Parallelizing Plan Recognition

Christopher W. Geib and Christopher E. Swetenham
School of Informatics
University of Edinburgh
10 Crichton Street,
Edinburgh, EH8 9AB, Scotland
cgeib@inf.ed.ac.uk and cswetenham@gmail.com

Abstract

Modern multi-core computers provide an opportunity to parallelize plan recognition algorithms to decrease runtime. Viewing the problem as one of parsing and performing a complete breadth first search, makes ELEXIR (Engine for LEXicalized Intent Recognition)(Geib 2009; Geib & Goldman 2011) particularly suitable for such parallelism. This paper documents the extension of ELEXIR to utilize such modern computing platforms. We will discuss multiple possible algorithms for distributing work between parallel threads and the associated performance wins. We will show, that the best of these algorithms will provide close to linear speedup (up to a maximum number of processors), and that features of the problem domain have an impact on the speedup.

Introduction

The ubiquity of multi-core processors provides an opportunity for algorithms that are easy to parallelize to realize significant runtime gains. However, the use of the kind of bounded parallelization available in these architectures has not been closely studied for most AI applications. Even with the ubiquity of libraries and packages supporting multithreading, most AI research has not focused on efforts to parallelize specific AI algorithms. That said, algorithms that are well suited to this kind of bounded parallelism, could benefit from a better understanding of the tradeoffs required to make full use of easily obtainable modern computer architectures.

This paper presents experimental results on the parallelization of a particular algorithm for the AI problem of plan recognition, namely the Engine for LEXicalized Intent Recognition (ELEXIR) system (Geib 2009; Geib & Goldman 2011). It will show that this algorithm can easily be parallelized to produce close to linear speedup if the correct method for work allocation is chosen. The paper will also show that specific features of the domain can have a significant impact on the achieved speedup.

Copyright © 2013, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

To this end, the rest of this paper will be organized as follows. First we will provide an overview of the ELEXIR system, and discuss the features of the algorithm that make it particularly well suited to parallelization. Next we will discuss four different algorithms for allocating work between the different precessing threads and their respective strengths and weaknesses. We will then discuss the results of testing these allocation algorithms in multiple domains and discuss the impact of various domain level features that can impact even the parallelized algorithm's performance. Finally we will draw conclusions that are applicable both to other plan recognition systems, as well as AI systems more broadly.

ELEXIR Background

Plan recognition is the process of inferring the plan being executed by an agent based on observations of the agent's actions and a library of plans to be recognized. Following other work on *grammatical methods*(Sidner 1985; Vilain 1990; 1991) for plan recognition, ELEXIR(Geib 2009) views the problem as one of *parsing* a sequence of observations, based on a formal grammar that captures the possible plans that could be observed. Space prevents a complete discussion of the ELEXIR system. Here we will cover only the basics of the algorithm and those details necessary to understand its parallelization. We refer the interested readers to(Geib 2009; Geib & Goldman 2011) for more details.

In ELEXIR, plans are represented using Combinatory Categorial Grammars (CCG) (Steedman 2000), one of the *lexicalized grammars*. Parsing in such grammars abandons the application of multiple grammar rules in favor of assigning a *category* to each observation and using *combinators* to combine categories to build a parse.

Plan Grammar Categories

To represent possible plans in CCG, each observable action is associated with a set of syntactic *categories*, defined recursively as:

Atomic categories : A finite set of basic action categories. $C = \{A, B, ...\}$.

Complex categories : $\forall Z \in C$, and non empty set $\{W, X, ...\} \subset C$ then $Z\backslash\{W, X, ...\} \in C$ and $Z/\{W, X, ...\} \in C$.

Viewing complex categories as functions, we will refer to the categories on the right hand side of a slash as *arguments* ($\{W, X, ...\}$) and the category on the left hand side as a *result* (Z). The direction of the slash indicates where in a stream of observations the category looks for its arguments. That is, the argument(s) to a complex category should be observed after the category for a rightward slash and will be called *rightward arguments*. The arguments for a complex category with a leftward slash, should be observed before it (*leftward arguments*), to produce the result. Finally, multiple arguments in set braces are unordered with respect to each other.

As an example consider the simple three step plan of picking up a cell phone, dialing a number, and talking on it. This plan could be represented by the following grammar:

CCG: 1

$$\begin{aligned} dialCellPhone &:= (CHAT/\{T\})\backslash\{G\}. \\ talk &:= T. \\ getCellPhone &:= G. \end{aligned}$$

Where *G, T*, and *CHAT* are basic categories, the actions of *talk* and *getCellPhone* each have only a single possible category, namely *T* and *G*, and the the action *dialCellPhone* has a single complex category that captures the structure of the plan for chatting to a friend.

It is also worth noting that lexicalized plan grammars also require a design decision about which actions should carry which parts of the structural information for a plan. We will call an action that has a particular category as its result an *anchor* for a plan to achieve that category. For example in the phone calling grammar *dialCellPhone* is the anchor for the plan to **CHAT**. However, as we can see in CCG: 2 and CCG: 3 we could have chosen *talk* or *getCellPhone* as the anchor by choosing a slightly different set of categories.

CCG: 2

$$\begin{aligned} dialCellPhone &:= D. \\ talk &:= (CHAT\backslash\{D\})\backslash\{G\}. \\ getCellPhone &:= G. \end{aligned}$$

CCG: 3

$$\begin{aligned} dialCellPhone &:= D. \\ talk &:= T. \\ getCellPhone &:= (CHAT/\{T\})/\{D\}. \end{aligned}$$

(Geib 2009) notes that the anchors chosen for a particular grammar can have a significant impact on the runtime of plan recognition. Some choices for the anchors result in a smaller number of possible parses. We will return to discuss this later.

Combinators

ELEXIR uses three *combinators* (Curry 1977) defined over pairs of categories, to combine CCG categories:

rightward application:
$$X/\alpha \cup \{Y\},\ \ Y \ \Rightarrow\ X/\alpha$$
leftward application:
$$Y,\ \ X\backslash\alpha \cup \{Y\} \ \Rightarrow\ X\backslash\alpha$$
rightward composition:
$$X/\alpha \cup \{Y\},\ \ Y/\beta \ \Rightarrow\ X/\alpha \cup \beta$$

where X and Y are categories, and α and β are possibly empty sets of categories. To see how a lexicon and combinators parse observations into high level plans, consider the derivation in Figure 1 that parses the observation sequence: *getCellPhone, dialCellPhone, talk* using CCG: 1. As each observation is encountered, it

$$\begin{array}{ccc} \text{getCellPhone} & \text{dialCellPhone} & \text{talk} \\ \hline G & (CHAT/\{T\})\backslash\{G\} & T \\ \hline \multicolumn{2}{c}{(CHAT/\{T\}} \ \ < & \\ \hline \multicolumn{3}{c}{CHAT} \ \ > \end{array}$$

Figure 1: Parsing Observations with CCG categories

is assigned a category as defined by the plan grammar. Combinators (rightward and leftward application in this case) then combine the categories. We will refer to each such parse of the observation stream as an *explanation*.

Stated briefly, ELEXIR performs plan recognition by generating the complete and covering set of explanations for an observed stream of actions given a particular grammar. It then computes a probability distribution over this complete set, and on the basis of this distribution can compute the conditional probability of any individual goal. While ELEXIR's probability model will not be relevant for our discussion and will not be covered here, there are some additional details of the parsing algorithm that make ELEXIR amenable to parallelization which we will discuss next.

Parallelizing ELEXIR: Theory

To enable incremental parsing of multiple interleaved plans, ELEXIR does not use an existing parsing algorithm. Instead it uses a very simple two step algorithm based on combinator application linked to the in-order processing of each observation and a restriction on the form of complex categories.

Assume we are sequentially observing the actions of an agent, and further suppose that the observed agent is actually executing a particular plan whose structure is captured in a category that we are considering assigning to the current observation. In this case, it must be true that all of the leftward arguments to the category have already been performed. For example, in the cell-phone usage case, we must have observed the action of getting the cellphone before the dialing action, otherwise it is nonsensical to hypothesize the agent is trying to chat with a friend.

To facilitate this check, ELEXIR requires that all leftward arguments be on the "outside" (further to the right when reading the category from left to right) of any rightward arguments the complex category may have. For example, this rules out reversing the order of the arguments to dialCellPhone in our example CCG: 1.

CCG: 4

$dialCellPhone := (CHAT/\{T\})\backslash\{G\}.$ *acceptable*

$dialCellPhone := (CHAT\backslash\{G\})/\{T\}.$ *unacceptable*

We call such grammers *leftward applicable*. This does not make a difference to the plans captured in the CCG, as the arguments are still in their correct causal order for the plan to succeed. However, this constraint on the grammar mandates that leftward arguments must be addressed first. In fact, accounting for a categories leftward arguments is the first step of ELEXIR's two stage parsing algorithm.

The restriction to leftward applicable grammars allows ELEXIR's parsing algorithm to easily verify that an instance of each of the leftward arguments for a category has previously been executed, by the agent, at the time the category is considered for addition to the explanation. If a category being considered for addition has a leftward argument that is not already present in the explanation (and therefore can't be applied to the category), ELEXIR will not extend the explanation by assigning that category to the current observation, since it cannot lead to a legitimate complete explanation.

Thus, for each category that could be assigned to the current observation, the first step of the parsing algorithm is to verify and remove, by leftward application, all of its leftward arguments. This is done before the category is added to the explanation. This means that the explanation is left with only categories with rightward arguments. Further, since none of the combinators used by ELEXIRproduce leftward arguments, for the remainder of its processing the algorithm only needs to consider rightward combinators. This feature enables the second step of the ELEXIR parsing algorithm.

After each of the possible applicable categories for an observation have been added to a fresh copy of the explanation, ELEXIR attempts to apply the rightward combinators to every pairing of the new category with an existing category in the explanation. If the combinator is applicable, the algorithm creates two copies of the explanation, one in which the combinator is applied, and one in which it is not. As a result, each rightward combinator can only ever applied once to any pair of categories. This two step algorithm both restricts observations to only take on categories that could result in a valid plan, and guarantees that all possible categories are tried and combinators are applied. At the same time, it does not force unnecessarily eager composition of categories that should be held back for combination with as yet unseen category. Effectively this is creating a canonical ordering for the generation of explanations. This is what makes the ELEXIR algorithm particularly amenable to parallelization.

ELEXIR uses this two step parsing algorithm to search the space of all possible explanations for the observed actions. Given the algorithm, any two explanations must differ either in the category assigned to an observed action, or to the rightward combinators that are applied. As a result, given this algorithm for parsing the explanations, it is not possible for two explanations that have been distinguished either by the addition of different categories or the application of different combinators to result in the same explanation for the observations.[1] This means each addition of a category to an explanation or the use of a rightward combinator splits the search space into complete and non-overlapping sub-searches. Such sub-searches do not depend on their sibling searches and can therefore be parallelized.

To summarize then, given the requirement of leftward applicable plan grammars, the two step parsing algorithm used by ELEXIR splits the search for explanations into non-overlapping sub-searches. Each such search can be treated as separate unit of work that can be done in parallel, with the complete set of explanations being collected at the end.

Parallelizing ELEXIR: Practice

Given a method to break up the search for explanations into disjoint sub-searches, parallelization of the algorithm still requires answers to the question: How will the work be scheduled for performance? Effectively scheduling work for execution across multiple threads means keeping all the available threads busy with work while satisfying the dependencies between units of work. The unit of work scheduling may also not directly correspond to a single subtask of the underlying problem. We could decide to batch several subtasks together to form a single work unit for scheduling. This means the choosing the size of work units requires making a tradeoff between the overhead of scheduling and the effectiveness of the work distribution. For example, in the limit, scheduling all the subtasks as one unit of work will give no multithreading at all. We will see that, the methods we investigated differ in the overhead of scheduling each unit of work, and in how effectively they keep threads busy.

To parallelize ELEXIR we first modified the algorithm to ensure the search could safely proceed across multiple threads. In our C++ implementation of ELEXIR, we replaced the standard memory allocator with, the *jemalloc* allocator(Evans 2006), which is designed for multi-threaded applications, has much better contention and cache behavior, and showed much better speedups with larger numbers of threads in exploratory test experiments.

[1]This does not mean that the system can only find a single explanation for a plan given a set of observations, but that each such plan will differ either in which observed actions are part of the plan, the categories assigned to the constituent observations, or the subplans composed to produce it. These are all significantly different explanations and need to be considered by the system.

We then implemented four different scheduling policies to allocate the work to be performed across available hardware threads, and compared these against the baseline runtime of the original single-threaded algorithm. All except the baseline implementation, were built to be configurable in the number of worker threads.

Some of our policies have the main thread distribute work to the worker threads, in which case the set of explanations after each observation are collected and redistributed to threads on the next observation. The others have the worker threads pull work when they are otherwise idle. This means these schedulers do not need to have all the worker threads complete their work and fall idle after each observation, but can instead keep all threads working until all the observations have been processed. We will highlight these distinctions for each of the implemented policies below:

1. The **baseline** implementation is the original implementation, albeit with the thread-safety guarantees in place.
2. The **naive** scheduler (Herlihy & Shavit 2012) implementation is a proof of concept for multithreading the algorithm; it spawns a new thread for each unit of work to be scheduled, and the thread is destroyed when the unit of work is completed. For each observation, one unit of work is produced for each thread, and the set of explanations is shared equally between units of work.
3. The **blocking** scheduler (Herlihy & Shavit 2012) gives each worker thread a queue, and the main thread distributes work to these queues on each observation. Threads can block on an empty work queue instead of repeatedly having to check the queue. As in the naive scheduler, explanations are redistributed equally among threads on each new observation.
4. The **global queue** (Herlihy & Shavit 2012) scheduler uses a single multiple-producer, multiple-consumer work queue shared between all the threads and guarded by mutex at both ends. Worker threads push new work into this queue as they produce new explanations, and fetch work from this queue when they fall idle. This policy has a second configurable parameter: the batch size, which specifies the maximum number of explanations to be added to a unit of work to be scheduled. The larger the batch size, the fewer units of work we need to schedule when processing, but the more potential there is for missed parallelism due to underutilization. By measuring the runtime with different batch sizes, We determined a batch size of 32 to be adequate, although larger values may preferable for large problems.
5. The **work-stealing** (Blumofe & Leiserson 1999) scheduler gives each worker thread a queue. When worker threads produce new explanations, they schedule new work units into their own queue, and threads which run out of work can steal work from other threads' queues. We implemented a lockless work-stealing queue due to (Chase & Lev 2005).

Real-World Domains

We tested the performance of the schedulers described above on three domains. First, a simplified robotic kitchen cleaning domain involving picking up objects and putting them away (XPER). This domain is based on the European Union-FP7 XPERIENCE robotics project(Xpe 2011). Second, a logistics domain (LOGISTICS), involving the transporting of packages between cities using trucks and airplanes. This domain is based on a domain in the First International Planning Competition(Long & Fox 2003). Third and finally, a cyber security based domain (CYBER) based on recognizing the actions of hostile cyber attackers in a cloud based network computing environment.

For each domain a problem with a runtime between a second and a minute for the baseline algorithm was generated by hand. This problem was then presented to each of the algorithms running on a multi-processor machine using 1 to 12 cores. We will present data on the *speedup* of each algorithm on the problem, defined as the single threaded runtime divided by the runtime with a larger number of threads. Ideally we would like to achieve linear speedup (speedup equal to the number of threads). In the following graphs, we compute the speedup against the baseline runtime of the original algorithm. In later figures, where the baseline implementation is not included, we instead compute the speedup by comparing the runtime for a single thread and the runtime for the current number of threads.

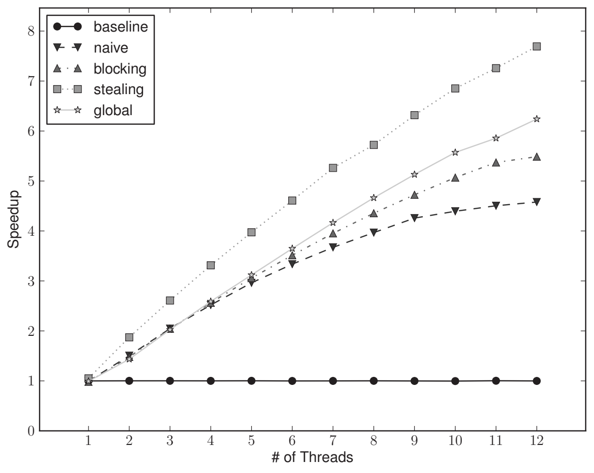

Figure 2: Speedup for CYBER domain vs. # of threads.

Figures 2, 3, 4 show the average speedup for each scheduler as we vary the number of threads available. Each data point was generated from the average of 20 runs. Comparing the results for different schedulers, on all three problem domains, the work-stealing scheduler remains the clear winner; the next best scheduler varies depending on the domains but the work-stealing scheduler dominates the others. The work-stealing scheduler

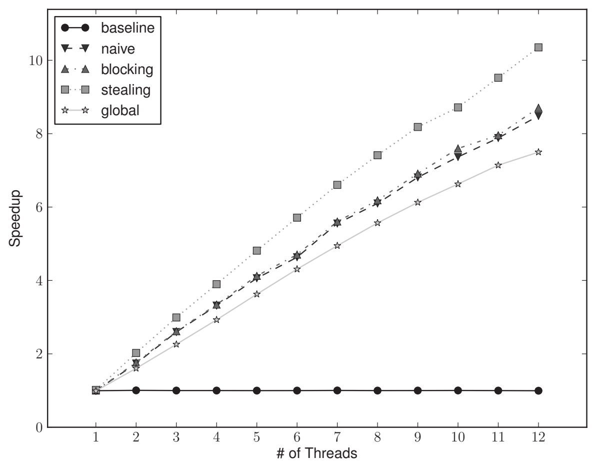

Figure 3: Speedup for XPER domain vs. # of threads.

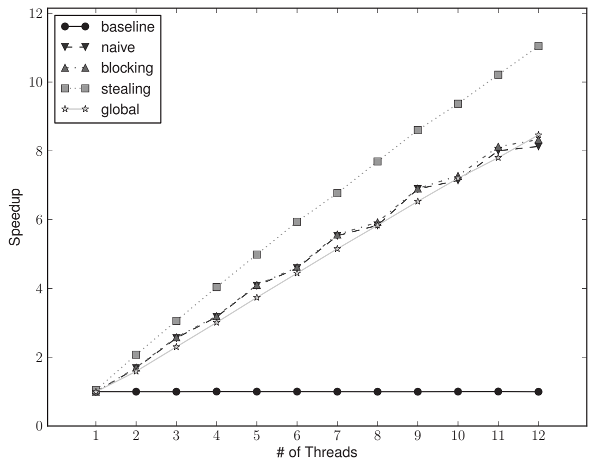

Figure 4: Speedup for LOGISTICS domain vs. # of threads.

does this by ensuring threads which are starved for work can rapidly find more, and the lockless work-stealing deque implementation has very low overhead. Given this convincing success, the remainder of our experiments focus on the work-stealing scheduler.

In Figure 5, we compare the speedups achieved on all three domains, using the work-stealing scheduler. The algorithm performs significantly worse on the CYBER domain than the XPER and LOGISTICS domains. Looking at the respective runtimes provides us with a clue as to why. The CYBER domain problem runs much faster than the others. For comparaison, with a single thread the CYBER domain problem runs in around 1 second, the LOGISTICS domain problem in around 25 seconds, and the XPER domain problem in around 60 seconds. This suggests, that the CYBER domain may simply have less to work to parallelize. Since the chief determiner of the runtime for the single threaded case is the number of explanations to be considered, we decided to explore if the structure of the plans in the domain could impact the speedup.

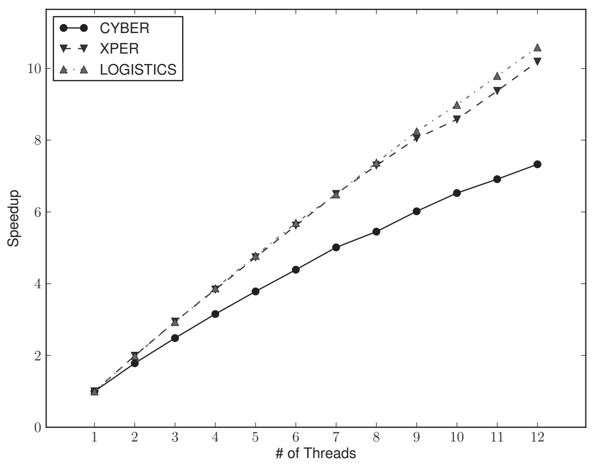

Figure 5: Speedup of **work-stealing** across all domains.

Synthetic Domains

To study how the structure of the plans within the domains affects the amount of work to be done and therefore the possible speedup, we created six synthetic domains, systematically varying the plan grammar, while maintaining the same input sequence of observations. We explored two different ways in which the plan grammer could be varied. First by changing the causal ordering of the actions within the plans, second by varying the anchor actions selected for the plans. We discuss each in turn. (Geib & Goldman 2009) showed that partial orderness in the plan grammar could result in large numbers of alternative explanations when using gramatical methods for plan recognition. We therefore explored two partially ordered plan structures (see Figure 6), which we will refer to as *order FIRST* where there is a single first element of the plan that all other actions must follow, and *order LAST* where there is a single last element that all actions must precede.

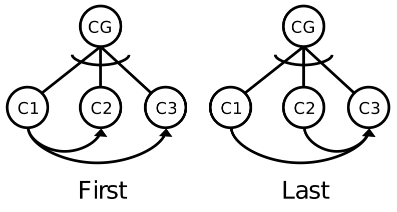

Figure 6: Causal structures for plans.

(Geib & Goldman 2009) also showed the effects of partial ordering can be influenced by the choice of anchors in a lexicalized plan grammar. Therefore, for our synthetic domains, we assumed complete tree structured plans of depth two with a uniform branching factor of three resulting in nine step plans. We then numbered the actions of the plan from left to right and on the basis of these indicies systematically varied the anchor of the plans from the far left to the far right. Given the branching factor of three for each subplan, this resulted in three possible values for the anchor which we will call: *anchor LEFT, anchor MID*, and *anchor RIGHT*, corresponding to the anchor being assigned to the leftmost action in the subplan the rightmost action of the subplan or the middle action in the subplan. As an example of only a sub part of the plan, the following is a set of CCG grammars for a three step, order FIRST plan, like that shown in Figure 6.

CCG: 5

FIRST-LEFT:

$act1 := GC/\{C2, C3\}.$

$act2 := C2.$

$act3 := C3.$

FIRST-MID:

$act1 := C1.$

$act2 := (GC\backslash\{C1\})\backslash\{C3\}$ *or* $(GC\backslash\{C1\})/\{C3\}.$

$act3 := C3.$

FIRST-RIGHT:

$act1 := C1.$

$act2 := C2.$

$act3 := (GC\backslash\{C1\})\backslash\{C2\}$ *or* $(GC\backslash\{C1\})/\{C2\}.$

As in the above grammars, in the future, we will denote each synthetic test domain grammar by its ordering feature and its anchor feature.

To quantify how much work is done by the algorithm for each grammar, during recognition we recorded the number of explanations that were generated both during the intermediate stages of processing as well as the final number of explanations generated for all of the domains. The results are presented in Table 1.

Domain	Intermediate	Final
FIRST-LEFT	1115231	330496
FIRST-MID	209	16
FIRST-RIGHT	5438	1296
LAST-LEFT	208326	48384
LAST-MID	1106489	416016
LAST-RIGHT	35	1
CYBER	74487	26632
XPER	710549	1149149
LOGISTICS	1628890	995520

Table 1: Explanations generated by each domain

To confirm our hypothesis that the number of explanations generated is a reasonable metric of the amount of time taken, Figure 7 is a scatter plot, showing the runtime of the work-stealing algorithm in seconds against the sum of the intermediate and final number of explanations for all of the domains. Note that FIRST-MID and LAST-RIGHT are basically on top of one another

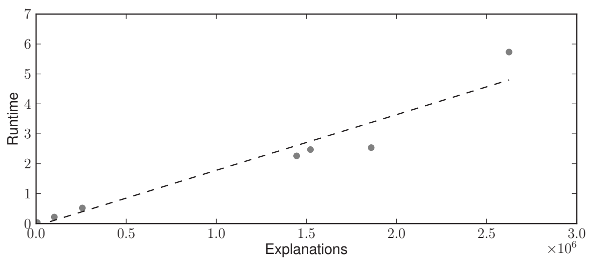

Figure 7: Runtime vs. total explanations, for all domains.

down almost on the origin. From this, we can see that the growth in runtime is roughly proportional to the total number of explanations generated for each problem, giving us strong reason to believe the total number of explanations is a reasonable metric for the amount of work done.

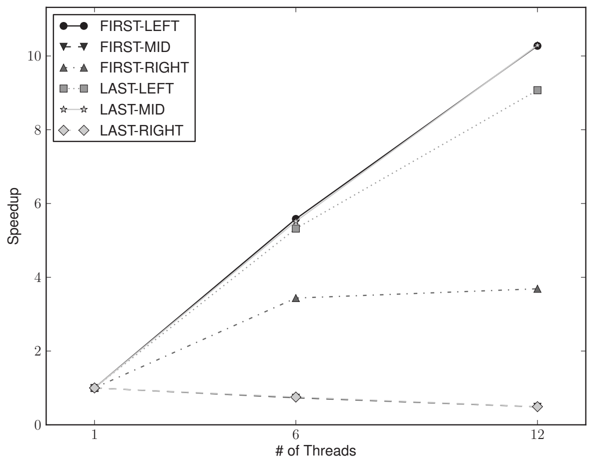

Figure 8: Speedup of the synthetic domain problems with increasing # of threads. The data points for the FIRST-LEFT and LAST-MID, as well as the FIRST-MID and LAST-RIGHT series overlap extremely closely.

Next, Figure 8 plots the speedup for the work stealing algorithm on the same observation stream for each of the synthetic domains. As expected it shows a clear difference in speedup depending on the structure of the plans and the grammar used to describe it. Comparing Figure 8 to Table 1 also shows a clear correlation. The LAST-RIGHT and FIRST-MID domains which generate only a handful of explanations have limited speedup, while the FIRST-LEFT and LAST-MID which generate tens of thousands of explanations and exhibit close to linear speedup. This gives us strong reason to believe that the differences in the speed up are a result of the differences in the number of explanations are generated.

This shows, that when more explanations are posible according to the grammar, more work is required, therfore more threads can be kept busy, and a greater speedup is achievable. However, the converse is also true. Fewer explanations in a domain, means that less work needs to be done, and for small enough problems there will be no significant gain in the runtime for a parallel implementation. Therefore, to help in real world deployment, we need to be able to identify when a parallel implementation is worth the cost.

To identify this, Figure 9 is a second scatter plot graphing speedup achieved with 12 threads against the base runtime with 1 thread for each of the problem domains. Its shows that for runs that take longer than around 5 seconds, we achieve 10-fold speedup, very close to the ideal, 12-fold speedup, making parallelism worth while. For shorter runs, there is much less benefit to the multithreaded implementation.

Our analysis also suggests that for real world domains with plan grammars with predominately LAST-RIGHT or FIRST-MID structure (where both the causal structure of the plan and the CCG grammar's anchors act to reduce the number of explanations) parallelism will be less helpful.

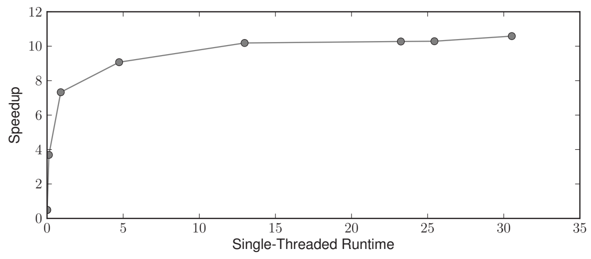

Figure 9: Speedup vs runtime, for all domains.

Conclusion

This paper has shown that parallelization using a work-stealing scheduling regime can be usefully applied to significantly speed up the processing of the ELEXIR plan recognition system. The multithreaded implementation discussed in this paper allows us to use the ubiquitous modern multi-core machines to explore domains which would previously have been computationally intractable. Further, it demonstrates that using the causal structure of the plan and correctly choosing the anchors for a CCG representation of plans can have a significant impact on the effectiveness of parallelization by preemptively taming of the complexity that results from partially ordered plans. Finally it suggests that parallelization should not be universally applied. For some domains and problems, the costs of parallelization may equal the gains, and it suggests some practical rules of thumb for when this may happen when using ELEXIR.

Acknowledgements

The work in this paper was supported by the EU Cognitive Systems project Xperience (EC-FP7-270273) funded by the European Commission.

References

Blumofe, R. D., and Leiserson, C. E. 1999. Scheduling multithreaded computations by work stealing. *J. ACM* 46(5):720–748.

Chase, D., and Lev, Y. 2005. Dynamic circular work-stealing deque. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, SPAA '05, 21–28. New York, NY, USA: ACM.

Curry, H. 1977. *Foundations of Mathematical Logic*. Dover Publications Inc.

Evans, J. 2006. A scalable concurrent malloc(3) implementation for freebsd.

Geib, C. W., and Goldman, R. P. 2009. A probabilistic plan recognition algorithm based on plan tree grammars. *Artificial Intelligence* 173(11):1101–1132.

Geib, C., and Goldman, R. 2011. Recognizing plans with loops represented in a lexicalized grammar. In *Proceedings of the 25th AAAI Conference on Artificial Intelligence (AAAI-11)*, 958–963.

Geib, C. 2009. Delaying commitment in probabilistic plan recognition using combinatory categorial grammars. In *Proceedings IJCAI*, 1702–1707.

Herlihy, M., and Shavit, N. 2012. *The Art of Multiprocessor Programming, Revised First Edition*. Elsevier.

Long, D., and Fox, M. 2003. The 3rd international planning competition: Results and analysis. *Journal of Artificial Intelligence Research* 20:1–59.

Sidner, C. L. 1985. Plan parsing for intended response recognition in discourse. *Computational Intelligence* 1(1):1–10.

Steedman, M. 2000. *The Syntactic Process*. MIT Press.

Vilain, M. B. 1990. Getting serious about parsing plans: A grammatical analysis of plan recognition. In *Proceedings AAAI*, 190–197.

Vilain, M. 1991. Deduction as parsing. In *Proceedings AAAI*, 464–470.

2011. Xperience project website. http://www.xperience.org/. Accessed: 30/01/2013.