The DelPhi Multiprocessor Inference Machine

W.F. Clocksin
Computer Laboratory, University of Cambridge
New Museums Site, Pembroke Street, Cambridge CB2 3QG

Abstract

In 1986 I proposed a new principle of executing Horn Clause programs with low overhead on multiple processors. This method does not involve the sharing of memory or the copying of computation state between processors. Under certain conditions, the method has the remarkable property of executing a given program in the minimum time theoretically required. Such optimal operation is not always possible, but performance of the implemented system is such as to render it of practical use. This paper describes the experience of implementing the method on a network of processors and of executing programs according to the method. This paper is more a retrospective than a tutorial, and so readers are referred to previous papers for introductory material and technical details.

History of the DelPhi Principle

After a departmental seminar in 1986 in which the distinguished visiting speaker (Professor D. H.D. Warren) described the state-of-the-art in parallel (multiprocessor) Prolog machines (*Cf* [7]), some colleagues (Hiyan Alshawi, Roger Needham, David Wheeler) and I came to the conclusion that current work was neglecting or palliating the memory and communications contention problems that naturally arise when executing Prolog on such machines. We considered that even the extreme case of *random search* of the proof tree resulting from the Horn clause program and goal clause – if conducted quickly enough – might in certain circumstances be more efficient than the sophisticated and intricate methods being investigated at that time (*e.g.* sharing and cacheing of variable bindings, load balancing by arbitration among processors, shared memory computer architecture). The idea behind random search is

to randomly *enumerate* paths through a proof tree and *test* them for success or failure. This is extremely easy to implement, so the question then became how this might be done efficiently and completely, and whether there might exist a natural way of accomplishing this by means of multiple processors. For example, the enumeration task can be distributed, and individual tests may proceed independently on distinct processors.

Following from this discussion, I devised the DelPhi principle [2,4,5], which states that the construction of computation state for each path (from the topmost goal node to a given terminal node) of a OR-only proof tree should be computed by a single processor associated with the path. Given multiple processors, the way to deploy them according to this principle is to enumerate all paths of the proof tree, and to associate each path with a processor. A path is represented by an *oracle*, a list of the nodes along a path. An oracle always begins with the root node of the tree, and terminates with either a success or failure node. The important thing about an oracle is that it is compact and context-free (contains no computation state such as variable bindings). For example, the complete OR-only binary tree of depth n has 2^n oracles, which are all binary strings of length n bits. Because a path of an OR-only tree contains no decisions and does not require state from 'across' the tree, a given processor in possession of the program is in principle capable of computing the path (choosing clauses as directed by the oracle and performing a series of unifications) without further inter-processor communications until the oracle is exhausted. A tree containing AND nodes may be converted to an OR-only forest by standard methods. Assuming the two subpaths of a binary AND-node are to be executed in sequence, the execution may be determined by an oracle string of the length of the sum of the lengths of the two subpaths. In practice, the forest is not constructed, but is tracked as execution proceeds (see Figs. 1,4 in [2]) using a stack.

In practice, the number of oracles will outnumber the number of available processors, and it is necessary to adopt an execution strategy that enumerates and allocates oracles in 'rounds', using information from each round to adjust the enumeration strategy. In principle, if on the first round an oracle terminated by a success node is executed, then the system will have found a solution in the minimum number of steps. Because the overhead required to enumerate and distribute oracles is very small (as they are context-free bit-strings), finding the solution in a minimum number of steps approximates to finding the solution in the minimum amount of time. Whether a successful oracle is allocated in an early 'round' depends on the ratio between the number of distinct paths and the number of available processors, and also on a random variable determined by the location of the goal in the proof tree.

Because in practice there are more paths than processors, strategies are an important part of any implementation, and are discussed in general in [5] and in detail in [6], but will not be further considered here. The strategies discussed in [5] require a limited amount of interprocessor communication, but improved strategies developed in [6] introduce load balancing together with limited backtracking without the need for interprocessor communication. The interesting point remains that even without such strategies, the simple DelPhi principle can give a practical and efficient implementation. The reason is that the outcome of an oracle test gives information about the shape and extent of the proof tree, and thus can influence the enumeration of oracles. For example, if an oracle is found to fail, there is no need to enumerate the oracles that extend it. This is not heuristic, but fact. The basic DelPhi implementation makes use of several such facts even before (heuristic) strategies are considered.

The DelPhi machine, so named because of the famous Oracle of Delphi, was predicted to have the following three characteristics.

(*i*) The best performance would be observed for highly nondeterministic programs of the type encountered by AI researchers. For more deterministic programs (such as matrix multiplication), no speedup would be observed as more processors were added. This is because deterministic problems are AND-heavy, making it necessary to search the major part (if not all) of the proof tree before the solution may be found. In the DelPhi context, this results in oracle strings of a length related to the number of nodes (leaves plus internal nodes) of the proof tree.

(*ii*) For nondeterministic problems, as the number of processors is increased, the machine would exhibit speedups until a certain point (that is, until no more or-parallelism is possible), but that the execution time should then *not increase significantly* as more processors are added without limit. I call this property ‘monotonic execution’ for lack of a better term (monotonic parallel time complexity is not what is being discussed here). This characteristic is unjustifiably neglected in the logic programming community. Most shared-memory systems do not exhibit monotonic execution because performance begins to decrease as processors are added beyond the bandwidth contention limit. Monotonic execution is a priority when considering a future in which the number of available processors is large (*i.e.* 10^n for $n > 3$) and unknown. Monotonic execution is essential for ‘MIP-mopping’: efficient utilisation of otherwise unused processor cycles on networks of thousands of distributed workstations. Monotonic execution is possible for the DelPhi machine because the processors are passive recipients of oracle strings. Processors do not need to communicate

between themselves or solicit work, vices which increase network demand (or shared-memory contention) by (at least) the square of the number of processors.

(*iii*) Given that oracles are compact and context-free, and that the speed of computation does not depend significantly on interprocessor communication, it is feasible to use a relatively 'low tech' implementation based on distributed workstations connected by an Ethernet. This requires no special hardware, and allows harmonious integration with other network users. Furthermore, because the processing of an oracle string is an atomic operation, it can be abandoned and restarted on another processor if the processor it was first assigned to crashes during execution. This robustness property is also important when considering 'MIP mopping'.

The DelPhi machine was intended to offer an ordinary Prolog top-level, and to conceal all the details of parallelism implementation from the user. Therefore, the principal groundrule was to support ordinary Prolog, without the use of programmer-supplied annotations. It was also considered necessary to retain the full non-determinism of Prolog, as this is a characteristic that renders it useful to AI-style programming. It was also considered important to retain extralogical primitives such as the 'cut', even if their use limits the parallelism that can be exploited by the system. Input/output primitives were not to be supplied.

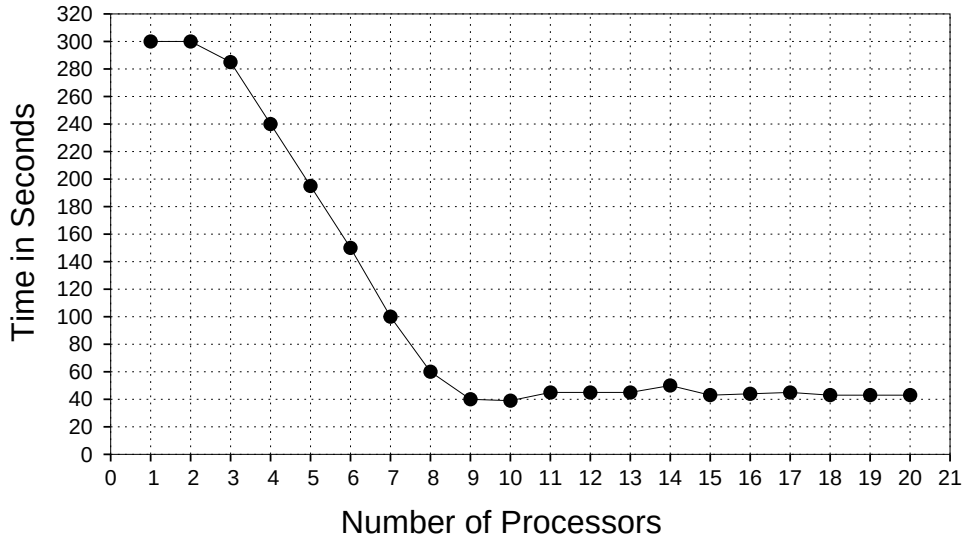
After a number of interpreted uniprocessor simulators (which I implemented in Prolog as it happens), Hiyan Alshawi [1] implemented a DelPhi model in Prolog on a network of five Sun workstations connected by an Ethernet. Oracles were represented as lists of integers, and were transmitted to processors by user-language-level I/O streams. Despite the efficiency limitations engendered by the need for the entire system to be implemented in Prolog, this system did demonstrate speedups related to the number of processors made available.

Or-Parallel Implementation

In 1987 it was decided to proceed with a compiler-based implementation on a network of 20 VAXstation-2000's running Ultrix and connected by an Ethernet. A number of refinements were made to the DelPhi model concerning the shortening of oracle strings when certain deterministic motifs appear in programs, and concerning the treatment of extralogical primitives such as the 'cut'. I devised new WAM instructions for choosing clauses based on the input oracle, and modified the SB-Prolog compiler and runtime-system accordingly. Oracles were represented as blocks of binary data. Carole Klein wrote routines that interfaced the SB-Prolog runtime-

system to the Unix socket I/O level, organised the system on a client-server model, and made it usable by the summer of 1989. Details of the implementation and an investigation of various execution strategies are given in [6].

The system behaved as predicted, with the appropriate speedups, monotonicity, and robustness. Given that the predictions were considered idealistically optimistic, everyone was surprised by the extent to which the actual performance was more favourable than expected. Typical system behaviour is illustrated in the following graph, showing performance of the system on the nondeterministic 8-Queens problem:

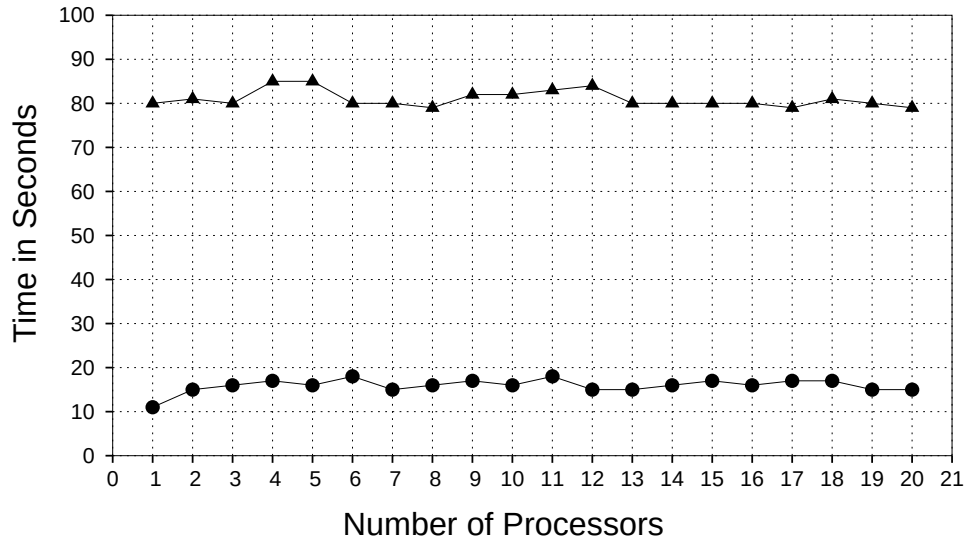


All timings in this paper are ‘wall clock’ times from start of goal to output of final solution, and include all setup costs and logging of results in ASCII files. Although not all researchers take such overheads into account, it is best to include them as they are unavoidable in any real system.

The concept of ‘monotonicity’ employed here is not the same as the mathematical textbook definition: I judge the above data to exhibit monotonicity (up to the known extent of the data) notwithstanding the presence of small fluctuations, notably the ‘blip’ at 14 processors. I interpret these as random fluctuations in system load and network congestion (the network is shared with about a hundred other users). They are not due to systematic increases in computation overhead, and thus are not comparable to the synchronisation and contention overheads obtained in shared memory systems.

Quite a different performance is shown for a deterministic problem: multiplication of matrices. The following graph shows performance on two problems: multiplication of two 20×20 matrices (the lower datapoints depicted as dots) and two 40×40 matrices

(the upper datapoints depicted as triangles):

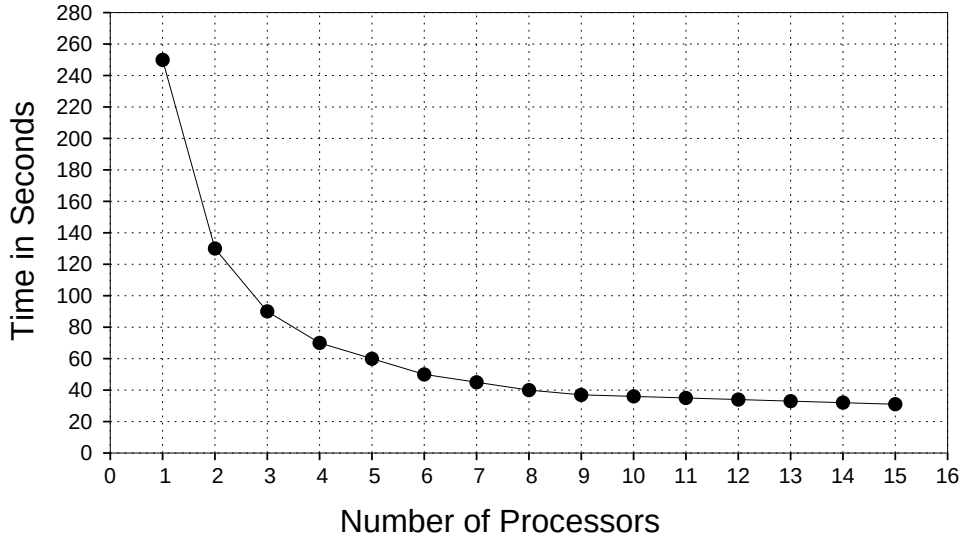


For a deterministic problem of this type there is no overt advantage arising from the use of Or-parallelism, and no speedup is seen when the number of processors is increased. However, there are two key observations to make. First, there is a relatively stable execution speed independent of the number of processors used, and therefore clear evidence of monotonic execution. Second, supposing an average execution time of 15s for the 20×20 matrix and given the complexity of matrix multiplication (n^3), execution times of about 120s would be predicted for the 40×40 matrix if there were no advantage in using multiple processors. However, the advantage in this case is demonstrated by the relatively small increase in computation time (*i.e*. less than a factor of 2^3) when the problem size is doubled. This is because processing relatively longer oracle strings (which is done without interprocessor communication and therefore without interruption) better pays back the cost of oracle management, and because the number of oracles to processed is related to the number of elements in the result matrix (*i.e.* a factor of n^2).

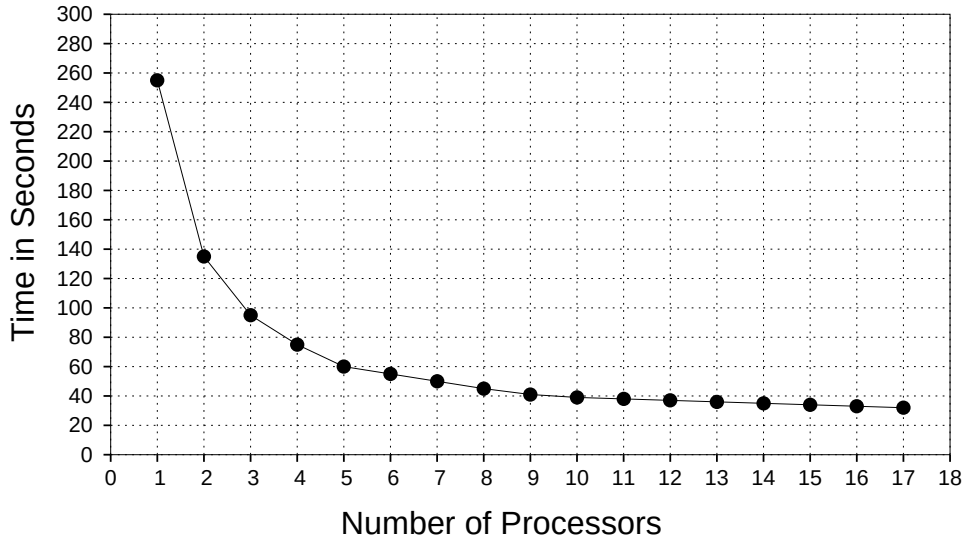
And/Or Parallel Implementation

Given that deterministic programs are not speeded up by adding more processors to the Or-parallel system, it was decided to attempt an And/Or-parallel implementation to determine whether any of the advantages of the DelPhi approach could be applied to deterministic problems in which communication across the proof tree can be used to establish consistent solution sets. Karen Wrench (now Bradshaw) devised and implemented a two-stroke computation cycle, also on a client-server model, in which partial solutions are generated by DelPhi-like oracle testing and sent to ‘answer

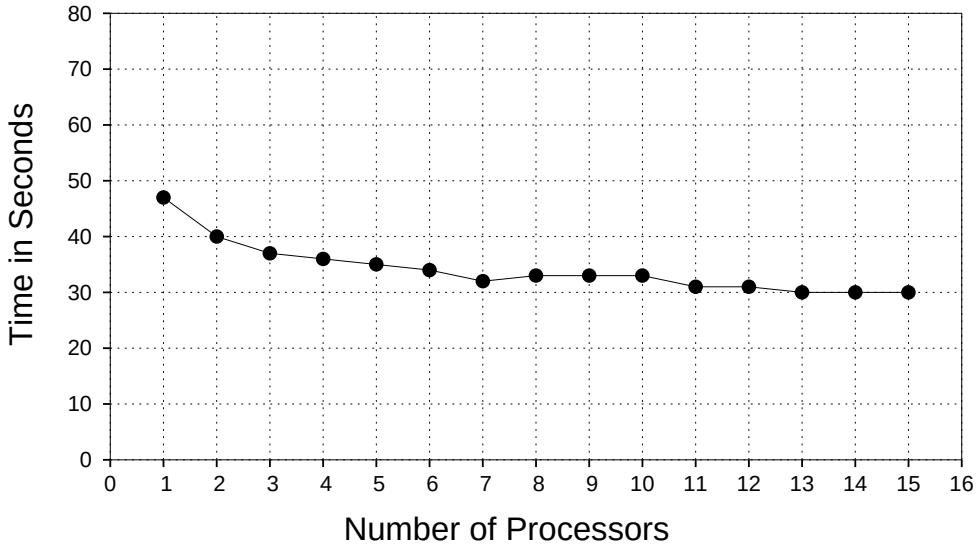
servers' where they are compared to form the solution. It was known at the outset that such a system had built-in performance limitations of the type found in shared-memory systems, but the purpose of the investigation was to determine the feasibility of a distributed and/or-parallel implementation and its characteristic behaviour. The performance on nondeterministic programs was comparable to the Or-parallel implementation for both speedup and monotonicity: the extra overhead introduced by the And/Or-parallel implementation was negligible. The performance graph for the 8-Queens problem illustrates this (in all the following graphs, the curve shown is the mean of ten observations, and thus exhibits the unrealistic smoothness of the mean):



For deterministic problems, the most favourable performance was seen on the Discrete Fourier Transform (DFT), in which a random order-64 complex polynomial is evaluated at the powers of its 64th roots of unity according to the recursive Danielson-Lanczos decomposition (see [3] for how to do this in Prolog):



However, no convincing speedups were demonstrated for other deterministic programs such as sorting and graph colouring. For example, the following graph shows performance on Quicksorting random 1000-element lists:



We suspect that the granularity of the problem is the deciding factor: the distributed element of computation in sorting and map colouring (simple comparisons) is too fine-grained to be expect to cover the cost of distribution, but the distributed element of computation in the DFT (basically following one path through a D-L decomposition) is sufficiently large. Also, graph problems in particular seem to be sensitive to the disposition of processors for reasons that are not entirely understood.

Discussion

It is tempting to criticise the DelPhi method on the questions of 'fairness' of load balancing and recomputation overhead. However, careful consideration shows these criticisms to be based on misleading intuitions about how time is spent in a distributed network of processors. The primary criterion of performance is the time taken for the whole system to execute the program. Questions of 'fairness' of processor load are secondary and irrelevant if they do not engage with the criterion for minimum execution time. Without analysing the program in advance, it is impossible to determine whether a given load balancing strategy is useful or fair, and under real conditions it is not obvious that a balanced or 'fair' load gives the fastest performance. The run-time load balancing strategies reported in [6,7] are heuristic and offer improvements in certain cases, although they must be put into perspective as additions to the simple DelPhi model.

Likewise, the criticism that recomputation leads to inefficiencies fails to appreciate that the *reason* for multiple processors in the DelPhi model is to quickly follow multiple paths that have subpaths in common. The critic might propose an 'efficient' alternative to DelPhi that arranges that unique subpaths are executed only once (or at least minimised) by means of task switching and sharing of variable bindings, but this is a false economy based on an intuition of what is more appropriate for sequential monoprocessors. Using the DelPhi model, common subpaths are followed by distinct processors because it is more efficient to use a unit amount of time to perform calculation than it is to perform communication. On a distributed multiprocessor the overheads of communication, synchronisation, and sharing program state (data structure) required to ensure minimal recomputation far exceed the amalgamated overhead of the DelPhi model.

Current Status

An improved distributed or-parallel system is currently being used on an experimental basis on the laboratory's local area network, where a 50-processor (DECstation 3000's) system is being implemented and tested by Paul Barham as part of a final year undergraduate project. This implementation has identified and remedied shortcomings in the Klein implementation concerning the interface to network communication routines.. By using more and faster processors it is hoped to observe system behaviour in a larger environment, thus gaining a more realistic assessment of the potential of DelPhi for 'MIP-mopping'. Also, it is necessary to have a better theoretical grasp of what the DelPhi principle makes possible, and what its limitations are. It is clear that the expectations of system behaviour engendered by the new implementation have outgrown the broadband network, and an alternative is required. A more comprehensive performance evaluation with standard benchmarks is needed, and it is hoped to undertake this in the coming year.

Acknowledgements

I thank Donald Gaubatz and Brian Rees for arranging an equipment donation from a Digital Equipment Corporation External Research Grant, without which this work would not have been possible. I thank Hiyan Alshawi, Carole Klein and Karen Wrench for their collaboration during this project, and Roger Needham and David Wheeler for fruitful discussions. I especially thank Carole Klein and Karen Wrench for their care in acquiring the performance data. The or-parallel performance graphs

were replotted from data in [6]; the and/or-parallel performance graphs were replotted from data in [9]. Any errors deriving from replotting are entirely my responsibility.

References

1. H. Alshawi and D.B. Moran. The Delphi model and some preliminary experiments. *Proc 5th Conf Symp Log Prog* (ed Kowalski and Bowen), MIT Press, 1578-1589.

2. W.F. Clocksin, 1987. Principles of the DelPhi parallel inference machine. *Computer Journal* **30**(5), 386-391.

3. W.F. Clocksin, 1988. A technique for translating clausal specifications of numerical methods into efficient programs. *Journal of Logic Programming* **5**, 231-242

4. W.F. Clocksin and H. Alshawi, 1986. A method for efficiently executing Horn Clause programs using multiple processors. Technical Report CCSC-3, SRI International (Cambridge Computer Science Centre).

5. W.F. Clocksin and H. Alshawi, 1988. A method for efficiently executing Horn Clause programs using multiple processors. *New Generation Computing* **5**, 361-376.

6. C.S. Klein, 1991. Exploiting or-parallelism in Prolog using multiple sequential machines. PhD dissertation. Reprinted as Technical Report 216, Computer Laboratory, University of Cambridge.

7. D.H.D. Warren, 1987. OR-parallel execution models of Prolog. Technical Report, Department of Computer Science, University of Manchester.

8. K.L. Wrench, 1990. A distributed and-or parallel prolog network. PhD dissertation. Available in summary form as Technical Report 212, Computer Laboratory, University of Cambridge.