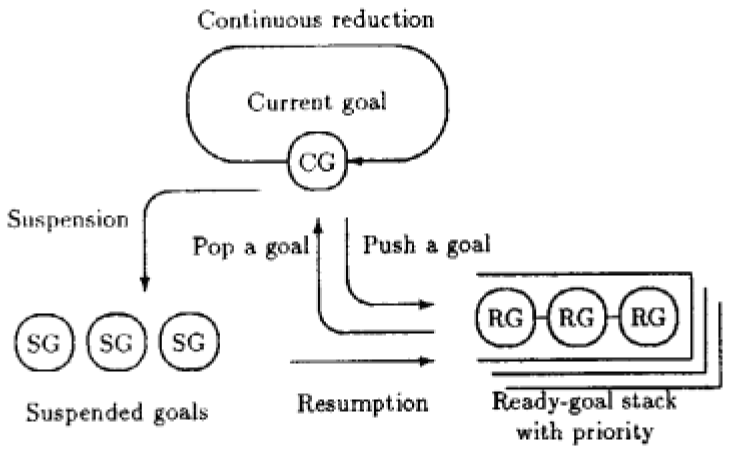

Figure 1: Goal State Transition

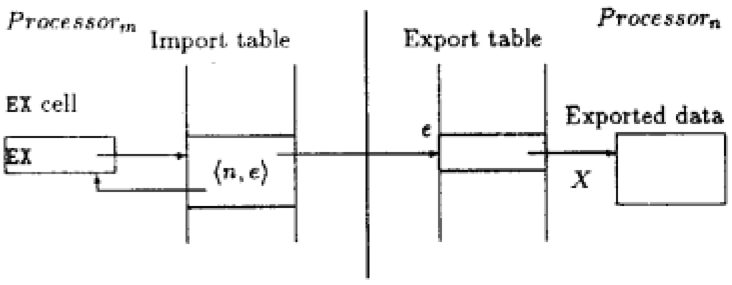

Figure 2: Export Table and Import Table

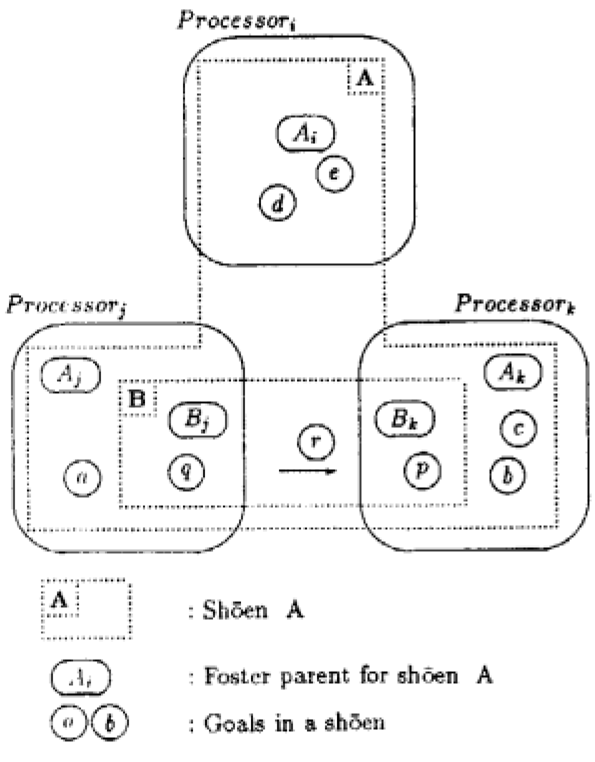

Figure 3: Shōen and Foster Parents