Implementing a Finite Domain Constraint Solving System in KLIC

Rong Yang

Department of Computer Science
University of Bristol
Bristol BS8 1TR, U.K.

Abstract

The paper presents CHUKL, a parallel finite-domain constraint solving system. The implementation is written in the concurrent logic programming language KL1, instead of C. The work challenges the assumption that committed choice concurrent logic programming languages cannot deal with don't know non-determinism. A dynamic forking scheme is designed and implemented to handle choicepoint making. The work also demonstrates that KLIC, an efficient implementation of KL1, is a powerful parallel programming tool which allows us rapid prototyping. The development on CHUKL has gone through several stages: three different versions of the runtime system were implemented. The performance of the latest version is reasonably close to CHIP which is a refined low-level implementation in C.

1 Introduction

KLIC [1] is an efficient implementation of the concurrent logic programming language KL1 [13], developed at ICOT as a part of the Fifth Generation Computer Systems Project of Japan. It compiles KL1 program into C code achieving high-performance and is portable to most Unix machines. Several parallel versions of KLIC are also implemented based on distributed memory and shared memory architectures [8, 9].

As is well known, KL1 is a simple but powerful language for concurrent computation. In this work, our general goal is to investigate whether we can widen its application area such that it can be used to implement non-deterministic logic programming systems. Normally, for efficiency reasons, logic programming systems are implemented in C or lower level languages. One might question using such a high level language now that so many successful C implementations exist. Our position is as follows.

Looking back, all parallel programming systems developed have involved a great deal of effort, perhaps several man-years. One can probably write a sequential prototype very quickly, but it might take 10 times more effort to make it work in parallel. One solution is to separate the effort into two levels. That is, if we have an intermediate language, which must be a concurrent language of course, then a parallel programming system can be built in two independent steps:

low level: implementing the intermediate language in C or other low-level languages;

high level: implementing the parallel system on top of the intermediate language.

The advantages of this two-level implementation approach are as follows. (i) The effort paid for developing the intermediate language doesn't only produce a system for a particular application but a general purpose software. Therefore, resources can be utilised more efficiently. (ii) One can leave the intermediate language to take care of low-level issues such as memory management, locking and synchronization. This allows a programmer to concentrate on high level issues and prototype new ideas quickly.

A key question is what the suitable intermediate language is. We believe that KLIC system is a good choice. It provides a declarative concurrent programming language, its implementation is efficient compared with other similar systems, and it is portable to many parallel machines [1].

In summary, the motivations of this work are to try out our idea of two-level implementation, to evaluate whether KLIC is a good choice as an intermediate language, and finally to investigate suitable programming methodologies for developing parallel logic programming systems in KLIC.

We selected finite domain constraint logic programming system as the first step to study. The CHUKL (**C**onstraint **H**anding **U**nder **KL**1) system is designed and implemented. It indeed required much less development time and achieved reasonably good performance as a prototype.

In this paper we present our practical experience and discuss our findings from this work. The rest of the paper is organised as follows. The next section describe finite domain constraints, the background of the work, to make the paper self-contained. Section 3 gives an overview of the CHUKL system. Section 4 presents in detail the CHUKL runtime system that we have implemented. Three different versions are discussed. In Section 5, we show the evaluation results of our work and compare CHUKL with CHIP, a well refined low-level implementation in C. Finally, conclusions and future work end the paper.

2 Finite Domain Constraints

In recent years, Constraint Logic Programming (CLP) has become an increasingly important research area. CLP languages extend logic programming with efficient methods for solving constraints on specific domains, thus providing a practical way to solve many kinds of problems, especially those involving combinatorial search. Many CLP languages and systems have been developed; perhaps the most successful in practice have been those based on finite domains, in which variables can be constrained to range over a finite set of possible values. The first finite-domain CLP language was CHIP [4], but this has been followed by others, including clp(FD) [3] and cc(FD) [5], among others. Here we shall use the name CLP(FD) to refer to a generic finite-domain CLP language.

In CLP(FD), some variables in a program may be designated as *domain variables*, ranging over a specified finite domain. Domain variables are treated differently by unification and constraints.

Generally speaking, a CLP(FD) program is composed of user defined Prolog-style predicates and CLP(FD) builtins. Three main types of builtins in CLP(FD) are as follows.

declaration builtin by which all domain variables are declared;

constraint builtin which includes a set of basic arithmetic constraints and symbolic constraints;

labelling builtin by which the domain variables are non-determinately assigned values.

The computation of CLP(FD) comprises two alternating phases:

- constraint propagation, and
- non-deterministic guessing.

In the first phase, all constraints are evaluated concurrently. The domain variables may be reduced to a smaller domain, or even a single value through the constraints' evaluation. The changes to a domain variable are propagated to related constraints. Therefore, an already evaluated but not yet solved constraint might activate again. This phase is just like the computation of concurrent logic programs. When no constraints are active, that is, we can't reduce domain variables' domains any further, the first phase terminates. In the second phase, we select one unbound domain variable, using its different possible values in the domain to create a non-deterministic fork.

3 The CHUKL system

CHUKL is a CLP(FD) system in KLIC. The general picture of the CHUKL system is shown in Figure 1.

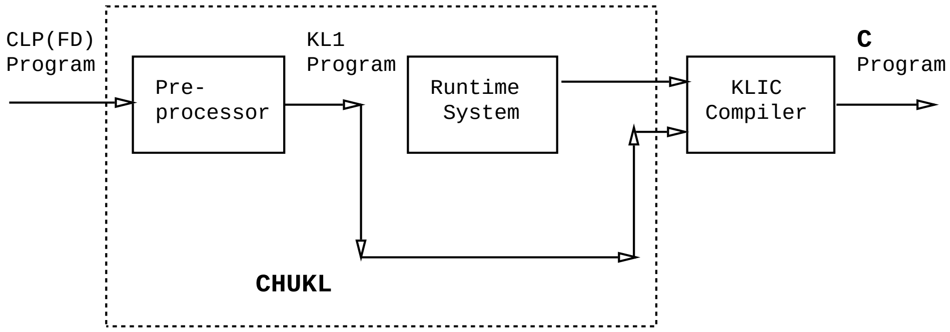

Figure 1: THE SYSTEM CONSTRUCTION OF CHUKL

There are two components in CHUKL: a *preprocessor* and a *runtime system.*

To run a CLP(FD) application, we first use the CHUKL preprocessor to transform it to a KL1 program. Next we apply the KLIC compiler to compile this KL1 program and link it with the CHUKL runtime system. This finally generates the C code of the application.

3.1 The Preprocessor

The transformation from CLP(FD) to KL1 is a source-to-source translation. In more detail, there two kinds of transformations:

- For CLP(FD) builtins, the preprocessor converts them to goals which call runtime libraries supported by the CHUKL runtime system.
- For non-builtins, the preprocessor translates them from Prolog syntax to KL1 syntax.

The first transformation is a kind of mapping operation, and is therefore very straightforward. But the second one is not trivial. The issue is how to translate a deterministic Prolog program to a KL1 program. To achieve this, we need a global analysis of the program to find out the mode information of each predicate and then moving the output unification from the head to the body. Our CHUKL preprocessor provides a mode analyser based on an algorithm given in [2]. Some significant improvements have been added to the algorithm. The details can be found in [14]. In this paper, we concentrate on presenting the runtime system.

3.2 The Runtime System

The CHUKL runtime system supports

- libraries of constraint solver;
- the machinery to control coroutining and forking.

The main issues arise from the following two aspects.

(a) How to represent domain variables

The most essential information attached to a domain variable is a set of the current possible values in the domain.

In existing low level implementations, the domain variable is represented as a special data structure which contains all necessary information, such as what the current possible values are and which constraints are dependent on the domain variable. Within a constraint, each domain variable is actually a pointer to its domain data structure. When the constraint needs to update a domain, the operation is like a destructive assignment in procedural languages: we overwrite an already instantiated value by a new value. For example, when the constraint `X > 3` is executed while `X` ranges from 0,...,9, the old interval 0,...,9 is replaced by a new interval 4,...,9. In low level implementations, we have the freedom to directly implement this kind of destructive assignment. However, in a high level concurrent language KL1, it is not easy to adopt the same method. Another issue relates to the and-parallel execution. When the constraint propagation phase is running in parallel, we need to guarantee that updating a domain is an atomic operation.

We have developed several different methods to implement domain variable which will be discussed in Section 4.

(b) How to deal with non-deterministic forking

As discussed above, the constraint propagation phase is same as the computation in a concurrent logic program. Therefore, it can be naturally fitted into KL1 very well: the constraints can be realised as KL1 processes and executed in parallel. But the non-deterministic phase is the problem. For many years, concurrent logic programming languages have been facing a major criticism from Prolog programmers, that is, they lack don't know non-determinism.

One approach was to convert a Prolog program to a concurrent logic program through a programming transformation [11, 7, 10, 12]. However, these have not been very successful in practice because they usually only work for a restricted subset of Prolog, and usually do not provide and-parallelism. In this work, instead of using programming transformation, we want to investigate the possibility of supporting non-deterministic forking at runtime. The basic idea is to have an extra process which monitors the whole computation: detecting deadlock and forking or-branches by copying the current state. We call it the *dynamic forking scheme*. Although this scheme might sound costly, we believe it is more powerful and general.

4 Three CHUKL Runtime Systems

4.1 CHUKL1: A Process-oriented Implementation

Our first implementation, CHUKL1 [15], simply followed a commonly used technique in concurrent programming: representing every object as a process. There are four kinds of processes:

- each constraint is a KL1 process;
- each domain variable is a KL1 process;
- for each or-branch, as mentioned above, there is a special process called a *controller* which monitors the computation, and deals with non-deterministic forking;
- Finally, a scheduler controls all or-branches, distributing or-branches among physical or virtual processors.

Figure 2 (a) shows the process structure of CHUKL1.

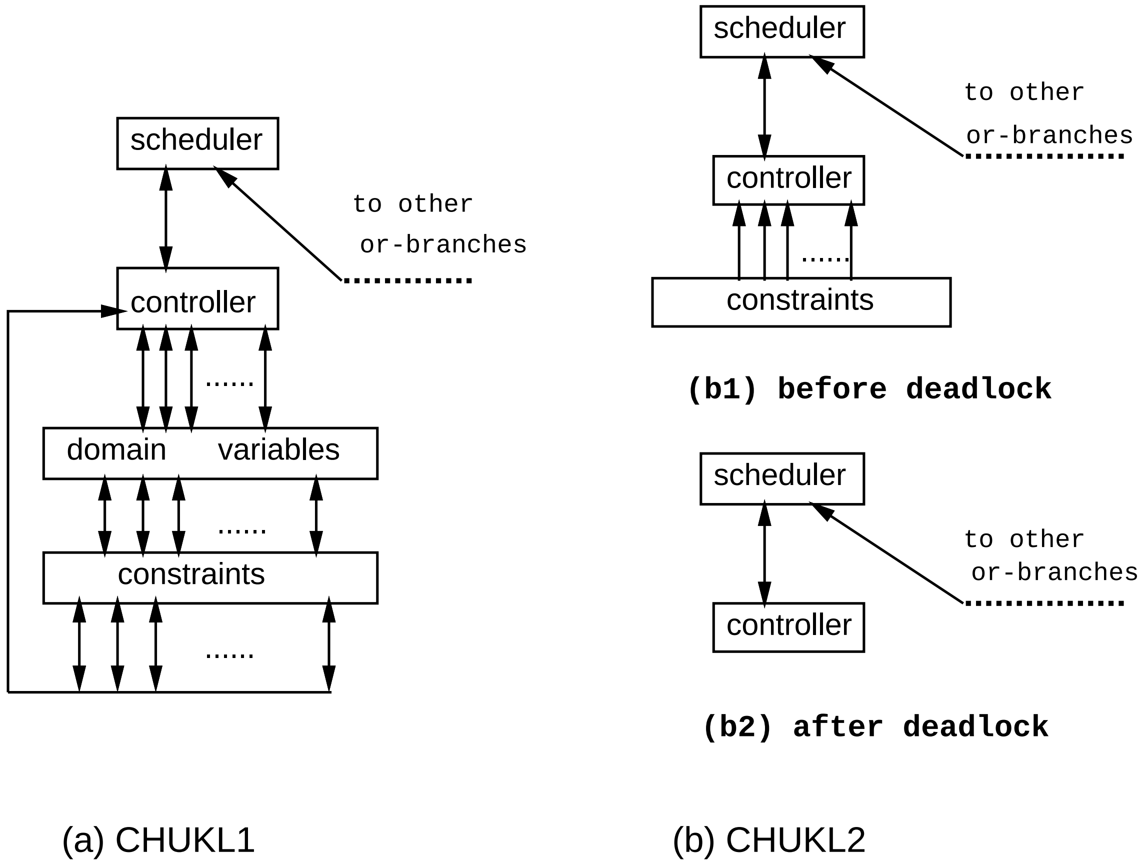

Figure 2: PROCESS STRUCTURE OF CHUKL1 AND CHUKL2

Obviously, the problem of destructive assignment to domain variables is solved by a using KL1 process to manipulate the domain. A domain variable X in a constraint is represented by `dv(StreamToX)` where `StreamToX` is a communication channel to X's process. For instance, when the constraint

```
4*dv(StreamToX)+dv(StreamToY) = 10 (i.e.  4*X+Y = 10)
```

is evaluated, its constraint process first sends messages through `StreamToX` and `SteamToY` to read the current values in X and Y. Then, if the constraint needs to update the domain of X or Y, it sends a message again. Note that updating a domain may not be successful, because several constraints may try to update a domain simultaneously with inconsistent value sets. Thus, after updating a domain, the constraint process needs a reply from the domain process to confirm whether the updating has actually been done or has failed.

The constraint processes are all connected to the controller as well as related domain processes. When a constraint is suspended, solved or failed, it reports its state to the controller.

The deadlock detection and forking are handled in the following way. When all constraints suspend, the controller checks whether all domain variables' latest domains have been read by all dependent constraints. If so, the deadlock is confirmed; the forking can go ahead. Otherwise, the controller tells the constraints to retry. In each domain process, we keep a list of constraints which have read the latest updated value for the purpose of detecting deadlock. To fork or-branches, the controller selects an unbound domain variable, reads its domain, then creates or-branches corresponding to the different values in the domain. Thus, non-deterministic forking problem is solved by adding a monitor process, the controller, to each or-branch.

Finally, we describe how the scheduler works. The scheduler is initialised by a given configuration which reflects, for example, how many physical or virtual processors are going to be used and what is maximum number of or-branches that can be explored simultaneously. All or-branches' controllers are connected with the scheduler, which monitors the computation tree. When an or-branch either fails or succeeds, it informs the scheduler of its termination. When a controller forks new N or-branches, it sends a request to the scheduler. The scheduler decides how many branches can be created or suspended depending on the current load balancing situation and the priority of the branches.

CHUKL1 is based on a naive model where everything is simply represented as a process. As we expected, its performance is quite poor: its sequential speed is about 100 times slower than an efficient C implementation, CHIP. However, from CHUKL1, we were convinced that our dynamic forking scheme is feasible. This is the main issue which we focused on.

4.2 CHUKL2: A Data-oriented Implementation

CHUKL2 was developed to overcome the inefficiency of CHUKL1 by improving the execution model.

We believe that the main reason for CHUKL1's poor performance is that the process communication was too heavy. There were four kinds of KL1 processes, and they are all communicating bidirectionally (bidirectional arrows in Figure 2 mean two-way communication). Moreover, or-branches were created in a distributed way: each process receives a fork message and necessary data and then forks itself independently. Therefore, creating a new or-branch requires a great deal of synchronization. We suspect that this would greatly affect parallel performance.

Based on the above considerations, in CHUKL2, we kept the dynamic forking scheme unchanged, but made the following improvements.

- Domain variables are no longer processes, but are represented as data. All domain variables are stored in a table which is shared by all constraints. Therefore, when a constraint needs to read or update a domain variable, it can access it directly. We use KL1's vector data structure to represent the domain variable's bitmap. To solve the destructive assignment problem, an unbound variable is used to represent the bit "1". An example of a domain table is shown in Figure 3.

- Most two-way communications between processes have been removed. This is partly achieved by removing domain processes. But we also specially changed the communication between controller and constraints from two-way to one-way. In CHUKL1, when a constraint suspends, it sends a suspend message to its controller and then waits for a reply: `resume`, `fail` or `fork`. In contrast to this, a constraint process only has a very short life in CHUKL2; if it suspends, the constraint process terminates and returns the constraint to its controller. As we can see in Figure 2 (b2), when the propagation phase completes, there are no constraint processes alive. All unsolved constraints are actually represented by data structures kept in the controller. This not only reduced the overhead for communication but also brought an advantage which is explained next.
- We made the forking procedures much simpler in CHUKL2: a new or-branch can be invoked by the controller process alone, without synchronization.

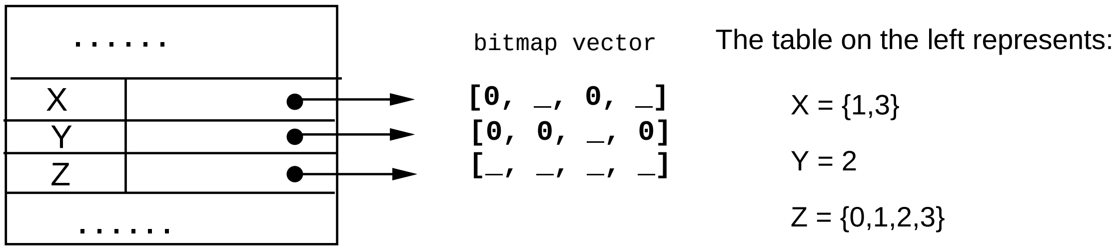

Figure 3: A Domain Table in CHUKL2

We view CHUKL2 as a data-oriented, rather than process-oriented, model.

Our experimental results for CHUKL2 are very encouraging. They show CHUKL2 is much faster than CHUKL1, giving a total speed improvement of up to a factor of 5.

4.3 CHUKL3: Applying Generic Objects to CHUKL2

In the KLIC implementation, a new feature called "generic objects" is provided. This is an interface to allow a user to add new data objects to the KLIC system. All operations on the new data objects are defined by the user and written in C.

In order to improve the performance further, we utilise the generic objects technique in CHUKL3 to implement a domain variable. A domain variable is not a KL1 term but an extended data type, a new object in KL1. The fields stored in a domain variable are as follows:

- the size of the domain,
- the current minimum value in the domain,
- the current maximum value in the domain,
- the bitmap of the domain's value,
- a field for the variable's value (when the value is still not determined, this field is -1),

- a list of suspended constraints which should wake up when the domain variable's minimum or maximum are changed, and
- a list of suspended constraints which should wake up when the domain variable is bound.

Using generic objects to create our own data object has brought many advantages:

- Less memory space is consumed: in CHUKL2, one "bit" in a bitmap actually occupies one *word*, but CHUKL3 uses a single bit.
- A lock can be introduced for each domain variable in the parallel version of CHUKL. This will allow a constraint to promote a domain's value as soon as it becomes determined. Therefore, other constraints can read updated values earlier, which will improve the speed. In CHUKL2, without the ability to lock a domain variable, the domain's value cannot be promoted until the constraint propagation phase is finished.
- The destructive assignment problem is efficiently solved at a low-level. As mentioned before, in CHUKL2, the bitmap had to be represented by vector of unbound variables (representing a '1' bit), and integer '0' (representing a '0' bit). Initially a bitmap is just a hollow vector (representing a bitmap with all bits set). When a value is removed, the variable was bound to '0', indicating that that value is no longer possible. This is of course inefficient. As well as the time and space overhead, there are problems with metaprogramming: to test an unbound variable in a concurrent logic programming environment needs great care. In CHUKL3, C's destructive assignment is used to update the fields of a domain variable. This is more efficient, and the metaprogramming problem no longer exists, because all domain fields are ground; this has greatly simplified the implementation. As for the current minimum and maximum values of a domain, in CHUKL2, we could not represent these in the domain table because of the destructive assignment problem; we had to store the min and max values locally in constraints. For example, in CHUKL2, if domain variables `X` and `Y` range from `0..3`, the constraint `4*X+Y = 10` is represented as

  ```
  4*dv(1,0,3)+dv(2,0,3) = 10
  ```

  where the first argument in structure `dv` is an index to the domain table and the other two arguments represent the current min and max. Whenever a domain variable changes its min or max value, CHUKL2 has to reconstruct a new constraint to keep the information updated. In CHUKL3, we store the minimum and maximum values globally in the domain table. Thus, the above constraint is simply represented as

  ```
  4*dv(1)+dv(2) = 10
  ```

  so no reconstruction is needed.
- In CHUKL1 and CHUKL2, we had to apply a busy waiting scheme to handle suspending and resuming constraints. In CHUKL3, we can easily implement two lists hooked on each domain variable: one represents constraints which are suspended on the domain's minimum and maximum values; another one represent constraints suspended on the domain variable's value. In this way, the constraints can be woken up whenever the corresponding condition changes.

The work on CHUKL3 has shown the great advantage of the "generic object" facility. The C code added to support the domain variable as a new data object is actually fairly small, about 900 lines including some comments. But the performance has been greatly improved. The detailed results can been found in the next section.

5 Evaluation Results

To evaluate the CHUKL system, we used the following typical benchmarks: two cryptarithmetic puzzles (send+more=money and roller-wheels=skoda), a car sequencing problem and a linear equation solving problem.

Figure 4 shows the execution times (in milliseconds) on each of the three versions of CHUKL. Figure 5 shows the speed of each CHUKL system relative to CHUKL1. These tests were carried out on a Sun Sparc20. CHUKL2 is significantly faster than CHUKL1, while the use of generic objects in CHUKL3 brings a further speed improvement of up to a factor of 9. On the whole, CHUKL3 is about 10 to 20 times faster than our original system, CHUKL1.

program	CHUKL1	CHUKL2	CHUKL3
smm	120	70	10
roller	37570	12900	3880
cars	30840	14120	1560
eq15	10449720	1951060	528510

Figure 4: Execution times with CHUKL1, CHUKL2, and CHUKL3

program	CHUKL1	CHUKL2	CHUKL3
smm	1	1.7	12.0
roller	1	2.9	9.7
cars	1	2.2	19.8
eq15	1	5.4	19.8

Figure 5: Speed comparison of CHUKL1, CHUKL2, and CHUKL3

To compare the memory usage of the three versions of CHUKL, we collected two statistics (Figure 6): the heap size allocated and the number of garbage collections. In each case, the heap size was initialised to 24576 words, KLIC's default size. Both CHUKL2 and CHUKL3, unlike CHUKL1, can run our four benchmarks without exceeding the initial heap size. Although we did not record a difference in heap size between CHUKL2 and CHUKL3, the number of garbage collections in CHUKL3 is about one third that of CHUKL2. This indicates that using generic objects improves not only speed but also memory usage.

To compare our work with a C implementation, Figure 7 shows the results from comparing CHUKL3 with CHIP, which is a commercial constraint logic programming system. The latest version of CHIP, version 4, was used, and the tests were run on a Sun Sparc Classic. The results show that CHUKL3 is reasonably close to CHIP, only about 3 to 8 times slower.

We believe that the CHUKL system can be optimized further by some more tuning.

To test the parallel performance, we have adapted the KLIC version of CHUKL1 and CHUKL2 to a PIMOS/KL1 version which runs on the PIM/p machine [6]. CHUKL3 can't be adapted to PIM/p because it lacks the "generic object" facility. The result from PIM/p is

program	CHUKL1		CHUKL2		CHKUL3	
	heap size	No. of GC	heap size	No. of GC	heap size	No. of GC
smm	24576	6	24576	2	24576	0
roller	98304	579	24576	725	24576	177
cars	393216	147	24576	799	24576	76
eq15	1572864	9217	24576	121606	24576	25103

Figure 6: Memory performance of CHUKL1, CHUKL2, and CHUKL3

program	CHIP	CHUKL3	CHUKL3:CHIP
smm	16	50	3.1
roller	1500	9830	6.6
cars	1216	3738	3.1
eq15	160766	1256280	7.8

Figure 7: Comparison of CHUKL3 with CHIP

also encouraging. We have been able to achieve about 7.5 times speedup with 8 processors. Note that this result was obtained by using one cluster of PIM/p. In the near future, we are planning to evaluate parallel performance in more detail by using the parallel version of KLIC.

Finally, having given the results on system's performance, we return to our initial idea of using two-level implementation to reduce the cost of parallel system implementation. The last table, Figure 8, shows the time spent on developing the three CHUKL systems and the size of the systems. The development time includes designing, coding and debugging, but does not include the time spent on adapting to other machines. The size of system includes the comment lines (of which there are not many). Note that CHUKL1 and CHUKL2 do not share any code, while CHUKL3 shares most of its KL1 code with CHUKL2. Figure 8 shows that, by using KLIC as an intermediate language, the development of a prototype CLP(FD) system can be done in a few months.

	size of system	developing time
CHUKL1	3740 lines of KL1	2 months/one person
CHUKL2	3290 lines of KL1	2 months/one person
CHUKL3	2800 lines of KL1 + 920 lines of C	1.5 month/one person

Figure 8: Cost of developing CHUKL

6 Conclusion and Future Work

The conclusions of this work can be separated into the following three aspects.

We have proposed a dynamic forking scheme to achieve non-deterministic choice in concurrent logic programming languages. The scheme was successfully applied in CHUKL. We believe that the scheme will be widely used to implement other non-deterministic logic programming systems in the concurrent logic programming framework.

The work also demonstrates that KLIC, an efficient implementation of KL1, is a powerful parallel programming tool which allows us to rapidly prototype our ideas. From our experience, the approach of two-level implementation is certainly useful. Note that this finding does not suggest that we should give up C to use a high level language like KLIC instead. Of course, we always need the efficient low level languages to maximise performance. But, as an alternative approach, especially when we need to quickly try out and adjust ideas, a high level, declarative and efficiently implemented system like KLIC is indeed a good choice. Besides KLIC's "generic object" facility, which allows the user to define their own object in C, has helped us to improve performance greatly with a little effort. Our experience of using "generic object" facility has probably indicated what the best solution of combining high and low level languages should be.

Finally, we summarise the conclusions on CHUKL. The initial results have shown that the basic performance of CHUKL is very promising. Compared with CHIP, which is a well refined low-level implementation in C, CHUKL is only about 3 to 8 times slower, which is a very encouraging result as a prototype implementation. Moreover, the memory usage is acceptable and the parallel speedups are also good.

As future work, we plan to improve CHUKL's performance further and we want to try CHUKL on the parallel version of KLIC if possible (at the moment, we do not have access to suitable parallel machines). We are also interested in applying the dynamic forking scheme and programming methods used in CHUKL to develop other non-deterministic parallel logic programming systems.

Acknowledgements

The development of the CHUKL system is supported by ICOT as part of the FGCS Follow-on Project. The work would not have been possible without help from Takashi Chikayama and Hiroyoshi Hatazawa of ICOT who gave much useful advice on the KLIC system and the PIM/p machine. The author would like to thank the Department of Computer Science at the University of Essex for allowing us to use the CHIP system on their machine. Finally, a special thank goes to Steve Gregory for his comments on the paper.

References

[1] T. Chikayama, T. Fujie, and D. Sekita. A portable and efficient implementation of KL1. In *International Symposium on Programming Language Implementation and Logic Programming*, pages 25–39, Madrid, September 1994.

[2] S. K. Debray. Static inference of modes and data dependencies in logic programs. *ACM Transactions on Programming Languages and Systems*, 11(3), July 1989.

[3] D. Diaz and P. Codognet. A Minimal Extension of the WAM for clp(FD). In *Proceedings of the International Logic Programming Conference*, MIT Press, June 1993.

[4] M. Dincbas, P. V. Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint logic programming language CHIP. In *International Conference on Fifth Generation Computer Systems 1988*, pages 693–702, ICOT, 1988.

[5] P. V. Hentenryck, V. Saraswat, and Y. Deville. *Design, Implementation and Evaluation of the Constraint Language cc(FD).* Technical Report No. CS-93-02, Dept. of Computer Science, Brown University, Jan 1993.

[6] K. Kumon, H. Hatazawa, et al. Architecture and implementation of PIM/p. In *International Conference on Fifth Generation Computer Systems*, June 1992.

[7] A. Okumura and Y. Matsumoto. Parallel programming with layered streams. In *Proceedings of the International Logic Programming Symposium*, October 1987.

[8] K. Rokusawa, A. Nakase, and T. Chikayama. Distributed memory implementation of KLIC. In *Proceedings of ICOT/NSF Workshop on Parallel Logic Programming and its Programming Environments*, University of Oregon Technical Report CIS-TR-94-04, March 1994.

[9] K. Rokusawa, A. Nakase, and T. Chikayama. Quiescence detection in a distributed KLIC implementation. In *EURO-PAR'95*, 1995.

[10] H. Tamaki. Stream-based compilation of ground I/O Prolog into committed choice nondeterministic languages. In *Proceedings of the International Logic Programming Conference*, June 1987.

[11] K. Ueda. Making exhaustive search programs deterministic. In *Proceedings of the International Logic Programming Conference*, June 1986.

[12] K. Ueda. Making exhaustive search programs deterministic, Part II. In *Proceedings of the International Logic Programming Conference*, June 1987.

[13] K. Ueda and T. Chikayama. Design of the kernel language for the Parallel Inference Machine. *Computer Journal*, December 1990.

[14] K. D. Varsamos. *The CHUKL Preprocessor: Transforming CHIP programs to KL1.* MSc Thesis, Dept. of Computer Science, University of Bristol, September 1994.

[15] R. Yang. CHUKL: Constraint handling under KL1. In *Proceedings of ICOT/NSF Workshop on Parallel Logic Programming and its Programming Environments*, University of Oregon Technical Report CIS-TR-94-04, March 1994.