

Parallelising Plan Recognition

Chris Swetenham



Master of Science
Artificial Intelligence
School of Informatics
University of Edinburgh
2012

(Graduation date: November 2012)

Abstract

TODO para1

TODO para2

Acknowledgements

TODO

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(Chris Swetenham)

Table of Contents

1	Introduction	1
2	Background	3
2.1	Plan Recognition	3
2.2	Combinatory Categorical Grammars	4
2.3	Applying CCGs to Plan Recognition	5
2.4	The ELEXIR Algorithm	5
2.5	Concurrency Techniques	6
2.6	Multicore Job Scheduling Techniques	6
2.7	Applying Job Scheduling Techniques to ELEXIR	6
3	Related Work	7
4	Initial Work	9
5	Method 0 - Original Single-Threaded ELEXIR	11
5.1	Implementation	11
5.2	Evaluation	11
5.2.1	XPERIENCE Domain	11
5.2.2	Logistics Domain	11
5.3	Analysis	11
6	Method 1 - One Thread Per Task	13
6.1	Implementation	13
6.2	Evaluation	13
6.2.1	XPERIENCE Domain	13
6.2.2	Logistics Domain	13
6.3	Analysis	13

7	Method 2 - One Queue Per Thread	15
7.1	Implementation	15
7.2	Evaluation	15
7.2.1	XPERIENCE Domain	15
7.2.2	Logistics Domain	15
7.3	Analysis	15
8	Method 3 - Lock-Free Work-Stealing	17
8.1	Implementation	17
8.2	Evaluation	17
8.2.1	XPERIENCE Domain	17
8.2.2	Logistics Domain	17
8.3	Analysis	17
9	Method 4 - Single Global Queue	19
9.1	Implementation	19
9.2	Evaluation	19
9.2.1	XPERIENCE Domain	19
9.2.2	Logistics Domain	19
9.3	Analysis	19
10	Evaluation	21
11	Conclusion	23
	Bibliography	25

List of Figures



Chapter 1

Introduction

Plan Recognition is a problem which consists of looking at a sequence of actions by an agent, and deduce possible goals that these actions achieve. Plan Recognition has many potential applications, including network intrusion detection, human-robot interaction and assisted care scenarios.

In many of these potential problem domains, including the ones mentioned above, it is desirable to perform plan recognition in real time, rather than offline as a batch computation. However currently for performance reasons it is often infeasible to perform Plan Recognition in real time.


In this project, we investigate several methods for parallelising a particular Plan Recognition algorithm in C++. We investigate the performance on two different problem domains, and how the performance scales up with the number of available hardware threads.

Chapter 2

Background

In this section, we discuss the background for the project and provide context for the work performed.

2.1 Plan Recognition

Plan Recognition looks at a sequence of actions by an agent, and attempts to deduce that agent's goals. Plan Recognition can be seen as the inverse of Planning, where given a goal the problem is to deduce a sequence of actions which achieve it. Plan Recognition  rates symbolically [TODO: is there any work on non-symbolic plan recognition?]; the actions observed are treated as symbols, and the rules operating on them operate on these symbols.

There are many possible situations where it is desirable to recognise the intentions of an agent from its observed actions. In network intrusion detection, we wish to identify any actions which could correspond to an attack, hidden in a large number of benign actions by users, in real time. [TODO: cite ?] In human-robot interaction, we wish a human and a robot to work together on a task. If the robot partner can observe the actions of the human partner, and from them understand what the human partner is currently trying to achieve, it can anticipate the needs of the human partner, and assist or offer advice. [TODO: cite JAST project]. In assisted care scenarios, a robot can again anticipate the needs of the human being assisted; it can also recognise unusual behaviour which would necessitate emergency response. [TODO: cite ?]

Since Plan Recognition takes symbolic input, some amount of transformation of the input may be required. For network intrusion detection, this could be a

simple filter on system logs. For more complex scenarios, Plan Recognition could tie in to **Behaviour Recognition**, which takes video and other real-time feeds and classifies the actions taken by humans using machine learning techniques.

Plan Recognition is **usually** performed in terms of some grammar for actions and goals. This grammar could be learned or crafted by the designer; in this project, we use two grammars crafted by hand.

2.2 Combinatory Categorical Grammars

The problem of parsing, where we analyse a sequence of tokens and produce a syntax tree according to a formal grammar, is very similar to the problem of plan recognition, so it is natural to consider the applicability of parsing techniques to this problem.


When parsing computer languages such as XML or C++, ambiguity is undesirable. An input text should admit to either a single interpretation or no interpretation at all, and it is the task of the language designer to resolve any possible ambiguities in the grammar.

When parsing natural languages such as English, a degree of ambiguity is unavoidable; in fact, such ambiguity can be exploited by speakers, for example in puns. Correspondingly, grammars and parsers for natural language must deal with potential ambiguity and multiple possible parses. Therefore, we can expect parsing techniques from natural language processing may be more amenable to being adapted to the problem of Plan Recognition.

Categorical Grammars assign *categories* to each input token or symbol, and combine them according to simple ~~application rules in a function argument relationship~~. Basic categories correspond to lexical classes such as “identifier” in a computer language or “noun” in a natural language. We denote basic categories by atomic terms such as N . ~~Complex categories are akin to~~ functions over categories: they look for arguments ~~directly~~ to their left (denoted \backslash) or right (denoted $/$) in the sequence of categories, and yield other categories as a result. A complex category taking an argument N to its left and yielding NP is denoted $NP\backslash N$; one taking the same argument to the right is denoted NP/N . [TODO footnote: some sources denote the leftward example above as $N\backslash NP$, preserving the ordering of the argument and result. For consistency we will always denote the arguments on the right and the results on the left.]

Taking a simple example in English: [TODO footnote: this example is taken from en.wikipedia.org/wiki/Categorial_grammar, retrieved 8 August 2012]

English sentence: “the bad boy made that mess”

Basic categories: [TODO: layout] N - noun NP - noun phrase S - sentence 

We then assign basic or complex categories to each word:

the bad boy made that mess
NP/N, N/N, N, (S\NP)/NP, NP/N, N

We can then combine the complex categories with their arguments and produce a parse. TODO: LaTeX for parse tree


Combinatory Categorial Grammars (hereafter ~~Combinatory Categorial Grammar~~ (CCG)s) extend categorial grammars with new reductions corresponding to combinators in the combinator calculus. Since complex categories can take arguments either to their left or right, each combinator has a leftward and a rightward version.

The leftward and rightward application combinators are the reductions already present in categorial grammars. The other two combinators commonly used in CCGs are composition and type-raising.

Composition corresponds to function composition. The rightward composition operation has the form: $X/Z \leftarrow X/Y, Y/Z$. TODO: more explanation

Type-raising transforms basic categories to complex categories. The rightward type-raising operation has the form: $T/(T \backslash X) \leftarrow X$. TODO: more explanation

2.3 Applying CCGs to Plan Recognition

~~The constraints on the ordering of actions in a plan are looser than the constraints on the ordering of words in natural language. Correspondingly,~~ we make some  modifications to the CCG formalism to apply it to Plan Recognition.

Firstly, it is possible for intervening actions to occur which are not part of the current goal; they may be part of some other goal, or entirely irrelevant to the goals we are interested in. We therefore allow complex categories to find their arguments anywhere to the left (for leftward-applying categories) or to the right (for rightward-applying categories), rather than directly to the left or right.

Secondly, certain actions to achieve a goal may in certain cases be executed in any order with equal results. Although this could be expressed in the CCG formalism, by including every possible permutation in the grammar, it is preferable

to allow complex categories to take sets of arguments which may be provided in any order. This is denoted as $A/B, C$ for a complex category taking arguments B and C to the right in any order, and yielding A .

2.4 The ELEXIR Algorithm

TODO describe the ELEXIR algorithm
 TODO state predicates
 TODO probabilities
 TODO bindings

2.5 Concurrency Techniques

TODO describe locking and lock-free techniques
 TODO describe mutexes, condition variables
 TODO describe memory barriers, atomic operations, hazard pointers (or later?)
 TODO describe performance issues: waiting, kernel switches, busy waits, starvation, priority inversion, cache contention and false sharing
 TODO describe correctness issues: missed wake-ups, deadlocks, static/dynamic hazards, ABA problem, ...

2.6 Multicore Job Scheduling Techniques

TODO describe single queue
 TODO describe multiple queues with main thread redistributing work
 TODO describe work-stealing
 TODO reference that XB360/dev conf talk on job queues?
 TODO mention dependencies between tasks, tasks producing other tasks

2.7 Applying Job Scheduling Techniques to ELEXIR

TODO describe how ELEXIR is well-suited to job scheduling - tasks generate the new tasks that depend on them, pure fan-out with no dependencies sideways and no shared resources needing to be updated
 TODO describe how we choose to split work
 TODO Work out the complexity (Could go in appendix)

TODO Work out the perfect T_1 to T_∞ speedup (Could go in appendix)

Chapter 3

Related Work

TODO

Chapter 4

Initial Work

TODO describe initial work fixing memory leaks, fixing reference counting

TODO describe tests added

TODO describe work making ref counting thread-safe and why we don't need hazard pointers for categories

TODO describe the tools and libraries used: boost::threads, valgrind, callgrind/cachegrind

TODO describe histogram of number of generated explanations for each existing one, and how this informs design choices

Chapter 5

Method 0 - Original Single-Threaded ELEXIR

TODO - short description of design.

5.1 Implementation

TODO - Describe implementation.

5.2 Evaluation

5.2.1 XPERIENCE Domain

TODO

5.2.2 Logistics Domain

TODO

5.3 Analysis

TODO

Chapter 6

Method 1 - One Thread Per Task

TODO - short description of design. Spawn a thread for each explanation, showing that multithreading the computation works.

6.1 Implementation

TODO - Describe implementation.

6.2 Evaluation

6.2.1 XPERIENCE Domain

TODO

6.2.2 Logistics Domain

TODO

6.3 Analysis

TODO

Chapter 7

Method 2 - One Queue Per Thread

TODO - short description of design. One queue per thread, main thread redistributes.

7.1 Implementation

TODO - Describe implementation. Phases, detecting completion, blocking and signalling on the queue.

7.2 Evaluation

7.2.1 XPERIENCE Domain

TODO

7.2.2 Logistics Domain

TODO

7.3 Analysis

TODO

Chapter 8

Method 3 - Lock-Free Work-Stealing

TODO - short description of design. Spawn a thread for each explanation, showing that multithreading the computation works.

8.1 Implementation

TODO - Describe implementation. Insertion of memory barriers. Hazard pointers and resizing. Expected properties in terms of load-balancing.

8.2 Evaluation

8.2.1 XPERIENCE Domain

TODO

8.2.2 Logistics Domain

TODO

8.3 Analysis

TODO

Chapter 9

Method 4 - Single Global Queue

TODO - short description of design. Single global queue, locking.

9.1 Implementation

TODO - Describe implementation.

9.2 Evaluation

9.2.1 XPERIENCE Domain

TODO

9.2.2 Logistics Domain

TODO

9.3 Analysis

TODO

Chapter 10

Evaluation

TODO here compare the different approaches, have graphs with results from all algorithms

TODO describe choice of metrics (or do this earlier?)

TODO describe the domains used (or do this earlier?)

TODO discuss hyperthreading and how it affects scaling of performance with number of threads (mention this in background too?)

TODO compare the speedup we get to a perfect T_1 to T_∞ speedup

Chapter 11

Conclusion

TODO draw conclusions from the results - why is work stealing so much better for this problem?

TODO future work: exploring only the most likely explanations given the observations so far

TODO future work: improving the memory allocation behaviour of the algorithm and testing different memory allocators better suited to multi-threaded code

TODO future work: parallellising the probability computation (would be automatic, and probably have better memory locality, if we did it incrementally)

TODO future work: incremental version, producing explanations each observations

TODO future work: discarding observations of a certain age if unused?

Bibliography