# Parallelizing Plan Recognition

Christopher W. Geib, Christopher E. Swetenham

■ *Modern multicore computers provide an opportunity to parallelize plan-recognition algorithms to decrease run time. Viewing plan recognition as parsing based on a complete breadth first search, makes ELEXIR (engine for lexicalized intent recognition) (Geib 2009, Geib and Goldman 2011) particularly suited for parallelization. This article 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 provides close to linear speedup (up to a maximum number of processors), and that features of the problem domain have an impact on the achieved speedup.*

The ubiquity of multicore processors provides an opportunity to speed up all computationally expensive algorithms. Any algorithm that can be parallelized can make use of the available multiple cores to drive down its overall run time. 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. We believe this is a result of two issues.

First, the processing performed by many AI algorithms is not obviously parallelizable and can require significant effort to make it so. For example, naive implementations of AI planning based on regression or progression search are difficult to parallelize. In most algorithms the search for a plan can arrive at the same world state by multiple search paths. This information must be maintained and shared between the parallel processing threads. To do this can require nontrivial restructuring of the algorithm.

Second, even given an algorithm that is easily amenable to parallelization, and given a desire to implement it, there is a question of how to apportion work to the various threads to gain maximum benefit. There are multiple algorithms for making this choice, and trade-offs between the cost of the additional code to support the threading and work allocation decisions. The efficacy of any particular method can best be evaluated empirically. This may necessitate multiple rounds of implementations and experimentation. That said, algorithms that are well suited to this kind of bounded parallelism could benefit from a better understanding of the trade-offs required to make full use of easily obtainable modern computer architectures.

  ISSN 0738-4602

This article 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 and 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 article will also show that specific features of the domain can have a significant impact on the achieved speedup.

The remainder of this article is organized as follows. First we will provide some background followed by 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 processing threads and their respective strengths and weaknesses. We will then present the results of testing these allocation algorithms in multiple domains and discuss the impact of various domain-level features that can affect 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. It is worth distinguishing plan recognition from activity recognition (Liao, Fox, and Kautz 2007). In general, the objective of activity recognition is to produce a labeling of a sequence of observations. For example imagine that we have a stream of video. Generally, the objective of activity recognition is the labeling of each frame of the video with an unstructured identifier that indicates the single action that is being done in that frame.

As a result, activity recognition often focuses on dealing with sensor noise in the environment. As such, methods and algorithms that have been successful at recognizing structure in noisy sensor streams like HMMs (Modayil, Bai, and Kautz 2008), CRFs (Liao, Fox, and Kautz 2007; Vail and Veloso 2008), and DBNs (Hoogs and Perera 2008) have been very successful at activity recognition.

Plan recognition, in contrast, is about combining the actions indicated by such labelings into more complex structures that capture the larger plans being executed. Generally if a plan has temporal extent and multiple substeps, plan recognition is interested in identifying where in the execution of the plan the agent is and those actions that are likely to be executed next.

In contrast, plan recognition is fundamentally about combining low-level observations into larger structures, that is, recognizing and building new structure. This has resulted in a different set of tools being used. For example: abstract HMMs (Bui, Venkatesh, and West 2002), graph covering algorithms (Kautz and Allen 1986), and even parsing (Vilain 1991, Geib 2009) have all been successfully used to do plan recognition. With this in mind, plan recognition and activity recognition can be seen as playing very different, but complimentary roles, with plan recognizers taking as input the activities generated by activity recognition algorithms.

With this said, this work falls squarely in the area of plan recognition. Following other work on grammatical methods for plan recognition (Sidner 1985; Vilain 1990, 1991), ELEXIR (Geib 2009, Geib and Goldman 2011) views the problem as one of parsing. That is, like natural language processing (NLP) in which a formal grammar specifies those sequences of words that form acceptable sentences of the language, we could imagine defining an action grammar that describes the set of all plans to be recognized. Recognizing a single plan can then be viewed as deriving a single parse for the sequence of observations based on the formal grammar for the plans.

To do complete probabilistic plan recognition we can view the problem as weighted model counting based on the possible parses. Each parse is viewed as an explanatory model or explanations for the observed actions. If we build a complete and covering set of such explanations we can then establish a probability distribution over this set to provide each explanation with a weight. On the basis of the probability distribution we can then compute the probability of any particular goal or plan structure by summing the probability mass of those explanations that contain it.

An obvious first reaction to this proposal is that computing the complete and covering set of all parses for the action grammar could be a very computationally expensive process. In our experience, there are a number of ways to reduce the number of parses that need to be generated, making this a viable approach for real-world application. However, for some domains, the number of possible parses is necessarily quite large, and this was one of the prime motivators for exploring the possibilities for parallelizing ELEXIR.

Note that in order for the weighted model counting approach to be easy to parallelize two things must be the case. First, it must be easy to parallelize the parsing of the observations into explanations. Second, computing the probability distribution must be easy to parallelize. For the grammar formalism that we have chosen and the probability model that we use both of these are true. In the following two sections we will discuss the details of the grammar and the probability models for ELEXIR.

### ELEXIR Plan Grammars

In ELEXIR, plans are represented using combinatory

**CCG: 1**
dialCellPhone := (CHAT/{T})\{G}.
talk := T.
getCellPhone := G.

**CCG: 2**
dialCellPhone := D.
talk := (CHAT\{D})\{G}.
getCellPhone := G.

**CCG: 3**
dialCellPhone := D.
talk := T.
getCellPhone := (CHAT/{T})/{D}.

*Figure 1. Three Different Grammars for the Plan of Using a Cell Phone.*

a. CCG: 1. b. CCG: 2. c. CCG: 3.

categorial grammars (CCGs) (Steedman 2000), one of the so-called lexicalized grammars (Schabes 1990). Lexicalized grammars are a relatively recent development in NLP. In lexicalized grammars, all language-specific information is moved into rich data structures called categories. The grammar's lexicon associates with each terminal symbol a set of such categories. Lexicalized grammars then use a small set of language-independent rules called combinators to combine lexical categories to produce a parse of the sequence of tokens into a complete sentence.

Thus, parsing in such grammars abandons the traditional search through the space defined by the application of multiple formal grammar rules. Instead parsing is viewed as a search defined by the category chosen for each terminal symbol and the set of combinators used to combine categories to build the parse.

For an action grammar the only significant difference is that the terminal symbols denote observable actions as opposed to words in NLP. In the following we will use the more general term *observable* to denote the terminal symbols of the grammar.

In CCGs, the categories that each observable is associated with are defined recursively as atomic or complex categories.

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

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

Like the observable actions they are associated with, each category can be thought of as a function. Basic categories are functions that take no arguments, while the slash in complex categories separates the arguments ($\{W, X, \ldots\}$) on the right side of the category from the result ($Z$) on the left side. The direction of the category's 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 (leftward arguments), should be observed before it, to produce the result. Finally, if a category has multiple arguments within a set braces, they are treated as 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 grammar in figure 1a. 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 action *dialCellPhone* has a single complex category that captures the structure of the plan for chatting to a friend.

Lexicalized plan grammars also require a design decision about which actions should carry which parts of the structural information for a plan. In CCG: 1 the *dialCellPhone* action was chosen to have a category that had most of the structure needed to recognize the plan for *CHAT*. 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 CCG: 1 *dialCellPhone* is the anchor for the plan to *CHAT*.

In the design of the grammar we could have made other choices. For example, in CCG: 2 and CCG: 3 we see what the grammar would have looked like if we had chosen *talk* or *getCellPhone* as the anchor for *CHAT.* This would have resulted in a different set of categories (note the introduction of the category *D*) and a different resulting lexicon (figure 1b).

The anchors chosen for a particular grammar can have a significant impact on the run time of plan recognition (Geib 2009). 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 2 that parses the observation sequence: *getCellPhone*, *dialCellPhone*, *talk* using CCG: 1. As each observation is encountered, it is assigned a category as defined by the plan grammar. Combinators then combine the categories to produce explanations. In this example, leftward application of the categories for *dialCellPhone* and *getCellPhone* is used first to produce *CHAT*/ {*T*}. Rightward application then combines it with *T* to produce a complete parse for a plan to *CHAT*. We will discuss the details of the parsing algorithm that does this next.

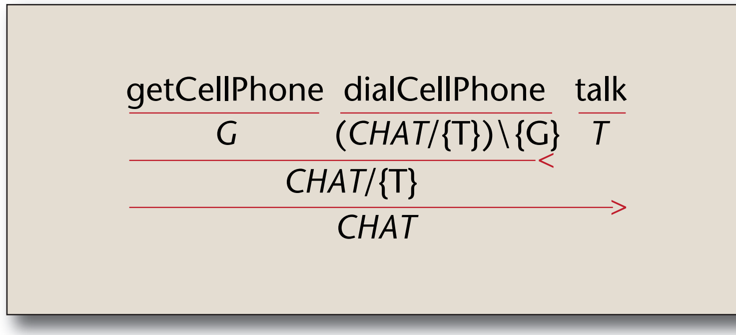


*Figure 2. Parsing Observations with CCG Categories.*

**CCG: 4**
dialCellPhone := (CHAT/{T})\{G}. acceptable
dialCellPhone := (CHAT\{G})/{T}. unacceptable

*Figure 3. CCG: 4.*

## ELEXIR Parsing Algorithm

To enable incremental parsing of multiple interleaved plans, ELEXIR does not use a preexisting parsing algorithm from NLP. 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 cell phone 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 (figure 3), this rules out reversing the order of the arguments to *dialCellPhone* in our example CCG: 1.

We call such grammars 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 easily to 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 ELEXIR produce 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 be applied once to any pair of categories. This is done so as not to force the combination of categories in case they are needed for application or composition with a category for an observation that has yet to be seen.

This two-step algorithm both restricts observations to take on only 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 an as yet unseen category.

The algorithm we have just discussed allows ELEXIR to build the complete and covering set of explanations for an observed stream of actions given a particular grammar. To compute the conditional probability for any particular goal it then needs to compute a probability distribution over this set. In the next subsection we discuss the construction of the probability of each explanation and, on the basis of this distribution, the conditional probability of any individual goal or plan.

## ELEXIR Probability Model

Traditionally in probabilistic plan recognition the objective is to compute the conditional probability for all of the possible root goals (Charniak and Goldman 1993). In weighted model counting, given that we can compute the exclusive and exhaustive set of explanations and that we can compute the conditional probability of each explanation, then the conditional probability for any given goal is given by the following formula:

Definition 1.1

$$P(goal \mid obs) = \sum_{\{exp_i \mid goal \in exp_i\}} P(exp_i \mid obs)$$

where $P(exp_i|obs)$ is the conditional probability of explanation $exp_i$ given the set of observations, *obs*. The conditional probability for the goal is just the sum of the probability mass associated with those explanations that contain the goal of interest.

Note this relies on (1) an exclusive and exhaustive set of explanations for the observations, and (2) being able to compute the conditional probability for each explanation, knowing there will be no more observations.

## The Probability of an Explanation

While there are a number of different probability models used for CCG parses in the NLP literature (Hockenmaier 2003, Clark and Curran 2004) we will extend a particularly simple one described by Hockenmaier (2003). For an explanation, *exp*, of a sequence of observations, $\sigma_1 \ldots \sigma_n$, that results in $m$ categories in the explanation, we define the probability of the explanation as:

Definition 1.2

$$P(exp \mid \{\sigma_1 ... \sigma_n\}) = \prod_{i=1}^{n} P(cinit_i \mid \sigma_i) \prod_{j=1}^{m} P(root(c_j))K$$

Where $cinit_i$ represents the initial category assigned in this explanation to observation $\sigma_i$, $root(c_j)$ represents the root result category of the $j$th category in the explanation, and $K$ is the constant product of the probability of each possible goal not being in the explanation. We provide motivation for these terms in this definition in turn.

The first product represents the probability of the given observations actually having their assigned CCG categories. This is standard in NLP parsing and assumes the presence of a probability distribution over the possible categories to which a given observation can be mapped. In NLP such probabilities are usually learned using large corpora of parsed text (Clark and Curran 2004). We note that ELEXIR allows the conditioning of this distribution based on the state of the world at the time the action is executed. Space constraints prevent providing a full exposition of ELEXIR's state model, but it does provided a facility to choose a distribution based on the state of the world at the time the action is executed.

The second product (and its associated constant) captures the probability that each category will not be part of a larger plan but instead represents a separate plan instance. This is not a part of traditional NLP models for two reasons. First, in NLP it makes no sense to consider the probability of multiple interleaved sentences. Second, in most NLP contexts the observations are known to be a whole sentence. Usually parsed text contains punctuation marks indicating sentence boundaries. In this setting it makes little sense to consider the probability that the sequence is anything but a single complete sentence.

However these assumptions do not hold for plan recognition. It is more than possible for a given sequence of observations to contain multiple interleaved plans of varying lengths, or to cover only fragments of multiple plans being executed (consider a set of multiday plans).

To address this we take the position that any action could be done for its own sake. Much prior work in plan recognition assumes a small distinguished set of acceptable goals. Instead, ELEXIR assumes that it is acceptable for any given action to be a root goal and to be executed by itself without regard to a more complex goal. Therefore, ELEXIR must be given a prior probability for each atomic category as a root goal. We would again note that ELEXIR actually allows the domain designer to condition the root probabilities based on the state of the world. In this case the conditioning must be done based on the initial state of the world. Again space constraints prevent a full exposition of ELEXIR's state model.

However such priors are not enough. To understand the second term in the above definition, we denote the set of all values of $root(c_j)$ for a given explanation, as *goals* (leaving the explanation implicit) and denote the probability of this particular set of categories being adopted as root goals as $P(goals)$. We also make the simplifying assumption of an independent prior for each category being a root goal. We represent the probability of an agent adopting a category $c$ as a root goal as $P(c)$ with each goal instance being chosen (or rejected) independently.

ELEXIR allows for multiple instances of a given category in goals (it is acceptable for $root(c_i) = root(c_j)$ where $i \neq j$). To do this, each goal is sampled as a geometric distribution. $P(c)$ represents the probability that category $c$ is a root goal in the explanation, and we keep sampling to see if there are more root instances of $c$. This means $P(c)^n(1 - P(c))$ represents the probability that there will be exactly $n$ root instances of category $c$ in an explanation. This is almost certainly an overestimate — intuitively the probability of multiple instances of a single goal decreases far more rapidly than this. Exploring more sophisticated models for this is an area for future research.

Assuming $|goals_c|$ represents the number of root instances of category $c$ in the explanation:

$$P(goals) = \prod_{c \in goals} P(c)^{|goals_c|}(1 - P(c)) \prod_{c \notin goals} (1 - P(c)).$$

Collecting all of the $1 - P(c)$ terms:

$$P(goals) = \prod_{c \in goals} P(c)^{|goals_c|} \prod_{\forall c \in C} (1 - P(c))$$

Now, the second term is a product over all the categories in the lexicon, and therefore a constant across all explanations and can be replaced with a constant $K$.

$$P(goals) = \prod_{c \in goals} P(c)^{|goals_c|} K$$

Rewriting in terms of the instances in the explanation yields the term seen in equation 1.2.

$$P(goals) = \prod_{j=1}^{m} P(root(c_j)K$$

With this understanding of ELEXIR's algorithm and probability model we can now discuss the features that make it amenable to parallelization.

## Parallelizing ELEXIR

ELEXIR's parsing algorithm not only makes effective use of the categories structure to reduce the search space, it also effectively creates a canonical ordering for the generation of explanations. This is what makes the ELEXIR algorithm particularly amenable to parallelization.

ELEXIR uses its 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 in the combinators that are applied. 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. Note, 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. As such, each addition of a category to an explanation or the use of a combinator splits the search space into complete and nonoverlapping subsearches. Such subsearches 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 nonoverlapping subsearches. 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.

We also note that the probability for each explanation can be computed in parallel. This computation depends only on the categories chosen for each observation and the set of root result categories in the explanation. The first of these can be maintained as the building of the explanation is performed. The second of these can be computed in a single pass over the explanation after the explanation has been built. It is only the final computation of the conditional probabilities for each individual goal that requires access to the complete set of explanations.

### Implementing Parallelization

Given a method to break up the search for explanations into disjoint subsearches, 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 choosing the size of work units requires making a trade-off 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,[1] which is designed for multithreaded applications, has much better contention and cache behavior, and showed much better speedups with larger numbers of threads in exploratory test experiments.

We then implemented four different scheduling policies to allocate the work to be performed across available hardware threads and compared these against the baseline run time of the original single-threaded algorithm. All implementations, other than the baseline, 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 is 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 five implemented policies next.

First, the *baseline implementation* is the original implementation, albeit with the thread-safety guarantees in place. This involved ensuring the reference-counting implementation used for releasing memory was thread safe, and guaranteeing that the shared data structures used were not modified for the duration of the search.

Second, the *naive scheduler* (Herlihy and 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.

Third, the *blocking scheduler* (Herlihy and 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.

Fourth, the *global queue* (Herlihy and 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 run time with different batch sizes, We determined a batch size of 32 to be adequate, although larger values may be preferable for large problems.

Fifth and finally, the *work-stealing* (Blumofe and 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 that run out of work can steal work from other threads' queues. We implemented a lockless work-stealing queue due to Chase and 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. 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 and 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 run time 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 multiprocessor machine using 1 to 12 cores. We will present data on the speedup of each algorithm on the problem, defined as the single threaded run time divided by the run time 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 run time of the original algorithm. This tells us how much faster we processed the input compared to using a single-threaded implementation. In later figures, where the baseline implementation is not included, we instead compute the speedup by comparing the run time for a single thread and the run time for the current number of threads.

Figures 4, 5, and 6 show the average speedup for each scheduler while varying the number of threads. Each data point was generated by averaging 20 runs. Comparing the results for different schedulers on all three problem domains, the work-stealing scheduler is the clear winner; the next best scheduler varies depending on the domains, but the work-stealing scheduler dominates the others.

The work-stealing scheduler does this by ensuring threads that 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 focused on the work-stealing scheduler.

Figure 7 compares 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

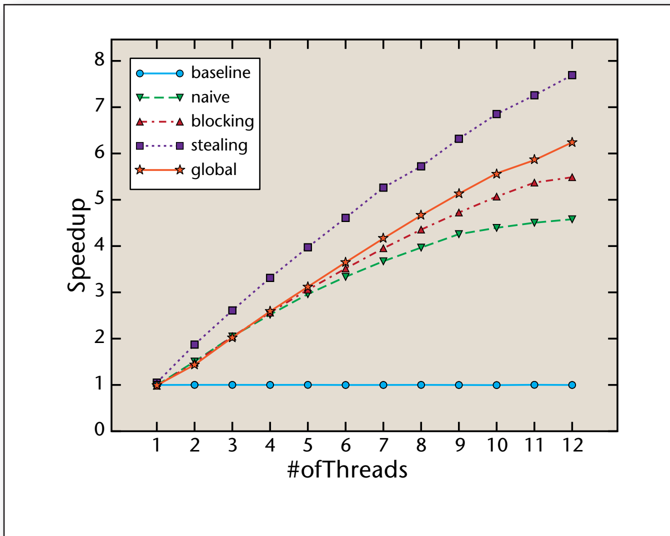


*Figure 4. Speedup for CYBER Domain Versus Number of Threads.*

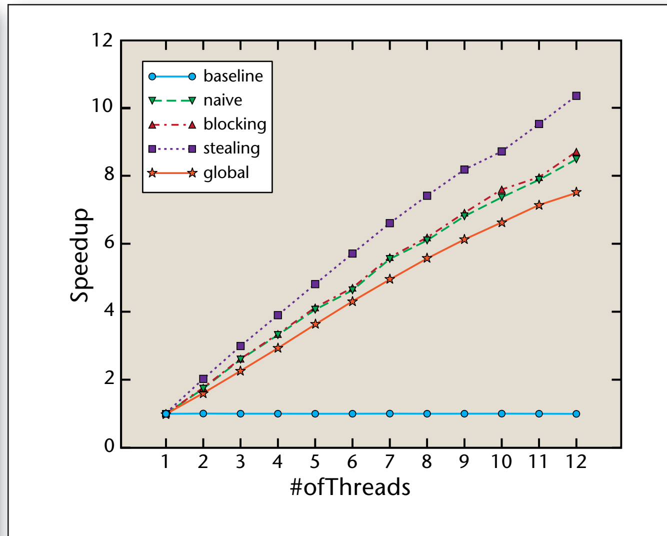


*Figure 5. Speedup for XPER Domain Versus Number of Threads.*

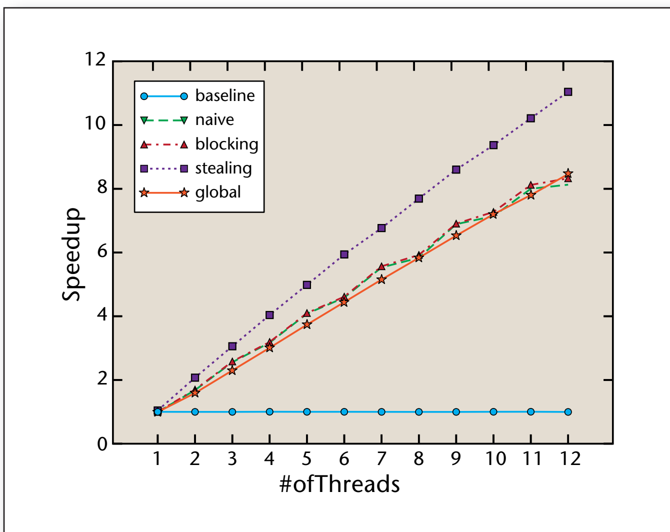


*Figure 6. Speedup for LOGISTICS Domain Versus Number of Threads.*

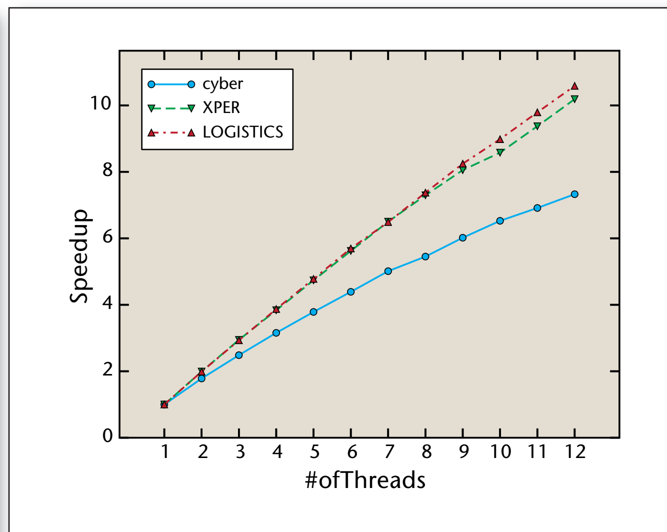


*Figure 7. Speedup of Work-Stealing Across All Domains.*

domains. Looking at the respective run times provides us with a clue as to why. The CYBER domain problem runs much faster than the others. For comparison, 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. Geib (2009) cites the number of explanations to be computed as the chief determiner of the run time for the single-threaded case, and that the structure of the plans and choice of anchors can significantly affect this. Therefore, we decided to explore whether the structure of the plans in the domain and the choice of anchors could affect the speedup.

## 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

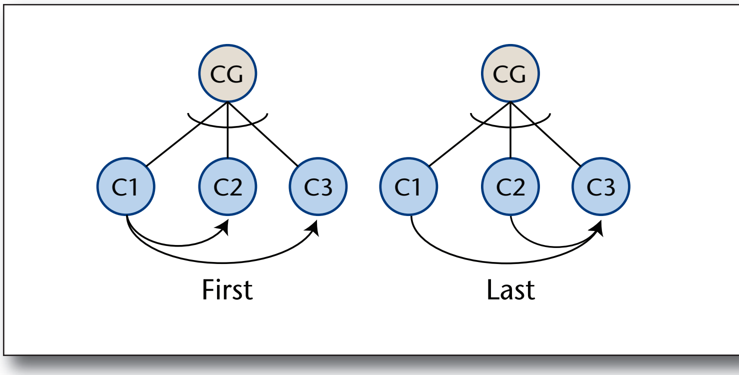


*Figure 8. Causal Structures for Plans.*

**CCG: 5**

*FIRST-LEFT:*
*act*1 := GC/{C2, C3}.
*act*2 := C2.
*act*3 := C3.
*FIRST-MID:*
*act*1 := C1.
*act*2 := (GC\{C1})\{C3}
*or* (GC/{C3})\{C1}.
*act*3 := C3.
*FIRST-RIGHT:*
*act*1 := C1.
*act*2 := C2.
*act*3 := (GC\{C1})\{C2}
*or* (GC/{C2})\{C1}.

*Figure 9. Set of CCG Grammars for a Three-Step, Order FIRST Plan.*

| 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.*

within the plans, second by varying the anchor actions selected for the plans. We discuss each in turn. Prior work (Geib and Goldman 2009) has shown 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 8), 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.

The effects of partial ordering can be influenced by the choice of anchors in a lexicalized plan grammar (Geib and Goldman 2009). 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 indices 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 subpart of the plan, figure 9 is a set of CCG grammars for a three-step, order FIRST plan, like that shown in figure 8.

As in the grammars in figure 9, 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. They confirm that varying the anchor feature can have a significant impact on the number of explanations generated by the algorithm, and thus the number of explanations can vary widely on the same domain with the same input if the grammar is different.

To confirm our hypothesis that the number of explanations generated is a reasonable metric of the amount of time taken, figure 10 is a scatter plot that shows the run time 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 down almost on the origin. From this, we can see that the growth in run time 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.

Next, figure 11 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 11 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, exhibit close to linear speedup. This gives us strong reason to believe that the differences in the speedup are a result of the differences in the number of generated explanations and therefore the number of processors that can be kept busy.

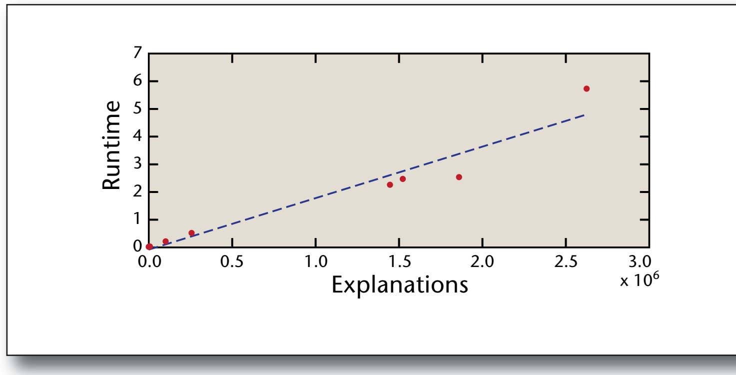


*Figure 10. Run Time Versus Total Explanations for All Domains.*

## Practical Implications

In the previous section, we have shown that the number of intermediate and final explanations can vary wildly depending on the structure of the domain. We now examine properties of all the domains examined so far, real and synthetic.

This indicates that when more explanations are possible according to the grammar, more work is required, therefore 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 run time 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 12 is a second scatter plot graphing speedup achieved with 12 threads against the base run time with 1 thread for each of the problem domains. It 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 worthwhile. 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. The amount of work required will already be reduced by the causal and gramatical constraints. Domains with grammars that do not use these tools to constrain the number of explanations will see significant benefits from parallelism.

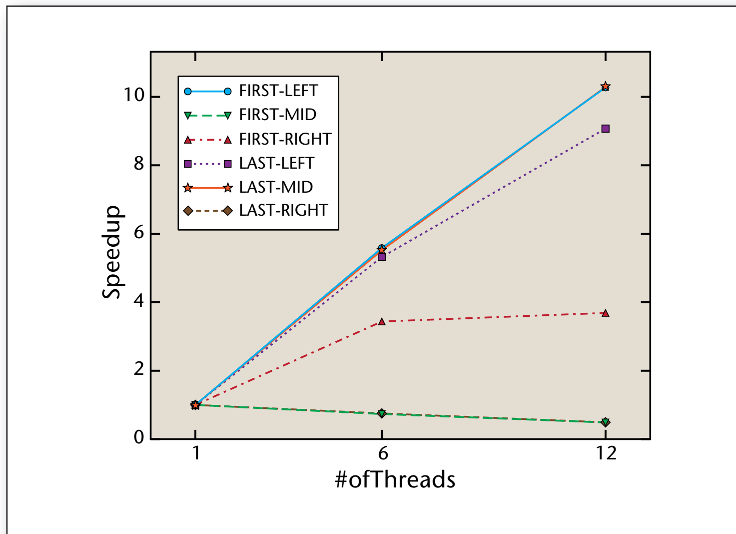


*Figure 11. Speedup of Synthetic Domain Problems with Increasing Number of Threads.*

The data points for the FIRST-LEFT and LAST-MID, as well as the FIRST-MID and LAST-RIGHT series overlap extremely closely.

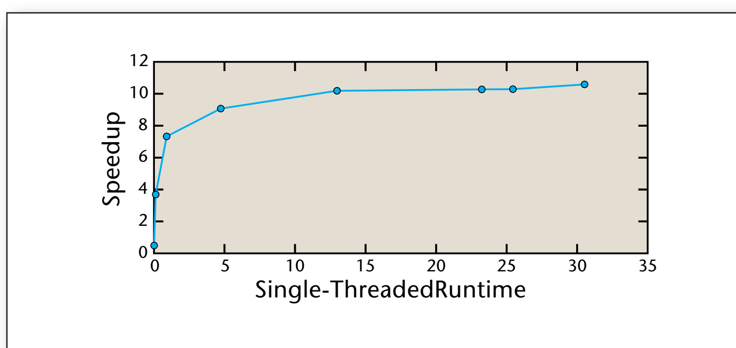


*Figure 12. Speedup versus Run Time for All Domains.*

## Conclusion

This article 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 article allows us to use the ubiquitous modern multicore machines to explore domains that 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 the complexity that results from partially ordered plans. Finally, it has shown that while parallelization is generally very helpful it 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.

In domains where there are a small number of possible explanations relative to the number of processors, and the length of the plans to be recognized is short, the costs of parallelization may very well outweigh the benefits. However, in such domains the single threaded implementation of ELEXIR already has very fast runtimes. Thus, the most encouraging result of this work is that the parallelization of the ELEXIR algorithm is most effective (speedup is greatest and the marginal costs of increasing parallelization lowest) precisely when the need is greatest, that is there are a large number of lengthy possible explanations for the observed actions.

The ELEXIR code base can be downloaded to enable others to experiment with it.[2]

## Acknowledgements

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

## Notes

1. See J. Evans (2006), A Scalable Concurrent Malloc(3) Implementation for Freebsd. people.freebsd.org/~jasone/jemalloc/bsdcan2006/jemalloc.pdf

2. www.planrec.org.

## References

Blumofe, R. D., and Leiserson, C. E. 1999. Scheduling Multithreaded Computations by Work Stealing. *Journal of the ACM* 46(5): 720–748.

Bui, H. H.; Venkatesh, S.; and West, G. 2002. Policy Recognition in the Abstract Hidden Markov Model. *Journal of Artificial Intelligence Research* 17: 451–499.

Charniak, E., and Goldman, R. P. 1993. A Bayesian Model of Plan Recognition. *Artificial Intelligence* 64(1): 53–79.

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: Association for Computing Machinery.

Clark, S., and Curran, J. 2004. Parsing the WSJ Using CCG and Log-linear Models. In *Proceedings of the 42th Annual Meeting of the Association for Computational Linguistics,* 104–111. Stroudsberg, PA: Association for Computational Linguistics.

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

Geib, C. 2009. Delaying Commitment in Probabilistic Plan Recognition Using Combinatory Categorial Crammars. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence* (IJCAI), 1702–1707. Palo Alto, CA: AAAI Press.

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. Palo Alto, CA: AAAI Press.

Geib, C. W., and Goldman, R. P. 2009. A Probabilistic Plan Recognition Algorithm Based on Plan Tree Grammars. *Artificial Intelligence* 173(11): 1101–1132.

Herlihy, M., and Shavit, N. 2012. *The Art of Multiprocessor Programming,* revised first edition. Amsterdam: Elsevier.

Hockenmaier, J. 2003. Data and Models for Statistical Parsing with Combinatory Catagorial Grammar. Ph.D. Dissertation, School of Informatics, University of Edinburgh, Edinburgh, Scotland.

Hoogs, A., and Perera, A. A. 2008. Video Activity Recognition in the Real World. In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence* (AAAI-08) 1551–1554. Palo Alto, CA: AAAI Press.

Kautz, H., and Allen, J. F. 1986. Generalized Plan Recognition. In *Proceedings of the Fifth National Conference on Artificial Intelligence* (AAAI-86), 32–38. Palo Alto, CA: AAAI Press.

Liao, L.; Fox, D.; and Kautz, H. 2007. Extracting Places and Activities from GPS Traces Using Hierarchical Conditional Random Fields. In *International Journal of Robotics Research* 26(1): 119 – 134.

Long, D., and Fox, M. 2003. The 3rd International Planning Competition: Results and Analysis. *Journal of Artificial Intelligence Research* 20: 1–59.

Modayil, J.; Bai, T.; and Kautz, H. A. 2008. Improving the Recognition of Interleaved Activities. Paper presented at the Tenth International Conference on Ubiquitous Computing, Seoul, South Korea, 21–24 September.

Schabes, Y. 1990. Mathematical and Computational Aspects of Lexicalized Grammars. Ph.D. Dissertation, Department of Computer and Information Science, University of Pennsylvania, Philadelpha, PA.

Sidner, C. L. 1985. Plan Parsing for Intended Response Recognition in Discourse. *Computational Intelligence* 1(1): 1–10.

Steedman, M. 2000. *The Syntactic Process*. Cambridge, MA: The MIT Press.

Vail, D. L., and Veloso, M. M. 2008. Feature Selection for Activity Recognition in Multi-Robot Domains. In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence* (AAAI-08) 1415–1420. Palo Alto, CA: AAAI Press.

Vilain, M. 1991. Deduction as Parsing. In *Proceedings of the Seventh National Conference on Artificial Intelligence,* 464–470. Palo Alto, CA: AAAI Press.

Vilain, M. B. 1990. Getting Serious About Parsing Plans: A Grammatical Analysis of Plan Recognition. In *Proceedings of the Eighth National Conference on Artificial Intelligence,* 190–197. Palo Alto, CA: AAAI Press.

**Christopher Geib** is an associate professor in the College of Computing and Informatics at Drexel University. His principle research interests are in artificial intelligence(AI) methods for intent recognition and planning more generally probabilistic reasoning about actions using formal grammars, and the interface between continuous control systems and logic-based reasoning systems. Geib's work on intent recognition has been applied in multiple application areas including assistive care for the elderly, human robot interaction, and computer network security.

**Christopher E. Swetenham** is a research associate on the University of Hong Kong's team in the DARPA Robotics Challenge. After completeing his undergraduate degree at the University of Cambridge, he spent six years working in the games industry. He then obtained a master's degree from the University of Edinburgh School of Informatics before pursuing a career in robotics.