Chart Parsing for Loosely Coupled Parallel Systems

Henry S. Thompson

Department of Artificial Intelligence
Centre for Cognitive Science

University of Edinburgh

Acknowledgements

The work described in this paper was made possible in part by the use of equipment made available by the Rank Xerox University Grants Programme, BBN Labs Inc., Artificial Intelligence Ltd., Xerox Palo Alto Research Center and Intel Scientific Corp.—to all of them my thanks. None of them are in any way responsible for the opinions expressed here.

0. Introduction

Of the parallel systems currently available, far and away the most common are loosely coupled collections of conventional processors, and this is likely to remain true for some time. By loosely coupled I mean that the processors do not share memory, so that some form of stream or message-passing protocol is required for processor-processor communication. It follows that in most cases the programmer must make explicit appeal to communication primitives in the construction of software which exploits the available parallelism. Even in shared-memory systems, the absence of parallel constructs from available programming languages may mean that appeal to a similar communication model may be necessary, at least in the short term.

Although not ideally suited to loosely coupled systems, the general problem of parsing for speech and natural language is of sufficient importance to merit investigation in the parallel world. This paper reports on explorations of the computation-communication trade-off in parallel parsing, together with the development of an portable parallel parser which will enable the comparison of a variety of parallel systems.

I. Parsing for Loosely Coupled Systems

Given the prevalence of loosely coupled systems, although one might suppose that shared-memory parallelism offers greater scope for the construction of parallel parsing systems, and parallel chart parsers in particular, none-the-less it is a good idea to look at what can be done in the loosely coupled case.

Loosely coupled parallel systems can be expected to do best, that is, show a nearly linear (inverse) relationship between solution time and number of processors, when the problem at hand is (isomorphic to a) tree-search problem with large initial fan-out and compact specifications of sub-problems and results. In such problems, the ratio of communication to computation is low, so the loose coupling does not significantly impede linear speed-up. Large problems can be broken down into as many pieces as there are processors, cheaply distributed to them, and the results cheaply returned.

Parsing of single sentences is not obviously suited to loosely coupled parallel systems. Whether one attacks single-sentence parsing by some form of left-to-right breadth-first parse, or by some form of all-at-once bottom-up breadth-first parse, very high communication costs would seem to be involved. The only hope would seem to be to pursue the latter route nevertheless, and see whether the communication costs can be brought down to an acceptable level. There are a number of different dimensions along which one might try to parallelise the parsing process, but insofar as they involve the distribution of sub-problems, they are highly likely to require the representation of partial solutions. Since this is a primary characteristic of the active chart parsing methodology, my investigations have focussed on parallel implementations of active chart parsers.

II. Parallelism and the Chart

We start with the observation that chart parsing seems a natural technique to base a parallel parser on. Its hallmarks are the reification of partial hypotheses as active edges, and the flexibility it allows in terms of search strategy, and it would seem straight-forward to adapt a chart parser doing pseudo-parallel breadth-first bottom-up parsing into a genuinely parallel parser. Indeed with a shared-memory parallel system, the BBN Butterfly™, I have done just that, and the result exhibits the expected linear speed-up. The approach used was simply to allow multiple processors to remove entries from the queue of hypothesised edges and add them to the chart in parallel, performing the associated parsing tasks and thereby in some cases hypothesising further edges onto the queue. Locks were of course required to prevent race conditions in updating the chart and edge queue, but instrumentation suggested that there was rarely contention for these locks and they had little adverse impact on performance.

Clearly this approach would not be appropriate in the loosely coupled case. One could of course use some system which supports virtual shared memory to implement a shared chart and edge queue. But this would defeat the whole purpose, as the parser would be serialised by the processor responsible for maintaining the shared structures. What I have explored instead is retaining the same granularity of parallelism, namely the edge, but accepting that at least some of the chart itself will have to be distributed among the processors.

III. Distributing the Chart

I have explored the approach of distributing the chart among the processors in several implementations of a chart parser for the Intel Hypercube™, a loosely coupled system, and for a network of Lisp workstations. A memory-independent representation of the chart is used, allowing edges to be easily encoded for transmission between processors. The chart is distributed among the processors on a vertex by vertex basis. Vertices are numbered and assigned to processors in round-robin fashion. Edges 'reside' on the processor which holds their 'hot' vertex, that is, their right-hand vertex if active, left-hand if inactive. From this it can be seen that once a new edge is delivered to its 'home' processor, that processor has all the edges required to execute the fundamental rule with respect to that new edge. Each processor also has a copy of the grammar, so it can perform rule invocation as necessary, and a copy of the dictionary, so that once the input sentence is distributed, pre-terminal edge creation can proceed in parallel.

The following three figures illustrate the distribution of vertices and edges for a simple example sentence and grammar, assuming a three processor system.

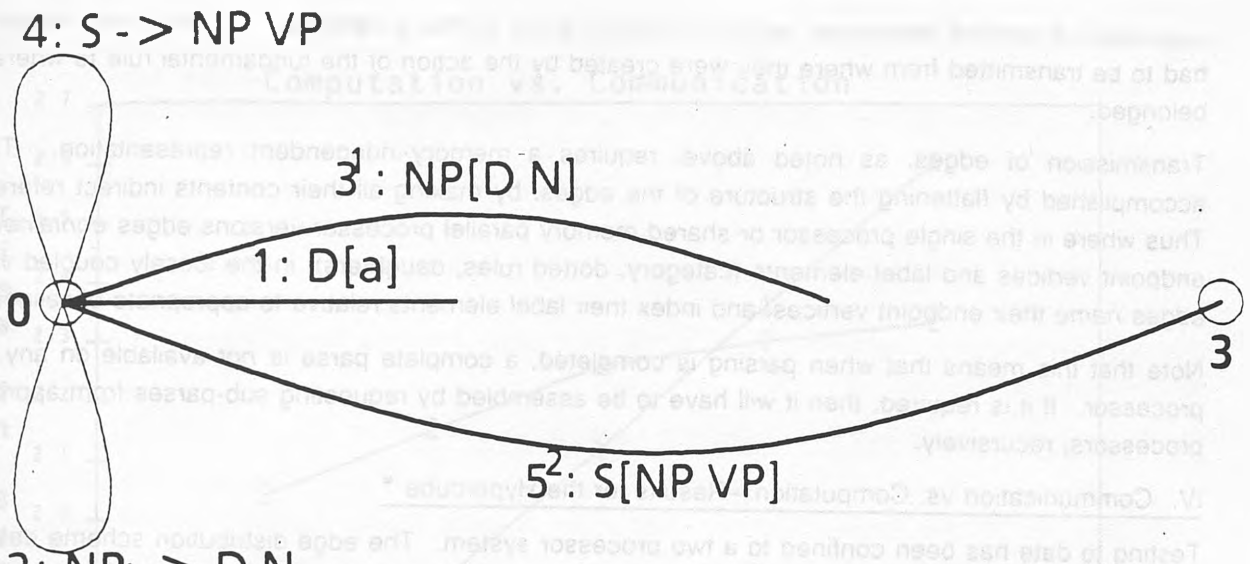

Figure 1a. Chart portion resident on processor 0

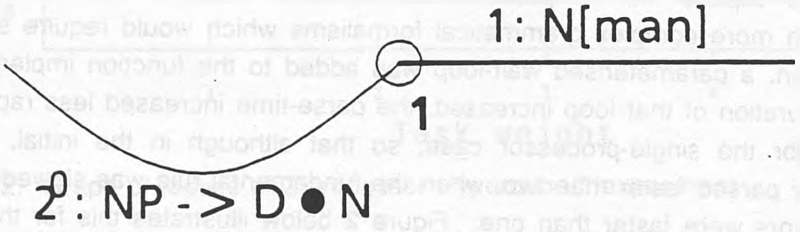

Figure 1b. Chart portion resident on processor 1

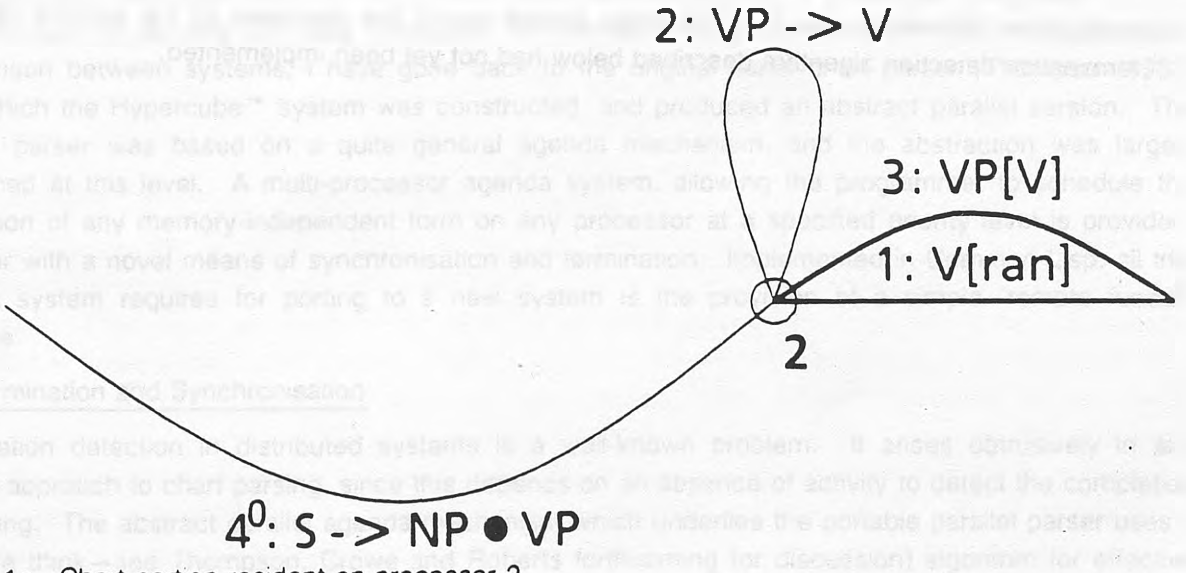

Figure 1c. Chart portion resident on processor 2

Vertices are numbered circles. Edges are thin if active, thick if inactive, and their contents are noted. They are numbered on a per processor basis. Those with superscripts, e.g. 4^0, are ones which originated on another processor, whose number is given by the superscript. Of the eleven edges, four had to be transmitted from where they were created by the action of the fundamental rule to where they belonged.

Transmission of edges, as noted above, requires a memory-independent representation. This is accomplished by flattening the structure of the edges, by making all their contents indirect references. Thus where in the single processor or shared memory parallel processor versions edges *contained* their endpoint vertices and label elements (category, dotted rules, daughters), in the loosely coupled version edges *name* their endpoint vertices, and index their label elements relative to appropriate baselines.

Note that this means that when parsing is completed, a complete parse is *not* available on any single processor. If it is required, then it will have to be assembled by requesting sub-parses from appropriate processors, recursively.

IV. Communication vs. Computation—Results for the Hypercube™

Testing to date has been confined to a two processor system. The edge distribution scheme described above was installed into an existing serial chart parser. Considerable care has been taken within the limits imposed by the host system communications primitives to keep communication bandwidth to a minimum (approx. 100 bits/edge in a single packet). Even with a relatively trivial grammar and lexicon and simple sentences of limited ambiguity, two processors are faster than one under some circumstances. In order to explore the computation/communication trade-off, and to simulate the operation of the system with more complex grammatical formalisms which would require substantially greater per-edge computation, a parameterised wait-loop was added to the function implementing the fundamental rule. As the duration of that loop increased, the parse-time increased less rapidly for the two processor case than for the single-processor case, so that although in the initial, un-slowed, condition, a single processor parsed faster than two, when the fundamental rule was slowed by a factor of around four, two processors were faster than one. Figure 2 below illustrates this for the sentence *The orange saw saw the orange saw with the orange saw* with a standard grammar which allows for PP attachment ambiguity and a lexicon in which *orange* is ambiguous between N and A and *saw* is ambiguous between N and V. The times plotted are to the discovery of the second parse, as the termination detection algorithm described below had not yet been implemented.

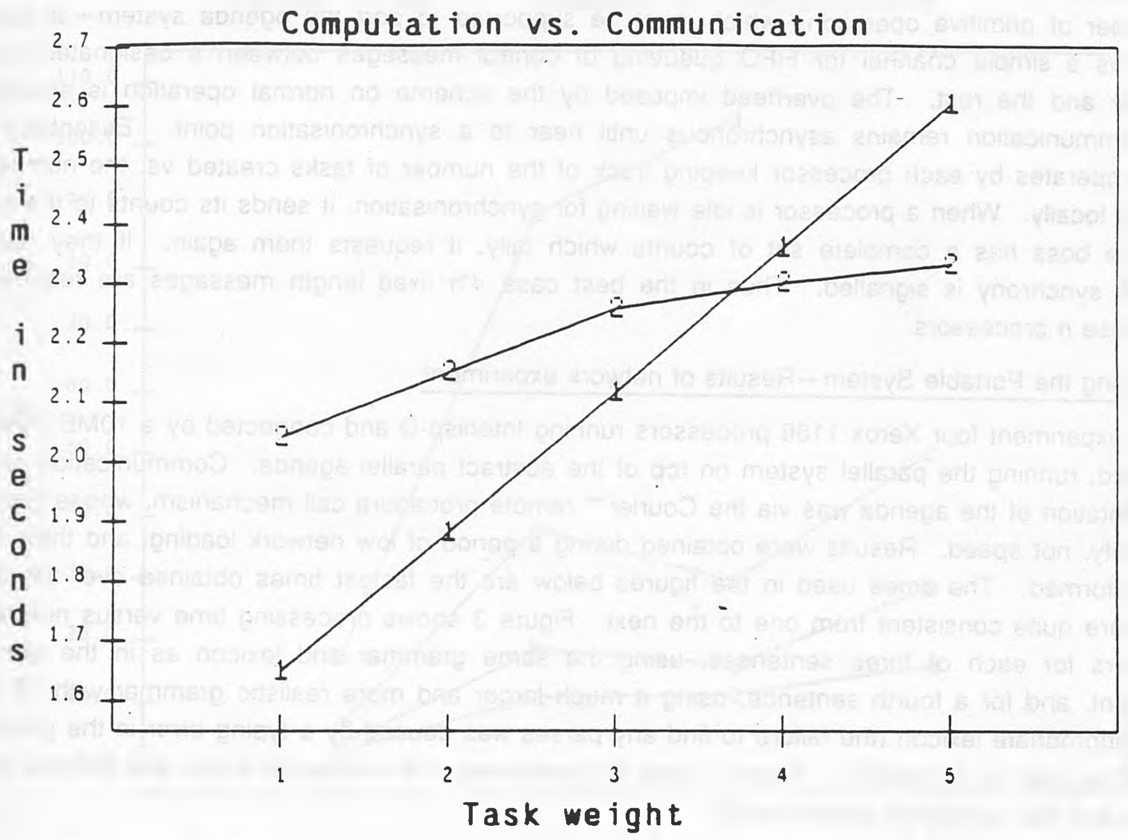

Figure 2. Graph of results of 2 processor Hypercube™ experiment

It is hoped that further experimentation with larger cubes will shortly be possible.

V. Towards Wider Comparability—The Abstract Parallel Agenda

With an eye to allowing an easy extension of this work to other systems, and more principled comparison between systems, I have gone back to the original serial chart parser (Thompson 1983) from which the Hypercube™ system was constructed, and produced an abstract parallel version. The original parser was based on a quite general agenda mechanism, and the abstraction was largely performed at this level. A multi-processor agenda system, allowing the programmer to schedule the evaluation of any memory-independent form on any processor at a specified priority level is provided, together with a novel means of synchronisation and termination. Implemented in Common Lisp, all this agenda system requires for porting to a new system is the provision of a simple 'remote funcall' primitive.

VI. Termination and Synchronisation

Termination detection in distributed systems is a well-known problem. It arises obtrusively in any parallel approach to chart parsing, since this depends on an absence of activity to detect the completion of parsing. The abstract parallel agenda mechanism which underlies the portable parallel parser uses a new (we think—see Thompson, Crowe and Roberts forthcoming for discussion) algorithm for effective synchronisation of task execution (of which termination is a special case). It is thus possible to reconstruct not only the prioritisation function of an agenda (run this in preference to this), but also the ordering function (run this only if that is finished). Unlike some existing termination algorithms, this one is particular appropriate where no constraints can be placed on processor connectivity (any processor may, and usually does, send messages to any other processor). It requires only a modest increase in the number of primitive operations which must be supported to port the agenda system—all that is required is a simple channel for FIFO queueing of control messages between a designated 'boss' processor and the rest. The overhead imposed by the scheme on normal operation is effectively zero—communication remains asynchronous until near to a synchronisation point. Essentially the scheme operates by each processor keeping track of the number of tasks created vs. the number of tasks run locally. When a processor is idle waiting for synchronisation, it sends its counts to the boss. When the boss has a complete set of counts which tally, it requests them again. If they haven't changed, synchrony is signalled. Thus in the best case *4*n* fixed length messages are required to synchronise *n* processors.

VII. Testing the Portable System—Results of network experiment

For this experiment four Xerox 1186 processors running Interlisp-D and connected by a 10MB Ethernet were used, running the parallel system on top of the abstract parallel agenda. Communication for the implementation of the agenda was via the Courier™ remote procedure call mechanism, whose hallmark is reliability, not speed. Results were obtained during a period of low network loading, and three trials were performed. The times used in the figures below are the fastest times obtained over the trials, which were quite consistent from one to the next. Figure 3 shows processing time versus number of processors for each of three sentences, using the same grammar and lexicon as in the previous experiment, and for a fourth sentence, using a much larger and more realistic grammar with 70 rules and an appropriate lexicon (the failure to find any parses was caused by a typing error in the grammar, detected too late for correction). Table 1 gives the sentences, the number of active and inactive edges involved and the number of parses found.

Sentence	active edges	inactive edges	parses
a: The orange saw saw the orange saw.	46	22	1
b: The orange saw saw the orange saw with the orange saw.	88	43	2
c: The orange saw saw the orange saw with the orange saw with the orange saw.	166	82	5
d: The front-end consists of those phases that depend primarily on the source-language.	285	58	0

Table 1. Sentences used in the network experiments

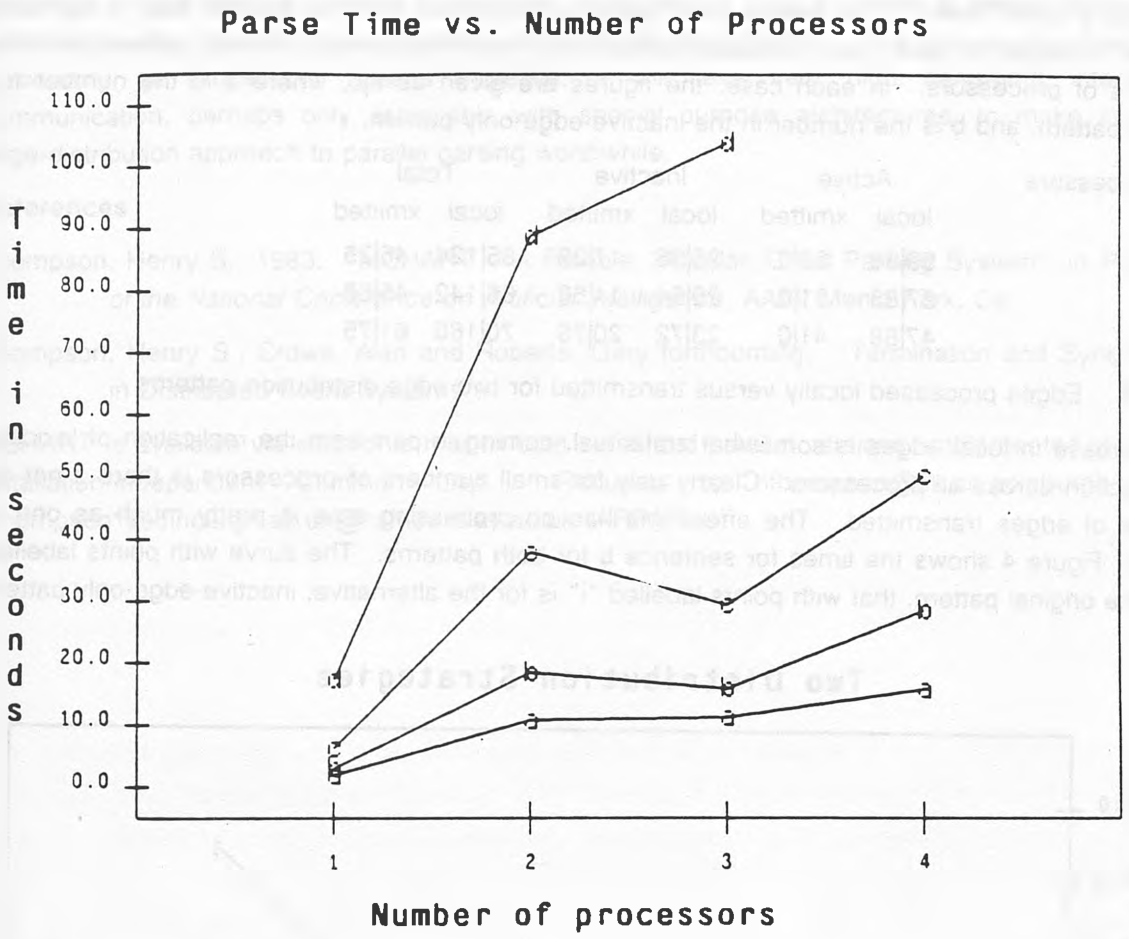

Figure 3. Graph of results of network experiment

Clearly not much encouragement can be taken from this experiment. Although there is some speed-up from two to three processors in some cases, overall the pattern is one where communication costs clearly dominate, so no advantage is gained. With slower processors and/or faster networks, we might hope to see better results, especially given the results in section IV, but the appropriateness of this approach to networked systems must remain in doubt in the absence of better evidence.

VIII. Alternative Patterns of Edge Distribution

One possible alternative decomposition of the task, which might offer some hope of improving the computation/communication trade-off, would be to transmit only inactive edges, but to send them to all processors. Then every processor would have the complete inactive chart, and could run active edges from start to finish without ever sending them anywhere. In order to distribute the computational load, rule invocation would be distributed on a per vertex basis. That is, each processor would only do bottom up rule invocation for those inactive edges which began at a vertex owned by that processor. The plus side of this route is that it sends only inactive edges around, which are simpler to encode, that the final result is available on a single processor, indeed on all processors, without having to be assembled, and that active edge processing is more efficient. The minus side is that the inactive edges have to be sent to all processors. In the simple example given in Figure 1, this actually balances out—four edges in the original implementation, two edges twice in the alternative one. A further experiment with the network system was conducted to explore this approach. The same sentences as before were used, but this time with the new edge distribution pattern. Table 2 below compares sentence b from Table 1, *The orange saw saw the orange saw with the orange saw*, in terms of the number of edges of each type processed locally and transmitted under the two patterns for different numbers of processors. In each case, the figures are given as a|b, where a is the number for the original pattern, and b is the number in the inactive-edge-only pattern.

# of processors	Active		Inactive		Total	
	local	xmitted	local	xmitted	local	xmitted
2	59|88	29|0	26|36	17|25	85|124	46|25
3	57|88	31|0	29|54	14|50	86|142	45|50
4	47|88	41|0	23|72	20|75	70|160	61|75

Table 2. Edges processed locally versus transmitted for two edge distribution patterns

The increase in local edges is somewhat artifactual, coming in part from the replication of lexical edge construction across all processors. Clearly only for small numbers of processors is there a net gain in number of edges transmitted. The effect this has on processing time is pretty much as one would expect. Figure 4 shows the times for sentence b for both patterns. The curve with points labelled "o" is for the original pattern, that with points labelled "i" is for the alternative, inactive-edge-only pattern.

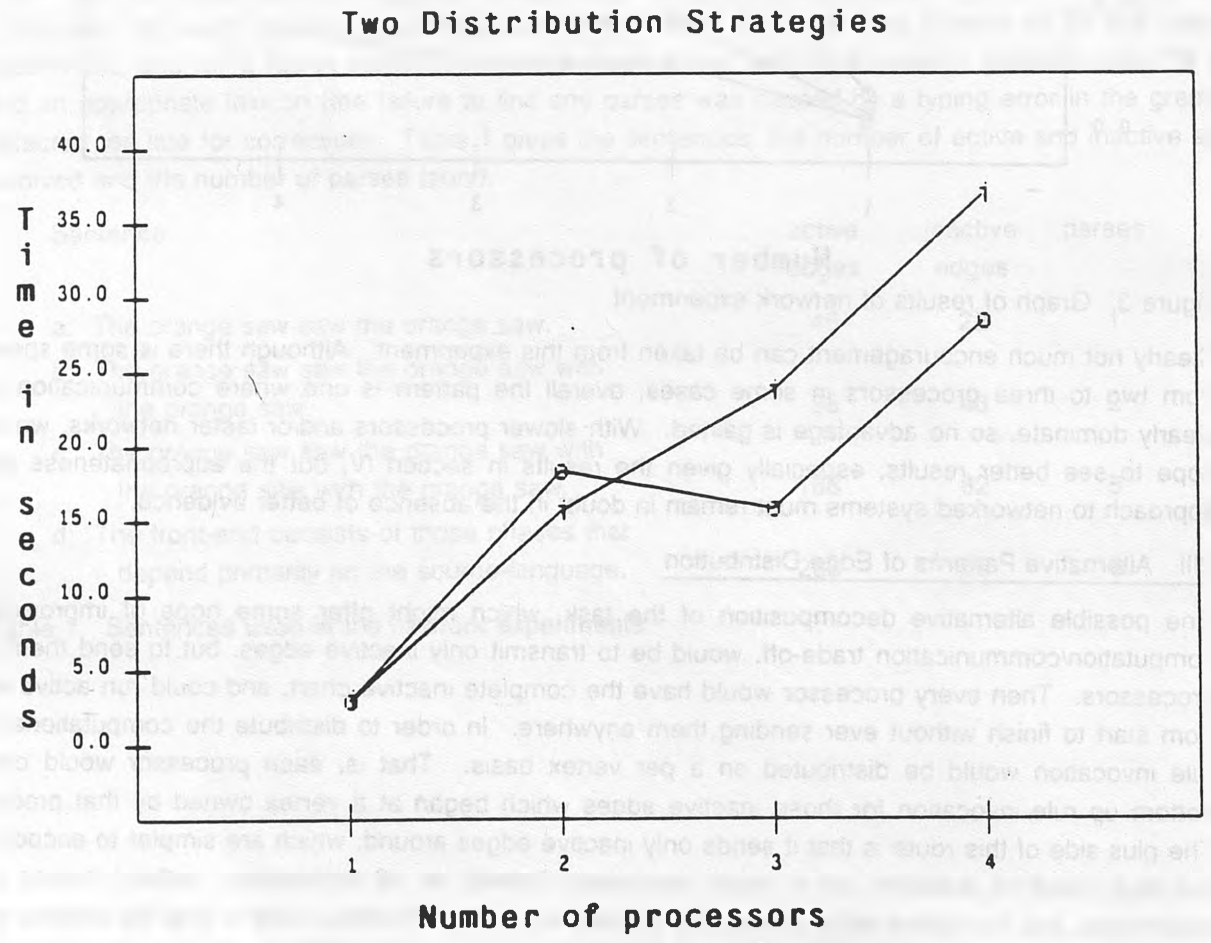

Figure 4. Graph of alternative distributions strategies for parsing sentence b

As expected, only in the two processor case do we see an advantage to the alternative approach. In general it is clear that the principle determinate of processing time is number of edges transmitted—the overhead in the network communication dominates all other factors. The obvious conclusion is that, particularly as processors speeds increase, it will take very high bandwidth inter-processor communication, perhaps only achievable with special purpose architectures, to make at least this edge-distribution approach to parallel parsing worthwhile.

References

Thompson, Henry S. 1983. "MCHART -- A Flexible, Modular, Chart Parsing System", in *Proceedings of the National Conference on Artificial Intelligence*, AAAI, Menlo Park, Ca.

Thompson, Henry S., Crowe, Alan and Roberts, Gary forthcoming. "Termination and Synchronisation in Distributed Event Systems".

MCHART is available via electronic mail in both serial and parallel versions, implemented in a relatively installation-independent Common Lisp. Requests to hthompson@uk.ac.edinburgh (JANET), hthompson%edinburgh.ac.uk@nsfnet-relay.ac.uk (ARPANet).