[9] L. V. Kale. Parallel architectures for problem solving. Technical Report UIUCDCS-R-85-1237, Department of Computer Science, University of Illinois at Urbana-Champaign, 1985.

[10] Z. Lin. Self-organizing task scheduling for parallel execution of logic programs. In *Proceeding of the 1992 Conference on Fifth Generation Computer Systems*, Japan, 1992.

[11] Z Lin. *Task Scheduling for Parallel Execution of Logic Programs.* PhD thesis, Department of Computer Science, University of Maryland, 1992.

[12] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1984.

[13] S. Mudambi. Performance of aurora on a switch-based multiprocessor. In *Proceeding of the North American Conference of Logic Programming, 1989.* MIT press, 1989.

[14] S. Mudambi. Performance of aurora on numa machines. In *Proceeding of the 8th International Conference on Logic Programming.* MIT Press, 1991.

[15] B. Ramkumar and L. V. Kale. Compiled execution of the reduced-or process model on multiprocessors. In *Proceeding of the North American Conference of Logic Programming, 1989.* MIT press, 1989.

[16] K. S. Trivedi. *Probability and Statistics with Reliability, Queuing, and Computer Science Applications.*

it is able to compile a Prolog specification into an executable for a network of workstations; it also allows easy integration with external C modules as demonstrated.

Contribution of this research is:

- studied and experimented with techniques to generate efficient C object code from Prolog-like specifications, which is motivated by the need to provide users with a means to attain high performance. We show that substantial performance gain can be achieved through proper handling of determinate predicates. We also show that transient variables ought to be detected and treated specially;
- studied and implemented a control mechanism, self-organizing scheduling, for parallel execution of logic programs in a loosely-coupled architecture;
- demonstrated once again that Prolog is a versatile language capable of many different tasks including parallel programming. A Horn clause program with a SET operator can describe, in a very natural way, fork-join graphs which are often used to represent parallel computation, without losing the simple and intuitive reading of the program.

We believe that more optimization is still possible in both the compiler and the kernel. We expect that the gap in terms of speed between a determinate Prolog program (or code segment) and a comparable C program would soon be narrowed dramatically. For certain types of programs in which a list data structure is necessary, and runtime type checking is not a major factor in performance the gap may be closed.

In the future, we will investigate the possibility of using other logic programming execution models that may maximize the benefit of compiled execution. We will also study the domain of applications most suitable for a logic oriented (hybrid) programming system, given its builtin mechanism for parallel execution and other advantages inherited from its Prolog ancestor. We are considering a large natural language processing Prolog program, which happens to have an accompanying C program.

Acknowledgments: We wish to thank Prof. Jack Minker for his guidance and many helpful suggestions. We also thank Michael Tan for her help in preparing this paper.

References

[1] Hassan Ait-Kaci. The Wam: A (real) tutorial. Digital Paris Research Laboratory, 1990.

[2] K. Ali and R. Karlsson. The Muse or-parallel Prolog model and its performance. In *Proceeding of the North American Conference of Logic Programming, 1990*. MIT press, 1990.

[3] M. Carlson, K. Danhof, and R Overbeek. A simplified approach to the implementation and-parallelism in an or-parallel environment. In *Proceedings of 5th International Conference and Symposium of Logic Programming*, pages 1565–1577, 1988.

[4] M. Carlsson. *Design and Implementation of an Or-Parallel Prolog Engine*. PhD thesis, Swedish Insitute of Computer Science, 1990.

[5] K. Clark and S. Gregory. Parlog: Parallel programming in logic. *ACM Transaction on Programming Languages and Systems*, 8(1):1–49, Jan 1986.

[6] K. Fuchi *et el.* Launching the new era. *Communication the ACM*, 36(3), March 1993.

[7] G. A. Geist. Network based concurrent computing on the pvm system. Technical report, Oak Ridge National Lab., 1991.

[8] M. Giuliano, M. Kohli, J. Minker, and I. Durand. Prism: A testbed for parallel control. In L. Kanal and V. Kumar, editors, *Parallel Algorithms for Machine Intelligence*. 1990.

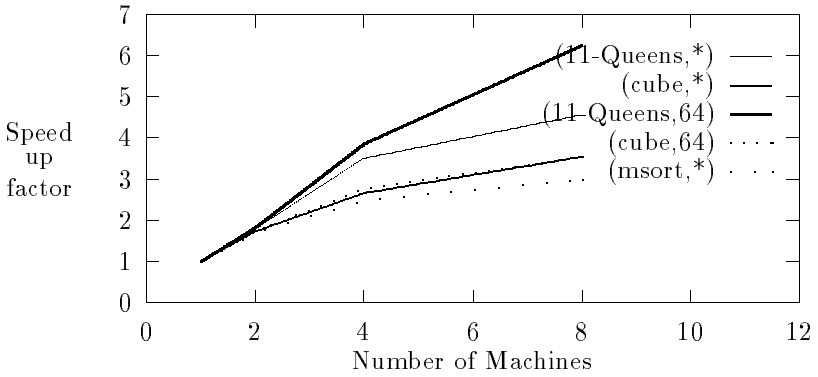

Figure 6: Speed-up on a DEC5000 workstation cluster. Line labels have the format (program_name,process_num). A "*" in the second field indcates using the default number of processes, i.e., the number of machines.

Finally, the *msort* program is amended to use a generic variable to store an array of 100,000 records, and to have the split() and merge() predicates defined in C. We also modified the program to invoke a quicksort C procedure once the original array is split into 16 partitions. Our compiler compiles the Prolog program as usual, and links it with the other C modules which contain the definition of split(), merge(), and quicksort(). Table 5 shows the sequential and parallel runtime[6] of this mixed Prolog-C program sorting 100,000 records. For comparison, the sequential execution time of a quicksort program in C which calls the qsort() library function is also given in the table. The result appears encouraging: 1) this program processes a fairly large data set which usually would not be attempted with Prolog; 2) the sequential Prolog-C program run at a speed comparable to a quicksort program in C, and 3) speed-up of a factor of 2 is obtained in a situation in which a few megabyte of data *have to be moved* around the network during the execution. We note that much better speed-up can be achieved if the problem at hand does not require transferring massive data through the network. For example, the *fib* program obtains a speed-up of 5.5 with 8 machines computing fib(30).

C quicksort	Prolog-C		
Sequential	Sequential	2 Machines	4 Machines
22.0	23.5	15.5	11.5

Table 5: Sorting 100,000 records: comparison between a quicksort procedure in C and a mixed Prolog-C program. The Prolog-C program is also executed on multiple machines.

6 Summary

A logic programming system is described in the paper. The system is implemented both on a single Unix workstation and in workstation clusters. We obtained experimental results showing the performance of this system. Our Prolog compiler is able to generate C object code that runs at a speed comparable to or even higher than that of Sicstus;

[6]The elapsed time from parallel execution is averaged from the best 3 out of 10 runs. Timing fluctuation is severe running this program, because a large amount of data needs to travel through the network of which we do not have exclusive use during the experiment.

System	fib(23)	9-Queens	Cube	Pattern	Zebra	Turtles
Nsys	7.3	10.1	7.1	0.8	0.9	2.8
Dsys	0.9	3.3	4.4	0.4	0.9	2.8
Ratio=$\frac{Nsys}{Dsys}$	8.1	3.1	1.6	2.0	1.0	1.0

Table 3: Effect of on-line execution of determinate predicates.

which is a WAM-based implementation. This suggests that WAM is probably superior in handling nondeterminate predicates.

Program	fib(26)	fib(27)	fib(30)	fib(33)
Dsys	2.1	3.4	14.3	60.8
"C"	1.0	1.6	7.1	29.7
Ratio=$\frac{Dsys}{C}$	2.10	2.13	2.01	2.05

Table 4: Runtime comparison between a compiled Prolog program and a C program. Both compute the Fibonacci number using an identical algorithm.

We mentioned in section 2 that *some* variables in a determinate predicate are *transient* variables which can be translated directly into C language variables local to the procedure corresponding to the predicate. Our compiler is not yet capable of detecting all transient variables. However, it accepts a command line flag which instructs the compiler to look for specially annotated variables (i.e., variables with a name starting with T_) in the program and treat them as transient variables. Table 4 shows the runtime of the *fib* program optimized by using transient variables, and compares the time against that of a comparably hand-written C program which implements the same algorithm[5]. The compiled Prolog program is about a factor of 2 slower than the C program. We attribute the difference to indirect variable access and, to a less degree, variable type checking by Prolog.

We select a few sufficiently large programs (i.e., programs with a sequential execution time > 5 seconds) which have a chance to be substantially sped up by running in a cluster of workstations to test the parallel execution mechanism in our system. These programs are compiled with the "-p" option to generate the parallel executable. In our system, a user program is started with a startup program, *pstart*, which takes as command line arguments the name of the user program, number of machines, and optionally the number of processes to run. Recall that with SOS, one may have an arbitrary number of processes running. The default number of processes is the same as the number of machines. Note that a PVM daemon process [7] is running in the background in every machine in the cluster when *pstart* is invoked. The elapsed time taken in the experiment includes the time to start up the user program, but does not include the time to start the daemon process. Figure 6 gives speed-up curves from the experiment. Speedup factor is computed as the average sequential elapsed time divided by the average parallel elapsed time (on DEC5000 workstations). All of the DEC5000 stations have 16 MB main memory. The sequential runtime on a DEC5000 is generally slower (approx. 10%) than that on a Sparc IPX which has 24 MB memory. These curves indicate that reasonably good speed-up can be achieved. Figure 6 also shows the difference between having one process running per machine, and having a large number of processes per machine. The latter helps balance the workload on machines as explained in section 3. Note that the difference is quite substantial with the *11-Queens* program using 8 machines. The *cube* program, however, has a very shallow search tree (24 potential branches at each node) and is not sensitive to this strategy. *Msort* is the mergesort program used as an example in section 3. The program sorts a list of 12,000 randomly generated integers.

[5] We shall clarify that the algorithm (see section 2) is by no means a good algorithm computing the Fibonacci number. Here we are interested only in studying how close a compiled Prolog program can approach, in terms of speed, to a program written in an imperative programming language using similar algorithm.

Option	Description
-c	Suppress linking.
-g	Turn on C debugging flag.
-I*dir*	Specify include file directory as *dir*. Default is the current directory.
-l*lib*	Link with library *lib*.
-o *name*	Name the executable as *name*. Default is the Prolog source file name without the ".p" extension.
-O	Turn on C compiler optimization.
-p	Produce parallel executable code (to be linked with PVM).
-r	Reverse upper/lower case convention in the Prolog program.
-t	Perform Prolog to C translation only.
-v	Optimize transient variables. Default is no optimization.

Table 1: Compiler options.

5 Experimental Results

A set of benchmark Prolog programs were run to compare the uniprocessor performance of our system with Quintus and Sicstus Prolog. This experiment is done on a Sparc-II IPX workstation when the load is light. The machine has 24 MB memory installed. All Prolog source code can be found in [11]. Most of these programs are topical symbolic processing programs in Prolog. The *msort* (mergesort) and *fib* (fibonacci) programs are included to demonstrate how to incorporate C procedures in a program in order to deal with problems for which Prolog may not be suitable.

Throughout this section we will use the symbol "Dsys" to denote our current compiler, and "Nsys" to denote the compiler without the determinate predicate detection function (for comparison). All timing statistics are in terms of seconds.

System	fib(23)	fib(26)	9-Queens	11-Queens	Cube	Zebra	Pattern	Turtle
Dsys	0.9	3.8	3.3	83.9	4.4	0.9	0.4	2.8
Scistus	2.5	10.4	4.4	121.5	4.6	1.1	0.6	1.8
Quintus	0.7	3.0	2.2	58.5	1.7	0.5	0.2	1.1

Table 2: Run time comparison with Scistus and Quintus

Table 2 shows the sequential run time[4] of the programs. The timing is averaged over five consecutive runs. Variance in execution time appears to be small and is not listed in the table. Our system generally yields an average runtime in between that of Sicstus and Quintus Prolog.

We described in the previous sections how determinate predicates are executed in our system. Table 3 shows the effect of executing determinate predicates on-line. The results reveal that substantial improvement can be achieved by compiling a determinate predicate directly in to a C procedure and then invoking the predicate on-line whenever possible. The few programs which do not yield any improvement are the ones which have no on-line calls at all. Our compiler, by default, detects determinate predicates and makes them on-line calls whenever possible. An interesting observation here is that, without on-line calls, our system could have been slower than Sicstus,

[4]That is the CPU time. Elapsed time is close to the CPU time as few I/O operations, including paging activities, are involved. We do not use elapsed time here as it is hard to measure in the other two systems against which we are comparing. Sicstus is its version 0.6 and Quintus is its Release 3.1.1

inserts the body-conj to a candidate procedure pool (work stack, see section 2). Then, the driver selects the next literal to invoke. For determinate predicates, the allocate-body-conjunction() function is replaced by:

```
...
allocate-args-for-ith-literal(args);
if (call-ith-literal(args) == FAILURE) return FAILURE;
...
```

Determinate predicates are executed on-line: call-ith-literal() invokes the procedure for the ith (determinate) literal in the clause body. Note that if all the body conjuncts are determinate, the procedure's body is a serial of C function calls.

The query of a program is translated into a special procedure with the name QUERY, which creates an initial process conjunction in the same way that we create the body conjunction of a clause. The QUERY procedure is the entry point of the Prolog program. Query and program can be compiled separately.

4.2 The Runtime Kernel

The runtime kernel includes the following components: a search driver, unification primitives, memory management primitives, and system built-in predicates.

The search driver gets the initial conjunction by calling the QUERY procedure which returns the initial query conjunction. Then the driver repeatly executes the following:

1. push the new conjunction into the work stack;
2. select a literal to invoke;
3. if the literal returns FAILURE then backtrack, otherwise repeat step 1 or 2 depending on whether or not a new body conjunction is created by the literal.

The iteration terminates when the initial conjunction is completed. The search driver uses a depth-first search strategy as the default. Prolog cut is supported. In the future we will incorporate more elaborate control strategies such as selective backtracking.

Unification primitives perform unification without the occur check. Some optimization has been incorporated. The compiler discovers special cases that require less extensive runtime check on variable types and produces special codes for them. For example, unification between constants is treated as a simple term matching test without calling the general unification routine. A separate stack (trail) is used to record bindings that need to be recovered upon backtracking. Each stack frame contains an address of a bound variable. Upon backtracking, bound variables are reset to free variables. Again, bindings to variables created in a determinate predicate need not be trailed, as no backtracking would ever occur for the predicate.

A third stack (a heap to be precise) is used for general storage including space for lists and structures. In addition, a portion of the heap is reserved for the SET operator to store results.

4.3 Compiler Options

A list of compiler options that have been implemented are given in Table 1. The compiler can be invoked with these options to produce an object C program or an executable with different characteristics.

No modification is necessary to the kernel in order to run the program sequentially or in parallel. Most parallel control codes are generated by the compiler. Therefore, these codes reside in the object program instead of the kernel. Overhead due to the few extra kernel instructions necessary to support parallel execution is negligible. For this reason, those instructions are made permanent in the kernel.

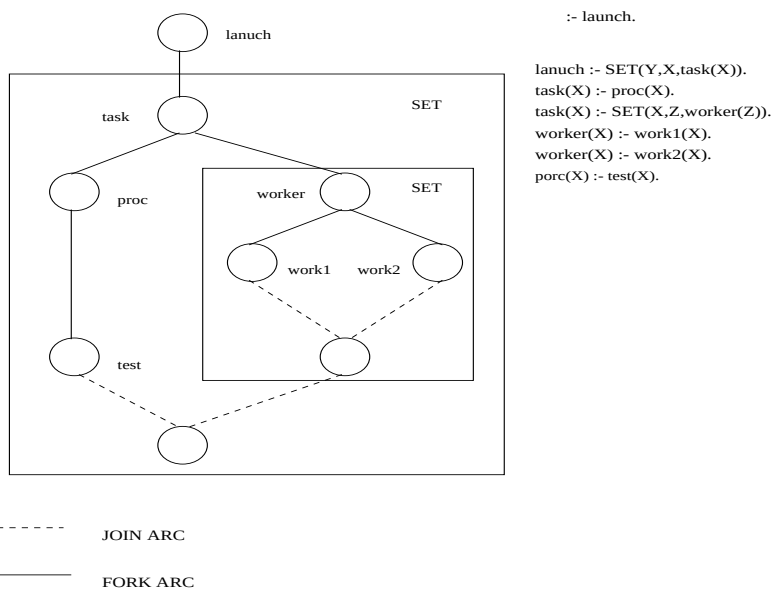

Figure 5: A sample fork-join graph and the Prolog program describing the graph.

```
    switch(clauseNumber)
    {...
    case i: ret-val=predicate-clause-i(argument-array, body-conj);
            if(ret-val ==SUCCESS) return SUCCESS;
            else {
                recover-environment();
                next-clause(&clauseNumber);
                }
    ...
    }
    return FAILURE;
}
```

Record-environment() records the state of the system (e.g., saves registers, stack pointers, etc.). Recover-environment() resets the stack pointers and undoes the variable bindings which occured in predicate-clause-i(). Predicate-clause-i() is the procedure for the ith clause of this predicate. For every alternative clause the compiler produces a procedure, called an AND procedure, which does unification and creates the body conjunction of the clause. A body conjunction is a list of procedures that corresponds to the body of the clause. The format of a standard AND procedure is:

```
predicate-clause-i(args, body-conj)
{
    ...
    allocate-ith-head-args(headarg[i]);
    if(unify(headarg[i], args[i]) == FAILURE) return FAILURE;
    ...
    allocate-body-conjunction(body-conj);
    return SUCCESS;
}
```

When the head unification succeeds, allocate-body-conjunction() generates the clause's body conjunction. The address of the conjunction is put in body-conj. The AND procedure then returns SUCCESS. The OR procedure that invokes this AND procedure also return SUCCESS. Control is thus turned back to the runtime driver, which

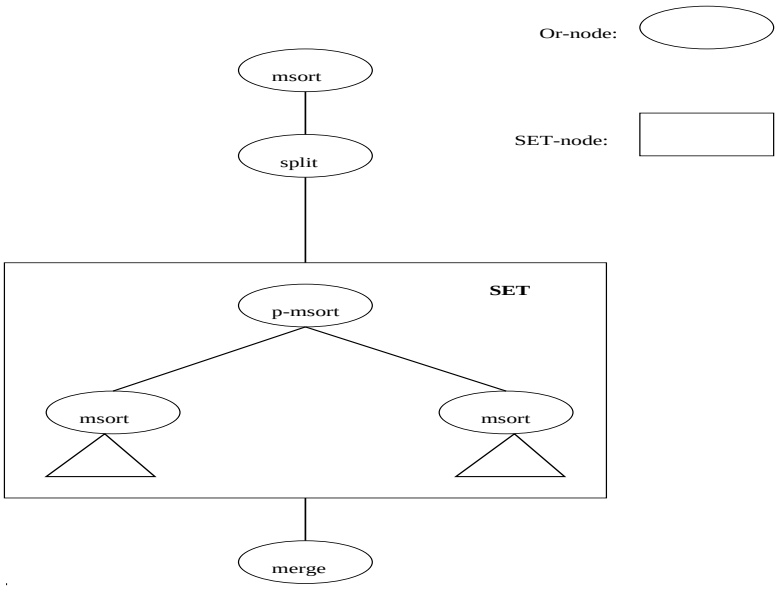

Figure 4: The search tree for the merge sort program.

Definition 3.2.1 SET Node: a SET node in a search tree represents the whole subtree generated by the predicate being computed in the set. A SET node can occur in any place in the tree just as if SET were a predicate.

Figure 4 depicts the search tree generated by the *msort* program. A search tree that contains SET nodes will be refered to as an OS-tree, in order to distinguish it from an Or-tree. It can be seen that SOS also applies to OS-trees. The only difference is that any process visiting a SET node will wait for others and join their results before proceeding further down the tree.

With the SET operator, one can easily encode parallel processes that are often represented as a fork-join graph. The coding is very natural, as illustrated in Figure 5. This allows parallel computation that observes the fork-join model to be conveniently written without exposing the programmer to the tedious communication protocols at the join phase. On the other hand, the SET operator can still be understood as computing all solutions to a given goal, should one choose not to use the notion of (concurrent) processes. Furthermore, programs written with the SET operator can be developed and verified on a sequential machine to eliminate logical errors before having them executed in parallel.

4 Implementation

Our system consists of two major components, a compiler and a runtime kernel. A brief description of these two components is given below.

4.1 The Compiler

The input to the compiler is a Prolog-like program[3] and the original goal. The output of the compiler is a set of C procedures. For every predicate, the compiler produces a procedure called OR procedure. An OR procedure selects one clause for the predicate upon an invocation. A standard OR procedure is of the format:

```
predicate-name(argument-array, body-conj)
{
    record-environment();
```

[3]Horn-clause programs are fully supported. Prolog built-in predicates are not yet fully supported. Other Prolog syntax such as the convention on lower/upper case characters is observed.

The merge sort program in Prolog:

```
msort([X],[X]) :-   !.
msort(Xs,Ys):-      split(Xs,Lts,Bgs),
                    msort(Lts,Ls),
                    msort(Bgs,Bs),
                    merge(Ls,Bs,Ys).
```

The program is transformed into:

```
msort1([X],[X]) :-  !.
msort1(Xs,Ys):-     split(Xs,Lts,Bgs),
                    SET([(1,Ls),(2,Bs)],LL,p_msort((Lts,Bgs),LL)),
                    merge(Ls,Bs,Ys).
p_msort((L,B),(1,LL)) :- msort1(L,LL).
p_msort((L,B),(2,LL)) :- msort1(B,LL).
```

Figure 3: A Merge Sort Program

which the interconnection network is slow compared to the CPU, and massively parallel multiprocessors, in which a large number of processors are involved and the scheduling algorithm may take time proportional to the number of processors in the system to decide from where and in what way to migrate a task. For these systems, we believe that SOS is a better alternative, despite the redundant computation involved.

3.2 SET as a Concurrency Control Construct

A *set* operator is used in Parlog [5] to compute all solutions to a predicate. The *bagof* operator in Prolog has similar function. We implemented a similar operator, SET, whose format is as follows:

$$SET(finalset, template, goal(input, template))$$

The SET operator evaluates the predicate *goal* with input variable *input*, and then returns all solutions with regard to the variable *template* in the variable *finalset*.

In addition to adhering to the above semantics, we make explicit the following:

If there are more than one processes computing the set, every process will wait at a barrier associated with the set after it complete its share of the computation, results from different processes are joined, and the final set is made available to all the processes.

Here the term "join" means combining the results from individual processes in such a way that *finalset* resembles the one obtained from a sequential execution. In our implementation, however, ordering of elements in the *finalset* may not be preserved in the same way as in Prolog, due to consideration of efficient implementation.

The proposed condition forces the SET operator to behave similar to that of a fork-join pair in procedural languages, with all the communication details encapsulated. Note that the SET operator still computes all solutions to the predicate that are inside the set. Therefore, the logical reading of a program with SET operators does not change in any way.

The SET operator can be used to express computation that is traditionally considered independent AND-parallelism in the literature. The use of the *bagof* predicate to achieve And-parallelism was studied in [3, 11], and it will not be further discussed in this report. Figure 3 is an example that shows how a merge sort program in Prolog is translated into one that uses the SET operator so as to achieve parallelism in the *msort* predicate. Notice that SET is invoked recursively.

The presence of SET operators in a program does not, in principle, alter the way we view the execution of a program: we still consider it a process of exploring a search tree. The tree now has two types of nodes: regular Or-nodes and the SET nodes. The new type of SET nodes are defined as following:

3. a process repeats step 2, until all alternatives branches it claims are explored.

The universal task distribution rule is a mapping between a set of processes and a set of tree nodes. It is a function F: process id $\rightarrow$ a set of nodes. A tree node is said to be claimed by a process when it is considered a part of the task for the process.

Step 2 in SOS can be repeated because of the following: at the very beginning every process knows that there are N processes visiting the root, and, each process can compute the number of children, k, of the root. The mapping from N processes to the k children is governed by the function F, which is known to all processes. Therefore, a process can determine:

1. which children shall be allocated by itself for further processing, and

2. which children shall be allocated by any other process.

In other words, a process is fully aware of which other processes (if any) will visit the same child nodes it claims. Furthermore, this information is obtained without any dialogue with other processes. Note that a process is uniquely identified by a process ID throughout the execution. Figure 2 illustrates the idea of SOS. The task distribution rule used in this example evenly divides the children of a tree node among processes visiting the node.

With SOS, load balancing is achieved by:

1. restructuring the program to obtain a better balanced search tree;

2. assigning processes to tasks with a statistically even task distribution rule. In our case, we use the equi-partitioning rule which was shown [10] to be optimal under the condition that nothing is known about the tasks before hand, and

3. allowing a greater number of processes than that of physical processors so that the load balancing problem can partially be taken care of by carefully matching processes to processors.

The first two approaches were studied in [11]; results indicate that many generate-and-test programs can indeed be restructured to obtain a better balanced tree, hence achieving a better balanced load distribution. The equi-partitioning rule was proven valid for tasks whose size follows certain probabilistic distributions. The intuition for the third approach is as follows: by assigning an equally large number of processes onto every processor, the weak law of large numbers [16] will take effect evening out the workload on each processor. In addition, some mapping function allows a processor to execute processes that are assigned to tasks (represented by one or more subtrees in the search tree) at different parts of the search space, realizing the effect of sampling the shape of the search tree. One of these functions, for example, is the division function: Processor_id = Process_id *modulo* Processor Number. This function is used in the current implementation. To illustrate how the mapping works, consider the example in figure 2. Suppose we have two processors. By the above mapping function, the first processor (id=0) will execute processes P_0 and P_2, the second processor (id=1) will execute P_1 and P_3. Despite the uneven shape of the tree, each processor still obtains a more or less even share of the workload.

One may observe that there is redundant computation involved with SOS. In fact, the top portion of the tree is computed by many processes. As shown in [11], the more the redundant computation is, the better the scheme can adapt to the shape of the search tree, or in other words, the more robust SOS will be in terms of balancing workload. The issue with SOS is thus the trade-off between *redundant computation* vs. *load balancing*, as opposed to *communication overhead* vs. *load balancing* as in conventional schemes. Given the fact that the difference in speed between CPUs and communication links is increasing (at least for the near future), we believe that the issue of redundant computation vs. load balancing is probably an easier one to deal with. For some architectures, dynamic task migrations may simply be too costly to perform. Such architectures include workstation clusters, in

[2] In previous reports the term "processor" was used. Recent studies suggested that an additional layer of mapping, one that maps virtual processes to physical processors, can yield even better result. This leads to the use of "process" instead of "processor" in our present discussion. Also, we note that a process is not necessarily an operating system process. In the current implementation, they are more like light-weight threads, manipulated solely by our own system.

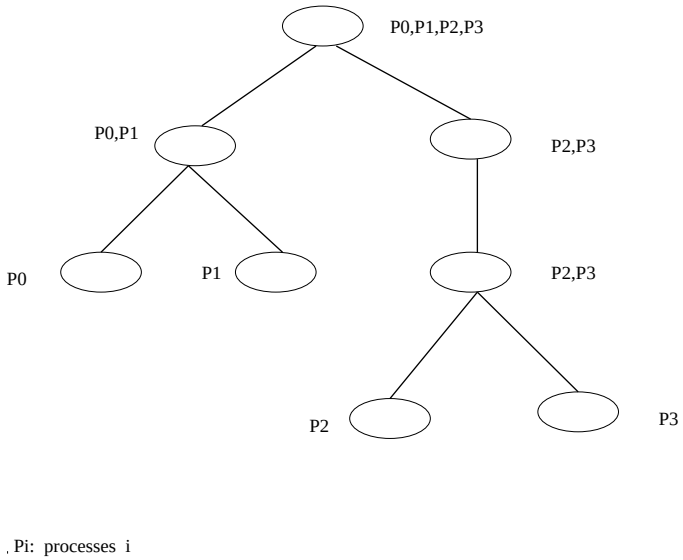

Figure 2: Example of SOS: four processes are cooperating to explore the search tree. Process IDs are listed next to the node visited by the processes.

3 Parallel Execution

A compiled Prolog program can be executed in parallel in our system. Control code for parallel execution is inserted into the object C program by the compiler, so as to minimize changes in the kernel. Parallel execution is currently supported on a DEC5000 workstation cluster connected by Ethernet. Our system exploits Or-parallelism. Independent And-parallelism can be achieved via the use of a SET operator. Interprocessor communication is implemented using the PVM (*Parallel Virtual Machine*) library, developed at the Oak Ridge National Laboratory [7]. Self-organizing scheduling (SOS) [10] is employed as the primary task scheduling mechanism.

3.1 Task Scheduling

Task scheduling in a parallel system usually involves moving computation from one processor to another in order to balance the workload. We call this process dynamic task migration. Dynamic task migration incurs overhead, which may offset the gain from maintaining a balanced load distribution. In a loosely-coupled system such as a cluster of workstations, the performance bottleneck is likely to be the interconnection network. This dictates that only coarse-grained tasks, i.e., those with a relatively high ratio of computation time to communication time, can effectively utilize the system's CPU power while avoiding the communication bottleneck. In other words, dynamic task migration must not occur frequently.

Self-organizing scheduling(SOS), proposed in [10], has been shown to be effective in attaining a relatively large computation granule for a given program, compared to existing scheduling schemes. The basic idea of SOS is to allow each process[2] to decide, independently, its share of a task belonging to a group of processes by following an universal task distribution rule. No shared work queue or interprocessor communication is required by SOS.

Consider a search tree (an Or-tree) that is implicitly defined by a program and a given query. Finding all solutions to the query is deemed as an exhaustive search in the tree. Suppose there are N processes participating in the search. SOS works as follows:

1. every process starts with the original goal, i.e., the root of the tree;
2. at each node, a processor expands all children of the node and claims those belonging to it according to an universal rule;

when fib() is invoked, only one of the three clauses will be evaluated eventually because the first argument in the clause head cannot be matched simultaneously (assuming that the first variable must be instantiated). This kind of predicate occurs often in Horn clause program as it is a standard way to define recursive relations: some clauses define base cases, while the others define recursive rules. Identifying this type of determinism requires elaborate compile-time analysis, which is out of the scope of this paper. In our system, we provide a conditional clause, the IF-THEN-ELSE clause as in Quintus and Sicstus, to allow a programmer to specify this type of determinism. An IF-THEN-ELSE clause is deterministic if and only if both the THEN part and the ELSE part are conjunctions of deterministic predicates. We note that in the current implementation, the condition test in the clause can only be a conjunction of deterministic predicates. With the IF-THEN-ELSE clause, fib() can be written as:

```
fib(N,X) :- IF N > 1
            THEN N1 is N-1, N2 is N-2, fib(N1,X1), fib(N2,X2), X is X1+X2
            ELSE X is 1 .
```

Deterministic predicates and non-deterministic predicates are treated differently due to the fact that nondeterministic predicates generate choice points which can only be handled inside the kernel. In contrast, many deterministic predicates can be executed on-the-fly. The body of a predicate is processed as follows: if a nondeterministic predicate is encountered, the predicate becomes a *candidate* for execution and it is handed over to the kernel where it waits for invocation; if a deterministic predicate is encounted, the predicate will be executed *immediately*[1]. We term such an invocation of deterministic predicates as *on-line* execution, as opposed to *off-line* execution of nondeterministic predicates. On-line execution is significantly more efficient than off-line execution because it avoids the overhead associated with an off-line call, such as the cost of assembling parameters for the call and book-keeping. In addition, some variables in a deterministic predicate are *transient* variables in the sense that they have a life span identical to that of the predicate to which they belong. Transient variables in a predicate can be compiled into C language variables which are local to the C procedure corresponding to the predicate, as opposed to other variables which must reside in the kernel's work stack. This provides additional saving in time for allocating and deallocating space, and more importantly, provides data access locality to which most operating systems are sensitive in order to achieve efficient memory management. A compiled deterministic program (i.e. a program that consists of only deterministic predicates), such as the fib() program written with the IF-THEN-ELSE clause can run at a speed of the same order of magnitude as a comparable hand-written C program, as will be shown later in the experiment section.

2.4 Generic Data Type

To facilitate an efficient interface with C procedures, our system supports a new data type which is normally not available in existing Prolog systems. We call it generic data type. The unification rule for a term of the generic data type is as follows: the term unifies *only* with an uninstantiated variable. While a generic term can be passed around in Prolog predicates, only a C procedure has the ability to manipulate the term, which is essentially an array with information about itself stored in the array header in a fixed format. The information includes the number of elements in the array, and the size of each element. The contents of the array are interpreted by the procedure that operates on the array.

2.5 Binding Method

We do not describe the details of the binding scheme in this paper as there is little novelty in it. We note that our binding scheme provides constant time variable access, and that binding recovery is done via the use of a trail, which is identical to that in WAM.

[1] Strictly speaking, only a subset of DP can be executed on-line, as we still observe the left-to-right literal selection rule of Prolog. Any predicate located to the right of a nondeterministic predicate has to be put off-line. We are looking into the possibility of using other execution models.

- Many deterministic predicates can be executed, as opposed to the abstract machine code approach in which all object code are interpreted.
- Many standard WAM code optimizations, such as eliminating unnecessary memory space for tail recursion, and optimizing parameter passing, etc., can be done at the C code level.

2.2 Execution Control Mechanism

Prolog evaluates literals in a clause body from left to right. This strategy is efficient to implement in the sense that no overhead is incurred at runtime due to literal selection. In fact, the WAM code for a Prolog clause is a fixed sequence of calls that correspond to a left-to-right ordering of literals in the body of a clause. In other words, control is hard-wired in the object program.

An alternative approach is to use a runtime driver. A runtime driver is a combination of data structures and functions which is responsible for determining the next literal to evaluate based on the evaluation outcome of the previous one, and perhaps on other information such as control heuristics and resource availability in the system. This approach had been taken by many systems including the reduced-Or Model [9], and PRISM [8]. The runtime driver approach offers flexibility in literal selection (which is not necessary for Prolog, but is desirable for logic programming in general). However, there appears to be a performance penalty associated with the runtime driver approach.

Our design adopts the runtime driver method. The system maintains a control stack (work stack) which mimics the search in an AND-OR tree defined by the program and the given query. A stack frame stores information about a predicate including its corresponding C procedure, arguments to be passed to the procedure, a counter indicating the next alternative choice point, a pointer to another stack frame where backtracking takes place upon a failure of the current predicate, and a pointer to an entry in another stack (an equivalent of the Trail in WAM), which keeps track of bound variables whose binding needs to be recovered upon backtracking. In addition, a work stack frame keeps control information for parallel execution.

We note that our implementation observes the Prolog left-to-right literal selection rule. The runtime driver is employed becauses it allows us to incorporate other control strategies we may choose, such as selective backtracking. It also appears to be convenient for supporting parallel execution in a way to be described in the next section.

2.3 Deterministic Predicate vs. Non-deterministic Predicates

We classify deterministic and non-deterministic predicates as follows:

Definition 2.3.1 Nondeterministic Predicate (NDP): A predicate is nondeterministic if only if

1. it is defined by more than one clause, or
2. its clause body contains nondeterministic predicate(s).

Definition 2.3.2 Deterministic Predicate (DP): A predicate is deterministic if it is not nondeterministic.

With this definition, DP includes all predicates defined by an unique clause whose body is a conjunction of deterministic predicates. Also, if p :- q, q :- p, and r :- r are the only clauses for p, q and r, they are all deemed to be deterministic predicates.

Notice that our classification of deterministic and non-deterministic predicate is *syntactic*. Some nondeterministic predicates may behave like a deterministic one at runtime. For example,

```
fib(N,X) :-  N > 1, N1 is N-1, N2 is N-2,
             fib(N1, X1), fib(N2, X2),
             X is X1 + X2.
fib(1,1).
fib(0,1).
```

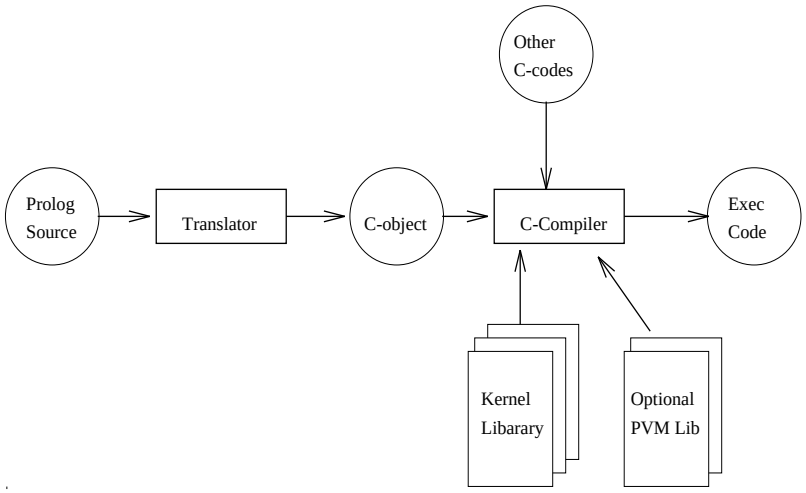

Figure 1: Program Compilation.

- Substantial speedup can be obtained by parallel execution.
- Performance results from running mixed Prolog-C programs appear to be encouraging.

The paper is organized as following: section 2 discusses system design issues; section 3 describes the parallel control mechanism; section 4 briefly outlines the implementation; section 5 gives experimental results; and section 6 concludes our work.

2 System Design

2.1 Target Language

A two-phase compiling approach is adopted in our system. A source Prolog program is first translated into C code. The C code is then compiled into executable code for the target machine. Figure 1 depicts the compilation process.

We choose C as the object language due mainly to the following considerations:

- The C compiler is standard on Unix and it is widely available on other systems nowadays. The object C program(s) generated by the translator are usable to all these systems. Also, the executable program is distributable without a supporting system such as a virtual machine emulator.
- The object C program can be linked with other C modules and libraries. This is a way to utilize existing modular compilation support from the C compiler and to tap into the vast program resources available in the form of C library calls. In our system, every Prolog predicate is translated into a C procedure. A user has the flexibility to decide *when* and *where* to substitute a possibly hand-written C procedure for a Prolog predicate. This is an effective though somewhat pragmatic way to overcome the inefficiency problem of Prolog, while being able to retain the elegant high level logical reading of a program (at least of a part of the program). With our system, one can prototype an application completely in Prolog and gradually fine tune the performance by replacing some predicates with C procedures.
- Architecture-specific program optimization can be handled by the C compiler that is developed specially for the architecture.

We believe that two-phase compiling can generate efficient object code, compared to a standard WAM [1] solution, for the following reasons:

empty). The intuitive interpretation of such a clause, or rule, is that if all B_i's are solved then H is considered solved. Evaluation of the program starts with a query written as $:-Q$, and uses resolution [12] to derive an empty clause should one exist. There may be multiple selections of rules at each resolution step, provided by rules with a matching head. All solutions to the query can be found by exhausting every possible alternative in the program. The answer to the query can be *true* or *false*. But at most of the time, the interesting solution to the query is the bindings on the variables in Q.

Logic programming has the very desirable characteristic of supporting implicit parallelism, that is, it offers a uniform view of computation independent of the underlying system architecture. Parallelism in a logic program can be extracted effectively as demonstrated by many previous studies [2, 4, 13, 14, 15].

However, to satisfy the demand for efficiency, in terms of speed and space, by users who are accustomed to imperative languages, logic programming alone is hardly adequate. It lacks the capability to describe the fine details of certain computation, which may be necessary to attaining satisfactory performance in a general purpose computer [6] . This limitation has caused Prolog to be viewed and treated as a special purpose language suitable only for limited types of tasks, in spite of its advantages such as being declarative and having a concise syntax and well-defined semantics.

We believe that this limitation can be alleviated without losing many of the language's advantages. The belief is based on the observation that one probably should adhere to the principle of separating logic and control in a program only when specifying the top level logic of the program, because it is neither necessary nor beneficial to force a logical language to carry out tasks for which procedural languages are known to be more suitable. Sometimes it may just be natural to encode the problem in a procedural manner, sometimes it is necessary to provide detailed control instructions to the machine in order to achieve good performance. This leads to the idea of a hybrid solution, one that maintains the logic orientation at the high level of a program while allowing procedural code at lower levels if desired. It is not clear at this point where to draw the line between the logical and procedural parts of a program. Our conjecture is that such a line exists in many applications where control coding is imperative at the low level procedures. But for the upper level specification, a logical language is sufficient and probably preferred due to its cleanness and possible parallelism.

While many Prolog programs exhibit parallelism in one form or another, even without explicit annotation, the question remains as to whether the language can effectively describe a parallel algorithm, should one exist. We claim that at least for the class of parallel algorithms representable as fork-join graphs, Prolog in conjunction with a special SET operator is sufficient. A fork-join graph can be described in a Horn clause program with the SET operator, as will be explained in the paper.

Due to the lack of experience in determining which part of a program should be written in a logical language and which part should be in a procedural one, we shall provide users with a means to switch easily from one form to the other. We believe that translating a Prolog predicate directly to a C procedure eases the transformation.

We develop a programming system which fully supports compiled execution of Horn clause programs (i.e., Prolog-like programs), and which allows easy integration with the C language. The system exploits parallelism in the Prolog part of the program, if it is running on a parallel machine. Our goal is to accommodate users who are interested in programming without being concerned with the underneath parallel architecture, and who need the capability to write efficient sequential code when necessary. The following problems are addressed in the implementation:

- how to generate efficient C code from the Prolog specification.
- how to support parallel execution on a coarse-grained system, i.e., a workstation cluster.
- how to provide an efficient interface with the C language.

The system is fully operational on UNIX workstations and workstation clusters. The following are highlights of our accomplishments:

- As a sequential Prolog system, the speed of compiled execution of programs compares favorably to that of Sicstus Prolog, a well-known non-commercial implementation of Prolog.

A Parallel Prolog Compiler and its Implementation

Wenfeng Li*
Department of Computer Science
University of Maryland
College Park, Md 20742
wenfeng@cs.umd.edu

Zheng Lin†
Department of Computer Science
University of Toronto
Toronto, Ontario M5S 1A1, Canada
zlin@cs.toronto.edu

Abstract

A logic programming system is presented in the paper. The system consists of a Prolog compiler and a runtime kernel. The Prolog compiler translates a Prolog specification directly into C code which may be executed on a sequential workstation or in a workstation cluster. We investigate techniques to generate efficient C code from a Prolog program as well as to provide a convenient interface with external C modules. We implement a parallel control scheme which appears to be particularly suitable for a coarse-grained parallel architecture. Experimental results show that 1) substantial gain in performance can be achieved through proper handling of determinate predicates in a Prolog program; 2) for a sufficiently large program, parallelism can be exploited effectively in a workstation cluster; and 3) an efficient C interface allows us to tackle problems which usually would not be attempted by using Prolog alone. As a sequential Prolog system, the speed of compiled execution of Prolog programs in our system compares favorably to that of Sicstus Prolog.

Keywords: Logic programming, compilation techniques, parallelism, task scheduling

1 Overview

A logic programming system is presented in this paper. The system is implemented on sequential workstations as well as in a DEC5000 workstation cluster connected by Ethernet. The system includes a Prolog compiler, and a runtime kernel. The runtime kernel is used to provide support for the execution of nondeterminate predicates in a logic program. There are a few features in the design that seem to make the system different from most other implementations:

- using the C language as the target language of the compiler.
- supporting parallel execution on a network of workstations.
- providing a mechanism for convenient integration with the C language.

This project is an initial step towards providing a convenient tool for distributed parallel programming. We opt for a logic oriented approach for reasons to be explained later. This paper covers mostly the design and implementation of the *logic programming* part of the project. It also provides experimental results showing the effectiveness of our approach.

First, we consider a logic program to be a set of Horn clauses written as $H :- B_1, B_2, ..., B_n$ where H, the head of the clause, is a positive literal and the B_i's, the body of the clause, are either positive or negated literals (possibly

*

†This research was partially supported by the National Science Foundation under grant IRI-89-16059, by the Air-Force Office of Scientific Research under grant AFOSR-91-0350, and by NSERC (Canada) CRD grant.