Proc. of FGCS '94, ICOT, Tokyo, December 1994

KLIC: A Portable Implementation of KL1

Tetsuro Fujise, Takashi Chikayama
Kazuaki Rokusawa, Akihiko Nakase

Institute for New Generation Computer Technology
4-28-21F, Mita 1-chome, Minato-ku, Tokyo 108, Japan
`{fujise, chikayama, rokusawa, nakase}@icot.or.jp`

Abstract

This paper describes an overview of the implementation of KLIC. KLIC is a portable implementation of a concurrent logic programming language KL1. The sequential core of KLIC shows reasonable efficiency in both time and space aspects. Two kinds of parallel implementation on a distributed memory and a shared memory, are designed with the policy to retain the efficiency of the sequential core. To realize this, the parallel processing portion of the implementations is built on *generic objects*, which provide a framework for flexible extensions without even slightly changing the core implementation.

1 Introduction

KLIC is a portable implementation of a concurrent logic programming language KL1, based on FGHC[19]. KL1 has been proved to be a practical tool for parallel processing software research through the development of the PIMOS operating system[4] and various application systems on parallel inference machines Multi-PSI[16] and PIM[11] in the Japanese Fifth Generation Computer Systems project.

Although KL1 runs efficiently on those parallel inference machines, such implementations have a serious disadvantage in that they are not portable and cannot be used on commercial machines. To solve this problem, a scheme that allows a very portable implementation of compiling into C was investigated. The language C was chosen as the intermediate language, because there are widely available compilers with excellent optimization quality for the language nowadays. Various demerits exist in using C as an intermediate language, we have designed an implementation scheme which detours them, and built KLIC.

The sequential core of KLIC shows reasonable efficiency in both time and space aspects. It runs about twice as fast as the native code generated by SICStus Prolog[2] for benchmark programs on SparcStation 10 model 30. The code size becomes larger than abstract machine code but is found to remain reasonable.

We designed the parallel implementation with the policy to retain the efficiency of the sequential core of KLIC. To realize this the parallel processing portion of the implementaion is built on *generic objects*.

Generic Objects provide a framework for flexible extensions of the implementation. The framework is object-oriented so that the core implementation does not have to know the representation nor manipulation of generic objects. When some manipulation is required on generic objects, the core asks the objects. Generic objects realized full functionality of the system with quite concise core implementation.

Features of KLIC implemented upon generic objects include built-in data types with foreign language interfaces, as well as multiple parallel implementation.

Two kinds of parallel implementation of KLIC are designed, called a distributed memory implementation and a shared memory implementation. The distributed memory implementation is based on a message passing scheme; the shared memory implementation is based on a shared heap area accessing scheme.

This paper gives an overview of implementation of KLIC and the structure of this paper is as follows. Section 2 describes the efficient technique of the compiling KL1 into C program. Section 3 describes the sequential core of KLIC, generic objects are described in Section 4. In Section 5, two kinds of the parallel implementation of KLIC are described. Section 6 gives a summary followed by concluding remarks.

Note that the KLIC system is described in [6].

2 C as the Intermediate Language

Language C was chosen as an intermediate language, mainly for establishing higher portability. The merits of using C and the difficulties as following faced upon translating KL1 programs to efficient C programs are summarized, and our solution to this problem is given.

2.1 Merits of Compilation into C

There are various merits in using C as the intermediate language to implement KL1, among which important ones are the following:

Portability The implementation can be made quite portable. Porting the system will require only adjusting some switches and recompiling.

Low-level optimization Some C compilers provide very good low-level optimization. By letting the C compiler take care of low-level issues, the language implementation can concentrate on higher-level optimization issues.

Linkage with foreign language programs Linking KL1 programs with programs written in C becomes quite easy.

2.2 Efficiency problems

Although compiling into C has the above-mentioned advantages, it is not easy to realize a reasonable efficiency for languages with an execution model quite different from C, such as KL1. Typical efficiency problems include the following:

Costly function calls The C language and its implementation is designed with fairly large functions in mind. Although function invocation and parameter passing overhead itself may be regarded not very large, dividing programs into functions makes program analysis more difficult, often resulting in less efficient object code. KL1 predicates are usually very small, often as small as one line of C code. Many of them are recursive, making inline expansion impossible. Thus, naive strategies such as compiling each KL1 predicate into one C function may result in quite inefficient code and should be not be used.

Register allocation control Certain global data, such as the free memory top pointer, is accessed very frequently. It would be best to keep such data in dedicated registers during the whole execution. In most C implementations, however, such control of register allocation is not possible.

Provision for interrupts Multiprocessor implementation should process interrupts from other processors. Interrupts may require allocating memory, enqueueing goals or instantiating variables. Data accessed in these operations is also frequently referenced and altered within the processor. Thus, data locking or inhibition of interrupts may be required. These are quite costly under conventional operating systems.

Object code size Compiling logic programming language programs tends to produce large native machine object code. Increased working set sometimes results in performance worse than an abstract machine interpreter. When compiled through C, this code size increase might be amplified. Using runtime subroutines may reduce the code size but may also reduce speed.

2.3 Solutions

Our solutions to the efficiency problems described above are as follows.

One module as one function Compiling the whole program into one function may be the best for reducing function call overhead. However, this will prevent separate compilation, which will be a serious problem with programs of practical size. Our solution is to let the user control the size of compilation. Each "module", which defines a set of closely related predicates is compiled into one C function. As far as predicates within the same module calling one another, control transfer is by goto and arguments can be passed through local variables, which might be allocated on machine registers.

Caching global variables To access to frequently referenced global variables with minimal overhead, such variables are cached in local variables. C compilers may allocate them on registers; even if not, accesses to local variables are less costly than to global variables with most modern processor architectures. Certain care should be taken to synchronize with interrupt handling. Even for synchronized runtime subroutines, passing all the cached variables back and forth is quite costly. Fortunately, we could design a maintenance principle for such variables so that only small numbers of them be passed and returned in most cases.

Efficient synchronization with interrupts Signal handlers are made to set a certain flag, which is examined when convenient for normal processing. This synchronizes interrupt handling with normal processing without expensive interrupt inhibition. This flag check is combined with a mandatory, heap overflow check for garbage collection, and thus made virtually costless.

runtime routines for exceptional cases
Read/write modes of variable occurrences can be decided during compilation much more easily for KL1 than for Prolog. Common cases are handled in-line while exceptional cases are dealt without significantly slowing execution.

3 Implementation of the Sequential Core

This section describes an outline of the experimental implementation of the sequential core, and its empirical evaluation results.

3.1 Experimental Implementation

This section describes the sequential core of our experimental implementation. The parallel implementation is built on this sequential core.

3.1.1 Data Representation

Every KL1 term is represented by a word (Fig. 1). The lowest 2 bits are used for basic data type tags. The rest represents an address or immediate data. Tags have to be removed to access memory referenced by tagged pointers, but this can be done easily by giving small offsets to register-based memory accesses. The tag bits distinguish

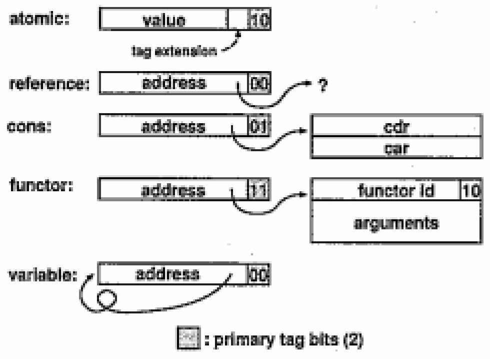

Figure 1: Basic Data Type

the following four basic types.

variable reference: The data part has the address of a variable cell. Uninstantiated variables are self-referential pointers as in WAM[20]. Variables with goals awaiting for their instantiation will be described later.

atomic data: For atomic data, 2 more lowest bits of the data part are used as a tag extension, which distinguish symbolic atom, integer, etc. The remaining bits represent the value.

cons: The data part has the address of a two-word memory block for the cell.

functor: The data part has the address of a memory block of a functor structure. The lowest bit of the first word of the block differentiates functor structures from generic data objects (see Sect. 4 for representation of generic objects.) For functor structures, the word contains a functor identifier. The rest is used for arguments.

The KL1 implementations on PIMs have a single reference count in points which allowed incremental garbage collection and destructure updates [3]. We decided not to adopt this scheme because no more tag bits can be efficiently manipulated on widely available hardware.

Atom and functor identifiers are determined at compilation time. The compiler driver keeps a database of atoms and functors used in the programs of a project and generate C header files with their definitions as macros. Separate C compilation is possible because definitions are only added and never deleted.

3.1.2 Goal Management

Ready goals are represented as goal records in the heap area (Fig. 2), and put into a LIFO goal stack, which is a linked list of goal records. In each reduction step, the topmost goal is poped up from the stack, reduced according to the program, and the resultant child goals, if any, are pushed back to the stack, except for the leftmost one, which is executed immediately. The second field of

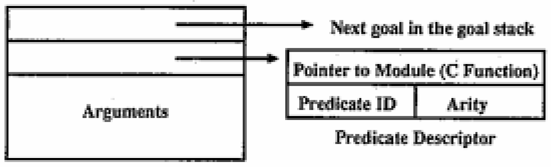

Figure 2: Goal Record

a goal record has a pointer to a predicate descriptor that contains a pointer to the code corresponding to the module (a C function) it belongs to, the predicate ID within the module (an integer), and the number of arguments. The rest contains arguments of the goal.

As KL1 allows programs to specify priority for each goal, either statically or dynamically depending on the partial results of computation, there can be multiple goal stacks corresponding to priority levels. The priority mechanism has been found to be very useful through application software research on PIM systems in describing various algorithms with speculative computation, and now is an indispensable feature of KL1. The last entry in the goal stack is a sentinel goal which schedules the goal stack with the next highest priority. The lowest priority goal stack uses the sentinel goal for termination detection.

Goal records are allocated on the heap with other data. This allows allocation of variable cells within goal records, which reduces the working set size and dereference cost. On the other hand, goal records are not reused incrementally, increasing garbage collection overhead.

A variable with goals waiting for its initiation is also represented by a pointer with the variable reference tag (Fig. 3). The pointer part references a *suspension record*

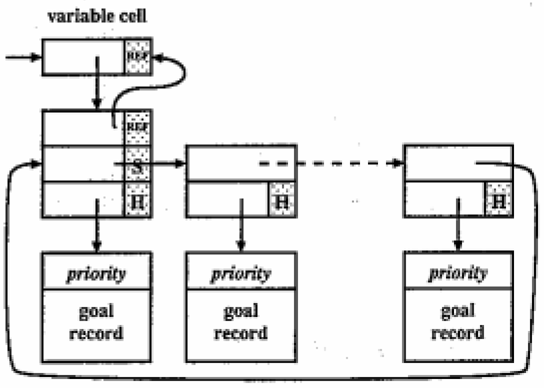

Figure 3: Suspended Goals

that starts a list of *hook records*. The first word of the suspension record contains a pointer back to the variable making a two-word loop[1]. The rest of the suspension record contains a *hook record* that records a goal waiting for the variable's value. Hook records, including the first one in in the suspension record, are linked by their first words to form a loop. This loop allows efficient unification of two variables each with many suspended goals.

The first field of goal records for suspended goals is used to keep their priority values, rather of linkage as for the goal stack. The lowest bit of the field is set for suspended goals. A goal awaiting for instantiation of one of a set of variables is referenced from multiple hook records. When the goal is activated by instantiation of one of the variables, it is sent back to the goal stack. As the first word of the goal record is used for chaining the goals in the stack, the lowest bit of the word is naturally cleared. On instantiation of other variables in the set, goals put back already can be recognized by this bit.

The hook records are deleted and reclaimed only by the garbage collector. Recognizing them as invalid at variable instantiation is less costly than incremental deletion, but may burden the garbage collector further.

3.1.3 Heap Area Management

The heap area is organized as shown in Fig. 4. Memory

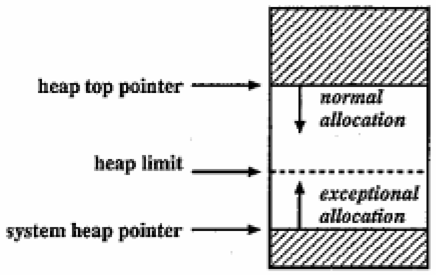

Figure 4: Organization of Heap Area

allocation is usually made from the heap pointer downwards. At the end of each reduction, the heap pointer is compared with the heap limit. If it points below the limit, the garbage collector will be invoked. The garbage collector uses a copying scheme with modifications to cope with directly referenced structure elements.

Allocations in certain runtime subroutines, such as those implementing some of the built-in predicates, are made with the system heap pointer which points to the bottom of the free heap area upwards. C allows only a one-word value to be returned efficiently as the result of functions. Returning multiple values has to be done through global variables or by passing variable addresses, which are significantly slower. Thus, for subroutines that allocate memory only in exceptional cases, it is more efficient to avoid passing and returning the heap top pointer.

When allocations are made at the bottom, not only the system heap pointer but the heap limit pointer as well is moved upwards by the same amount, always keeping a certain gap between the system heap pointer and the heap limit. The size of the gap is kept greater than the maximum memory allocation in one goal reduction made in the compiled code. Runtime subroutines check this dynamically at memory allocation.

When the garbage collector copies data from the old space to the new space, it makes no distinction between the system heap and the normal heap; active data in the system heap is copied to the top of the new space.

3.1.4 Interrupt Handling

On multiprocessor implementation, normal goal reductions may be interrupted by other workers. On interrupts (e.g. Unix *signal*), the signal handler will be used only for notification of interrupts: it will set a flag in

[1]This idea is originated Hiroshi Nakashima.

a global variable and, at same time, modify the heap limit[2], so that the check at the end of the current reduction will find it. The garbage collector called through the check can examine the flag and determine the need for a garbage collection and/or interrupt handling, The interrupt handler will be called as normal processing. Thus, no extra interrupt checking is required during normal processing, except that the heap limit has to be stored in memory.

The same mechanism is used to notify and resume newly available goals with higher priority, by instantiating their hooked variables. It is also used by the stepping tracer.

3.2 Performance

We compared the sequential core of KLIC with a representative Prolog system, **SICStus Prolog** version 2.1 patchlevel 8, and two experimental logic programming systems, **Aquarius Prolog** version 1.0[10] and **JC** version 2.0[9], a Janus[18] to C compiler. For SICStus, both native code(fastcode) and abstract machine code(compactcode) were measured.

The execution time for the benchmark programs are shown in Table 1.

Table 1: Comparison of execution time

program	K	Sf	Sc	A	J
nrev	2,430	4,789	10,440	1,610	2,740
qsort	3,300	7,320	14,980	1,920	3,240
times10	2,420	5,659	12,569	3,090	3,240
divide10	2,900	5,890	14,390	3,240	4,370
log10	1,060	3,319	6,299	1,830	1,670
ops8	1,620	4,460	9,270	2,560	2,330
primes	1,600	2,869	5,349	2,780	2,170
tak	3,620	6,789	14,820	1,460	1,590
mean	2,205	4,908	10,343	2,219	2,532

K: KLIC; Sf: SICStus `fastcode`; Sc: SICStus `compactcode`; A: Aquarius; J: JC. Timing data are in milliseconds. The "mean" shows geometric means.

When compared to SICStus, the sequential core runs about twice as fast as native code and 3 to 6 times as fast as abstract machine code. Aquarious and JC show similar speed with the sequential core. Both considerably outperform the sequential core with tak. One reason may be in the implementation scheme of the goal stack. As the sequential core allocates goal records in heap, garbage collection overhead becomes very high for programs that make many non-tail recursions. Actually, simply setting a larger heap size speeds up the execution to 3,170 ms.

Aquarious is faster for programs with much integer arithmetics. This may be at least partly due to the global analysis that can delete redundant dereference and type checks. This suggests the direction of further improvements of the sequential core.

Further details can be found in [5].

[2]Real heap limit is saved in a separate variable.

3.3 Portability

The sequential core has been ported to many systems, Major systems which KLIC has ported to are shown in Table 2.

Table 2: Portability of the sequential core

Model	OS	Manufacturer
SparcStation	SunOS, Solaris	Sun Micro.
DEC 3000	OSF/1	DEC
RS 6000	AIX	IBM
HP 9000	HP-UX	HP
IRIS	IRIX	SGI
EWS 4800	EWS-UX/V	NEC
Luna 88k	Mach	OMRON
M-880	HI-OSF/1-MJ	Hitachi
S-3800	HI-OSF/1-MJ	Hitachi
IBM AT clone	MS-DOS, OS/2, Linux	IBM etc.

4 Generic Objects

The generic objects feature allows easy modification and extension of the system without changing the implementation core. This section describes why and how such a feature is incorporated and how it is used.

We borrowed the basic idea of generic objects from AGENTS[14], modified and extended it for KLIC. In AGENTS, the distinction of three categories of generic objects described below is not made; there are no generator objects and the functionality of consumer objects is provided by *ports* which are data objects, as *ports* are names of streams rather than streams themselves.

4.1 Objectives

With the KL1 implementation on PIMs, we experienced severe difficulties in trying out different parallel execution schemes, as the schemes were too integrated into the system core. This gave rise to the principle that system extensibility and modifiability should be put above bare efficiency. On the other hand, as the system is for stock hardware, only a limited number of tag bits can be handled efficiently. We thus needed some other ways to distinguish various built-in data types.

Generic objects were introduced to KLIC to achieve these two objectives simultaneously. Some of the standard built-in data types and non-local data references for parallel implementation were implemented as generic objects. The core runtime system and compiled code only know that there are data types generically called "generic objects". Generic objects of all classes have the same interface; new object classes can be freely added without changing the system core.

Bodies of data objects can be allocated to the heap area[3], and they can contain both KL1 data and C data there. This means that structured data in C, that is, tag-less data is allowed to be allocated on the heap area by the framework of data objects. The garbage collector calls the *gc* method of their own. This method can be defined easily. This idea originated in AGENTS.

4.2 Three Categories of Generic Objects

KLIC has three categories of generic objects: data objects, consumer objects and generator objects.

4.2.1 Data Objects

Data objects are treated exactly as data of extended data types. As described in Sect. 3, only four basic data types are supported in the KLIC system core. Data objects provide a framework to add and maintain more data types in the whole KLIC system.

Data object are accessed via built-in predicates and generic method calls. Method calls have the interface `generic:Method(Obj, Args, ...)`. They are logically immutable objects without time-dependent states. Time-independence only means that they always look the same to the KL1 program; physical representations may be modified. For example, multiversion arrays are data objects, but their actual representations are mutated on update. Multiversion vectors and character strings are implemented as data objects, and various other object, such as bignum are also planned. New data objects can be defined easily in C.

4.2.2 Consumer Objects

Consumer objects behave exactly like suspended goals which associate with logic variables. They are activated by variable instantiation i.e. consumer objects are *data-driven* objects. When a variable associated with a consumer object is instanciated, the unifier subroutine recognizes that, and calls the *unify* method of the object. Consumer objects are also mutable, time-dependent objects. A consumer object is allowed to re-associate with another logic variable again after it has been activated by the instantiation of its pre-associated variable. For instance, suppose a logic variable `OBJ` is associated with some consumer object. When the object is activated by variable instantiation, `Obj = [Message|NewObj]`, the unifier calls the *unify* method of the object. The object performs the task specified by `Message` or creates a new goal to do the job, and re-associates itself with `NewObj` again (Fig. 4.2.2). Thus, if the task is simple enough, the overhead of goal suspension and resumption can be avoided. Note that this is not disturbing the pure semantics of the language; consumer objects behave exactly like a KL1 process waiting for instantiation of a variable. If the task specified by `Message` is a method call, the *unify* method of the objects is treated as a *generic* method just like a *body generic* method (see below) for data objects. Since the association of an object with a new variable implies the change of the process to a new state, objects can rewrite their own bodies every method call. Therefore, in the case that a C program with side-effects is to be included in a KLIC program, the C program should be implemented as a consumer object. Stream mergers and file I/O interface objects are implemented as consumer objects. For example, the *unify* method of the I/O interface object for message `put(C)` will simply write the character `C` to a file. In other words, `putc(C)` is one of the *generic* methods.

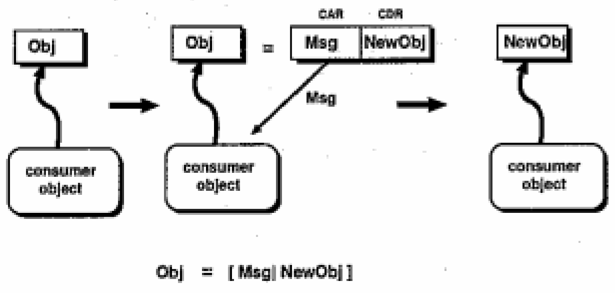

Figure 5: Re-association of Consumer Object

4.2.3 Generator Objects

Generator objects associate with logic variables. These logical variables look like a reference to a cell which has been instantiated or may be instantiated with some value. Unlike consumer objects, generator objects are activated by dereference operations, that is, a generator object is a demand-driven object. Generator objects define a *generate* method. When the goal suspension handling routine finds that a suspended variable is associated with a generator object, the *generate* method of the object will be invoked. The object may immediately generate some value and instantiate the associated variable, or it may spawn a goal that will eventually do it. Generator objects are also mutable, with time-dependent

[3]Bodies of constants of data object described below may be allocated outsides of the heap area

states, and allowed to associate with another logic variable like consumer objects. Generator objects can also implement lazy computation. Generator objects are activated only when the values they generate are required by some other goal.

4.3 Representation of Generic Objects

All kinds of generic objects have a pointer to its method table (Fig. 6). The rest of the object is accessed only through their methods; the core implementation does not touch the object itself. Common operations, such as copying for garbage collection, are standard methods shared by all object classes. Non-standard methods are logically called through the *guard* and *body generic* method. Actually all non-standard methods of some class may be implemented only on a function which performs its *generic* method.

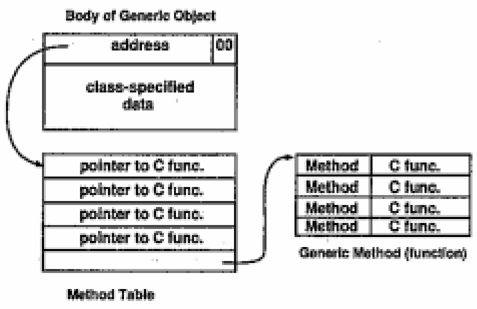

Figure 6: Body of Generic Object

Data objects are represented by pointers to bodies of data objects with the functor tag (Fig. 7). Data objects are distinguished from functors by the lowest bit in the first word of the block.

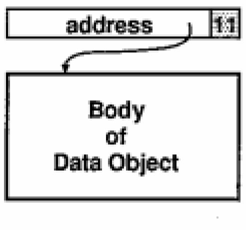

Figure 7: Representation of Data Object

Consumer objects are referenced by the second word of hook records (Fig. 8), as for suspended goals. Consumer objects are distinguished from suspended goals by the lowest bit of this second word.

Generator objects are referenced by the second word of suspension records (Fig. 9). Generator objects are distinguished from the hook records list by the lowest bit of this second word.

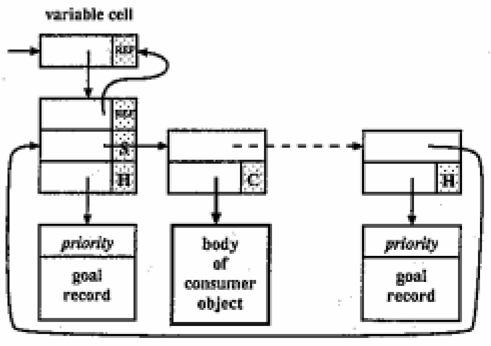

Figure 8: Representation of Consumer Object

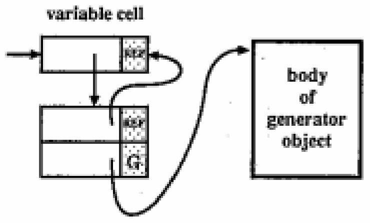

Figure 9: Representation of Generator Object

4.4 Standard Methods

Generic objects of all classes have the same standard methods. The following describes major ones.

body unify The body unify method is invoked by the body unifier.

guard unify Only data objects have a guard unify method. This method is invoked by the guard general unifier.

generate Only generator objects have a generate method. This method is invoked when the goal suspension handling routine finds that a suspended variable is associated with a generator object.

gc The gc method is invoked by the garbage collector. This method copies a generic object to a new heap area when the garbage collector has found the object.

body generic Only data objects have a body generic method. This method is a meta-method which calls user-defined methods appearing in body parts of clauses.

guard generic Only data objects have a guard generic method. This method is a meta-method which calls user-defined methods appeared in guard parts of clauses.

encode Consumer objects do not have this method. It is used for parallel implementation when objects are translated to packets for communication with other processors. Consumer objects do not have this method because making their copies will change the semantics.

5 A Parallel Implementation of KLIC

This section describes the parallel implementation of KLIC.

Two different implementation on, distributed memory and shared memory, were designed depending on the memory architecture to use. Both were designed with a common policy that no modification should be made to the sequential core in order to retain its efficiency. Under this policy, most of the extensions required for the parallel implementations were realized using generic objects.

This section gives the use of generic objects and the scheme of interrupt handling for parallel implementations are described. Overviews of the two types of parallel implementation are described. Brief description of a performance tuning tool is also given.

5.1 Common Mechanisms

5.1.1 Generic Objects for Parallel Implementation

In parallel implementations, non-local references are represented as generator objects. On suspension of a goal, the *generate* method of a generator object associated with a suspension reason variable is called. It may send a value fetching message to a remote processor or read data from shared memory. If it does not get a value immediately, the generator object mutates to a consumer object or another generator object and waits for the associated variable to be instantiated or dereferenced. In this way, remote references are easily implemented with generic objects.

5.1.2 An Interrupt Handling for Parallel Implementation

On parallel implementations, a worker that sends goals or data (the "sender") notifies their arrival to another worker (the "receiver") which should receive them. To realize that, the sender rewrites the heap limit of the receiver to zero, i.e., the receiver is reported the arrival of goals or data as a request of a garbage collection. Thus, a KLIC interrupt handling routine may receive messages or goals from other processors between reductions in a synchronous manner. Unix *Signals* or active messages are often used for rewriting the heap limit.

5.2 The Distributed Memory Implementation

5.2.1 Basic Design

In a distributed memory implementation, goal and data migrations among workers are performed with message exchanges. The scheme of this implementation inherits its basic design from the distributed implementation on PIMs[4] (e.g. two level addresses, local and global, are used with address translation table called "*export table*").

In the distributed memory implementation, portability was put above efficiency.

Most part of the communication processing is implemented with generic objects as methods of generic one called only on demands, it does not degrade the performance of the sequential core. Machine dependent communication procedures are strictly separated from the management of the high-level communication such as distributed unifications, giving high portability.

5.2.2 Configuration

To execute a KLIC program in parallel, a communication library is attached to the sequential core. The core runs on each processing node as a worker.

Creation of workers and communication with other worker are done via a communication library. This library is divided into two levels for portability; "KL1 communication library" and a "machine-dependent communication library".

The KL1 communication library is directly invoked by the sequential core, and performs KL1-level management and control, including goal migration, distributed unification, distributed garbage collection and distributed termination detection.

The machine-dependent communication library is invoked by the KL1 communication library. It initiates workers and passes actual messages among them.

The configuration of this implementation is shown in Fig. 10. It can be ported by replacing the machine-dependent communication library for the target machine architecture and communication media.

[4]Further details can be found in [17].

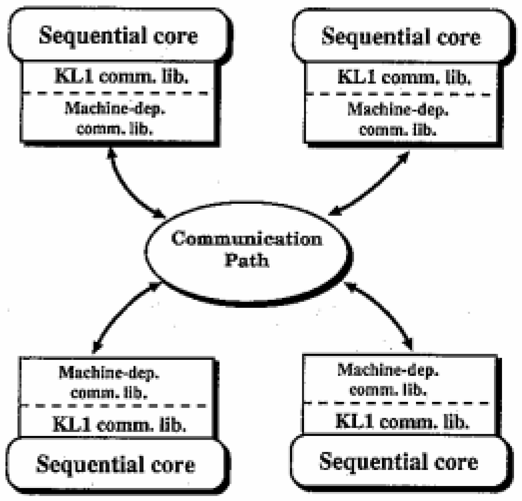

Figure 10: Configuration

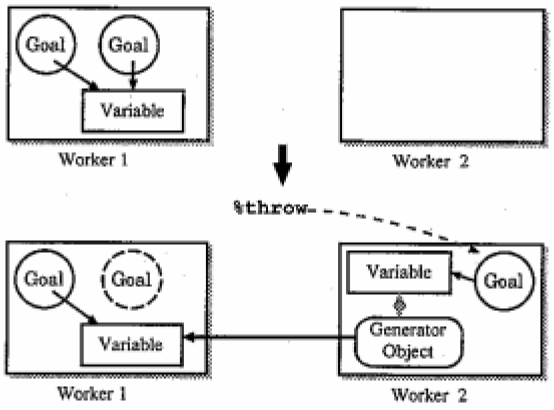

Figure 11: Goal Migration with Variable

5.2.3 The KL1 Communication Library

The KL1 communication library manages goal migration, distributed unification, distributed garbage collection, and distributed termination detection. The basic scheme of the distributed implementation is based on those of Multi-PSI and PIMs, and optimized to fit for KLIC.

Goal migrations should be explicitly specified by a goal distribution pragma, "`@node(X)`". For instance,

```
a(X):- true | b(X)@node(1).
```

means to migrate a goal `b` to worker 1.

Goal and data migration is performed as follows.

- Migration of Goal
  - When a goal is migrated to another worker, the goal is encoded in the message named "*%throw*" and sent to the other worker.
  - If this goal has arguments, this message has to include the encoded data of these arguments.
  - When the other worker receives this message, the *%throw* is decoded to a new goal on this worker (Fig. 11).
- Migration of arguments of a goal
  - Arguments of a goal are encoded when the goal migrates to the other worker.
  - When the argument is an atomic or structured constant data, it is encoded as it is.
  - When the argument is a structured data (except constants) or a variable, this is decoded to a variable associated with a remote reference called an *external reference* to the structured data or the variable on the sender worker. This means arrival of a *%throw* may make remote references from the receiver worker to the sender worker. As described above, *external references* are represented as generator objects.
- Unification
  - When unification of concrete value with a variable associated with an *external reference* is made on the receiver worker, the generator object, which is an *external reference*, sends back the message "*%unify*" to the sender worker and where it is done.
  - The *%unify* message includes a unified data. This data is an atomic or a structured data whose elements are encoded to *external references* to the receiver worker (Fig. 12).
  - If this element has already been an *external reference*, it is sent as it is.
  - The variable on the sender is unified with the value decoded this message.

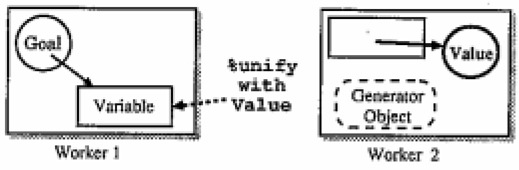

Figure 12: Unification

- Dereference

- When a dereference to a variable associated with an *external reference* is made on the receiver worker, the *external reference* sends back the message "*%read*" to the sender and mutates itself to a consumer object which waits for a unification message described below from the sender or a unification on the receiver.
- If the variable on the sender is not yet instantiated, this variable is associated with a consumer object for sending a reply value to the consumer object on the receiver after the variable is instantiated to some value (Fig. 13).
- If the variable on the sender has already been instantiated to a value, the message "*%answer*" including the value is sent and a message named a "*%release*" is sent back to the sender to removes the reference (Fig. 14).

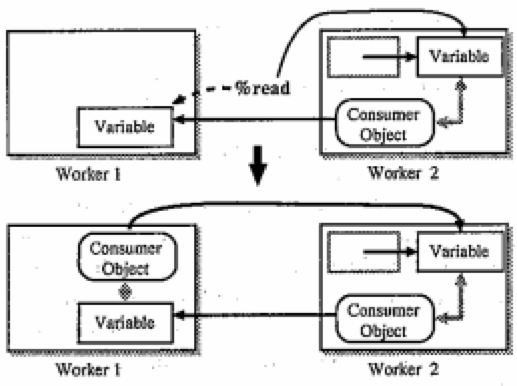

Figure 13: Dereference (1)

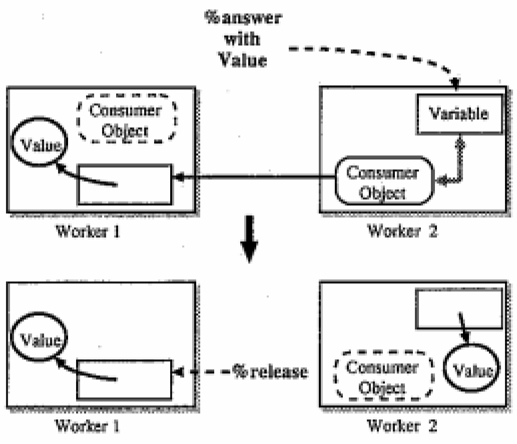

Figure 14: Dereference (2)

5.2.4 The Machine-Dependent Communication Library

The machine-dependent communication libraries have two functions: initiation of workers and actual message communication among them.

Initiation: In initiation, workers are spawned and communication paths are established mutual. Special workers for console I/O and termination detection are also invoked. The console worker is used for interaction for tracing.

Message passing: This implementation performs message passing simply. It is not necessary to send sophisticated messages passing primitives, such as for broadcasting or message tagging.

Message sending and receiving procedures are as follows: initiating a send buffer, packing data into buffer and sending the message; initiating a receive buffer, receiving a message into the receive and unpacking data from the buffer.

The following summarizes typical communication paths.

General purpose message-passing library: Several message-passing libraries have been developed recently, such as PVM[8] and MPI[7]. These libraries have been ported to many kinds of parallel and distributed machines. KLIC on these libraries are portable.

Shared memory: On shared-memory machines, shared memory segments can be used for high performance message paths.

Machine-specific communication path: Most parallel machines have their own message passing libraries for maximizing parallel execution performance.

We adopted PVM as the basis of a machine-independent portable parallel version of KLIC.

5.2.5 Performance

We show parallel speed-up of two kinds of implementation where communication paths are PVM and shared memory in Table 3. Programs for this comparison are n-queen (13), the distinct order factorization[15] and genetic sequence alignment analysis[13]. All measurements are on SparcCenter 2000, with 40MHz Super-Sparc processor with 1 M Byte of external cache and 16 Kbyte of internal cache. The result of distinct order factorization shows the PVM version got similar speed up as the shared memory version. The alignment analysis program got lower performance on both of them, but the result of PVM version got worse than that of the shared memory version.

Table 3: Execution times
(a) PVM

13-queen

Workers	1	2	4	8	12	16
Wall Clock	176	88	47	28	20	19
Speed Up	1.0	2.0	3.7	6.3	8.8	9.3

Distinct Order Factorization

Workers	1+1	2+1	4+1	8+1	12+1	16+1
Wall Clock	149	75	38	19	13	10
Speed Up	1.0	2.0	3.9	7.8	11.5	14.9

Alignment Analysis

Workers	1+1	2+1	4+1	8+1	12+1	16+1
Wall Clock	116	106	66	57	48	52
Speed Up	1.0	1.1	1.8	2.0	2.4	2.2

(b) Shared Memory (NORMA)

13-queen

Workers	1	2	4	8	12	16
Wall Clock	165	85	45	24	16	14
Speed Up	1.0	1.9	3.7	6.9	10.3	11.8

Distinct Order Factorization

Workers	1+1	2+1	4+1	8+1	12+1	16+1
Wall Clock	145	73	37	18	12	10
Speed Up	1.0	2.0	3.9	8.0	12.0	14.5

Alignment Analysis

Workers	1+1	2+1	4+1	8+1	12+1	16+1
Wall Clock	64	54	39	23	19	16
Speed Up	1.0	1.2	1.6	2.8	3.4	4.0

Timing data are in seconds.

5.2.6 Portability

This version of KLIC has been ported or is being ported on several systems in Table 4.

5.3 The Shared-memory Implementation

5.3.1 Basic Design

Shared-memory implementations of concurrent logic programming languages developed so far have taken either a UMA model (all memory is shared) or a NORMA model (no memory is shared). The PIM implementation adopted a UMA model. In that implementation, many lock operations degrade the performance of each worker.

Our implementation adopted such memory model that each worker owns its local heap area in addition to a shared heap area, in order to decrease the number of lock operations even in garbage collections. Since basic data types of KLIC has no reference count, the garbage collection requests may occur more frequently than PIM's, but its is more significant to decrease the number of lock operations.

Further details can be found in [12].

Table 4: Portability of the distributed implementation

Model	OS	Comm. Path	Manufactor
SparcStation	SunOS	PVM	Sun Micro.
SparcStation	Solaris	PVM	Sun Micro.
SparcCenter	Solaris	PVM	Sun Micro.
SparcCenter	Solaris	Shared Memory	Sun Micro.
DEC 7000	OSF/1	PVM	DEC
RS 6000	AIX	PVM	IBM
Paragon	OSF/1	PVM	Intel
CM5	SunOS	CMAML	TMC
AP 1000[†]	[‡]	[‡]	Fujitsu
Cenju-3[†]	Mach	MPI	NEC
SR 2001[†]	HI-UX	Express	Hitachi

[†] porting
[‡]AP 1000 has a private OS and a communication path.

5.3.2 Local and Shared Heap Areas

Memory is divided into local heap areas and a shared heap area (Fig. 15). Each worker has a local area which it can read and write without coordinating with other workers. A worker can also read and write to and from the shared heap area, but it must coordinate with other workers to do so since some of them may be accessing the data simultaneously.

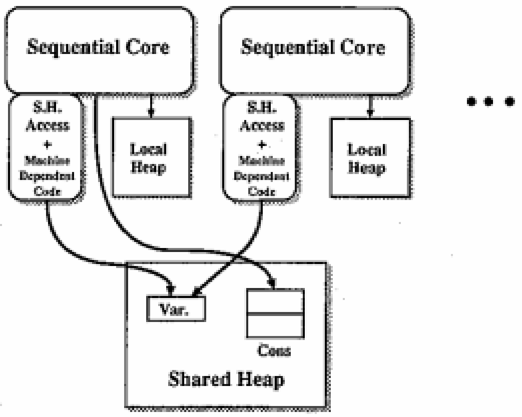

Figure 15: Configuration

A local heap area consists of the maintenance data for dynamic allocation of local goals and KL1 data. The maintenance data includes the local heap top pointer, the root of the local goal stack, etc. The shared heap area consists of the maintenance data for worker interaction and the shared heap for dynamic allocation of shared goals and KL1 data. The shared maintenance data includes external goal pools (one for each worker), interrupt flags (which of course include a local heap limit) for each worker, etc. A worker's external goal pool holds the goals given to the worker by other workers for load distribution. Goals are inserted by workers other than the owner worker and removed by the owner worker for execution. Interrupts flags are allowed to be set by other workers.

Each local heap area is divided into two spaces for copying garbage collection. No lock operations are required to collect garbages in local heap areas.

The shared heap area is divided into three spaces which are used in a circular manner. At any instant, there is one old space, one new space, and one unused space. Active data can be in the old or new space, or extend across both of them, but there must be no pointers from the new space to the old space.

The unused space is used for the asynchronous garbage collection as described below.

5.3.3 Goal Distribution

When a goal moves, the data directly and indirectly referenced by it is copied to the shared heap area.

For example, in the following code,

```
p(X) :- ... | ...,
q(X,foo(Y))@node(4), ...
```

the goal `q(X,foo(Y))@node(4)` is allocated in the shared heap area. It involves copying the data structure pointed to by the variable X from the local to the shared heap area, and allocation of the compound term `foo(Y)` in the shared heap area. `Y` is a new shared variable. If `X` is an uninstantiated variable, a shared variable is allocated in the shared heap area and the local variable will be converted to a reference to the shared variable. After the goal has been copied to the shared heap area, the pointer to it is inserted into the external goal pool of the target worker, the interrupt flag for the target worker is set so that the new goal can be scheduled right after the current reduction. Otherwise, the target worker is not interrupted. A worker checks its external goal pool whenever it moves to a lower priority level.

Note that these shared variables are implemented with generator and consumer objects.

5.3.4 Asynchronous Garbage Collection of Shared Heap Area

In our experience, synchronous garbage collection of the shared heap area (i.e.when all workers stops normal execution and garbage collection is started at the same time) makes the shared bus a bottleneck, there is little locality in garbage collection.

To remove this bottleneck, an asynchronous garbage collection scheme is designed for the shared-memory garbage collection while others are executing normal code.

The asynchronous garbage collection requires three spaces: old space, new space, and unused space. Each worker has one "current" space (either the old or the new) for allocation of shared data. Each space has a list of the workers currently using the space.

When some worker detects memory shortage in the old space, it copies data it references to the new space, while other workers may be still executing normal KL1 code. The new space becomes the current space for the worker.

Suppose that the new space has run out. If the old space is still used by some workers, they are interrupted to be forced to do garbage collection. They copy the active data in the old space to the unused space. At this point, the spaces are rotated: the old space becomes the unused space, the new space becomes the old space, and the unused space becomes the new space.

5.3.5 Performance

We compared a shared memory implementation with a sequential core of KLIC using the 12-queen program and evaluated the performance this implementation. The results are shown in Table 5. All measurements are on SparcCenter 2000, with 40MHz Super-Sparc processor with 1 M Byte of external cache and 16 Kbyte of internal cache.

Table 5: Comparison of execution times

The type of KLIC	Wall Clock
Sequential version	22,390
Shared Memory version (1 worker)	22,400
Shared Memory version (12 workers)	2,340

Timing data are in milliseconds.

The result shows this implementation does not degrade the performance of the sequential core and 9.5 fold speedup is achieved with 12 workers.

5.3.6 Portability

This implementation is fairly portable and has been ported to several machines (Table 6).

Machine dependent routines are as follows:

shared memory segments: This implementation uses the *mmap* function for getting shared memory segments.

Table 6: Portability of the shared implementation

Model	OS	Manufacturer
SparcCenter	Solaris	Sun Micro.
DEC 7000	OSF/1	DEC

lock: The lock operation has to be implemented separately for each machine.

store ordering: For example, the store barrier operation is needed for the implementation on SparcCenter and DEC 7000.

5.4 A Tuning Tool

We ported a runtime monitor, one of the PIMOS realtime visual tuning tools[1]. This tool displays that each column represents load averages of all workers in various colors (a gray scale is used in this paper). As shown in Fig 5.4, this tool displays a load average of a worker at specified interval on an element of each column. Each column appears at the interval, is displayed from left to right on the screen.

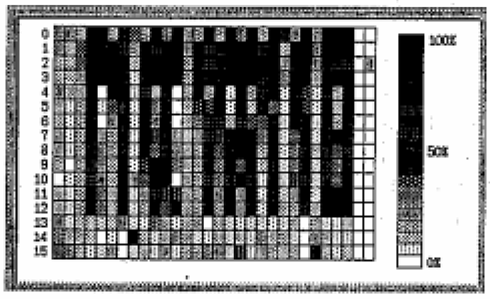

Figure 16: Runtime Monitor

In PIM, the load average of each worker is measured by the firmware. We adopted the following probabilistic method similar to *"prof"* command of Unix to avoid recompiling KL1 programs.

1. Each worker sends signals to itself by an interval timer (an interval is about 10 ms, which does not affect the performance much). These signals are counted, and another counter is incremented if it is idle when the worker receives this signal (this counter is called the "idle counter" in this paper). An active message from the profiling worker to a target worker can be also used, too.

2. The profiling worker collects values of all counters from all workers on some interval (which we usually use about two seconds). The profiling worker sends ratios of idle counts and signal counts to the display worker. For example, suppose a worker has 100 signal counts and 50 idle counts, this means 50% load.

This probabilistic method can be applied to collect other profile information.

6 Concluding Remarks

The preceding sections gave an overview of KLIC. These parallel implementations are based on the non strict argument passing feature of the programming language KL1 and can be made portable.

Various efforts are on-going to make the system more useful in parallel software research, including the following:

- Implementation of more language features
- Providing better software development tools, such as tuning tools
- Further optimization through static analyses
- Various automatic load balancing libraries

7 Acknowledgments

Hiroshi Yashiro contributed in early design phases of the system. Daigo Sekita contributed to the runtime library of the sequential core. Masao Morita and Nobuyuki Ichiyoshi contributed to the shared memory implementation.

Discussions with Kazunori Ueda, Hiroshi Nakashima and other members of KLIC Task Group helped the design.

Finally, the authors would like to express their gratitude to the users of KLIC, who gave helpful comments.

References

[1] Aikawa, S, Kamiko, M. et al.: "ParaGraph: A Graphical Tuning Tool for Multiprocessor Systems", *Proceedings of FGCS'92*, pp.286-293, 1992.

[2] Carlsson, M., Widén, J. et al.: SICStus Prolog User's Manual, 1993.

[3] Chikayama, T., Kimura, Y.: Multiple Reference Management in Flat-GHC, *Proceedings of ICLP'87*, pp.276-293, 1987.

[4] Chikayama, T., Sato, H., Miyazaki.T.: Overview of the parallel inference machine operating system (PIMOS), *Proceedings of FGCS'88*, pp.230-251, 1988.

[5] Chikayama, T., Fujise, T. et al.: A Portable and Efficient Implementation of KL1, *Proceedings of PRILP'94*, Lecture Notes in Computer Science, #884, Springer-Verlag, 1994.

[6] Chikayama, T.: Parallel Basic Software, *Proceedings of FGCS'94*, 1994.

[7] Dongarra, J. J., Hempel, R. et al.: A proposal for a userlevel, message passing interface in a distributed memory environment, Technical Report TM-12231, Oak Ridge National Laboratory, 1993.

[8] Geist, A., Beguelin, A. et al.: PVM 3 USER'S GUIDE AND REFERENCE MANUAL, Technical Report TM-12187, Oak Ridge National Laboratory, 1994.

[9] Gudeman, D. et al.: "`jc`: an efficient and portable sequential implementation of janus", *Proceedings of JICSLP*, The MIT Press, 1992.

[10] Haygood, R.C.: Aquarius Prolog User Manual, 1993.

[11] Hirata, K., Yamamoto, R.: Parallel and Distributed Implementation of Concurrent Logic Programming Language KL1, *Proceedings of FGCS'92*, pp.436-459, 1992.

[12] Ichiyoshi, N., Morita, M. et al.: A Shared-Memory Parallel Extension of KLIC and Its Garbage Collection, *Proceedings of Workshop on Parallel Logic Programming attached to FGCS'94*, 1994.

[13] Ishikawa, M., Toya, T. et al.: Parallel Application Systems in Genetic Information Processing, *Proceedings of FGCS'94*, 1994.

[14] Janson, S., Montelius, J. et al.: AGENTS user manual, SICS technical report, SICS, 1994.

[15] Murao, H., Fujise, T.: Modular Algorithm for Sparse Multivariate Polynomial Interpolation and its Parallel Implementation, *Proceedings of PASCO'94*, pp.304-315, 1994.

[16] Nakajima, K., Inamura, Y. et al.: Distributed Implementation of KL1 on the Multi-PSI/V2, *Proceedings of ICLP'89*, pp.436-451, 1989.

[17] Rokusawa, K., Nakase, A. et al.: Reference Loops Management in a Distributed KLIC Implementation, *Proceedings of Workshop on Parallel Logic Programming attached to FGCS'94*, 1994.

[18] Saraswat, V.A., Kahn, K. et al.: "Janus: A step towards distributed constraint programming", *Proceedings of NACLP*, The MIT Press, 1990.

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

[20] Warren, D. H. D.: An abstarct Prolog instruction set, Technical Note 309. SRI International, 1983.