Implementing Unification in a Distributed Environment

(Extended Abstract)

Anna Ciampolini , Evelina Lamma, Paola Mello, Cesare Stefanelli

Dipartimento di Elettronica, Informatica e Sistemistica
Universitá di Bologna
Viale Risorgimento 2
40136 Bologna - Italy
Phone: +39 51 6443033
Fax: +39 51 6443073
{anna,evelina,paola,cesare}@deis33.cineca.it

When implementing logic programming in a distributed architecture (see for instance [10, 3, 5, 11]), data structures are distributed among different nodes. Processes do not share memory, thus each interaction and information exchange is performed through message-passing. Message passing communication is more expensive than communication by means of a shared memory. We have to pay attention to reduce the amount of inter-node communication and to maintain quick responses.

Since in logic programming information exchange is obtained by using unification, one of the basic choices when considering a distributed implementation is how and when to access remote data structures in order to perform the unification and reduce the amount of the inter-node communication. In a distributed environment, performing the unification involves sending and receiving messages containing information about the goal arguments.

Since the underlying architecture is fully distributed local data structures and variables can be made accessible to other processes by sending messages. A message can carry either the overall values of variables (infinite level copying) or only remote references to these data structures (0-level copying). Also an intermediate approach can be followed, copying data structures till a certain level, and then sending them to other processes. The trade-off between these choices (full-copying or remote referencing) heavily depends on the underlying architecture, and on the cost of communication in particular.

In a distributed system, where the cost of remote accesses is relatively high, it should be better to copy data for later accesses in many cases. However, a high level copying may cause unnecessary duplication. This consideration has driven the choices adopted in the literature, where a fixed level of copying is adopted depending on the underlying architecture (see for instance [9] and [4]).

However, fixing a specific policy of copying might not be the optimal choice for all cases. Which of the policies of copying is more efficient also depends, in fact, on the particular program and how the arguments are to be used. For this reason, instead of fixing only one particular policy of copying at implementation level , it could be of interest adopting a more flexible and finer policy.

In this paper, we address this issue by implementing a simple logic language with OR parallelism. Since the implementation we consider is fully distributed, we cannot rely on a shared memory or on a common address space to represent variable bindings for OR parallel logic programs. The main problem we face is how and when to access remote data structures to perform the distributed unification. In order to reduce the amount of the inter-node communication, instead of fixing a particular policy of copying, we adopt an implementation scheme where the level of copying can be specified for each goal argument. As a consequence, we introduce in put/set and get/unify Warren Abstract Machine (WAM, [12])

instructions the level of copying as a parameter which can be specified during the compilation phase depending both on the particular program and architecture. In particular, put/set instructions are used for preparing goal arguments and put them into a goal message sent to each child process. When a put_value (put_structure, put_list, or set_value) instruction is executed, a level of copying L can be specified. If this is the case, data structures are built on the message exactly with the level of copying L (L equal to -1 corresponds to an infinite level of copying).

If a remote reference is encountered in building the message, a dereferencing message is sent to the owner of the remote information, specifying the appropriate level of copying. After the put phase, which prepares the goal arguments, call instructions take care of broadcasting the goal message to each child process. The parent process then non-deterministically waits for an answer from one of its descendent, and finally propagates the result RviaS a proceed instruction (backward propagation phase).

When an OR process receives a goal message, it reconstructs the goal arguments on its heap, and performs the head unification by means of get and unify instructions. When a get_value (get_structure, get_list, or unify_value) instruction is executed, a level of copying L can be specified in order to guarantee that the data structures to be unified with the head arguments can be locally accessed. If the data structures built on the local heap have a level of copying lower than L, a dereferencing message is sent to the owner of the remote information, with the appropriate level of copying.

We avoid the problem of common unbound ancestor variables already faced, for instance, in [7, 13, 1] by introducing a suitable data structure for unbound ancestor variables. In this way, by specifying an appropriate level of copying both for put/set and get/unify instructions, inter-process communication can be reduced.

In order to have some results independent of the particular (distributed) architecture chosen for the real implementation, we isolate some basic time parameters. Given a term and a particular policy of copying, we are able to determine the overhead due to remote dereferencing in terms of two basic parameters which depend only on the chosen architecture (i.e., the time spent for message transmission and local memory access). Thus, given a program and a particular architecture we can tune the different levels of copying adopted in put/get instructions in order to reduce the overhead due to message exchange in distributed unification.

Experimental results have been obtained by considering a (tightly-coupled) transputer-based architecture [6], and future work will be devoted in porting the same system on a loosely -coupled one (i.e. a network of Unix workstations). Of course, the real optimal level of copying could be statically determined in all cases only if we know all the nesting of structures to which the program variables will be bound during the computation. To address this issue, abstract interpretation [2, 8] could be a useful technique to be applied.

References

[1] J.S.Conery: Binding Environments for Parallel Logic Programs in Non-Shared Memory Multiprocessor. *IEEE Symposium on Logic Programming*, pp. 457-467. IEEE Computer Society Press, 1987.

[2] S.K. Debray: Static Analysis of Parallel Logic Programs. *5th International Conference on Logic Programming*, (R. Kowlaski, K. Bowen, eds.). Seattle (USA), The MIT Press, 1988.

[3] P. Kacsuk: A Parallel Prolog Abstract Machine and Its Multi-Transputer Implementation. *The Computer Journal*, V. 34, pp. 52-63, 1991.

[4] N.Ichiyoshi, T. Miyazaki, K. Taki: A Distributed Implementation of Flat GHC on the Multi-PSI. *4th Int. Conference on Logic Programming*, pp. 257-275. The MIT Press, 1987.

[5] N.Ichiyoshi, K. Rokusawa, K. Nakajima, Y. Inamura: A New External Reference Management and Distributed Unification for KL1. *Int. Conference on Fifth Generation Computer Systems*, v.3, pp. 904-913. Tokio, 1988.

[6] INMOS, Transputer Reference Manual, Prentice Hall, 1988.

[7] G.Lindstrom: OR parallelism on applicative architectures. *Second International Logic Programming Conference*, pp. 159-170. Uppsala, Sweden, July, 1984.

[8] K. Muthukumar, M.V. Hermenegildo: Determination of variable dependence information at compile time through abstract interprtetation. *North American Conference on Logic Programming.* Cleveland, USA, The MIT Press, 1990.

[9] K. Nakajima et al.: Distributed Implementation of KL1 on the Multi-PSI/V2. *6th International Conference on Logic Programming*, pp. 436-451, (G. Levi, M. Martelli eds.). Lisbon (Portugal), The MIT Press, 1989.

[10] L.M. Pereira , R. Nasr: DeltaProlog: A Distributed Programming Language. *Int. Conference on Fifth Generation Computer Systems*, Tokio, 1984.

[11] S.Taylor, S. Safra, E. Shapiro: A Parallel Implementation of Flat Concurrent Prolog. *International Journal of Parallel Programming*, v. 15, pp. 245-275, 1987.

[12] D.H.D. Warren: An Abstract Prolog Instruction Set. SRI Technical Note 309, SRI International, 1983.

[13] D.S. Warren: Efficient Prolog Memory Management for Flexible Control Strategies. *Int. Symposium on Logic Programming*, pp. 198-202, 1984.