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

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. This can be seen as the inverse of Planning, where given a goal the problem is to deduce a sequence of actions which achieve it.

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 an 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 preprocessing 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 a system which takes video and other real-time feeds and classifies the actions taken by humans using machine learning techniques.

Some Plan Recognition techniques use a formal grammar to define actions and plans to be recognised. This grammar could be learned or crafted by the designer; in this project, we use two grammars crafted by hand.

2.2 Combinatory Categorial Grammars

The problem of plan recognition is very similar to the problem of parsing, where we analyse a sequence of tokens and produce a syntax tree according to a formal grammar; 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 a sequence of actions into possible explanations, a degree of ambiguity is unavoidable; for example, if we observe a truck driving from Edinburgh to Newcastle, it may be making a delivery locally in Newcastle; or it may be driving there in its way to York. Grammars and parsers for natural language must deal with potential ambiguity and multiple possible parses in human language. Therefore, we can expect parsing techniques from natural language processing may be more amenable to being adapted to the problem of Plan Recognition than parsing techniques for computer languages.

Categorial Grammars assign *categories* to each input token or symbol, and combine them according to simple rules. *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 A. *Complex* categories are akin to functions over categories: they look for arguments 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 A to its left and yielding B is denoted $B\backslash A$; one taking the same argument to the right is denoted B/A. [TODO footnote: some sources denote the leftward example above as $A\backslash B$, 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: replace with an example parse from the XPERIENCE or logistics domains]

English sentence: “the bad boy made that mess”

Basic categories:

N - noun

NP - noun phrase

S - sentence

We then assign basic or complex categories to each word:

$$\begin{array}{cccccc} \text{the} & \text{bad} & \text{boy} & \text{made} & \text{that} & \text{mess} \\ NP/N, & N/N, & N, & (S\backslash NP)/NP, & NP/N, & N \end{array}$$

We can then combine the complex categories with their arguments and produce a parse. Combining the categories follows the rules: $X \leftarrow X/Y,\ Y$ and $X \leftarrow Y,\ X\backslash Y$:

. $NP/N,\ N/N,\ N,\ (S\backslash NP)/NP,\ \underbrace{NP/N,\ N}$

. $NP/N,\ N/N,\ N,\ \underbrace{(S\backslash NP)/NP,\quad NP}$

. $NP/N,\ \underbrace{N/N,\ N},\quad (S\backslash NP)$

. $\underbrace{NP/N,\quad N},\quad (S\backslash NP)$

. $\underbrace{NP,\quad (S\backslash NP)}$

. S

Combinatory Categorial Grammars (CCGs) extend categorial grammars with new reductions corresponding to combinators in combinatory logic. 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

In applying CCGs to Plan Recognition, we seek to assign (basic or complex) categories to the input sequence of actions, and by reductions using a set of combinators, produce a set of explanations for the input. Adjacency of grammar elements is not as crucial in Plan Recognition as in Natural Language Processing, and so 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 [TODO: cite Hoffman].

2.4 The ELEXIR Algorithm

The ELEXIR algorithm [TODO: cite] provides several additional features to CCGs used for plan recognition. *Features* give first-order variable arguments to basic categories, which refer to labels in the domain. For example, `putAwayC(cup1)`. This allows categories in the domain to refer to many different potential objects, rather than have to repeat similar categories for each potential object in the domain. The same feature label refers to the same object in all basic categories within a complex category. When categories are combined with combinators, categories which were previously tested for equality must now be *unified*, with unbound features in one bound to corresponding bound features in the other, and corresponding bound features tested for equality. *States*, defined in terms of first-order predicates, are tracked through each explanation, each action updating the current state predicates. Assignments of categories to each action can be made conditional on certain predicates holding in the current state. *Probabilities* for each explanation are computed, based on probabilities assigned in the input to categories and assignments of categories to actions. These probabilities can be made conditional on the current state predicates.

The ELEXIR algorithm restricts itself to three combinators: left application, right application, and right composition. It builds up a list of explanations by successively adding categories for new observations to existing explanations, and then applying any applicable combinators to pairs of categories in the explanation. Each input action can modify the state predicates, and can be assigned one or more categories, depending on whether preconditions on their applicability are satisfied by the current state. Combinators are applicable if the relevant arguments or results can be unified. When a new category is added for an observed action, at least one new explanation is produced for each existing one: the one in which no combinators are applied.

2.5 Concurrency Techniques

We can split our discussion of concurrency techniques between blocking and non-blocking methods.

Blocking methods include those involving mutual exclusion locks (mutexes), semaphores, condition variables, and other constructs where a thread may block on a concurrency construct indefinitely. [TODO: expand further on each of these?]

Non-blocking methods several possible levels of guarantee. *Wait-free* algorithms guarantee that threads will always complete in a finite number of steps, but are usually expensive. *Lock-free* algorithms only guarantee that some thread will always be making progress, and can be much cheaper than wait-free algorithms. Non-blocking algorithms depend on lower-level primitives than blocking ones; most commonly, the *Compare-And-Swap* (CAS) operation available on modern processors along with memory fences which provide some guarantees with respect to the ordering of memory accesses. Non-blocking algorithms are harder to implement and reason about than blocking methods, but can provide improved performance and lower overhead in return.

[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 Task Scheduling Techniques

Many problems can be broken down into individual units of work to be executed across several threads. These tasks may have dependencies on other tasks which they require to be complete before they can begin; and they may generate new tasks to be scheduled. There are many possible techniques for scheduling work across threads, with different overheads and complexities of implementation. The lower the overhead of scheduling, the more fine-grained we can make the tasks, and the better we can load-balance to ensure all threads perform useful work.

One of the simplest possible implementations uses a single global queue for all tasks. Access to the queue is guarded by a global lock. Tasks are enqueued at the tail of the queue, and dequeued at the head. This implementation guarantees that a thread will be able to find a task if one is available, but the contention by all threads on a single global lock means that the overhead of this method is potentially high especially for short-lived tasks.

A potentially better implementation uses one queue of work for each worker thread. This reduces contention when threads are requesting work from the queues. In order to ensure that work is distributed to all threads, the main thread must periodically distribute work to all the threads, and worker threads must feed any new tasks to be scheduled back to the main thread. While this reduces contention, worker threads may not be able to find tasks to execute if they are available but the main thread has not scheduled them yet.

The final case we will examine is a lock-free work-stealing scheduler, with one queue for each worker thread. In a work-stealing scheduler, threads which find their own queue empty will select a random *victim* thread and attempt to steal a task from their queue. New tasks will be added to the thread's own queue. In this way, work will be distributed between threads, but most of the accesses will be uncontended accesses to a thread's own queue. This can be implemented lock-free and has potentially the lowest contention of the methods mentioned here while balancing work well across threads.

[TODO cite XB360/dev conf talk on task queues? Or see if it has any references of its own? +cite Herlihy&Shavit]

2.7 Applying Task Scheduling Techniques to ELEXIR

In the ELEXIR algorithm, each existing explanation is processed independently when a new action is added to the observations. Each explanation generates one or more new explanations, with no dependencies on other computations. This makes ELEXIR well suited to parallelising using task scheduling. We can assign to each task some number of explanations, depending on the overhead of the method we are using.

[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

Before starting with the implementation of multithreading, we evaluated the existing ELEXIR codebase to detect any potential issues. We used the *Memcheck* tool within the *Valgrind* program on Linux to detect memory leaks which we then fixed. Valgrind dynamically translates the machine code being executed and allows its tools to insert instrumentation before the code is actually executed. Memcheck tracks memory allocations and can warn of both memory leaks and invalid memory accesses.

The reference-counting was changed to use atomic incrementing and decrementing of reference counts, and this was sufficient to ensure thread safety, since threads incrementing the reference count always own at least one reference to it through the expression currently being processed. In order to verify that the changes made to the code did not break existing functionality, we added tests for the results of the algorithm on one of the domains. We also added unit tests for some of the new functionality.

We considered several possible libraries for implementing multithreading. We chose the *Boost Threads* library, which is part of the *Boost* project, over the POSIX threading API for reasons of both convenience and portability; although the code has not yet been ported entirely to Windows, the added code was designed with this future port in mind.

Chapter 5

Method 0 - Original Single-Threaded ELEXIR

The first method we include for comparaison is a mostly unmodified single-threaded implementation.

5.1 Implementation

On each new observation, the entire current list of explanations is looped over by the main thread, and a new list of explanations is generated.

[TODO: code snippet]

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

As a proof of concept for multithreading the algorithm, this method spawns a new thread for each task.

6.1 Implementation

On each new observation, the main thread spawns a new thread for each task. Each task processes its expressions and produces new resulting expressions. The main thread waits for all the spawned threds to complete, and collects their results, then spawns a new set of threads on the next observation.

[TODO: code snippet]

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

This method uses one queue per thread, guarded by a mutual exclusion lock.

7.1 Implementation

The main thread creates a queue for each worker thread, then spawns the threads. On each new observation, the main thread partitions the work evenly between the worker thread queues, and waits for them to finish processing them, finally collecting the new explanations.

TODO - Describe blocking and signalling on the queue. TODO - Describe detecting completion. TODO - Describe boost::bounded_buffer.

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

This method uses one lock-free queue for each worker thread. Other threads can steal from this queue if they run out of work in their own queue.

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

This method uses a single global queue of tasks, guarded by a mutual exclusion lock.

9.1 Implementation

The main thread adds a single empty explanation to the global queue, then spawns a worker thread for each hardware thread available. These threads all attempt to lock the queue and extract work from it. If they cannot lock the queue, they sleep until the lock becomes available. If they lock the queue and find it empty, they sleep for a fixed period of time, then try again. Once they complete the task, they lock the queue to put all the newly generated work on the queue, then resume trying to pull a task from the head of the queue.

[TODO: code snippet]

9.2 Evaluation

9.2.1 XPERIENCE Domain

TODO

9.2.2 Logistics Domain

TODO

9.3 Analysis

TODO

Chapter 10

Evaluation

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

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: parallelising 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