Implementing Prolog on Distributed Systems: *n-parallel* PROLOG

Douglas Eadline
Paralogic, Inc.
115 Research Drive, Bethlehem, Pa 18015
(610) 861-6960
deadline@plogic.com

Abstract: A method for distributing Prolog across multiple processors is described. Design criterion and rationale are delineated along with the assumptions behind the methodology. The underlying assumptions are based on the need to provide each processor with maximum sequential work. The parallelization mechanism was implemented using a meta-interpreter that includes new predicates for dependency detection, job queuing, queue solving, and queue maintenance. Debugging and side effects issues are addressed. Benchmark results are also presented.

1. History

In 1988, we began the design of Prolog that would operate in parallel over any number of computing processors. As a commercial company, our goal was to provide a high level tool for systems programmers using parallel or distributed systems. To date, the result of our efforts have produced several prototypes and a commercial version which goes by the name of *n-parallel* PROLOG.
During the course of development, we learned several things about implementing Prolog in a distributed environment and the market for such a tool. A description of our current technology is provide herein.

2. Design Criteria and Rationale

In the initial phases, we made some decisions about the design criteria. Many of these decisions were based upon the need to provide a commercially viable product. In general we designed for conformance with existing standards wherever possible. The following goals were designated for the project:

a. **Compatible with existing Prolog** - We determined that a "new" parallel language would be beyond our ability to commercialize. We decided that Prolog provided a good language to express parallelism and there is an expanding pool of Prolog applications and programmers.
b. **Scalable Executables** - We considered that the need to recompile for a different number of processors would be an excessive burden on the end user. This feature would necessitate a run-time load balance instead of a compile time allocation. The "n" in *n-parallel* PROLOG signifies the ability of our Prolog to run any number of processors (from 1 to "n").
c. **Processor/Environment Independence** - The ability to support many different processors and architectures was beyond the abilities of the company. We therefore decided to use ANSI C [1] as an intermediate language for our compiler and to work within any communication standards that had been

established (e.g. PVM). Providing a portable solution, we also allow customers to protect their software investment if changes occur in the hardware environment.

d. **Extensible** - In addition to portability, we wanted to make sure that the user could embed C code in our Prolog and allow our Prolog to be called from C. The use of a Prolog to ANSI C compiler provided this absolute integration feature.
e. **External Database Interface** - We wanted to allow for large external database integration. While this feature is not fully implemented, we have made provisions in the design of *n-parallel* PROLOG for this feature.

3. Design Assumptions and Conclusions

In order to achieve our goals, we based our design on the following assumptions.

The hardware environment will consists of a processor with local store and at least two communication channels to other processors.

This assumption covers most commercially available parallel processing machines (e.g. nCUBE, SP2, CM-5) and includes workstation networks. It also allows the use of shared memory machines that support messaging.

Therefore, the parallel Prolog system is based on a message passing model and the Prolog program (database) must be replicated on each processor.

Parallelism is based on concurrent searching of branches in the application.

Baring side effects, each node point in the Prolog program is considered to be a candidate for a concurrent search. Node points can be OR parallel or AND parallel (with some restrictions). This type of parallelism is "easy" to identify.

Therefore, the parallel Prolog system will look for independent branches that can be searched in parallel. OR parallelism will be based upon multiple instances of a predicate. AND parallelism will be based upon the resolution of independent sub-goals within a predicate. (Independent sub-goals are those that do not share variables.).

On a single processor, a sequential resolution mechanism (e.g. WAM) is more efficient than a parallel resolution mechanism.

All other things being equal, a parallel resolution mechanism by its nature will require more "book keeping" than a sequential resolution mechanism on a single processor [See Note 1]. This additional overhead often produces single processor results that are sometimes slower than a sequential resolution mechanism for a given program. In general, a complete parallel resolution mechanism is not required for every node point in the program particularly if processors are not available.

In addition, communication overhead which is not present in sequential mechanisms, will be present in distributed methods. Unless some consideration of computation vs. communication time is included in the parallel resolution mechanism, large amounts of overhead can be wasted on "simple" operations.

Therefore, the parallel analysis should be limited and applied only where needed. The bulk of the processing should be done by a sequential resolution mechanism and communication overhead should be kept to a minimum.

Furthermore, since the goal is to have the bulk of the processing done sequentially on individual processors, the parallel "portion" of the program should only contribute to a small amount of overall efficiency and can, therefore, be based on an interpretive runtime analysis.

There will be more branch points than processors.

For many "parallel" applications, there are often many more possible parallel branch points than available processors. Due to the cost associated with parallelism, it is important to limit the amount of parallelism to the actual number of processors.

Efficient parallel execution using a parallel resolution mechanism assumes an unlimited number of processors and low communications overhead. In a real hardware environment, this is never the case.

Therefore, a load balancing mechanism must be employed so that identification of parallel branches will be limited to the availability of processors. **Enough parallelism must be found to saturate the processor network with highly efficient sequential processing tasks**.

4. Implementation

The above ideas were implemented and tested using several prototypes based on several different hardware platforms. Current work is being performed using the nCUBE 2 parallel computer and networks of SUN workstations. Previous papers describe some of the other aspects of the implementation [2,3,4].

The two key components of *n-parallel* PROLOG are the identification step and the distribution step. Keep in mind, that these steps occur at runtime and, therefore, allow *n-parallel* PROLOG to be implemented on top of an existing sequential Prolog system [See Note 2].

Identification Step

After studying the prototype results, we determined that the identification phase should occur at run time. Although this may seem an inefficient method, our studies indicate that the actual amount of time need to identify enough parallelism to keep the network busy is not very large.

We allow compiled applications to be run in parallel, but, any portion of the code that is to be subject to parallel analysis must be an interpreted module or be delineated with a "parallel" directive. This segregation of sequential and parallel portions allows the sequential portions of the program to be as "fast as possible" using established sequential Prolog compiling technology, and the parallel portion of the program to be efficiently analyzed at run-time.

In addition, we decided to implement the automatic identification step in Prolog as a meta-interpreter. This feature allows the user to modify the identification step or skip it altogether and use the distribution predicates directly in their application. A complete discussion of the meta-interpreter is not provided due to space limitations.

Future plans include the movement of the meta-interpreter into the compilation phase. Before this can occur, however, information about the estimated communication time and estimated computation time for specific hardware platforms must be available to the compiler. It is not possible to make accurate assessments of parallel efficiency unless this information is employed. The present meta-interpreter also provides a flexible environment to test various parallel strategies.

Automatic identification is facilitated by the following predicates in the meta-interpreter. Each processor in the network is running the meta-interpreter and can accept goals or sub-goals to be solved.

> ***parallel(+Query)*** - parallel/1 will search for the first node point in the program that has two or more possible OR or AND search paths. Once a node point is found, the parallel search STOPS and the node point is solved using the distribution step. OR parallelism is checked first, then AND parallelism.

or_parallel(+Query) - or_parallel will search for the first node point in the program that has two or more possible OR parallel search paths. Once a node point is found, the parallel search STOPS and the node point is solved using the distribution step.

and_parallel(+Query) - and_parallel/1 will search for the first node point in a program that has two possible AND parallel search paths. Once a node point is found, the parallel search STOPS and the node point is solved using the distribution step. After the AND parallel predicates have been solved, the rest of the predicate is examined to see if there are any remaining predicates. Any remaining predicates are solved in an OR parallel fashion using the bindings returned from the previous AND parallel solution.

Because each processor tries to identify one level of parallelism, excessive parallel identification is avoided. Once the parallel node in the search tree has been identified, the distribution step takes over. In other words, once each processor finds some parallelism, it tries to solve these tasks as fast as possible and does not look for any more parallel node points.

The following predicates are used by the meta-interpreter to identify parallel portions of the code.

branch(+Instance,+Goal,?Tail) - branch/3 searches the internal database, looking for a specific instance of a predicate. The Prolog database is searched for predicates that can be matched against Goal. The first two arguments of branch must be instantiated at the time of the call. Goal must be a valid Prolog goal representing the head of a Prolog predicate. Instance, an integer of value greater or equal to zero, represents the instance of the predicate to be used in the match with Goal. If the match is successful, Tail is bound to the dereferenced tail of the predicate; if there is no tail, then Tail is bound to true. If the match fails, then Tail is bound to fail. branch/3 will fail only if it cannot find an instance of a predicate to match against Goal. branch/3 will work with all predicates, both user and system. This predicate is used to determine OR parallelism.

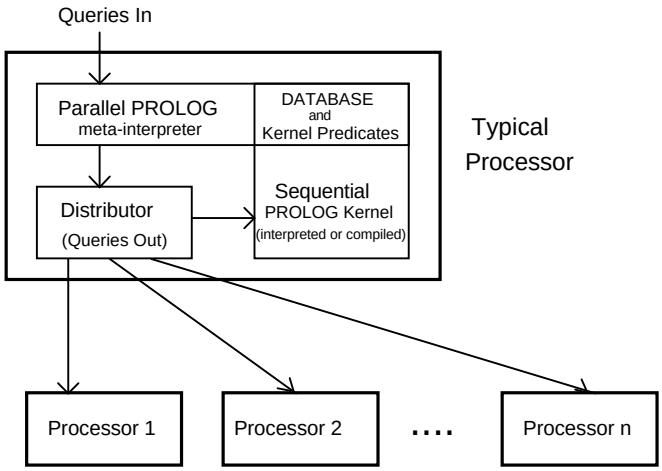
independent(+Fst_Goal,+Sec_Goal) - independent/2 expects both arguments to be instantiated to some Prolog goal. The purpose of this predicate is to check whether both goals share unbound variables, or whether they can be treated as two independent goals. This predicate is used to determine AND parallelism.

Example:

```
% Assume an internal database with these rules:
%   drinks(joe,milk).
%   drinks(tim,beer):- likes(tim,beer).
%   drinks(ann,soda):- there_is(soda).
?-branch(1,drinks(X,Y),Tail).
X = tim
Y = beer
Tail = likes(tim,beer) ->;
no
?-branch(0,drinks(X,Y),Tail).
X = joe
Y = milk
Tail = true ->;
no
?-branch(0,drinks(joe,soda),Tail).
Tail = fail ->;
no
?-branch(3,drinks(X,Y),Tail).
no
?-independent(hello(A,joe,C),hello(B,tim,F)).
yes
?-independent(hello(A,joe,C),hello(tom,tim,A)).
no
```

Distribution Step

During the identification phase when an OR parallel or a restricted AND parallel node point has been identified, a "parallel queue" is filled with these independent paths, represented by "sub-queries". (Sub-queries are dereferenced and represent the current state of the program and, therefore, can be used to launch independent program threads on other processors with the same database.) The queue is FIFO and is given to the distributor. The distributor will try to first distribute tasks on the parallel queue by requesting processors. If after trying to distribute tasks to other processors the parallel queue is not empty, tasks are solved using an internal sequential kernel (compiled or interpreted). To ensure load balancing during sequential operation, the distributor will continually request processors, provided there are tasks on the parallel queue. The processors receiving sub-queries repeat the process until the network is saturated with queues that are solved sequentially. This algorithm allows parallel execution on any number of computing elements, including a single processor. Furthermore, once the network has been saturated, parallel threads are evaluated in an optimum sequential fashion. This process is described graphically below:



The parallel queue is supported by the following set of predicates:

queue_query(?Query,?Receipt,+DatabaseChanges,?Destination) - queue_query/4 places a query in the parallel queue. The parallel queue is a permanent data structure that remains after backtracking. Queued queries remain in the queue after the query has been completed. Query represents a valid Prolog goal. Receipt is an integer that identifies the Query in the queue. Receipt may or may not correspond with the location in the queue. DatabaseChanges is a reserved variable and should always be set to true. Destination is the destination of the query. If Destination is bound, it must be to a valid processor number. If Destination is unbound, then any processor may be used to solve the query.

queue_solve(-Result) - queue_solve/1 solves any unsolved queries in the parallel queue. This query actually calls the distribution mechanism used by *n-parallel* PROLOG. Result is the result of solving all queries on the queue. If they all failed, then Result gets instantiated to fail. If at least one query succeeded, then Result gets instantiated to true. Solved queries remain on the queue until they are specifically dequeued.

dequeue_query(?Receipt) - dequeue_query/1 removes a query from the queue of jobs. Receipt is the receipt bound by the queue_query predicate. dequeue_query will fail if Receipt is invalid. If Receipt is unbound, then the entire queue is cleared. This predicate has no effect on the state of the query in the queue.

queue_length(?Length) - queue_length/1 reports the number of jobs currently in the parallel queue. All Prolog jobs (parallel queries) are kept in the queue until they are explicitly de-queued.

queue_head(?Receipt) - queue_head/1 returns the Receipt for the first query in the queue. If Receipt is bound, then it must match the Receipt in the head of the queue, otherwise it will fail.

retrieve_bindings(+Template,+Receipt) - retrieve_bindings/2 allows the programmer to retrieve the bindings for a query that has been solved in the parallel queue. The bindings are determined by a matching Template with the query. (i.e. Template is the query before it was solved in the queue.) Receipt is the receipt of the query on the queue. The query must be in the parallel queue. retrieve_bindings will backtrack until there are no more bindings found.

rwb(+NodeNumber,+Template,+Receipt) - rwb/3 (return with bindings) is a special predicate that is only used when bindings must be returned from a query on the parallel queue. rwb is appended to parallel queries so that the bindings are returned to the correct processor. Bindings are made using the retrieve_bindings predicate. Since NodeNumber, Template, and Receipt must all be bound, these bindings must take place on the processor sending the query.

node_number(?Current) - node_number/1 binds Current to the number of the processor that is currently executing the query. If Current is instantiated to an integer value, it only succeeds if the query gets executed on the processor specified by Current.

Example:

```
example(X,Y):-
  node_number(NodeNumber),
  % Receipt is assigned by queue_query
  queue_query((likes(joe,X),rwb(NodeNumber,
    likes(joe,X) ,Receipt1)),
    Receipt1,true,_),
  queue_query((likes(bill,Y),rwb(NodeNumber,
    likes(bill,Y),Receipt2)),Receipt2,true,_),
  queue_solve(_),
  retrieve_bindings(likes(joe,X),Receipt1),
  retrieve_bindings(likes(bill,Y),Receipt2).
```

The use of a queue and the subsequent queue predicates allows for the run time load balancing mechanism to be "below" the user. This feature allows the actual distribution algorithm to be tuned to each hardware platform with out changes to the users program.

In addition, it is possible for the user to explicitly use the queue predicates and eliminate the identification step altogether. Working at this level will reduce the portability of the users program.

5. Debugging

Debugging is very similar to sequential Prolog. When a trace or debug predicate is initiated, the user will see a standard execution trace until the parallel queue has two or more queries. Then the queue is displayed and the user is given the option to follow any job in the queue, watch the solution of the queue, or turn off tracing. If the branch being traced is transferred to another processor, then the debugging is automatically transferred to that processor.

Other predicates that are useful for debugging are:

remote_debug(+Node) - remote_debug/1 starts a debugging session on a remote processor. Node must be a valid processor number. The debug session is the same as using the debug predicate.

remote_trace(+Node) - remote_trace/1 starts a tracing session on a remote processor. Node must be a valid processor number. The trace session is the same as using the trace predicate.

6. Side Effects

Support for side effects has been included in *n-parallel* PROLOG. The FIFO queue allows assumptions about execution precedence to be preserved from the sequential domain to the parallel domain.

! - *cut* behaves exactly as in the sequential domain but, in the parallel domain, cut creates a hold situation on a processor until its success or failure can be determined, thus causing a postponement of some parallelism. The effect of the *cut* on a previous branch point depends upon whether the *cut* gets executed. If a failure occurs before the *cut* is executed then alternatives at the choice point can be attempted. However, if the query succeeds past the *cut*, the alternatives are never executed. Thus, *cut* causes a hold situation on a processor until the outcome of the *cut* can be determined.

Assert/Retract/Abolish - The assert, retract and abolish predicates present an interesting challenge for parallel operation. Since all three predicates can effectively change a Prolog program during operation, the Prolog program (database) needs to be updated throughout the network or branches using an "invalid" database which must be re-evaluated.

Currently, *n-parallel* PROLOG uses a database tracking mechanism whereby changes to the database are monitored and the results of any changes are transmitted up to the parent processor. All invalid branch points cause by a change in a query before it is in the queue, must be re-evaluated. Obviously, the re-evaluation of branch points causes a loss of parallelism. Therefore, over-use of *assert*, *retract* and *abolish* can cause a loss of parallel performance.

In a parallel computing sense, overuse of assert, retract, and abolish can be considered "global" data operations and thus require careful programming and algorithm analysis for optimum results.

7. Sequential Performance

The sequential performance of our compiler is given below. The compiler produces ANSI C output which is then compiled with the host C compiler. See reference 1 for more information.

Table 1: Compiled Sequential Performance

Benchmark Program	(SUN,SPARC/40MHz) KLIPS	(SGI,R4000,50 MHz) KLIPS
Naive reverse (30)	80	216
Serialize	46	93
8 queens (naive)	69	151
8 queens (smart)	158	330
Ackerman function	167	352
Zebra	27	52

8. Parallel Performance

Parallel tests were performed on an nCUBE 2 parallel computer. The nCUBE 2 is a message passing hypercube design. Each nCUBE 2 processor runs at 20 Mhz. A simple "parallel" benchmark was constructed to test the basic scalability of n-parallel PROLOG. The benchmark is provided in listing 1 in the Notes and Listings section. The benchmark does not use the meta-interpreter, but rather the program divides the amount of work evenly among processors using the *queue* predicates. As an example, the query "bench(100000)" can be used to run the benchmark.

Table 2: KLIPS Scalability

number of processors	KLIPS	*speed-up
1	15	1.0
4	45	3.0
8	100	6.7
16	227	15.1
32	472	31.5
64	936	62.4
128	1914	127.6
256	3646	243.1

*speed-up = KLIPS for n processors/KLIPS for 1 processor

Preliminary performance of *n-parallel* PROLOG meta-interpreter was tested over a range of processors. Three types of performance were measured - OR parallelism, AND parallelism, and combined OR/AND parallelism. The benchmark was designed to produce a large amount of different types of parallelism. Unlike the above benchmark, the processors are not evenly balanced by the user, but rather the meta-interpreter/distributor attempts to distribute the work evenly among processors. Refinements in the distribution mechanism are expected to improve the performance on larger numbers of nodes.

Since the tests were designed to measure how well the meta-interpreter/distributor works, the entire program was interpreted. Any goal not appearing as an argument to a "parallel/1" predicate can be compiled.

OR Parallelism

The OR parallel performance was tested by entering the query: "`or_bench(100)`". Performance of a larger number of processors remains "constant" because there are not enough parallel jobs in the program to supply the processors. The results are given in following table.

Table 3: OR Parallel Tests

number of processors	time in seconds	speed-up*
1	3085	1.0
4	1286	2.4
8	563	5.5
16	365	8.45
32	181	17.0
64	90	34.3
128	103	30.0

*speed-up = time for 1 processor/time for n processors

AND Parallelism

In the current meta-interpreter the AND parallel mechanism is a binary algorithm. (e.g. it will only detect two independent goals) Therefore, the most speed-up expected per processor is around two times. This performance is by design. When combined with OR parallelism, AND parallelism can cause an exponential explosion in parallel tasks. This situation may overwhelm lower numbers of processors and degrade performance. The AND parallel mechanism is user modifiable and the meta interpreter can be modified to detect more independent goals. The test was run using the query: "`and_bench(200)`". The results are given in the following table.

Table 4: AND Parallel Tests

number of processors	time in seconds	speed-up*
1	316	1.0
4	197	1.60
8	177	1.78

*speed-up = time for 1 processor/time for n processors

OR/AND Parallelism

A combined OR/AND test was devised. The test was run by entering: "`or_and_bench(100)`". The meta interpreter first finds two independent goals: "`or_bench3(X,Y)` *and* `bench(X,yes)`" Each is sent to a separate processor. The or_bench3/2 goal spawns eight sub-goals that can be sent to other processors. When the two goals are solved, then the remaining goal "`or_bench1(Y)`" is solved. The results are given in following table. As in the OR parallel example, increased performance at larger numbers of processors remains "constant" because there are not enough parallel jobs in the program to supply the processors.

Table 5: OR-AND Parallel Tests

number of processors	time in seconds	speed-up
1	2745	1.0
4	1315	2.1
8	490	5.6
16	329	8.3
32	181	15.2
64	117	23.5
128	123	22.3

*speed-up = time for 1 processor/time for n processors

A Real Application

n-parallel PROLOG has been used to search the Human Genome Database [5-6]. The following results were reported for a typical search. An nCUBE 2 was used for the tests.

Table 6: Human Genome Application

number of processors	time in seconds	*speed-up
1	73,380	1.0
4	26,968	2.7
8	11,606	6.3
16	5,503	13.3
32	2,892	25.4
64	1,781	41.2

*speed-up = time for 1 processor/time for n processors

9. Conclusion

The current version addresses much of the design criteria. Parallel operation was successfully implemented using the meta-interpreter. There are several areas that we plan to address in the future. These areas include optimizing the distribution mechanism, improving the assert/retract/abolish performance, adding lower level communication predicates, a distributed cache Systems, implementing fully asynchronous I/O, and moving the detection phase into the compiler.

10. References

[1] "Design and Implementation of a Prolog to C Compiler, Anatholy F. Dedkov and Douglas J. Eadline, Post ILIPS '94 Workshop, November 1994, Ithaca , NY.
[2] "Making Prolog Parallel", Douglas J. Eadline, AI Expert, Vol. 4, No. 7, July 1989.
[3] "Mapping Search Graphs onto Arbitrary Transputer Networks", Douglas Eadline, Transputer Research and Applications 3, Alan S. Wagner ed., IOS Press, Washington, 1990.
[4] "How will you write software for a 1000 node parallel computer?", *Parallelogram*, #52, March/April 1993.
[5] Searls, David B., Prolog and the Human Genome Project, *International Conference on Prolog Applications*, London, April, 1992.
[6] Searls, David B., The Linguistics of DNA, *American Scientist*, Vol. 80 No. 6, Nov/Dec 1992, pp. 579-591.

11. Notes and Listings

[1] At the time *n-parallel* PROLOG was first developed (1989-1990), many parallel resolution mechanisms for message passing systems required much more overhead than a sequential mechanism. In particular, finding idle processors at every potential node point is not possible due to the large amount of communication overhead. A design for a fixed number of processors was considered, but then scalable executables would not be possible.

[2] While in principle it is possible to implement some of the *n-parallel* PROLOG mechanism with on top of other Prolog implementations, it may not be possible to implement all features due to the need to access the internal data structures.

Listing 1:

```
/* par_nrev.pro NPP benchmark Program (compiled portion).
 *
 * COPYRIGHT 1994, Paralogic, Inc.
 * All Rights Reserved.
 *
 */

data([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,
      21,22,23,24,25,26,27,28,29,30]).

:-mode nrev(i,o).
nrev([],[]).
nrev([Element|List],Reversed) :-
  nrev(List,Reversed0),
  append(Reversed0,[Element],Reversed).

:-mode append(i,i,o).
append([],List,List).
append([Element|Front], Back, [Element|List]) :-
  append(Front,Back,List).


bench_mark(Times,Lips) :-nodes(N),
  adjust_times(N,Times,ActualTimes),
  write('Actual Times = '),write(ActualTimes),nl,
  time_loop(Times, LoopTime),
  time_nrev(Times, NRevTime),
  Lips is (ActualTimes*496)/(NRevTime - LoopTime).

adjust_times(N,Times,ActualTimes):-
  Times < N,
  !,
  ActualTimes is Times.

adjust_times(N,Times,ActualTimes):-
  ActualTimes is integer((Times/N))*N.

time_loop(Times, 0).   /* forget about this for now */

%time_loop(Times, Time):-
%  time(Start_time),             /* starting  time */
%  run_loop(Times),
%  time(End_time),             /* finishing time */
%  Time is End_time-Start_time.

%run_loop(Times) :-
%  up_to(Times),
%  fail.
%run_loop(Times).

time_nrev(Times, Time) :-
  data(List),
  time(Start_time),                        /* starting  time */
  pnrev_loop(Times, List),
  time(End_time),
  abstime(Start_time,Start),
  abstime(End_time,End),
  Time is End-Start.

snrev_loop(Times,List) :-
  up_to(Times),
  nrev(List,Rev),
  fail.
snrev_loop(Times,List).

up_to(N) :- N > 0.
up_to(N) :-
  N > 0,
```

```
  M is N-1,
  up_to(M).

abstime(time(Hr,Min,Sec,Hun),Abs):-
  Abs is Hr*3600 + Min*60 + Sec + Hun/float(100).


set_interval(1,M,M):-!.
set_interval(Nodes,M,N):-
  M is N/(Nodes-1).

set_nodes(1,1):-!.
set_nodes(EndNode,Nodes):-EndNode is Nodes-1.

pnrev_loop(Times,List):-
  nodes(Nodes),
  set_interval(Nodes,Interval,Times),
  set_nodes(LastNode,Nodes),
  for(_,1,LastNode),
  queue_query(snrev_loop(Interval, List),_,fail,_),
  fail.
pnrev_loop(_,_):-
  queue_solve(_),
  dequeue_query(_).

bench(Times):-
  bench_mark(Times,Lips),
  write('Logical Inferences/Second = '),
  write(Lips).
```

Listing 2:

```
/* n-parallel PROLOG benchmark Program.
 *
 * COPYRIGHT 1994, Paralogic, Inc.
 * All Rights Reserved.
 *
 */

data([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,
                        21,22,23,24,25,26,27,28,29,30]).
:-mode nrev(i,o).
nrev([],[]).
nrev([Element|List],Reversed) :-
        nrev(List,Reversed0),
        append(Reversed0,[Element],Reversed).

:-mode append(i,i,o).
append([],List,List).
append([Element|Front], Back, [Element|List]) :-
        append(Front,Back,List).


bench_mark(Times,Lips) :-
        time_loop(Times, LoopTime),
        time_nrev(Times, NRevTime),
        Lips is (Times*496)/(NRevTime - LoopTime).

time_loop(Times, Time):-
        clock(Start_time), /*  starting  time */
        run_loop(Times),
        clock(Stop_time),           /*   finishing time */
        Time is Stop_time-Start_time.

run_loop(Times) :-
        up_to(Times),
        fail.
run_loop(Times).

time_nrev(Times, Time) :-
        data(List),
        clock(Start_time), /*  starting  time */
```

```
        nrev_loop(Times, List),
        clock(Stop_time),           /*   finishing time */
        Time is Stop_time-Start_time.

nrev_loop(Times,List) :-
        up_to(Times),
        nrev(List,Rev),
        fail.
nrev_loop(Times,List).

up_to(N) :- N > 0.
up_to(N) :-
        N > 0,
        M is N-1,
        up_to(M).


bench(Times,Test):-
        bench_mark(Times,Lips),
        succeed(Test).

succeed(yes).

or_bench(X):-parallel(or_bench0(X)).

or_bench0(X):-or_bench1(X).
or_bench0(X):-or_bench1(X).
or_bench0(X):-or_bench1(X).
or_bench0(X):-or_bench1(X).
or_bench0(X):-or_bench1(X).
or_bench0(X):-or_bench1(X).
or_bench0(X):-or_bench1(X).
or_bench0(X):-or_bench1(X).
or_bench0(X):-or_bench1(X).
or_bench0(X):-or_bench1(X).

or_bench1(X):-parallel(or_bench2(X)).

or_bench2(X):-bench(X,no).
or_bench2(X):-bench(X,no).
or_bench2(X):-bench(X,no).
or_bench2(X):-bench(X,no).
or_bench2(X):-bench(X,no).
or_bench2(X):-bench(X,no).
or_bench2(X):-bench(X,no).
or_bench2(X):-bench(X,no).
or_bench2(X):-bench(X,no).
or_bench2(X):-bench(X,no).



and_bench(X):-and_parallel(bench(X,yes),bench(X,yes)).


or_and_bench(X):-parallel(or_and_bench1(X)).

or_and_bench1(X):-or_bench3(X,Y), bench(X,yes),or_bench1(Y).

or_bench3(X,Y):-parallel(or_bench4(X,Y)).

or_bench4(X,100):-bench(X,yes).
or_bench4(X,100):-bench(X,yes).
or_bench4(X,100):-bench(X,yes).
or_bench4(X,100):-bench(X,yes).
or_bench4(X,100):-bench(X,yes).
or_bench4(X,100):-bench(X,yes).
or_bench4(X,100):-bench(X,yes).
or_bench4(X,100):-bench(X,yes).
```