KLIC: A Portable Parallel Implementation of a Concurrent Logic Programming Language

Takashi Chikayama

Department of Electronic Engineering, University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo Japan

Abstract. This paper gives an overview of KLIC, a portable parallel implementation of a concurrent logic programming language KL1. Its portability is realized by the compilation scheme using the language C as an intermediate language and its object-oriented system extension framework called "generic objects," that allows realization of different communication schemes on exactly the same sequential kernel implementation.

Two completely different parallel implementations of KLIC have been actually built using this framework, one based on shared memory and the other on message passing. Both have exactly the same sequential kernel, which has competent sequential performance. Interprocessor communication is realized in an object-oriented manner through "generic objects" which encapsulates implementation details.

KLIC has been ported to many different systems, including several multiprocessor systems and systems consisting of network-connected workstations. Several experimental parallel application software systems run on these implementations.

1 Introduction

A concurrent logic programming language KL1[13] was chosen as the interface between the hardware and software research in the Japanese Fifth Generation Computer Systems project. The language has been proved to be a practical tool of parallel processing software research through the development of the operating system PIMOS[2] and various application software on parallel inference machines, Multi-PSI [12] and PIM [4].

Such implementations, however, had a serious disadvantage that they were not portable; although they are efficient, they run only on specially devised hardware. A portable byte-code implementation exists, but with limited performance.

To solve the problem, a scheme that allows a highly portable implementation of compiling into C was investigated. The language C was chosen as the intermediate language, as excellent optimizing compilers for the language are widely available nowadays. We have designed an implementation scheme which detours various possible demerits of using C as an intermediate language and built an experimental system named KLIC.

The sequential core of KLIC showed reasonable efficiency in both time and space aspects. It ran about twice as fast as the native code generated by SICStus Prolog for benchmark programs on SparcStation 10 model 30. The code size became larger than abstract machine code but was found to remain reasonable.[3]

This paper describes an overview of the KLIC implementation of KL1, focusing on how different parallel implementations are built on the same sequential core.

2 Sequential Kernel

This section gives an overview of the design of the sequential kernel of KLIC and the results of its evaluation. See [3] for further details.

2.1 C as the Intermediate Language

We decided to use the language C as the intermediate language to implement KL1. There are various merits in using C as the intermediate language, in which most important ones are the following.

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

Low-level optimization Some C compilers provide very good low-level optimization. By letting 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. In addition, in Unix-like systems where C is "the" language, most other language systems provide certain C language interface. Thus, KL1 programs can also be linked with programs in almost any languages without much effort.

Although compiling into C has the above-mentioned advantages, it is not easy to realize reasonable efficiency for languages with an execution model quite different from C, such as KL1. Typical efficiency problems and our remedies to them are as follows.

Costly function calls The language C and its implementations are designed having in mind that functions are not too small. Although function invocation and parameter passing overheads themselves may not be large, dividing programs into functions makes program analysis more difficult, often resulting in less efficient object code. Predicates of KL1 are usually very small, often as small as one line of C code. Many of them are recursive, prohibiting inline expansion. Thus, naive strategies such as compiling each KL1 predicate into one C function may result in quite inefficient code.

We decided to compile one program KL1 "module" that defines a set of closely related predicates into one C function. Compiling the whole program into one function may be the best for object code quality, but this will

prevent separate compilation, which will be a serious problem with programs of practical size. Our solution lets the user control the size of compilation. As far as predicates within the same module are calling one another, control transfer is by goto and arguments can be passed through local variables, which might be allocated on machine registers.

Register allocation control Certain global data, such as the free memory top pointer, are accessed very frequently. It would be best to keep such data on some dedicated registers during the whole execution. Some compilers allow such control but, as the feature is non-standard, portability of reasonable efficiency is not realized.

To access frequently referenced global variables with lowest cost, such variables are cached on local variables. C compilers may allocate them on registers. 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 of such variables so that only small numbers of them are to be passed and returned in most cases.

Provision for interrupts Multiprocessor implementations should process interrupts from other processors. Data accessed during interrupt handling are also frequently referenced and altered in normal processing within the processor. Thus, certain locking on data or inhibition of interrupts may be required. These are quite costly under conventional operating systems.

We decided to synchronize interrupts with normal processing. Signal handlers are made to set a certain flag, which will be examined at timing convenient for normal processing. This flag check is combined with another mandatory check, heap overflow check for garbage collection, and thus made with virtually no cost. This will be discussed further below.

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.

Fortunately, 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 handled by the runtime routines. Object code size was thus kept reasonable without significantly slowing down the execution.

2.2 Interrupt Handling

On multiprocessor implementations, normal goal reductions may be interrupted by other processors. On interrupts (*signals* of Unix), the signal handler will set a flag in a global variable and, at the same time, modifies the limit of heap usage so that the heap overflow check at the end of the current reduction will find it. The garbage collector called through the check examines the flag and tells the need of a garbage collection and/or interrupt handling. Thus, no extra check of

interrupts is required during normal processing, except that the heap limit has to be stored in memory rather in a register.

The same mechanism is also used to for other purposes: to implement priority scheduling, stepping tracer, etc.

2.3 Compilation

The KL1 to C compiler is written in KL1 itself. The code generated is a straightforward sequence of C program code, with some C-language level macros for ease of system maintenance. This C code is compiled into relocatable objects by the C compiler of the host system, with any low-level optimization. Then the relocatable objects are linked together with a runtime library archive of KLIC and possibly with other programs written in other languages to make the executable (Fig. 1).

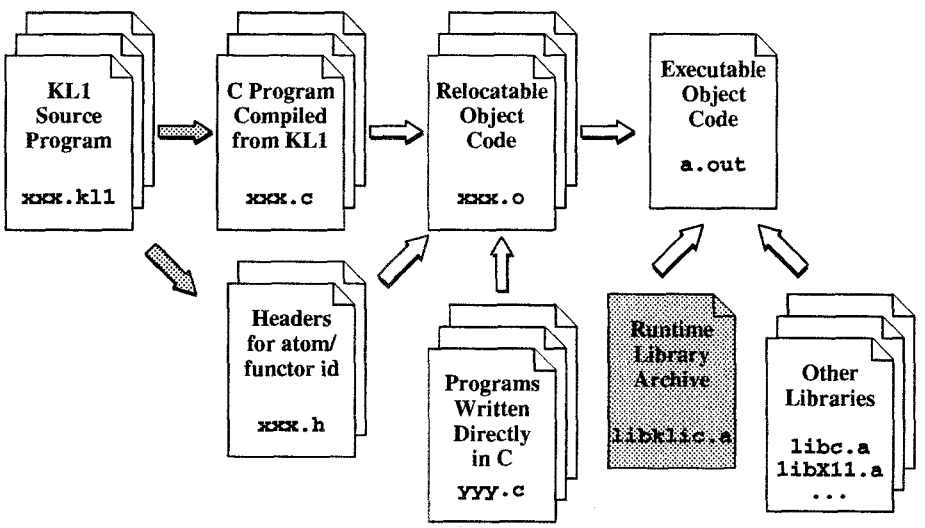

Fig. 1. Compilation and Linkage

What makes sequential and parallel systems different is the runtime library archive. The relocatable objects are exactly the same for both.

2.4 Performance Evaluation Results

Evaluation of the sequential core of KLIC is made through comparisons with similar language processing systems. The result suggests that the execution speed of the sequential kernel of KLIC is about twice as fast as native code generated by SICStus Prolog [1] and comparable to experimental optimized implementations

such as Aquarius Prolog [6] or JC [5]. Object code size is found to be a little smaller than that of SICStus Prolog.

One problem with KLIC is its slow compilation speed. It is more than ten times slower than SICStus Prolog. It is mainly due to the slowness of the C compiler, not the KL1 to C translator.

3 Generic Objects

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

We borrowed the basic idea of generic objects from AGENTS [7], modified and extended it for KLIC. In its original form in AGENTS, distinction of the three kinds of generic objects described below was not made; there are no generator objects and the functionality of consumer objects is provided by a separate feature called of *ports*.

3.1 Objectives

With the KL1 implementations on PIM [4], we experienced severe difficulties in trying out different parallel implementation schemes. The reason was that one predefined scheme was integrated too firmly into the system core. This pointed to us a moral that system extensibility and modifiability should be put above bare efficiency. On the other hand, as the KLIC 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 at a time. Some of the standard built-in data types and non-local data references for parallel implementations are represented 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.

3.2 Three Kinds of Generic Objects

KLIC provides three kinds of generic objects.

Data Objects are immutable objects *without* time-dependent states. They are accessed via built-in predicates and generic method calls of the form: `generic:Method(Obj, Args,...)`. Data objects are no more than a way to provide more built-in data types than those can be represented by the limited number of tag bits, except that adding new types does not require changes in the kernel implementation.

Consumer Objects are *data*-driven objects *with* time-dependent states. They look like a logical variables with a goal awaiting for its instantiation. They are

activated by variable instantiation, such as `Obj = [Message|NewObj]`. The unifier in the runtime kernel recognizes that the instantiated variable is associated with a consumer object, and calls the "unify" method of the object. The object performs the task specified by `Message` or creates a new goal to do the task, and associates itself with `NewObj` again. Thus, if the task specified is simple enough, the overheads of goal suspension and resumption can be avoided. Note that this is not disturbing the pure semantics of the language.

Generator Objects are *demand*-driven objects *with* time-dependent states. When the goal suspension handling routine finds that a variable suspended on is associated with a generator object, the "generate" method of the object will be invoked. The object may generate some value immediately and instantiate the associated variable, or it may spawn a goal that will eventually do it.

Generic objects are used to implement various features of KLIC. Among others, parallel implementations make extensive use of generator and consumer objects, which are described below.

4 Parallel Implementations

Two completely different parallel implementations are built upon the same sequential core, using the mechanism of generic objects. One of the parallel implementations is meant for shared-memory multiprocessors and uses shared memory for shared data [8]. The other implementation is for distributed memory systems and uses message passing mechanisms for communication between processors [11]. Both of them run exactly the same C code compiled from KL1. The only difference is that the runtime systems used are different.

4.1 Shared-Memory Implementation

The shared-memory parallel implementation uses two kinds of heap areas: local and global.

Worker processes running in parallel are associated with a local heap area, which contains data used exclusively by a single worker. Workers have different address spaces and thus local heaps may be allocated to the same logical address. Without any parallel execution specification, programs proceed using local heaps only. Once some data become visible from some other worker, they are copied to the global shared heap so that other workers can access. The key idea is not to allow any direct pointers from the global heap to any of the local heaps.

Most shared-memory multiprocessor mechanisms do not physically distinguish shared and local memory areas. However, coherent cache mechanisms are designed to provide faster accesses to data that reside only in the cache memory of one processor. The logical separation of local and global heap is expected to make accesses to exclusive data to stay within cache, not disturbing the shared bus connecting processors, which is the most precious resource of the system.

Copying to the global heap occurs in the following cases.

- When a goal is sent to some other worker. All its arguments are copied to the global heap. If they are nested structures, the whole structure is copied recursively. This occasion is explicit in the program and the runtime system can handle it easily.
- When some data structure is unified with a logical variable already shared among workers, that is, variables in the global heap. The data structure is copied to the global heap recursively. This is *not* explicit in the program, as whether lexically the same variable mean a local or global depends on dynamic behavior of programs. To detect such cases, variables in the global heap are represent as a generic object, whose *unify* method takes care of copying.

As the language is side-effect free, we only have to take care of logical variables. Elements of data structures are never *updated.*

Garbage collection of local heaps can be done independently by each worker process. If the whole local heap fits in cache, the shared bus will not be used by this local garbage collection.

We use a copying garbage collection scheme for the global heap. Each worker copies data it requires to the new space. However, it must be done with certain care so that workers sharing the same data structures should not become confused. A simple method is to synchronize all the workers, stop normal processing and do garbage collection simultaneously. However, it will congest the shared bus, as garbage collection is an operation that has low access locality. To avoid bus congestion, we developed a method to let workers copy the data they need asynchronously. As other workers are continuing normal processing possibly using the data being copied, we adopted a subtler scheme [8].

4.2 Distributed Memory Implementation

The distributed memory implementation adopts basically the same scheme as we used on parallel inference machines [10]. Each processor has its own local memory and there is no shared memory space. References to data in memory of a remote processor are represented as generic objects. They contain an indirect global pointer, consisting of the identifier of the remote processor and the index in the *export table* of the remote processor. All the data referenced from outside of a processor are registered in this export table. This allows local relocation of data, which enables asynchronous and independent garbage collection by each processor.

When a goal requires some data in the memory of a remote processor, the "generate" method of the object representing remote data is invoked, as it is the rule for generator objects. This will actually send a "read" message to a remote processor to fetch the relevant data. The processor with data returns the data in a "answer" message in its response, or, if the data is not generated yet and what is referenced is still an unbound variable, associates the variable with a data-driven consumer object that will respond to the original processor on unification with concrete data.

Another implementation layer is provided for increased portability. Beneath the layer of passing messages such as "read" and "answer" is a layer for physical communication. The physical layer is made separate because different computer systems have different interprocessor communication mechanisms suited for specific hardware architecture. Several different versions of the physical communication layer have been implemented, using PVM, MPI, Express, TCP/IP, CMAML for CM5, and "get & put" features on AP1000. As the physical communication layer is rather thin (a little more than 1,000 lines of C code), the KLIC system can be ported without too much effort to systems with different architecture, avoiding overheads of providing the uniform interface of standard message passing libraries.

4.3 Performance of Parallel Implementations

We have not yet evaluated parallel implementations in depth. Only a few serious application programs are tested on them so far.

The shared-memory implementation showed 9-fold speed-up of using 12 processors on SparcCenter 2000 on small benchmark programs.

The performance of the distributed memory implementation heavily depends on the physical communication layer used. We developed a version based on the distributed memory model, but using shared memory as message passing media, to see how it works with the fastest possible physical communication. This system showed up to 15 times speed-up for a distinct order factorization program (a part of symbolic formula manipulation system) using 17 processors of SparcCenter 2000 [9]. An implementation using slower physical communication layer, such as PVM over ethernet, showed much lower performance as was expected.

5 Conclusion

A scheme for portable implementation of a concurrent logic programming language KL1 is described. The scheme employs the strategy to compile KL1 source code into C for increased portability, rather than directly generating machine code. The sequential core of its experimental implementation shows that efficiency in both time and space aspects can be achieved even by such an indirect code generation scheme. The "generic object" feature of the system is so powerful that different parallel implementations could be built upon it. The side-effect-free nature of the language allowed clean design of data and memory management in parallel implementations.

Various efforts are still 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

Acknowledgments

The system was implemented at Institute for New Generation Computer Technology (ICOT) as a part of the FGCS follow-on project supported by the Japanese government during fiscal years of 1993 and 1994. The implementation team consisted of Tetsuro Fujise, Daigo Sekita, Kazuaki Rokusawa, Akihiko Nakase, Masao Morita, Goichi Nakamura, and the author.

References

1. M. Carlsson, J. Widén, J. Andersson, S. Andersson, K. Brootz, H. Nilsson, and T. Sjöland. *SICStus Prolog User's Manual*, 1993.
2. T. Chikayama, H. Sato, and T. Miyazaki. Overview of the parallel inference machine operating system (PIMOS). In *Proceedings of FGCS'88*, pages 230–251, Tokyo, Japan, 1988.
3. T. Chikayama, T. Fujise and D. Sekita. A portable and efficient implementation of KL1. In *Proceedings of PLILP'94*, pages 25–39, Springer-Verlag, Berlin, 1994.
4. A. Goto, M. Sato, K. Nakajima, K. Taki, and A. Matsumoto. Overview of the parallel inference machine architecture (PIM). In *Proceedings of FGCS'88*, Tokyo, Japan, 1988.
5. D. Gudeman, K. De Bosschere, and S. K. Debray. `jc`: an efficient and portable sequential implementation of Janus. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*. The MIT Press, 1992.
6. R. C. Haygood. *Aquarius Prolog User Manual*, 1993.
7. S. Janson, J. Montelius, K. Boortz, P. Brand, B. Carlson, R. C. Haygood, B. Danielsson, and S. Haridi. AGENTS user manual. SICS technical report, Swedish Institute of Computer Science, 1994.
8. N. Ichiyoshi, M. Morita, and Takashi Chikayama. A shared-memory parallel extension of KLIC and its garbage collection. in *Proceedings of Workshop on Parllel Logic Programming attached to the International Symposium on Fifth Generation Computer Sytems 1994*. ICOT, December 1994.
9. H. Murao and T. Fujise. Modular algorithm for sparce multivariate polynomial interpolation and its parallel implementation. In *Proceedings of PASCO'94*, pages 304–315, 1994.
10. K. Nakajima, Y. Inamura, N. Ichiyoshi, K. Rokusawa and T. Chikayama. Distributed implementation of KL1 on the Multi-PSI/V2. in *Proceedings of the Sixth International Conference on Logic Programming*, 1989.
11. K. Rokusawa, A. Nakase, and T. Chikayama. Distributed memory implementation of KLIC. *New Generation Computing*, 14(3), May 1996, *to appear*.
12. 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 FGCS'88*, Tokyo, Japan, 1988. Also in New Generation Computing 7–2, 3 (1990), pp. 179–195.
13. K. Ueda and T. Chikayama. Design of the kernel language for the parallel inference machine. *The Computer Journal*, 33(6):494–500, December 1990.