PARALLEL ACTIVE CHART PARSING

G. Tsilikas and M. Fleury
Electronic Systems Engineering Department, University of Essex, Colchester

Keywords to describe this work: parallel computation, speech understanding

Key Results: Speed-up results from a local agenda and shared-memory architecture.

How does the work advance the state-of-the-art?: Realistic performance has been achieved compared to previous parallelizations.

Motivation(problems addressed): The fine-grained and synchronous algorithmic structure impeded parallelization.

Introduction

In the field of linguistics, Active Chart Parsing (ACP) is an algorithm that generates all possible parsings of a sentence, given an ambiguous grammar. Therefore, ACP would be one part of a natural language interface to a database server, if real-time processing could be achieved. Parallel processing is one way to achieve real-time performance. A previously reported parallelizations of ACP [1] on a Network of Workstations (NOW) showed no speedup, rather the reverse, an unsurprising conclusion, as ACP is inherently fine-grained and synchronous. Partial results, *i.e.* tentative or definite identification of phrases, are stored for later use in the sequential algorithm, and continually exchanged in any parallel algorithm. A further difficulty for any parallelization is that there is an ordering constraint, as, at least in the English language, aggregated phrases are formed from adjacent words or phrases.

The reported implementation was through the Orca language, which adopts a shared-object model with implicit message passing. Replication of shared object state is reported to have resulted in a slow-down, as local changes of state required a broadcast. Therefore, the ACP algorithm is appropriate for a physically shared-memory address space, where no replication is required. In this paper, we report a parallelization on a commodity four-processor Dell multiprocessor (700 MHz with 1 GB memory, 1 MB L2 cache). Such processors, which are cost effective due to their commercial use as web servers, are programmed in parallel mode using multithreading.

Active Chart Parsing algorithm

Input to the algorithm is a putative sentence (or phrase) to be parsed, and a set of grammatical rules. In the sequential algorithm, each word of the sentence is initially stored in a stack called an `Agenda`. At this stage, a word is also an `Edge`, or grammatical unit. An `Active Edges` represents a hypothesis that an aggregate edge can be formed by combination with an `Inactive Edge`. Likewise, `Inactive Edges` are matched to `Active Edges`, but there is an added possibility of making a new edge identification by re-applying the grammatical rules. Any edges formed are stored in either an `Active Edge` or `Inactive Edge` array. Edges formed are also pushed onto the `Agenda`, which represents a set of tasks executed in last-in first-out (LIFO) order. Edges taken from the `Agenda` are matched against edges held in either the `Active` or `Inactive Edge` arrays. Termination is when there are no further edges in the `Agenda`.

Two parallel algorithms were implemented. The first creates a global agenda and assigns a new thread to each edge taken from that global agenda. Around 300 threads were created for a fourteen word sentence. Given this large number of threads, a variant of the first algorithm restricts the number of threads to the members of a thread pool. The motivation is to reduce thread creation overhead. In the second parallel algorithm, once

a thread has been assigned an initial edge, formed from a word in the sentence, it then puts that edge on a local `Agenda`, and continues to push any further identified edges onto the local `Agenda`. It also copies those edges into the appropriate global edge array. Therefore, access to the two global edge arrays was synchronized. In this second parallel algorithm, it is important to ensure that if an identified edge is held by a thread that it is placed in a global array to avoid another thread terminating prematurely through 'edge starvation'.

Results

The first parallel algorithm's performance, Table 1 60 rules, was disappointing, as it was worse than that of the sequential version. It was surmised that this was due to the overhead of creating threads. Restricting the number of threads to the number of processors (4), through use of a thread pool, algorithm 1P, only served to increase the disparity. This was because edges must wait until a thread becomes available. Increasing the number of threads to twenty did improve the performance of algorithm one to the same level as algorithm two, *i.e.* a small speed-up. A further increase in the number of threads in the pool to thirty resulted in very much the same timings, and, hence, this result is not recorded in Table 1. The main problem with a thread pool is knowing the optimal number to use.

Algorithm two did produce a small speed-up. The times recorded in Table 1 are the minimum (modal) timings from up to twenty runs, as, due to the thread scheduling order, there was variation in timings. For example, the timing for threaded algorithm two ranged between 0.039 to 0.050 s for 60 rules. Moreover, as the variable scheduling order could lead to results not being available when the list of edges was inspected, a final parsed sentence did not always result when using algorithm one. Therefore, algorithm two, using a local agenda, was not only faster, but also was completely reliable. Therefore, algorithm one is not to be recommended.

A small list of rules (60 in total) was responsible for the first set of results in Table 1. Rules in ACP include identification of a word as a noun, adjective, and so on, as well as rules identifying composite grammatical syntax. Clearly 60 rules is not a real-world task, and, therefore, the number of rules (principally words) was increased to 8,300 by incorporating the XTAG vocabulary. This resulted in an increase in thread granularity. In the second part of Table 1, the speed-up from using algorithm two is now a little better (1.4) when using the new rule list, though with the original 14 word sentence. The increase in granularity is also reflected in the 20 thread pool version of algorithm one. Additionally, the greater granularity resulted in algorithm one also becoming completely reliable, as there was evidently no opportunity for unfortunate orderings in the output of results.

Algorithm	Time (s)
Sequential	0.045
Threaded (1)	0.096 (0.10)
Threaded (1P-4)	0.071 (0.085)
Threaded (1P-20)	0.079 (0.084)
Threaded (2)	0.039 (0.042)
60 rules	
Sequential	0.42
Threaded(1)	0.38 (0.40)
Threaded(1P-4)	0.36 (0.50)
Threaded (1P-20)	0.33 (0.35)
Threaded (2)	0.29 (0.32)
8,030 rules	

Table 1: ACP timings (4 processors)

Conclusion

By transfer from a distributed memory to a shared-memory architecture, a reasonable parallel decomposition of ACP can be achieved, if the correct parallel algorithm is selected. Fine-grained, synchronous algorithms, of which ACP is one, are difficult to accelerate, not least because of the problem of multithreading programming. This argues for further development of libraries and software tools for this class of parallel computing environment.

[1] A. R. Sukul. Parallel implementation of an Active Chart parser in Orca. Technical report, Vrije Universiteit, Amsterdam, 1996.