DEADLOCK-FREE PARALLEL PROCESSING

A.O.Hill, M.R.C. Brain Metabolism Unit,
University of Edinburgh, Edinburgh.

H.R.A.Townsend, Machine Intelligence Research Unit,
University of Edinburgh, Edinburgh, Scotland.

Introduction

Recent developments In programming languages for A.I. such as Manner (1), Popler (2) and Conniver (3) have mainly been concerned with pattern matching techniques applied to a relational database. Consequently little attention has been paid to the problems of multiprocessing required to operate in a real-time environment.

Consider the problem of manipulating a pile of blocks. If some accident or deus ex machina happens to move a block after a "picture" is taken then this will only he noticed If and when the next "picture" is taken. What we would really like is to be able to monitor the external environment and to roport back if any significant alteration ocoun.

We are interested in the analysis of the electroencephalogram (EEG) where there is an extremely high rate of continually changing data. This rate is so high that even with the current generation of computers the processing speed is the major limitation and the sheer volume of data requires that it should be processed reasonably expeditiously.

Fortunately the problem is much less difficult than would at first appear. Only a few patterns are of Interost, or at least are known to be useful. Many of these pattorns are transient, and in some EEGs they may be absent (a fact which is often relovant). Tho presence of some of the "interesting patterns'* can be detected in real-time but the detailed description of the structure of a short segment of EEG utillses a much larger amount of processing time.

This problem is very similar to the real-timo blocks problem. We cannot stop monitoring the external world since we might miss some relevant event but we can pass information *over* to a collection of parallel processes which, relieved of the necessity to monitor the data can then build up a database and perhaps plan somo active intervention.

In this paper we shall examine the problems of writing programs for cooperating parallel processes , especially tho problems of the deadly embrace (4) where a set of processes cannot proceed because of mutual interference. We shall not at this stage consider the problems of maintaining a relational database which can be altered by several processes.

Systems of Cooperating Parallel Processes

We have been developing methods for constructing systems of parallel processes which will not have deadlock problems. One major requirement is that it should not matter whether the processes are situated in the one area of common core under tho control of the same processor or whether they are distributed over several processors with communicating data links. Each process is considered to be completely autonomous and is written a* a self-contained module which has a number of "ports" through which it can communicate with the outside world and other processes. A system ts constructed by connecting pairs of ports and communication is achieved by the sending and receiving of messages along the routes.

In the simplest version each route can only store one message and a message can only be sent if the route is free. If it is occupied then the process sending the message is held up until the route becomes free. Similarly a message can only be received if there is a message on the route and any process trying to receive a message from an empty route is held up until a message *ia* available. In more complicated versions it would be possible for a route to store more than one message but so far no great need for this has arisen.

The equivalent of Dijkstra's P and V operations (5) appears in the fact that sending and receiving a message must be indivisible operations but the advantage is that since these operations are common to all processes thay can be handled by the system thus relieving the programmer from any need to become involved in the conceptually difficult details.

Analysis of the Complete System

It would be useful if a system of processes could be checked and any possible deadly embraces discovered. Under the formalism presented here this becomes a relatively simple procedure.

Each process can be modelled as a (possibly non-deterministic) finite-state machine with the states as the stages between message passing and the transitions occurring when messages are sent or received. Since each process only communicates with the outside world by means of these messages the details of the code of the process become irrelevant.

This finite-state description can be coded into a simple finite-state language to provida a more succinct representation of the process.

e.g. Bending a message:- PUT 0 0->I;
describes the action of sending a message on route 0 and going from state 0 to state 1.

Similarly s- GET I 2->3;
describee the aotion of receiving a message on route 1 and going from state 2 to ststr 3.

Therefore a simple system of two processes in which the first sends a message to the second and gets a reply can be written as

```
PROCESS ONE
PUT 0 0->I
GET 1 I->0;

PROCESS TWO
GET 0 0->I
PUT 1 I->0;
```

As wlth other methods for investlgating deadly embraces (0) the graphs of the processes can be joined to form one composite graph. The vertices of this graph are the states of the processes and also of the routes. if we restrict the transitions so that only one process may change state at a time then the edges that leave a vertex represent all the currently allowable transitions.

For the system described above:

Process		Route		
ONE	TWO	0	1	
0	0	0	0	(initial state)
1	0	1	0	(proc. ONE sends a message)
1	1	0	0	(proc. TWO receives a message)
1	0	0	1	(proc. TWO sends a reply)
0	0	0	0	(proc. ONE receives a reply)

In this example there is no branching.

It is a simple matter to examine the coding for each individual process and to discover its transitions. The compound states can then be automatically generated. There is no theoretical reason why several processes should not change etate at the same time but in praotice this increases the complexity of the graph and has not as yet been implemented.

Definition of a "Deadly Embrace"

By convention each process starts in state 0 and may have initialisation stages followed by a recurrent sequence of states. if there were no initialisation stages then a definition similar to that given by Llwellyn (7) would be appropriate. A deadly embrace exists if and only if from any state (simple or composite) another state oan be reached from which it is impossible to return to the original state. With the addition of initialisation stages however this no longer remains true since the program never returns to them under normal function and the definition of a deadly embrace muat be tailored to take this into account.

Another point which emerged in testing multi-process systems is the else of the composite graph (8). If the system Is correct the total number of states of the composite process is usually quite small, but when there are errors a inn— effect Is that the graph becomes so big that it cannot be atored. We therefore require some method of analyaing the graph without necessarily generating and storing it completely In order to discover whether we have a complex but correct system or to locate any deadlock states.

This can be done by factoring the graph Into ita strong components, a strong component being a aet of vertices which are mutually reachable, tills factored graph la very much smaller than the unfactored graph and oan more easily be stored.

TarJan's algorithm (SO supplies a technique for finding the strong components and oan be modified ao that deadlock states oan be detected without necessarily generating the complete graph. It also has the merit that computation increases linearly with the number of vertices. The graph is explored in a depth first tree search collecting information on the way. It is not intended to give a formal proof of the algorithm but merely to point out the main features of relevance to the detection of deadly embraces,

The algorithm starts with all nodes unranked and all strong components zero. As the exploration proceeds the nodes are ranked in the order in which they are first met and values assigned to the strong components where possible. All nodes which have been ranked but as yet have no strong component assigned to them are held on a stack. The strong component of a node x la assigned as the minimum rank of any node y which can be reached from x where the strong component of y has not yet been assigned. There is no need to start off with the complete graph, since the neoessary nodes oan be developed as the search proceeds.

This algorithm was implemented using the method described by Knuth (10> and has indeed proved to be very efficient in finding strong components. There are however greater benefits to be had in the discovery of possible deadlock states.

If there are no Initialisation stages in any prooeaa then aa mentioned above there will be the possibility of deadlock if there is more than one strong component in the graph. In order to allow for initialisation stages the strong components can be claased as either terminal or non-terminal i.e. whether it is ever possible to get out of that group of states. Thus whether there arc initialisation atages or not there exists the possibility of deadlock if there are two or more terminal strong components.

We are now left with the problem that although it 5is feasible to check for deadlock states in a multiprogramming »y»tem, this has to be done as a separate analysis and not concurrent with systora operation. While this has proved very useful in some EEG applications (11) it has considerable disadvantages in complex programs where processes may be continually added and deleted and wo have found it necessary to restructure the problem in order to obviate continual reanalysis.

One factor which often led to considerable difficulty was the existence of cycles where one process sent a message to a second process and this second process could either reply directly or send a reply via a third process. The system was restructured to avoid this problem.

A Practical Implementation

The current version is implemented on a GEC 2050 computer which is a mini-computer very well suited to multiprocessing. The basic structure is that communicating processes arc connected as a tree, there being separate trees for non-communicating systems. Kach process may own any number of dependent processes but n process may have only one owner.

Message passing is as described above except that the massages must always be paired i.e. if a message in sont to a process then at some stage a roply must be received and certainly before another message is sent on the same route.

In ordor to accommodato messages of variable length the actual signalling of a message takes only 1 bit of 8torAgo while the buffer for the mossage is located in the dependent process and contains its size.

This storage is then unusual in that the variables located there only have validity at restricted stages of communication between the processes. In the resting state they are valid for the owning message until the PUT flag is set at which time their validity ceases. When the message is received by the dependent process the storage buffer becomes valid for that process and remains so until the roply is sent. The buffer then becomes available for the owning process after the reply is received.

The tree is dynamic in that a process may ask for another process to be attached to it if either that procoss is froo and in core or is held on disc. Similarly when a process has f1nishod with a dependent process it may disconnect it. If the dependent process is not finished then it becomes the root of a new tree and there is a now completely independent parallel system. If it has finished then it bocomos available for garbage collection when more processes require to bo loaded.

A cyclic structure cannot be created beoause a process cannot become attached unless it is currently unattached, a process may only have one "owner" at a time. An informal proof that the tree structured system will not suffer from deadlock states is as follows:*

If there are separate non-interacting tree systems then deadlock in one will be independent of the states of the other trees.

Each tree has a root with no "owner". If a message is sent by this root process to a dependent process then a reply must be received even if this reply only signifies that an error has occurred. This implies that the root process can never be held up trying to send a message. Deadlock states will therefore appear as the absence of a reply, i.e. the deadlock must be produced by the dependent process and not by the root.

Consider a dependent process. The rules are that if a message is received from an owning process then a reply must be sent, perhaps after the procoss in its turn has sent messages and received the replies. The reply may signify that an error has occurred giving information about the procoss that found the error and the type of error, A reply will only fall to be sent if there is an irrecoverable software error in that process which is not by definition a "deadlock". The structure is so arranged that a message exchange is always originated by an owner and completed by the reply (the "principle of politoness"). Since the system is tree-structured this argument can be continued by induction until th© tips of the branches are reached. Thus there will only be deadlock if it is produoed by the terminal processes.

The terminal processes however simply receive a roeSHage from their owner, carry out some computation or input/ouput and then reply. Since the owning process is always expecting the reply the terminal process can never be hold up i.e. the system is deadlock free.

Furthermore since these rules are structural they can be checked syntactically for each process as it is compiled i.e. a dynamic parallol processing system which does not have deadlock states can be constructed.

Limitations

Parallel processing may be achieved in our system by

a) An owner, after instructing a slave to carry out some procedure, continuing to compute in parallel for some time before expecting a reply.

b) An owner issuing several messages to slaves, which will carry out their computations in parallel, and then collecting the replies seriatim.

c) A mixture of these two mechanisms.

The order in which computations will be

completed must be specified In advance aa no mechanism la provided tor accepting replies In unspecified order or for abandoning a computation which la going on for too long (e.g. if the answer has been found by a different parallel branch). We think however that auch mechanisms can be incorporated wlthout destroying the desireable tree structure.

A more fundamental limitation is that resource deadlocks can still occur due to competing independent process systems, perhaps because there is not enough core available for the next process that either system requires, or e.g. aa in the claaaic situation whore one owns the reader and wants the lineprinter while the other owns the llneprinter and wants the reader.

In the completely dynamic case a system of processes may not "know" what resources will be required until a considerable tree structure has been created and there may be little option but to terminate one of the competing systems (even In this event a properly designed error reply should be able to limit the damage to part of the tree, and permit recovery).

Where requirements can be predicted in advance a static analysis can be undertaken. Any set of Interconnected processes forming a system (perhaps dynamic) known to be deadlock-free may be replaoed - as far as the non-deterministic model of the whole set of systems is concerned by a single composite prooess. This process can claim a resource by placing a message on a route (which la equivalent to setting a semaphore) and relinquish it by picking up the message. This new model can be analysed by the methods discussed in the earlier section to discover if deadlock states are possible.

Conclusion

Constructing parallel processing systems out of modular processes which coranunicato by passing messages provides a simple and efficient structure to expand artificial intelligence techniques to the real-time world. Static systems can be checked for deadlock by means of Tarjan's algorithm and by Imposing structure on the message passing dynamic deadlock-free systems can be created.

Acknowledgements

We should like to thank Dr.n.Burstall for pointing out Tarjan's algorithm.

References

(1) Hewitt,C. PlANKERs A Language for Manipulating Models and Proving Theorems in a Robot. MAC Al Memo 168, 1970.

(2) Daviea.D.J.M. POPLER: a POP-2 Planner. MIP-K-89 Department of Machine Intelligence and Perception, University of Edinburgh, 1971.

(3) McDermott,D.V. and Sussman,G.J. The C0NN1VER Reference Manual. A.I. Memo 259 M.I.T. - A.I. laboratory 1972.

(4) Dijkstra.E.W. Solution of a Problem in Concurrent Programming Control. CACM Vol.o,196S,p9.

(5) Dijkstra,E,W. Cooperating Sequential Processes. Programming Languages. F.Genuya (Ed.) Academic, New York,1966,

(0) Gilbert,r• and Chandler,W.J. Interference between Communicating Parallel Processes. CACM Vol. 15, 1972,p427.

(7) Llwellyn,J.A. The Deadly Embrace - a Finite State Model Approach. The Computer Journal Vol. 16,1973, p223.

(8) Hill,A.Q«₉ and Townsend,H.R.A. The Computer Journal Vol. 18,1975, p94.

(9) Tarjan,R. Depth-first Search and Linear Graph Algorithms. S1AM J. Computing Vol.1,1972,pi46.

(10) Kn\jth,D.£. Structured Programming with Go To statements. ACM Computing Surveys, Vol. 6, 1974, p261.

(11) Townsend,H.R.A. In the footsteps of the Amoeba or J&ilti-Processing Without Tears. MIP-R-97, School of Artificial Intelligence, University of Edinburgh, 1972.