

DISPO: Distributed Multi-threaded Execution of Prolog Programs

A. S. Mohamed, A. Galal, I. Khalil, K. Sobh, M. Selim

Department of Computer Science
The American University in Cairo
P.O. Box 2511, Cairo - Egypt
Email: sameh@aucegypt.edu

Abstract

DISPO is a Distributed Prolog Virtual Machine for interpreting and executing Prolog programs through distribution and multi-threading. It is a distributed version of a standard Prolog interpreter designed for exploiting OR-parallelism and pipe-lined AND parallelism. Some distinctive features of DISPO is that it speeds up the execution of Prolog programs and generates all possible solutions for a query, i.e. it works as an all-solutions Prolog virtual machine. The effectiveness of its distribution technique and load balancing equations resulted in a 70% average speedup in the execution time of a number of Prolog benchmark programs. These results are presented graphically with a detailed analysis of system performance.

1. Introduction:

The language constructs of Prolog lend themselves to parallelism, since a set of rules that can be used to solve a given query can be seen as a logical ORing, while the literals in the bodies of each of these rules can be seen as a logical ANDing. This has come to be known as inherent AND/OR parallelism (Shizgal 1990). There are many implementations of parallel prolog such as PARLOG, 3DPAM and PADMAVATI Prolog, exploiting OR-parallelism or Restricted OR-parallelism, with several forms of AND parallelism as PURE AND-parallelism, and pipe-lined AND-parallelism (Kacsuk and Wise 1992), (Karlsson 1992). These systems have been developed on either multiprocessor machines with shared memory or on a large number of transputers such as the OPERA machine that is developed on a super node containing 256 processors (Kacsuk and Wise 1992). All these systems have introduced new language constructs to Prolog to allow for the explicit distribution of tasks and their scheduling.

DISPO is the first distributed implementation of Prolog that provides the user with two types of total transparency: actual (distribution on several servers), and virtual (multi-threading on the same server) OR-parallelism. The system is implemented on a UNIX platform with a number of servers connected through an ethernet LAN, each using a multithreaded inference engine and duplicates the knowledge base. The goal is to achieve efficiency regarding speed through the distribution of ORs on remote servers using a load balancing technique that decides on

the fastest servers, as well as the multithreading algorithms used by each server to receive more than one task at the same time and execute them concurrently.

2. Knowledge Base Granularity Calculation

The Knowledge base is made up of a list of rules and facts. Each rule/fact is identified by a serial key. Each rule is made up of a head followed by a list of sub-goals. It is worth noting that in Prolog, the scope of the variables is within the same rule. The complexities of Prolog knowledge bases vary from one application to another. It is not always the case that a large program has to have a high complexity and a small program has to have a low complexity. Programs could get complex by applying a small set of rules on a small set of data recursively.

In order to obtain an efficient distribution, an accurate estimate of the execution time of each sub-goal has to be calculated. DISPO's granularity calculation method takes into account both the inference time for a goal and the unification time (which for one goal may seem negligible). The importance of taking the unification time into account becomes very clear in the execution of programs that has large number of parameters in the literals of its set of rules and facts.

In traditional methods for calculating granularity, the unification time for "foo(z,b)" fact is unity (1). The deficiency of this calculation becomes apparent when facts with say hundred of parameters are estimate to take the same time as facts with only one parameter which is true when it comes to inference time. On the other hand the real weight during the proof process is not only the inference time but the unification time which is calculated first and then the inference time is added to it. This would make a great difference when balancing a knowledge base with facts of different sizes because then the unification time makes a difference. The unification time for "foo(z,b)" fact in DISPO's new granularity calculation method is 2. The predicate "bar(a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p)" is to have 8 times as much load on the processor as the "foo(z,b)" although they are the same number of facts (bodies/Ors). On the other hand, using the regular method of finding granularity, the two predicates would have been estimated to have the same load in the execution time which is not true.

In the case of a rule, the same method is used for each sub-goal of the rule's body.

Example 1:

bar(A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P):-foo(A,L),
foo(B,O), foo(P,J).

$$\text{Granularity} = 16(\text{head}) + 8(1^{\text{st}} \text{ pred.}) + 8(2^{\text{nd}} \text{ pred.}) + 8(3^{\text{rd}} \text{ pred.}) = 40$$

Here we assume that we have four predicates of “foo()” and four predicates of “bar()” stored in the knowledge base. Since the ORs (bodies) form our basic unit of division and distribution, the significance of this calculation method is shown when calculating the granularity of the top level OR, in the next example.

Example 2:

top(A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P):-
foo(A,L), foo(B,O), foo(P,J).
top(A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P):-
bar(A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P),
bar(P,O,N,M,L,K,J,I,H,G,F,E,D,C,B,A),
bar(A,P,B,O,C,N,D,M,E,L,F,K,G,J,H,I).

Granularity of body 1 (OR 1)=

$$16(\text{head}) + 8(1^{\text{st}} \text{ predicate}) + 8(2^{\text{nd}} \text{ pred.}) + 8(3^{\text{rd}} \text{ pred.}) = 40$$

Granularity of body 2 (OR 2)=

$$16(\text{head}) + 64(1^{\text{st}} \text{ predicate}) + 64(2^{\text{nd}} \text{ pred.}) + 64(3^{\text{rd}} \text{ pred.}) = 208$$

3. Multi-threaded Inference Engine

The main element of inference is a **resolver**. Resolver try to substitute rules into other subgoals in attempting to reduce the resolvent to an empty set which denotes the existence of a solution.

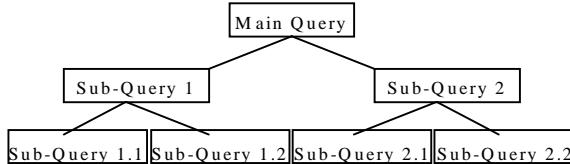


Figure 1: Query breakdown during inference

The inference process uses unification for proving goals. Proving the goals is done by successive reduction of goals till a final result is reached (see figure 1). The inference process starts by having a goal G and a program P. The process of reduction is done by reducing each goal to all its possible bodies and then recursively applying the same process on the bodies. During this process, the most general unifier (MGU- the result of unification) is substituted within the rules every time the unification is checked.

Example 3:

Assume there is a program P:

append([], Ys, Ys).

append([X|Xs], Ys, [X|Zs]):-append(Xs, Ys, Zs).

Consider solving the query append([a,b],[c,d],Ls).

The query is first checked and is found to unify with the second rule with an MGU={X=a,Xs=[b],Ys=[c,d],Ls=[a|Zs]}

The MGU is substituted in the rule being resolved. Then the query being proved is reduced and the loop continues till there are no more rules left. Then the result is returned as the set of substitutions made. The trace of this program will look as follows:

append([a,b],[c,d],Ls)	Ls=[a Zs]
append([b],[c,d],Zs)	Zs=[b Zs1]
append([], [c,d], Zs1)	Zs1=[c,d]

After that, the substitutions are propagated backwards and returned as the result. In DISPO, this example would create three threads for the three iterations of the recursion. Threads are natural environment for executing recursive functions since they have by definition identical execution images of the code, and they are created on the same machine with less network overhead to execute more tasks.

The next example illustrates the first level resolving process with reduction by using a dynamically expanding tree of resolvers that are recursively created.

Example 4:

Illustrates the execution of the query father(X,Y) on the following knowledge base:

father(ahmed,mohamed).
father(aly,mohamed).
father(X,Y):-male(Y), son(Y,X).
son(ahmed,tarek).
male(ahmed).
male(mustafa).

Resolvent 1 : father(X,Y)

Resolvent 2: male(Y) son(X,Y)

Resolver 1 gets reduced to three different resolvers each for a unifying body in the Knowledge Base:

- 1.2: father (ahmed,mohamed)
- 1.3: father(aly,mohamed)
- 1.4: male(Y), son(Y,X)

Resolver 1.2 and 1.3 are reduced to NULL which means that a solution has been found and the two solutions father(ahmed,mohamed) and father(aly,mohamed) are part of the solution. The next level of reduction of the third resolvent (1.4) is resolvent 2.

The first sub-goal in resolvent 2 is then matched with the knowledge base and two predicates are found that match it male (ahmed) and male(mustafa). Leading to respective MGUs {Y=ahmed} and {Y=mustafa}. Thus the MGUs get substituted in the rest of the subgoals in the resolvent and the next level reduction looks as follows:

Resolvent 2.1 then unifies with son(ahmed,tarek) and gets reduced to NULL which means that another solution has been found son(ahmed,tarek).

Resolvent 2.2 on the other hand fails to reduce and stops. In DISPO, this example would create two threads for both levels of resolvers above. The first one terminate execution unsuccessfully.

4. Distribution Equations and Technique

4.1. Exploiting The Fastest Server Technique (ETFS)

The DISPO machine uses a distribution technique which we nick named ETFS that stands for “*Exploiting The Fastest Server*”. The following is the algorithm for the ETFS technique:

1. The loads of the available servers are calculated using a load equation that depends on time stamping for network overhead and instruction time (see section 4.2 next).
2. The tasks’ granularities are calculated and obtained from the knowledge base (as described in section 2 above).
3. Calculation of the sequential time for execution on the local server using the following formula:

$$\text{Sequential_time} = \sum (f(g_i) * \text{Instruction_time} * \text{number_of_inst}).$$

- where $0 < i \leq N$, $N = \#$ of ORs,
- Instruction_time = time taken to execute one instruction on the local server.
- number_of_inst = number of code instructions executed by our inferencing algorithm.
- $f(g_i)$ = complexity of the algorithm for the calculated OR granularity.

4. The lightest task (smallest granularity) is scheduled on the fastest remote server, if the execution time exceeds the sequential execution time calculated, then all tasks are scheduled on the local server, otherwise we move on to schedule tasks on other servers.

5. Then the heaviest task (with the largest granularity) is scheduled on the fastest server. The time to execute this task is calculated as:

$$\text{Longest_time} = (f(\text{OR}) * \text{Instruction_time} * \text{number_of_inst}) + \text{Network_Overhead}$$

6. If the calculated time is found to be less than the sequential time then we move to the scheduling of the rest of the tasks (step 7). Otherwise all tasks will be scheduled on the local machine and executed in a multi-threaded mode.

7. The scheduling occurs so that we try to have the rest of the tasks take nearly the same amount of time on the other servers as will the heaviest task on the fastest server. The algorithm proceeds as follows:

- a. Using the Longest_time (LT) calculated in step 5, we calculate the required granularity (G) needed for an unassigned server using the equation:

$$G = f \left(\frac{(\text{LT} - \text{Net_overhead} - A_T)}{(\text{number_of_inst} * \text{Inst_time})} \right)$$

where A_T is the time taken to execute other previously scheduled tasks on this specific server.

- b. A search is performed within the given list of OR tasks to find a task with a granularity approximately equal to the one required. If no such grain is found we proceed.

- c. Steps a and b are repeated for the rest of the unassigned servers.

8. If some of the tasks are not yet scheduled on a server, we go to step 9, otherwise scheduling is complete and we proceed to sending the tasks for execution.

9. We look for the assigned server with the smallest assigned time and schedule on it the first task in the unscheduled task list, comparing it to the longestest_time and the sequential execution time making sure that it does not execute both, if it does, then the task is scheduled on the local server, having its assigned time updated, then the longest time is compared to the servers’ longest assigned time where it replaces the latter if it is greater than it. Then steps 5 through 7 are repeated with priority to scheduling tasks on unassigned servers until all tasks have been scheduled on a server.

4.2. Load Balancing Equations:

The equations used to calculate the loads of the servers have several parameters:

Instruction Time = time taken to execute a single instruction by the Prolog virtual machine (I).

Network Overhead = time spent over the network to communicate tasks and receive results (NOH).

Granularity = number of predicates need to be inferenced to reach a solution for the head with weights assigned to predicates according to their type AND/OR (g).

F(g) = Complexity Function.

Longest_Exec_Time = Longest execution time of a set of tasks on a server (L_T).

Assigned_Time = Current time spent by a processor to process its tasks (A_T).

Number_Of_Inst = The number of instructions in the algorithm of the inference (NO_IN).

i = The order of a task in the task list..

The Equations using these parameters are:

Network Overhead (NOH) = $(\text{Client_Rcv_T} - \text{Client_Snd_T}) - (\text{Server_Snd_T} - \text{Server_Rcv_T})$

Magic Cookie (MC) = $I + \text{NOH}$.

Sequential_Execution_Time (SET) = $\sum_{i=1}^n (F(G_i) * I * \text{NO_IN})$.

$G = f \left(\frac{(\text{LT} - \text{NOH} - A_T)}{(\text{NO_IN} * I)} \right)$.

NSAT = $\sum_{i=1}^n (f(\text{OR}_i) * I * \text{NO_IN} + \text{NOH})$.

4.3. Example:

The following table represents a sample of the data used by the scheduling algorithm. Using this data we provide a numerical calculation of the speedup achieved by the DISPO virtual machine, while the actual results and graph illustrations are discussed in section 5.

Serv. name	Serv. type	inst Time (μs)	number of Instructions	NOH (ms)	MC	Task #	G
CS	Local	0.489	100	0	0.5	1	19
CS1	Remote	0.155	100	3.59	3.6	2	18
CS7	Remote	0.155	100	5.25	5.3	3	15
						4	13
						5	12
						6	11
						7	10
						8	7

Table 1: Represents a sample of the data used for scheduling tasks and speed up calculations.

The following is a trace of our algorithm using the above data:

1. Sequential Execution Time

$$\text{SET} = 0.489 \times 100 \times (19 + 18 + 15 + 13 + 12 + 11 + 10 + 7) = 5.13 \text{ msec.}$$

2. Time of Task#8 → CS1 = 3.69 msec (fastest remote server), which is less than the sequential execution time, therefore the algorithm continues with the scheduling.

3. Scheduling Task#n (n=1) → CS (fastest server), SAT = 0.93 msec, LET = 0.93 msec.

4. The required grain for CS1 and CS7 fails as their network overhead is greater than the longest execution time (LET).

5. In this example steps 3-4 were repeated with tasks (2,3,4,5) scheduled on CS, SAT=3.76 msec, LET=3.76 sec.

6. The required grain for CS1 $G = (\text{LET} - A_T - \text{NETOH}) / (I \times \text{NO_IN}) = 17$, the closest grain is 11 (task 6) which is scheduled on CS1 having its SAT=3.67 msec.

7. LET = 3.76 msec, a grain is calculated for CS7 but fails as its network overhead exceeds the value of LET.

8. The algorithm schedules Task#7 → CS, SAT=4.25 msec, LET= 4.25 msec.

9. The grain calculation for CS1 and CS7 fails with the new value of LET, and as a result the algorithm schedules Task#8 → CS, SAT = 4.28 msec, LET = 4.28 msec.

10. At this point all tasks have been scheduled.

5. Results

5.1. Loading Times

The following table includes data concerning Prolog files used as benchmarks (see (Kacsuk and Wise 1992) for listing). These benchmarks represent various combinations between breadth of the graph of the knowledge base (high/low), depth of the graph of the knowledge base (high/low) and dependencies between the ANDs of a rule (existant/non-existent, heavy/light).

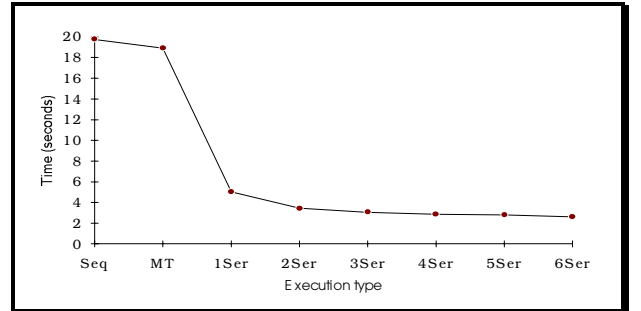
File Name	# F	# R	#Rec Rules	Total Literals	Time (ms)
reverse.pl	2	2	2	7	6.57
colors1.pl	6	1	0	15	15

queens.pl	3	5	3	20	14.7
or1.pl	9	8	0	25	10.5
or3.pl	11	10	0	31	13.1
map.pl	27	2	0	35	22
trial2.pl	7	12	2	34	11.9
colors.pl	6	4	0	42	19.9
usa_map.pl	30	1	0	136	37.8
family.pl	19	10	0	49	11
nlp2.pl	34	8	0	55	19.9
or2.pl	23	25	0	73	23.8
cousin.pl	33	20	0	97	19.3
deep_back.pl	44	40	0	124	39

Table 2: Represents the loading times, time/literal for several benchmarks with varying degrees in the breadth, depth and dependencies of the graph of the knowledge base represented.

5.2. Execution Time

The following graphs represent the execution times for one of the benchmarks. The variations in the execution time results from the variation in the degrees of breadth, depth and dependencies between the literals of the knowledge base used. The graphs compare the sequential execution (on local server) with the multi-threaded execution (MT on local server) and the distributed execution using the local and remote servers (local + n servers), i.e 1 Ser = local server plus one remote server.



Graph. 1: Represents the execution time of the Colors.pl file

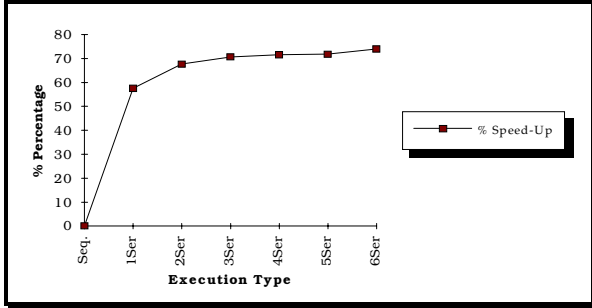
In some graphs we observed that the multi-threaded execution is slower than the sequential one and thus parallelism on one machine hinders execution. This occurred because of the nature of the graph in the deep_back.pl and OR3.pl programs where the graphs have a very high breadth, low depth and no dependencies. Thus the time of forking the threads is larger than the sequential execution time because of the large number of threads needed and the small depth of each task.

5.3. Performance Results

The following graph represents the average percentage of speed up achieved in the total execution time with respect to the number of servers used.

5.4. Execution Time Division

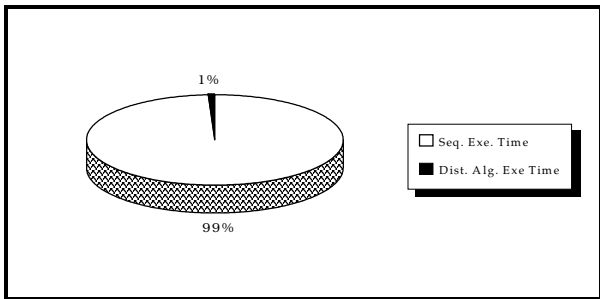
The following graphs represent the time distribution between the major activities done during the sequential, multi-threaded and distributed execution of our system. In the next table we present the average running and job execution time for all of the benchmarks used.



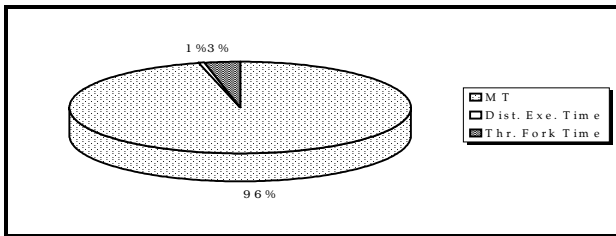
Graph. 2: Represents the speed-up over all benchmarks used vs. the number of servers.

Algorithm and/or Job	Average Time (msec)
Scheduling Algorithm Execution	0.0005
Thread Forking	0.027
Sequential Execution	0.68
Threaded Execution on LOCAL	0.96
Distributed Execution	0.93

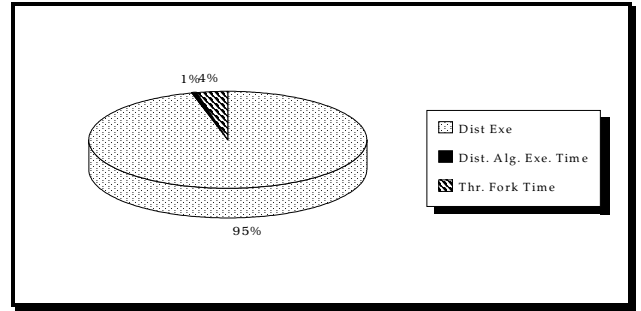
Table 3: Represents the average times for thread forking, Scheduling algorithm execution time, Sequential, Multi-threaded, and Distributed Multi-threaded execution times.



Graph. 3: Represents the percentage of execution time consumed by the scheduling algorithm in the sequential execution type.



Graph. 4: Represents the percentage of time consumed in thread forking and by the distribution algorithm from the multi-threaded execution.



Graph. 5: Represents the percentage of time consumed in thread forking and by the distribution algorithm from the distributed execution

6. Conclusion

From some results, we observe that the cases in which threaded execution takes longer time than sequential or distributed ones are due to the overhead of forking threads. This indicates that our Distributed Prolog Engine operating in the multi-threaded mode on all servers (local, remote) operates with the utmost efficiency on very complex KB as the colors, cousin, and deep_back problems where these problems have either a large breadth or depth or both regardless of the degree of dependencies between the ANDs of the body. Dispo's inference engine when operating in the multi-threaded mode on only the local server has its efficiency decreased because of the thread forking overhead especially with graphs of very large breadth at the top level, no considerable depth and no dependencies as Deep_back, and OR3 problems. When executing simple Prolog knowledge bases with small breadth, depth and low/no degree of dependency between the ANDs using multi-threaded or distributed, multi-threaded DISPO virtual machine's execution time is increased due to overhead. ETFS distribution algorithm and load balancing equations have proved a 70% average accuracy with respect to the actual execution time (sequential, threaded, distributed). By distributing the interpretation of sufficiently large Prolog programs a speedup of 71.5% was achieved using UNIX based machines (SPARC) connected via an ethernet LAN.

7. References

- (Kacsuk and Wise 1992) Kacsuk P., and Wise M. Implementation of Distributed Prolog. John Wiley & Sons.
- (Karlsson 1992) Karlsson R. A High Performance OR-Parallel Prolog System. The Royal Institute of Technology: Sweden.
- (Shizgal 1990) Shizgal I. "The Amoeba-Prolog System." The Computer Journal 33 (2).