Parallelising Plan Recognition

Chris Swetenham

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

(Graduation date: November 2012)

Abstract

Plan Recognition is an area of Artificial Intelligence research with many potential applications in which it is desirable to perform it in real time. In order to devise an scalable, efficient solution for plan recognition, we study the ELEXIR algorithm for plan recognition in C++. We find it well suited to parallelisation using multithreaded job scheduling techniques and apply several scheduling techniques, including lock-free work stealing. We analyse and report on the speed and scalability of these techniques when applied to plan recognition.

Acknowledgements

Thanks to my supervisor, Chris Geib, for his support during this project.

Declaration

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

(*Chris Swetenham*)

Table of Contents

1 Introduction 1

2 Background 3

2.1 Plan Recognition . . . 3
2.2 Combinatory Categorial Grammars . . . 4
2.3 Applying CCGs to Plan Recognition . . . 5
2.4 The ELEXIR Algorithm . . . 6
2.5 Concurrency Techniques . . . 7
2.6 Multicore Task Scheduling Techniques . . . 8

3 Related Work 11

4 Initial Work 13

5 Method 0 - Original Single-Threaded ELEXIR 17

5.1 Implementation . . . 17
5.2 Evaluation . . . 18

6 Method 1 - One Thread Per Task 19

6.1 Implementation . . . 19
6.2 Evaluation . . . 20

7 Method 2 - One Queue Per Thread 23

7.1 Implementation . . . 23
7.2 Evaluation . . . 25

8 Method 3 - Lock-Free Work-Stealing 27

8.1 Implementation . . . 27
8.2 Evaluation . . . 30
    8.2.1 Batch Size Parameter . . . 30
    8.2.2 Thread Count . . . 30

9 Method 4 - Single Global Queue 33
- 9.1 Implementation 33
- 9.2 Evaluation 35
  - 9.2.1 Batch Size Parameter 35
  - 9.2.2 Thread Count 35

10 Evaluation 39

11 Conclusion 43

Bibliography 45

List of Figures

6.1 Method 1 runtime with 1 to 4 threads on Sherlock, XPERIENCE 20
6.2 Method 1 runtime with 1 to 4 threads on Sherlock, Logistics . . . 21
6.3 Method 1 runtime with 1 to 16 threads on Catzilla, XPERIENCE 21
6.4 Method 1 runtime with 1 to 16 threads on Catzilla, Logistics . . . 22

7.1 Method 2 runtime, 1-4 threads on Sherlock, XPERIENCE 25
7.2 Method 2 runtime, 1-4 threads on Sherlock, Logistics 26
7.3 Method 2 runtime, 1-16 threads on Catzilla, XPERIENCE 26
7.4 Method 2 runtime, 1-16 threads on Catzilla, Logistics 26

8.1 Method 3 runtime, varying batch size on Sherlock, XPERIENCE . 30
8.2 Method 3 runtime, 1-4 threads on Sherlock, XPERIENCE 31
8.3 Method 3 runtime, 1-4 threads on Sherlock, Logistics 31
8.4 Method 3 runtime, 1-16 threads on Catzilla, XPERIENCE 31
8.5 Method 3 runtime, 1-16 threads on Catzilla, Method 3, Logistics . 32

9.1 Runtime, varying batch size on Sherlock, Method 3, XPERIENCE 36
9.2 Method 4 runtime, 1-4 threads on Sherlock, XPERIENCE 36
9.3 Method 4 runtime, 1-4 threads on Sherlock, Logistics 37
9.4 Method 4 runtime, 1-16 threads on Catzilla, XPERIENCE 37
9.5 Method 4 runtime, 1-16 threads on Catzilla, Logistics 37

10.1 All methods throughput, 1-4 threads on Sherlock, XPERIENCE . 39
10.2 All methods throughput, 1-4 threads on Sherlock, Logistics 40
10.3 All method throughput, 1-4 threads on Catzilla, XPERIENCE . . 40
10.4 All methods throughput, 1-4 threads on Catzilla, Logistics 41
10.5 Histogram of generated expression counts 42

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 propose to improve the performance of plan recognition by parallelising the core of the algorithm, allowing it to take advantage of modern multicore machines. 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.

We will first cover the background for the project, and discuss related work. We then discuss the initial work done to prepare the algorithm for a multicore execution. For each of the algorithms studied, we discuss its particular implementation, and study its performance as we scale up the number of available threads. Finally, we compare the results for the algorithms studied, draw conclusions, and explore avenues for future work.

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. 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 [1]. 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 [2]. 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 [3].

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. In this project, we use two grammars crafted by hand to fit different problem domains. The formal grammars are defined in the language of *Combinatory Categorial Grammars*, discussed below.

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 [4] 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. [1]

Taking a simple example from the Logistics domain:

```
observations: [
  loadTruck(p0, t0, c1),
  driveTruck(t0, c1, c0),
  unloadTruck(p0, t0, c0)
];
```

---

[1] 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.

We have three observations: package `p0` is loaded into truck `t0` at location `c1`, the truck drives to location `c0`, and the package is unloaded there. We then assign basic or complex categories to each observation:

```
loadTruck(p0, t0, c1) :
  inTruckC(p0, t0)
driveTruck(t0, c1, c0) :
  (deliverPkgC(o, c0)/notInTruckC(o, t0))\inTruckC(obj, t0)o
unloadTruck(p0, t0, c0) :
  notInTruckC(p0, t0)
```

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 $X \leftarrow Y,\ X\backslash Y$. Note that package `o` is unbound until the complex category is unified with an argument.

$$\underbrace{\texttt{inTruckC(p0, t0), (deliverPkgC(o, c0)/notInTruckC(obj, t0))\textbackslash inTruckC(o, t0)}}\texttt{, notInTruckC(p0, t0)}$$

$$\underbrace{\texttt{deliverPkgC(p0, c0)/notInTruckC(p0, t0), notInTruckC(p0, t0)}}$$

$$\texttt{deliverPkgC(p0, c0)}$$

We reduce the input to an explanation consisting of a single basic category, `deliverPkgC(p0, c0)`. *Combinatory Categorial Grammars* [4] (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: $X/Z \leftarrow X/Y,\ Y/Z$. Type-raising transforms basic categories to complex categories. The rightward type-raising operation has the form: $T/(T\backslash X) \leftarrow X$.

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 [5].

2.4 The ELEXIR Algorithm

The ELEXIR algorithm [6] 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*. When we unify two categories, if the same feature is bound in both of them, it must be bound to the same value; otherwise unification fails. If the same feature is bound in one of them and unbound in the other, the unbound feature takes the value of the bound feature. *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.

In the ELEXIR algorithm, each existing explanation is processed independently when a new action is added to the observations. For each existing explanation, the algorithm tries to apply the three combinators mentioned above between pairs of categories. Each successful application creates a new explanation; the case where no combinators are applied is also a valid new explanation. This process does not refer to any other existing explanations, and does not make modifications to the problem state outside the set of current explanations. This makes ELEXIR well suited to parallelising using task scheduling: depending on the overhead of the method we are using, we can create tasks containing some small or large number of explanations, and schedule these tasks to be executed independently across multiple threads. We now look at techniques for parallelism and how we might apply them to speed up this algorithm.

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. Mutexes are used to guard access to an object. Before accessing the object, threads must lock the mutex. Only one thread at a time is permitted to hold the mutex locked; other threads which attempt this will sleep until the lock becomes available. This mechanism allows an object designed to be accessed in a single-thread manner, be safely accessed by multiple threads. Condition variables provide a way for threads to wait until a condition is satified by a shared object, for example for a queue to become non-empty. Threads which modify the object in such a way that the condition might now hold where it previously did not, should notify the condition variable, which will then wake the sleeping threads. These threads will then test whether the condition does indeed hold. Condition variables are often used in conjunction with mutexes, to guard access to the object being waited on.

Non-blocking methods offer 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.

There are many potential performance issues in multi-threaded code. The two most relevant to this project are *lock contention*, where threads spend a lot of time waiting on the same mutex for access to a shared object; and *cache contention*, where threads concurrently access memory in the same cache line, causing delays as it is acquired by the caches of the processors the threads are executing on. Results from cachegrind allow us to discover potential cases of cache contention.

2.6 Multicore Task Scheduling Techniques

Many problems can be broken down into individual units of work to be executed across several threads [7]. 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. We will discuss several possible techniques for job scheduling on multicore machines, which we will evaluate in parallelising the ELEXIR algorithm.

Global Queue 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.

Blocking Queues 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.

Work-Stealing Queues 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 created by a thread 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.

Chapter 3

Related Work

We now look at some related efforts in terms of Plan Recognition and parallelisation. Various Approaches to probabilistic Plan Recognition based on Hidden Markov Models (HMMs) and Conditional Random Fields (CRFs) exist. The AHMM model [8] allows online, probabilistic Plan Recognition, including a representation of the state of the world and its effect on the planning of agents; however, it does not deal with interleaved or partially-ordered plans. Graph-based methods reduce plan recognition to a graph covering problem. The algorithm in [9] is able to perform Plan Recognition lazily, computing explanations efficiently and only as needed, but it does not provide probabilities for the explanations, and does not deal with interleaved or partially-ordered plans.

There is extensive literature on parallelising search problems [10], and some of this could be applicable to Plan Recognition algorithms. However, since ELEXIR performs an exhaustive rather than heuristic search, they are not directly applicable to this project. There has been some work on parallelising the parsing problem [11], however this applies mainly to traditional parsing problems based on context-free grammars with no ambiguity, where only a single parse will result from a given input. These methods do not need to deal with contextual effects and multiple parse explanations. They are applicable to much larger inputs presented in batch form, such as program source millions of lines long. Parallelisation strategies are therefore quite different from the ones we consider, for example by starting simultaneous parses at multiple points in the input stream and combining the resulting partial parse trees.

Clark and Curran have developed a parallel implementation for learning of CCGs in Natural Language [12], which runs in parallel and distributed fashion on a Beowulf cluster. However, the parsing aspect is not a complete parallel parser like the one we develop here.

The influential Cilk algorithm [13] schedules strict computations with dependencies between tasks. This paper discusses the advantages of work-stealing over work-sharing algorithms, and gives bounds on the efficiency of work-stealing. Chase & Lev describe an algorithm [14] for a lock-free work-stealing queue, which we base ourselves on for our own implementation.

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 reference-counting smart pointers rather than the manual incrementing and decrementing initially present. This helped fix a number of memory leaks. It was also modified 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 explanation 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 [15], 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.

The executable compiled for evaluation runs takes as parameters the domain and problem files to be processed. In addition, command-line arguments allow specifying the number of explanations to allocate in each task, and the maximum number of worker threads to spawn. Except in Method 1, it will never spawn more worker threads than there are hardware threads available.

The code for this project was developed using Boost version 1.46.1 on Ubuntu Linux 12.04 64-bit, and also tested and run using Boost version 1.41 on Scientific Linux 6.2. The machines used for evaluation are Sherlock, a 4-core Intel i5 desktop PC with 8GB of memory running the Ubuntu configuration, and Catzilla, a 16-core AMD Opteron multi-user server with 64GB of memory running the Scientific Linux configuration. The code was built using GCC under the -O3 optimisation level for evaluation.

Boost 1.46.1 is the currently packaged version of Boost on Ubuntu 12.04. Unfortunately it suffers from an occasional deadlock issue due to locking order when a sleeping thread is interrupted by calling `boost::thread::interrupt()`. We encountered this issue during development which for certain build configurations and datasets would very reliably cause deadlocks. Interruption was being used to exit threads when processing was complete. Interruption causes an exception to be thrown in the interrupted thread; we replaced calls to `thread::interrupt()` with a task that throws this exception when invoked. The entry function of the worker thread then catches this exception and exits the thread.

Each algorithm is implemented with a similar structure: a class encapsulates the algorithm, and is instantiated based on the selected algorithm for each run. This class has a method `explainObservations` which is called on the main thread, and creates the worker threads. Each of the threads runs a static method `thread_fn` on the class, which interacts with a `WorkerData` structure containing the queues or other relevant information. The worker thread method is mostly independent of the nature of the work being done. Finally, tasks are represented by a `TaskData` structure which contains the expressions for the task, and any other relevant data. This structure has a method `execute` which is passed a pointer to the `WorkerData`, processes the expressions, and passes the results back into new tasks or to the main thread.

The two problem domains used for evaluation are the XPERIENCE domain, which involves hands picking up and moving balls around and into cups, and is based on the XPERIENCE robotics project [16]; and the Logistics domain, which involves trucks and airplanes transporting packages between cities, and is based on the Logistics domain in the First International Planning Competition [17].

During evaluation, each configuration was run 5 times, on both domains, on both machines, varying either the batch size or the maximum number of worker threads. We have focused on measuring the runtime for the core of the algorithm, ignoring in our timings the parsing of the input files, the probability computations, and output of the results.

Code listings will be provided to highlight certain details of implementation. In these listings, details such as i/o and runtime statistics gathering have been omitted for clarity.

Chapter 5

Method 0 - Original Single-Threaded ELEXIR

The first method we include for comparison is a mostly unmodified single-threaded implementation, after the code had been made thread-safe and memory leaks addressed.

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. The batch size and thread count parameters are ignored.

```
void explainObservations(Problem const* problem, list<Explanation*>* results)
{
  // Initialise the set of explanations with a single empty explanation
  curExps.push_back( new Explanation() );
  // Loop over the observations
  for ( size_t i = 0; i < problem->obs.size(); ++ i ) {
    if ( curExps.empty() ) { return; } // Found no consistent explanations
    // Process all explanations on the main thread
    while ( !curExps.empty() ) {
      Explanation* lep = curExps.front();
      lep->extendExplanation( &problem->lexicon, &problem->obs[ lep->obsIndex ], &newExps );
      delete lep;
      curExps.pop_front();
    }
    curExps.splice(curExps.end(), newExps);
  }
  results->splice(results->end(), curExps);
}
```

5.2 Evaluation

The algorithm was evaluated on the XPERIENCE and Logistics domains, on the Sherlock and Catzilla machines. Since this method does not scale with the number of threads, we will report baseline performance numbers, shown in Table 5.1.

Domain	Sherlock Runtime (seconds)	Catzilla Runtime (seconds)
XPERIENCE Domain	5.63	14.67
Logistics Domain	1.36	3.59

Table 5.1: Average Runtimes for Method 0

Chapter 6

Method 1 - One Thread Per Task

Before implementing the methods discussed in Section 2.6, as a proof of concept for multithreading the algorithm, we implemented a scheduling algorithm which spawns a new thread for each task, one task per hardware thread up to the thread limit.

6.1 Implementation

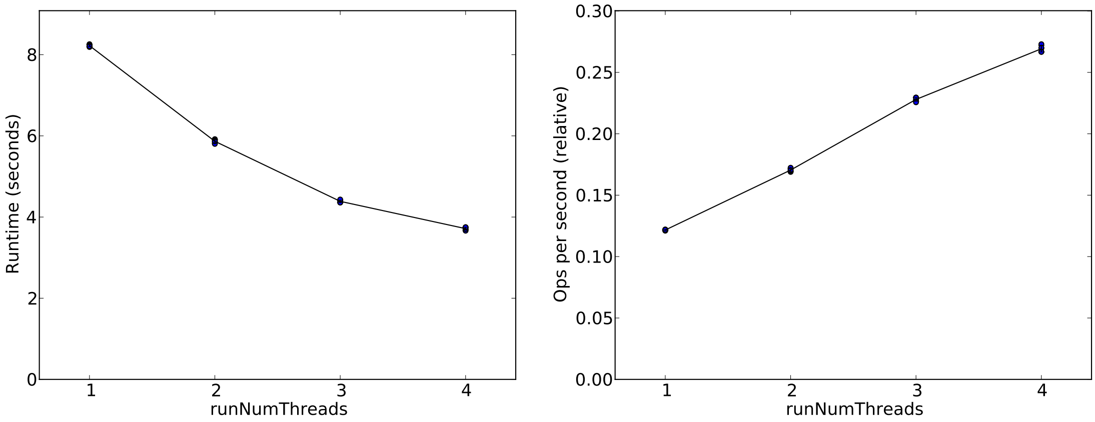
On each new observation, the main thread spawns a new thread for each task, equal to the number of hardware threads available or the thread limit. Each task processes its explanations and produces new resulting explanations. The main thread waits for all the spawned threads to complete, and collects their results, then spawns a new set of threads on the next observation. This implementation ignores the batch size parameter and partitions the explanations evenly between tasks.

```
void explainObservations(Problem const* problem, list<Explanation*>* results)
{
  // Initialise the set of explanations with a single empty explanation.
  curExps.push_back(new Explanation());
  // Loop over the observations
  for ( unsigned i = 0; i < problem->obs.size(); ++ i ) {
    if ( curExps.empty() ) { return; } // Found no consistent explanations
    // Distribute the explanations into one task per thread
    int i = 0;
    while ( !curExps.empty() ) {
      tasks[i].push_back(curExps.front());
      curExps.pop_front();
      i = (i + 1) % numThreads_;
    }
    // Launch the threads and wait for them to complete
    for (int i = 0; i < numThreads_; ++i) {
      threads.add_thread(new boost::thread(&thread_fn, problem, &tasks[i], &taskResults[i]));
    }
    threads.join_all();
    for (int i = 0; i < numThreads_; ++i) {
      curExps.splice(curExps.end(), taskResults[i]);
    }
  }
  results->splice(results->end(), curExps);
}

static void thread_fn(Problem const* problem, list<Explanation*>* exps, list<Explanation*>*
    out)
{
  while ( !exps->empty() ) {
    Explanation* lep = exps->front();
    lep->extendExplanation(&problem->lexicon, &problem->obs[lep->obsIndex], out);
    delete lep;
    exps->pop_front();
  }
}
```

6.2 Evaluation

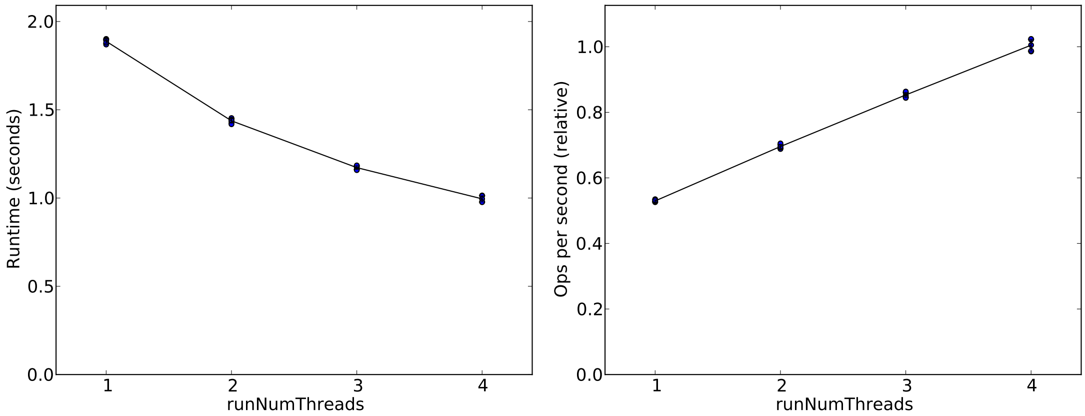
The algorithm was evaluated on the XPERIENCE and Logistics domains, on the Sherlock and Catzilla machines, with different thread count limits. For each set of runs, we plot the runtime in seconds and the relative throughput. If the algorithm scales perfectly, the throughput graph should be linear.



(a) Runtime

(b) Throughput

Figure 6.1: Method 1 runtime with 1 to 4 threads on Sherlock, XPERIENCE

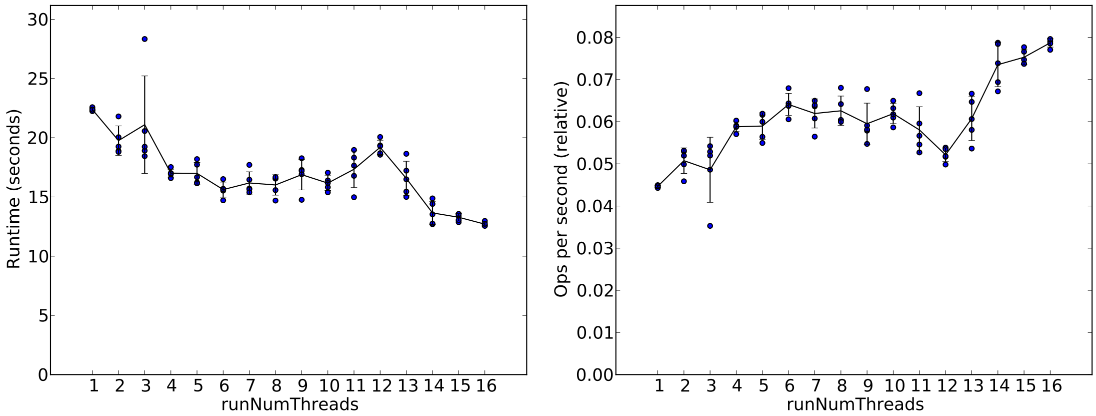


(a) Runtime

(b) Throughput

Figure 6.2: Method 1 runtime with 1 to 4 threads on Sherlock, Logistics

Despite the very simplistic approach to task scheduling taken in this approach, we achieve a good almost-linear speedup on Sherlock with this method, although the speedup visibly tails off at the 4th thread.

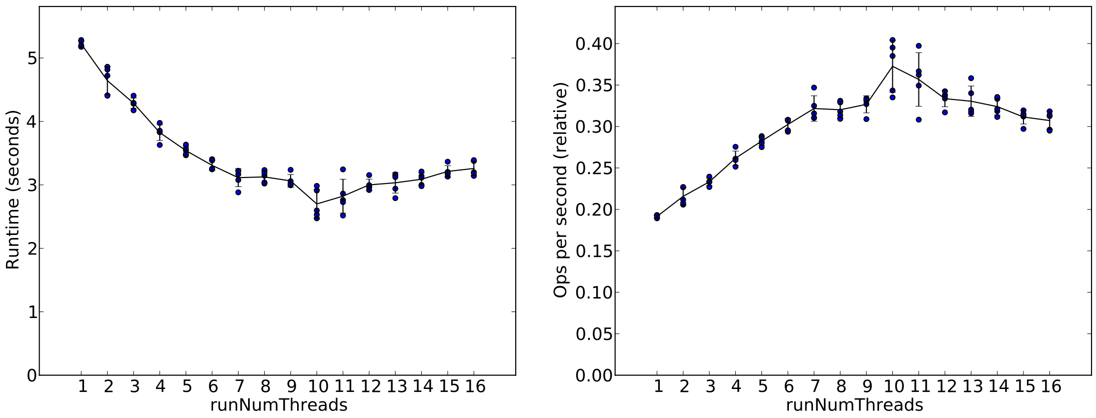


(a) Runtime

(b) Throughput

Figure 6.3: Method 1 runtime with 1 to 16 threads on Catzilla, XPERIENCE

Catzilla provides a more challenging environment for evaluation. Since this is a multi-user system, the runtimes vary much more, but we can see in the XPERIENCE domain that the performance increases near-linearly up to 4 threads, then hardly increases after this. In the Logistics domain, the performance increases near-linearly up to 6 threads before it begins to level off.



(a) Runtime (b) Throughput

Figure 6.4: Method 1 runtime with 1 to 16 threads on Catzilla, Logistics

Chapter 7

Method 2 - One Queue Per Thread

In this section we discuss the implementation of the blocking queue method. This method uses one blocking 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.

The implementation of the queue is based on boost::bounded_buffer from the documentation of the Boost Circular Buffer library. Internally, it uses a bounded circular buffer guarded by a mutex. Worker threads trying to fetch work wait on a condition variable which is signalled when work is added to the buffer. The main difference between this method and the previous one is that the threads sleep between observations rather than exiting and being recreated.

Completion is detected using a completion barrier as described in The Art of Multiprocessor Programming [7]. A count of active threads is maintained and atomically incremented and decremented by worker threads as they become busy or idle.

This implementation does not take into account the batch size parameter, since the main thread is responsible for redistributing the work. The main thread creates as task for each worker thread and splits the explanation to be processed evenly between them.

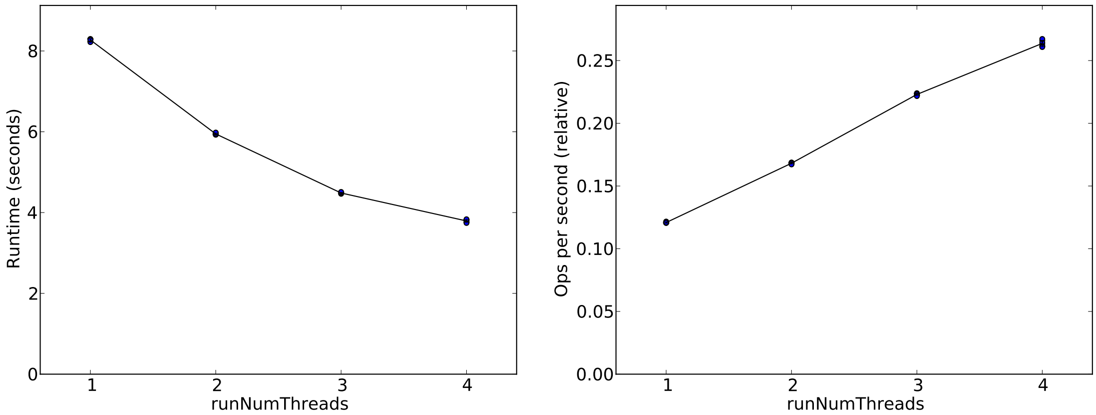
```
void explainObservations(Problem const* problem, list<Explanation*>* results)
{
  for (int i = 0; i < numThreads_; ++i) {
    workerData[i].problem = problem;
  }
  // Initialise the set of explanations with a single empty explanation.
  curExps.push_back( new Explanation() );
  for (int i = 0; i < numThreads_; ++i) {
    threads.add_thread(new boost::thread(&thread_fn, &workerData[i]));
  }
  // Loop over the observations
  for ( unsigned i = 0; i < problem->obs.size(); ++ i ) {
    if ( curExps.empty() ) { return; } // Found no consistent explanations for the
        observations
    // Distribute the explanations into one task per thread
    int i = 0;
    while ( !curExps.empty() ) {
      taskData[i].exps.push_back(curExps.front());
      curExps.pop_front();
      i = (i + 1) % numThreads_;
    }
    for (int i = 0; i < numThreads_; ++i) {
      barrier.incActive();
      workerData[i].queue->push_front(&taskData[i]);
    }
    // Wait for them to complete
    while (!barrier.allInactive()) {
      boost::this_thread::sleep(boost::posix_time::millisec(1));
    }
    for (int i = 0; i < numThreads_; ++i) {
      curExps.splice(curExps.end(), taskData[i].results);
    }
  }
  results->splice(results->end(), curExps);
}

static void thread_fn(WorkerData* workerData)
{
  Queue* queue = workerData->queue;
  TerminationBarrier* barrier = workerData->barrier;
  try {
    while (true) {
      TaskData* task;
      queue->pop_back(&task);
      task->execute(workerData);
      barrier->decActive();
    }
  } catch (boost::thread_interrupted& e) {
    // exit thread
  }
}
```

```
void Task::execute(WorkerData* workerData) {
  if (exit) { throw boost::thread_interrupted(); }
  Problem const* problem = workerData->problem;
  while ( !exps.empty() ) {
    Explanation* lep = exps.front();
    lep->extendExplanation(&problem->lexicon, &problem->obs[lep->obsIndex], &results);
    delete lep;
    exps.pop_front();
  }
}
```

7.2 Evaluation

The algorithm was again evaluated on the XPERIENCE and Logistics domains, on the Sherlock and Catzilla machines, with different thread count limits. For each set of runs, we plot the runtime in seconds and the relative throughput.



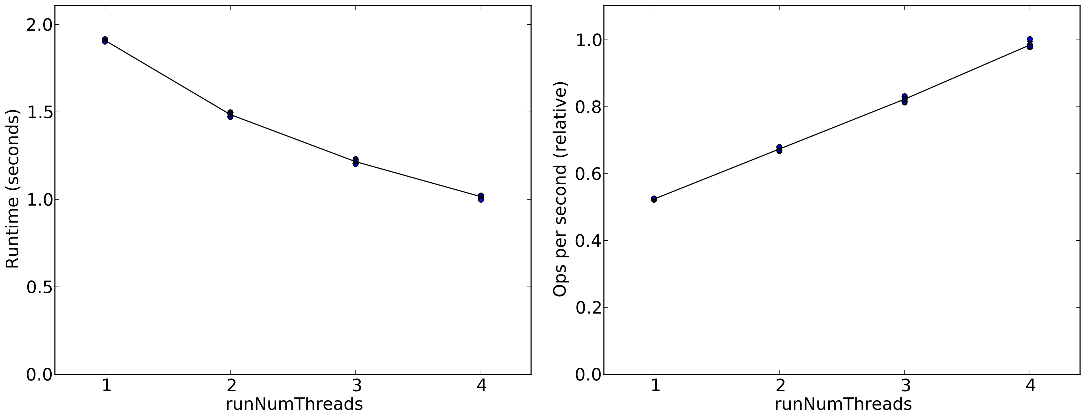
(a) Runtime

(b) Throughput

Figure 7.1: Method 2 runtime, 1-4 threads on Sherlock, XPERIENCE

Both domains on Sherlock show very similar characteristics to the previous method, with almost linear speedup.

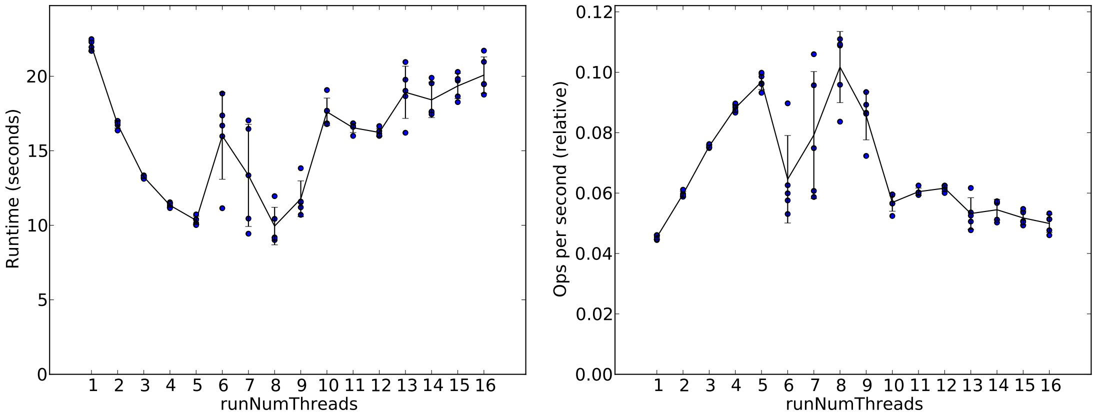
The runs on Catzilla are harder to analyse, but we can see that 5 threads seems the best number for the XPERIENCE domain, after which we have some erratic data points with the running time eventually rising again as the number of threads is increased. For the Logistics domain, performance seems to scale near-linearly up to 8 threads, after which it remains more or less constant.



(a) Runtime

(b) Throughput

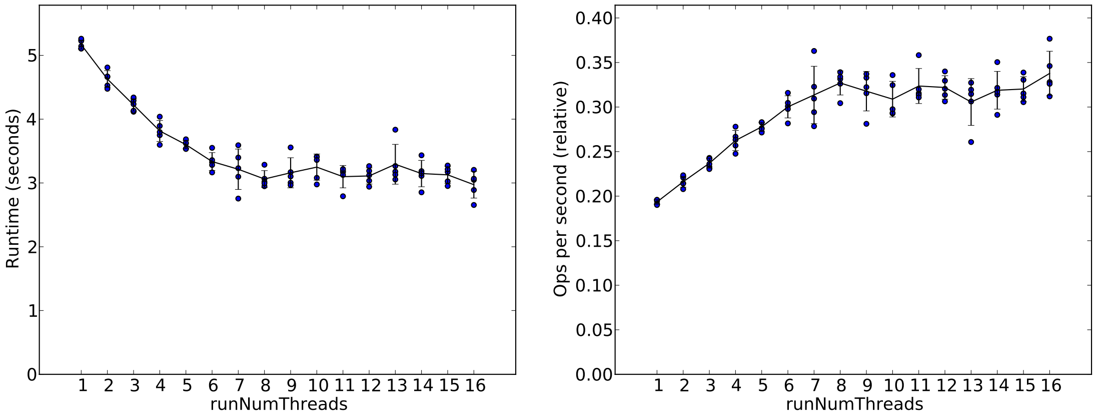
Figure 7.2: Method 2 runtime, 1-4 threads on Sherlock, Logistics



(a) Runtime

(b) Throughput

Figure 7.3: Method 2 runtime, 1-16 threads on Catzilla, XPERIENCE



(a) Runtime

(b) Throughput

Figure 7.4: Method 2 runtime, 1-16 threads on Catzilla, Logistics

Chapter 8

Method 3 - Lock-Free Work-Stealing

In this section we discuss the implementation of the work-stealing queue method. 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

The implementation of this scheduler is based on [14], also described in The Art of Multiprocessor Programming [14], which describes a lock-free work-stealing queue where stealing threads steal a single task from the tail of the victim thread's queue, and threads otherwise insert and retrieve work from the head of their own queue. The queue can be resized safely if there is not enough space to insert work at the head of the queue. The original implementation described is appropriate for the language Java where memory is managed by garbage collection, and reads and writes to volatile variables are not reordered. In order to implement this in C++, the replacement of the queue's buffer when it is reallocated had to be guarded using *hazard pointers* [18] and the ordering of reads and writes to variables accessed locklessly enforced using compiler-specific memory fence directives.

In terms of performance, we expect this implementation to have low contention due to the lack of locking, and good load-balancing since idle threads will steal work from other threads. During initial testing, we found that performance would drop as the number of threads increased from 2 to 4 threads. Using cachegrind to produce a differential profile between a run with 1 thread and a run with 4 threads, we discovered that idle threads were spending a lot of time in random number generation, which is used when selecting a victim thread to steal from. The insertion of a short sleep after failed stealing attempts resolved this issue.

```
void explainObservations(Problem const* problem, list<Explanation*>* results)
{
  // Initialise the set of explanations with a single empty explanation
  TaskData initTask;
  initTask.exps.push_back(new Explanation());
  queues[0]->pushBottom( initTask );
  for (int i = 0; i < numThreads_; ++i) {
    workerData[i].problem = problem;
  }
  // Launch threads
  for (int i = 0; i < numThreads_; ++i) {
    threads.add_thread( new boost::thread(&thread_fn, &workerData[i]) );
  }
  // Wait for them to complete
  while (!barrier.allInactive()) {
    boost::this_thread::sleep(boost::posix_time::millisec(10));
  }
  // Force them to exit
  for (int i = 0; i < numThreads_; ++i) {
    TaskData exitTask;
    exitTask.exit = true;
    lexCore::writeBarrier();
    queues[i]->pushBottom(exitTask);
  }
  threads.join_all();
  for (int i = 0; i < numThreads_; ++i) {
    curExps.splice(curExps.end(), taskResults[i]);
  }
  results->splice(results->end(), curExps);
}

static void thread_fn(WorkerData* workerData)
{
  VictimPicker* victimPicker = workerData->victimPicker;

  std::vector< WorkStealingQueue<TaskData>* >& queues = *workerData->queues;
  TerminationBarrier* barrier = workerData->barrier;
  boost::posix_time::millisec const sleepTime = boost::posix_time::millisec(1);

  TaskData task;
  try {
    while (true) {
      // Take work from own queue
      while (queues[workerData->workerID]->popBottom(&task)) {
        task.execute(workerData);
      }
      barrier->decActive();
      // Try stealing work
      while(true) {
        int victim = (*victimPicker)();
        if (!queues[victim]->isEmpty()) {
          barrier->incActive();
          if (queues[victim]->popTop(&task)) {
            task.execute(workerData);
            break;
          }
          barrier->decActive();
        }
        if (barrier->allInactive()) { return; }
        boost::this_thread::sleep(sleepTime);
      }
    }
  } catch (boost::thread_interrupted const& e) {
      // Exit thread
  }
}

void Task::execute(WorkerData* workerData)
{
    if (exit) { throw boost::thread_interrupted(); }
    Problem const* problem = workerData->problem;
    while ( !exps.empty() ) {
      Explanation* lep = exps.front();
      if ((size_t)lep->obsIndex < problem->obs.size()) {
        list< Explanation* > results;
        lep->extendExplanation(&problem->lexicon, &problem->obs[lep->obsIndex], &results);
        delete lep;

        // Push results as new tasks
        while (!results.empty()) {
          for (int i = 0; i < workerData->maxBatch && !results.empty(); ++i) {
            TaskData newTask;
            newTask.exps.push_back(results.front());
            (*workerData->queues)[workerData->workerID]->pushBottom(newTask);
            results.pop_front();
          }
        }
      } else { // No more explanations, add this to the list of completed explanations
        workerData->results->push_back(lep);
      }

      exps.pop_front();
    }
  }
```

8.2 Evaluation

8.2.1 Batch Size Parameter

In order to estimate the best value for the batch size parameter, we ran the implementation with 4 threads on the XPERIENCE domain, varying the batch size.

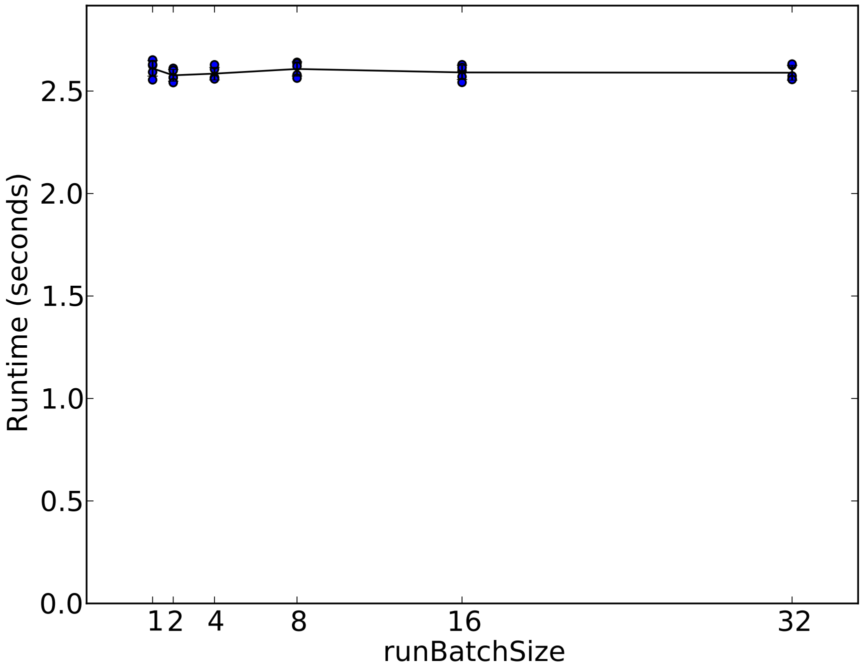


Figure 8.1: Method 3 runtime, varying batch size on Sherlock, XPERIENCE

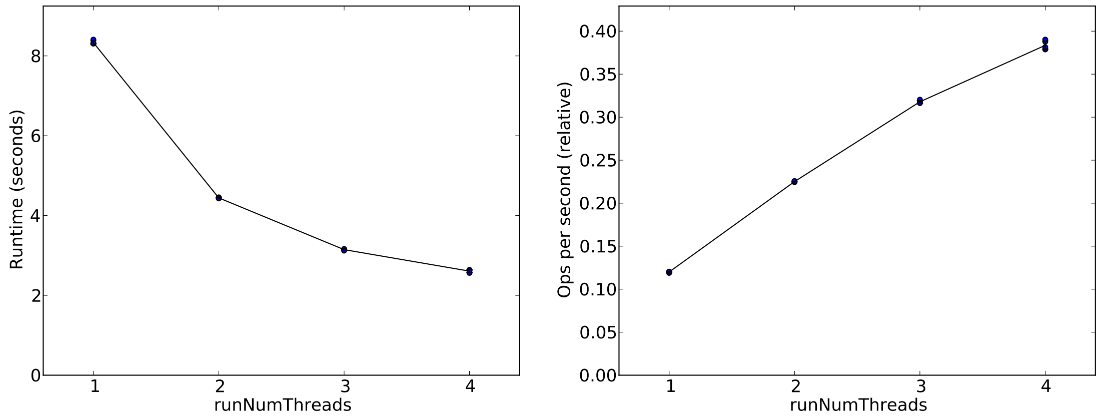
We can see in Figure 8.1 that the batch size does not significantly affect the runtime for this algorithm. All further tests were performed with a batch size of 8.

8.2.2 Thread Count

The algorithm was again evaluated on the XPERIENCE and Logistics domains, on the Sherlock and Catzilla machines, with different thread count limits. For each set of runs, we plot the runtime in seconds and the relative throughput.

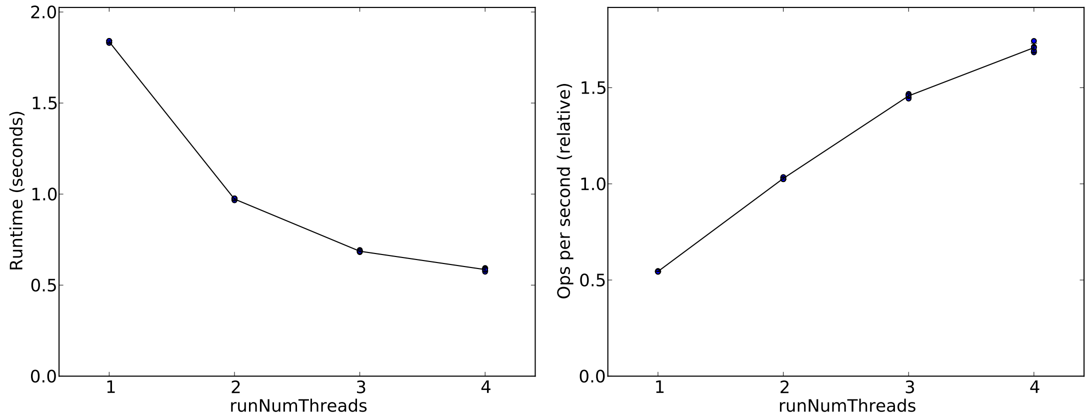
Runs on Sherlock again show a near-linear speedup with slight tailing off when we reach 4 threads.

The runs on Catzilla show a sharp speedup at first, followed by a slow decline. The XPERIENCE domain peaks at around 5 threads, and the Logistics domain around 7 threads.



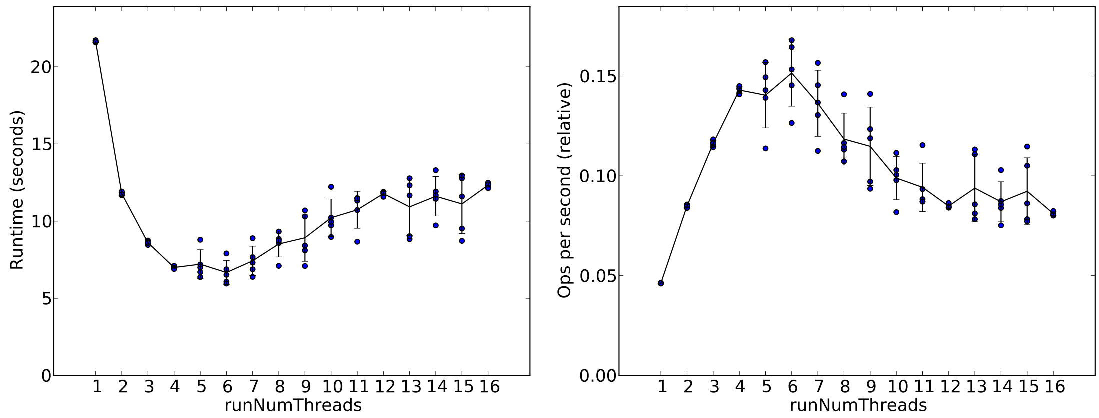
(a) Runtime (b) Throughput

Figure 8.2: Method 3 runtime, 1-4 threads on Sherlock, XPERIENCE



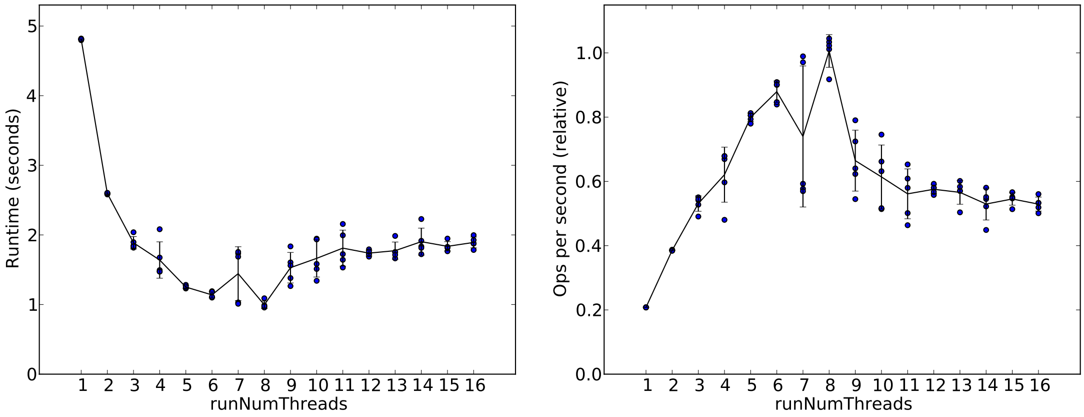
(a) Runtime (b) Throughput

Figure 8.3: Method 3 runtime, 1-4 threads on Sherlock, Logistics



(a) Runtime (b) Throughput

Figure 8.4: Method 3 runtime, 1-16 threads on Catzilla, XPERIENCE



(a) Runtime

(b) Throughput

Figure 8.5: Method 3 runtime, 1-16 threads on Catzilla, Method 3, Logistics

Chapter 9

Method 4 - Single Global Queue

In this section we discuss the global queue method. 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.

```
void explainObservations(Problem const* problem, list<Explanation*>* results)
{
  for (int i = 0; i < numThreads_; ++i) {
    workerData[i].problem = problem;
  }

  // Initialise the set of explanations with a single empty explanation
  {
    TaskData* newTask = new TaskData();
    newTask->exps.push_back(new Explanation());
    boost::mutex::scoped_lock lock(mutex);
    queue.push_back(newTask);
  }

  // Create worker threads and wait for them to complete
  for (int i = 0; i < numThreads_; ++i) {
    threads.add_thread( new boost::thread(&thread_fn, &workerData[i]) );
  }
  while (!barrier.allInactive()) {
    boost::this_thread::sleep(boost::posix_time::millisec(10));
  }
  threads.interrupt_all();
  threads.join_all();

  for (int i = 0; i < numThreads_; ++i) {
    curExps.splice(curExps.end(), taskResults[i]);
  }
  if ( curExps.empty() ) { return; } // Found no consistent explanations for the observations
  results->splice(results->end(), curExps);
}

static void thread_fn(WorkerData* workerData) {
  TerminationBarrier* barrier = workerData->barrier;
  boost::mutex* mutex = workerData->mutex;
  Queue* queue = workerData->queue;

  boost::posix_time::millisec const sleepTime = boost::posix_time::millisec(10);

  try {
    while (true) {
      TaskData* task = NULL;
      {
        boost::mutex::scoped_lock lock(*mutex);
        if (!queue->empty()) {
          task = queue->front();
          queue->pop_front();
        }
      }

      // If succeeded, execute, otherwise sleep
      if (task) {
        task->execute(workerData);
        delete task;
      } else {
        barrier->decActive();
        if (barrier->allInactive()) { return; }
        boost::this_thread::sleep(sleepTime);
        barrier->incActive();
      }
    }
  } catch (boost::thread_interrupted const& e) {
    // exit thread
  }
}

void execute(WorkerData* workerData)
{
  if (exit) { throw boost::thread_interrupted(); }
  Problem const* problem = workerData->problem;
  std::list<Explanation*> results;
  while ( !exps.empty() ) {
      Explanation* lep = exps.front();
      exps.pop_front();
      if ((size_t)lep->obsIndex < problem->obs.size()) {
        lep->extendExplanation(&problem->lexicon, &problem->obs[lep->obsIndex], &results);
        delete lep;
      } else {
        // No more explanations, add this to the list of completed explanations
        workerData->results->push_back(lep);
      }
    }

    // Push new expression back onto queue as tasks
    boost::mutex::scoped_lock lock(*workerData->mutex);
    while (!results.empty()) {
      TaskData* newTask = new TaskData;
      for (int i = 0; i < workerData->maxBatch && !results.empty(); ++i) {
        newTask->exps.push_back(results.front());
        results.pop_front();
      }
      workerData->queue->push_back(newTask);
    }
  }
```

9.2 Evaluation

9.2.1 Batch Size Parameter

In order to estimate the best value for the batch size parameter, we again ran the implementation with 4 threads on the XPERIENCE domain, varying the batch size.

We can see in Figure 9.1 that the batch size parameter does have an effect on the runtime of this algorithm, suggesting that contention for the lock is worse with small tasks leading for more frequent insertions and removals on the global queue. In this test the runtime is similar from 8 explanations per batch upward, so all further tests were performed with a batch size of 8.

9.2.2 Thread Count

The algorithm was again evaluated on the XPERIENCE and Logistics domains, on the Sherlock and Catzilla machines, with different thread count limits. For each set of runs, we plot the runtime in seconds and the relative throughput.

The runs on Sherlock again show a near-linear speedup.

The runs on Catzilla show a linear speedup at first, followed by a peak and tailing off. Both domains peak at around 8 threads.

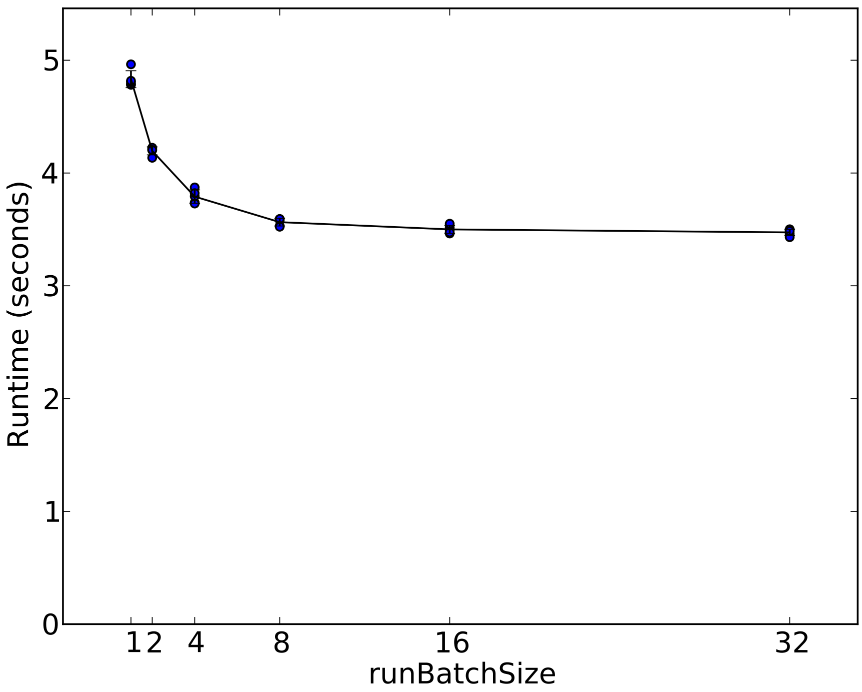
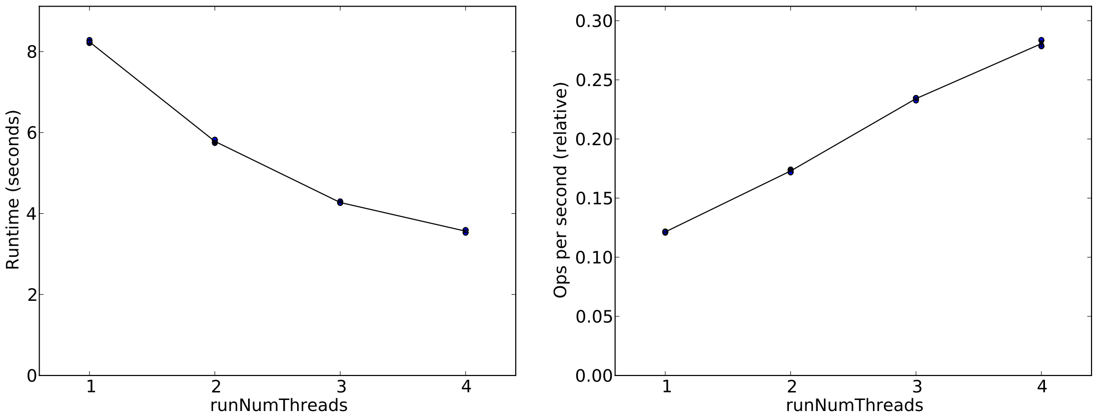


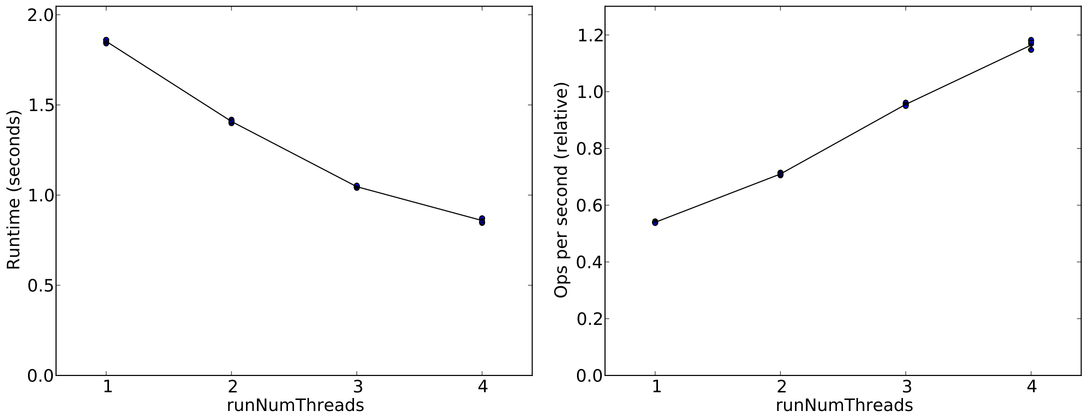
Figure 9.1: Runtime, varying batch size on Sherlock, Method 3, XPERIENCE



(a) Runtime

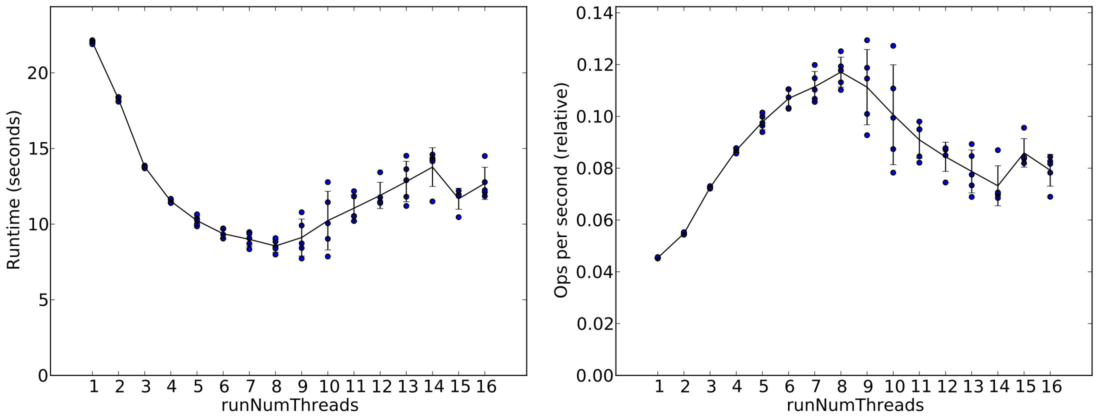
(b) Throughput

Figure 9.2: Method 4 runtime, 1-4 threads on Sherlock, XPERIENCE



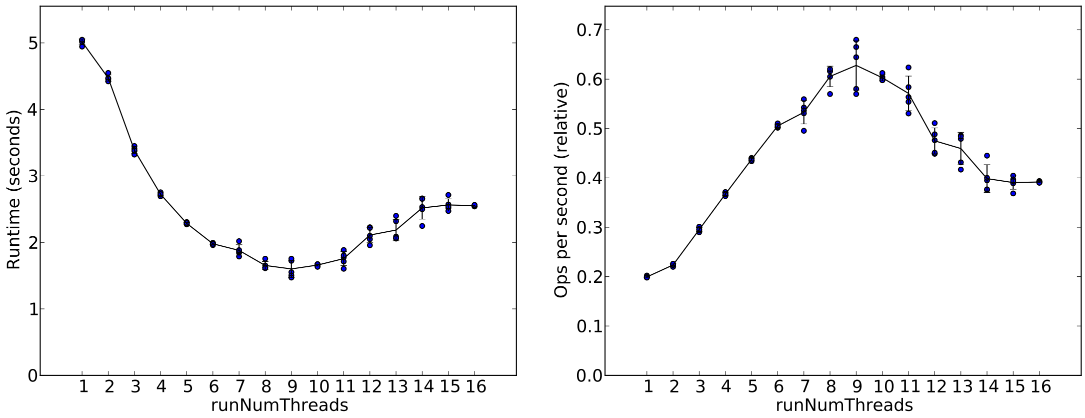
(a) Runtime (b) Throughput

Figure 9.3: Method 4 runtime, 1-4 threads on Sherlock, Logistics



(a) Runtime (b) Throughput

Figure 9.4: Method 4 runtime, 1-16 threads on Catzilla, XPERIENCE



(a) Runtime (b) Throughput

Figure 9.5: Method 4 runtime, 1-16 threads on Catzilla, Logistics

Chapter 10

Evaluation

If we overlay the results from the previous sections, we can compare the performance of the different algorithms presented thus far.

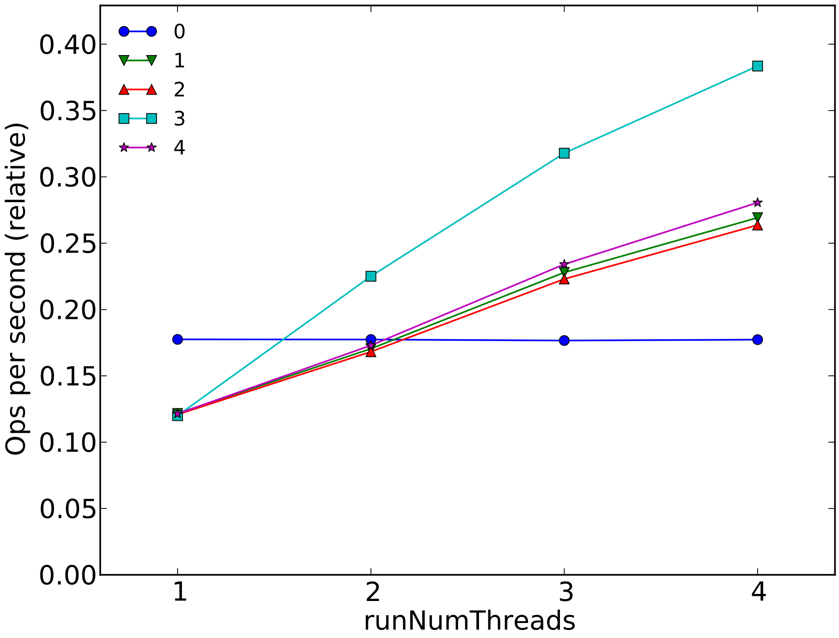


Figure 10.1: All methods throughput, 1-4 threads on Sherlock, XPERIENCE

Looking at throughput on Sherlock using the XPERIENCE domain in Figure 10.1, we can see that all the algorithms have a certain cost above the baseline single-threaded method. Method 3, the work-stealing queue, becomes better than baseline starting from 2 threads, and remains better than the other algorithms by a comfortable margin. The other algorithms still manage to scale linearly, becoming better than the baseline by 3 threads, and are very close to each other in terms of performance.

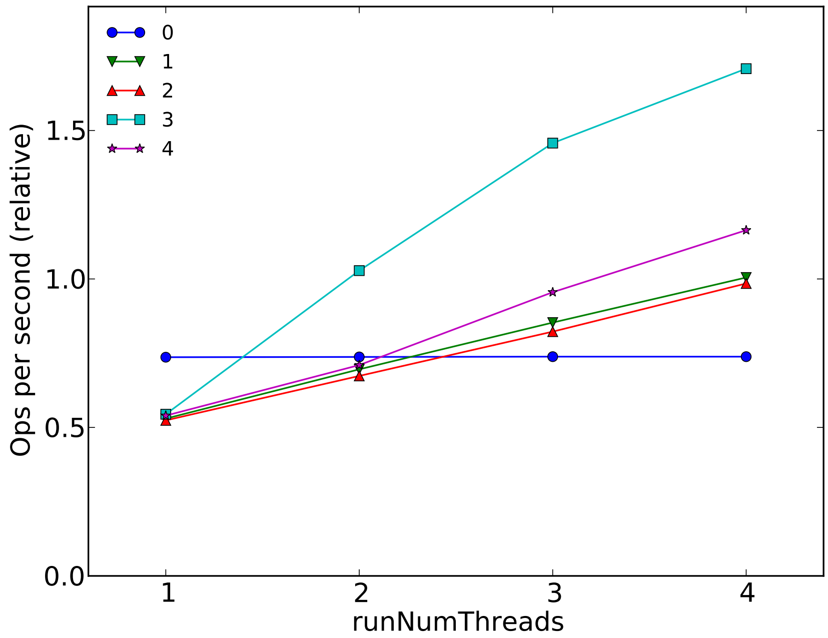


Figure 10.2: All methods throughput, 1-4 threads on Sherlock, Logistics

The results on the Logistics domain in Figure 10.2 are very similar, although on this domain Method 4, the global queue, manages to do better than Methods 1 & 2. The work-stealing queue remains comfortably faster than all the others.

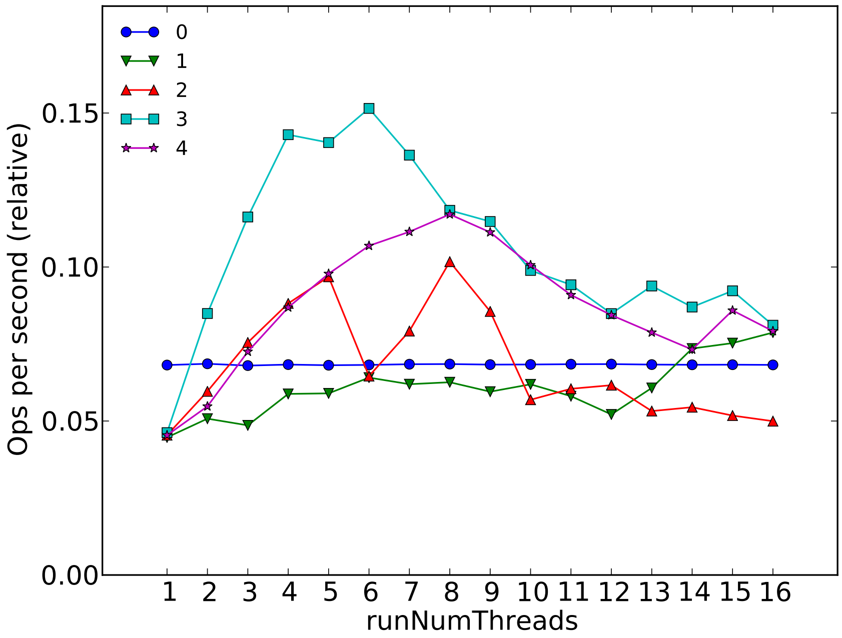


Figure 10.3: All method throughput, 1-4 threads on Catzilla, XPERIENCE

On Catzilla, the different algorithms differentiate themselves a little further.

In Figure 10.3, we can see Method 3, the work-stealing queue remains much faster. Here, Method 1 struggles to ever climb past the baseline single-threaded performance, suggesting that thread creation overhead may be much higher on this machine. Method 2, the blocking queues, and Method 4, the global queue, start very close but the global queue eventually overtakes the blocking queue in terms of performance.

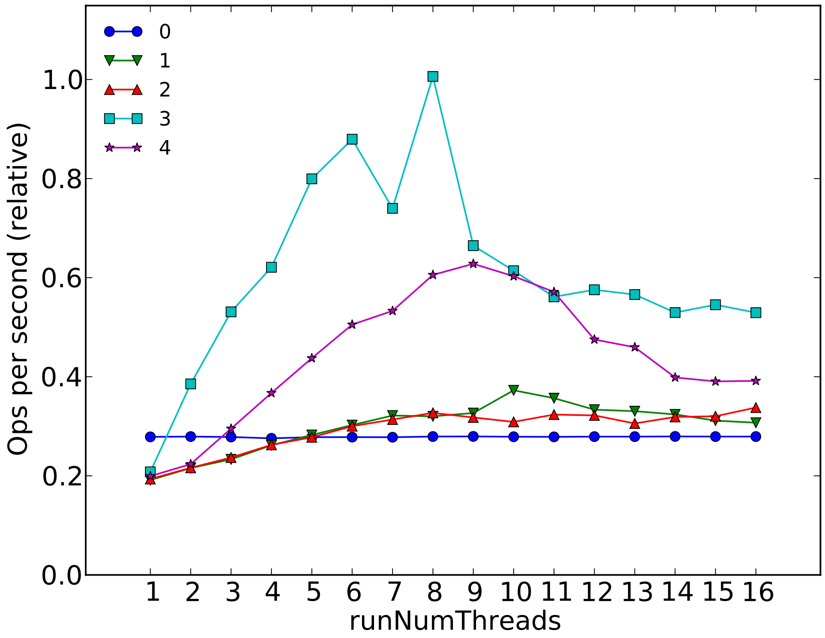


Figure 10.4: All methods throughput, 1-4 threads on Catzilla, Logistics

In Figure 10.4, we see the same results for the Logistics domain. Here Method 2, the blocking queue, performs even worse relative to Method 4, the global queue. The global queue scales well, but the work-stealing queue remains by far the fastest.

By studying the cachegrind annotated emulation results in runs using Method 3 with 1 to 4 threads on Sherlock using the XPERIENCE domain, we were able to observe an increase in the cost of memory allocation, specifically in `malloc` and `free` and child functions in the language runtime. These increases were attributed to increased costs in memory access and increased cache misses. Since these results come from an annotated translation of the machine code and a simulation of cache costs, these results are suggestive but not conclusive.

In general, we have seen that the Logistics domain seems to scale up to a larger number of threads than the XPERIENCE domain, and we have seen that a possible cause of scaling difficulties comes from cache contention during memory allocation. Another compelling piece of evidence comes from graphing the number of new explanation generated by each existing explanation, when a new observation is processed.

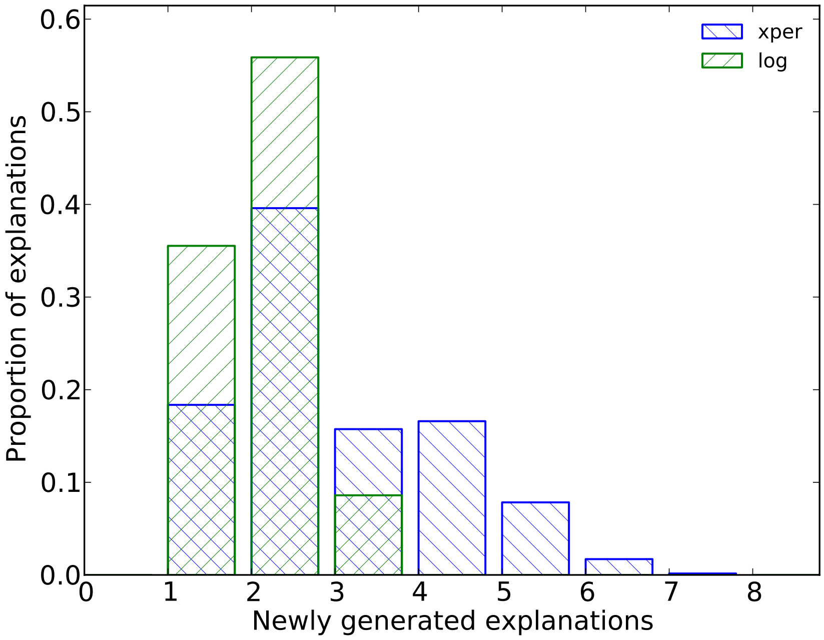


Figure 10.5: Histogram of generated expression counts

In Figure 10.5, we see that each new observation in the Logistics domain typically produces only one or two explanations to replace it. However in the XPERIENCE domain, a significant proportion of explanations will produce 3 to 5 new ones to replace it. This would imply an increased rate of memory allocation and more cache contention.

Chapter 11

Conclusion

In all our tests, the lock-free work-stealing queue performs far better than any of the other algorithms studied, a performance that seems to justify its higher complexity of implementation. This algorithm avoids locking, leading to low contention between threads, and automatically load-balances, leading to better utilisation of worker threads.

We have seen evidence suggesting that the problem domain has an effect on the scalability of plan recognition, and that this may be related to the memory allocation behaviour during execution. Future work could explore how the structure of the domain leads to fewer or more explanations being generated at each stage, and perhaps how to optimise a domain for scalability. The use of specialised memory allocators, and modifications to the algorithm to reduce memory allocations, could also be explored.

We have investigated and successfully parallelised the main part of the ELEXIR algorithm. The largest part of the remaining runtime comes from the probability computation post-process, which seems especially expensive now that we have sped up the main part. Further work could explore parallelising this section of the algorithm to further speed up the computation.

The ELEXIR algorithm produces a complete list of possible consistent explanations and their probabilities. Client code may not need such a full exploration of the possibility space, and instead need only a list of the most probable explanations. The ELEXIR algorithm could be adapted to expand only the most probable current explanations. As new observations arrive, older unexplored possibilities may become more likely, and be explored further.

In order to apply the ELEXIR algorithm to real time scenarios, an incremental version would be desirable, maintaining a list of explanations and updating it with new observations rather than updating it as a batch. This would mean incrementally computing the probabilities, which would naturally mean the parallelisation of the probability computation as suggested above.

Bibliography

[1] C. W. Geib and R. P. Goldman, "Plan Recognition in Intrusion Detection Systems", in *DARPA Information Survivability Conference and Exposition (DISCEX)*, 2001.

[2] M. E. Foster, T. By, M. Rickert, and A. Knoll, "Human-Robot Dialogue for Joint Construction Tasks", *ICMI*, 2006.

[3] M. E. Pollack, "Intelligent technology for an aging population: The use of AI to assist elders with cognitive impairment", *AI Magazine*, vol. 26, no. 2, pp. 9-24, 2005.

[4] M. Steedman, "The Syntactic Process", MIT Press, 2000.

[5] B. Hoffman "Integrating Free Word Order Syntax and Information Structure", in *Proceedings of the seventh conference on European chapter of the Association for Computational Linguistics (EACL'95)*, pp. 245-252, 1995.

[6] C. W. Geib, "Delaying Commitment in Plan Recognition Using Combinatory Categorial Grammars", in *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 1702-1707, 2009.

[7] M. Herlihy and N. Shavit, "The Art of Multiprocessor Programming, Revised First Edition", Morgan Kaufmann, 2012.

[8] H. H. Bui, S. Venkatesh, and G. West, "Policy recognition in the Abstract Hidden Markov Model", *Journal of Artificial Intelligence Research*, vol. 17, pp. 451-499.

[9] D. Avrahami-Zilberbrand and G. A. Kaminka, "Fast and Complete Symbolic Plan Recognition", in *Proceedings IJ-CAI*, 2005.

[10] G. R. Andrews, “Foundations of Multithreaded, Parallel, and Distributed Programming”, Addison-Wesley, 2000.

[11] D. Grune, “Parsing Techniques: A Practical Guide”, Springer, 2008.

[12] S. Clark and J. R. Curran, “Log-Linear Models for Wide-Coverage CCG Parsing”, in *Proceedings of the 2003 Conference on Empirical Methods in Natural Language Processing*, pp. 97-104, 2003.

[13] R. D. Blumofe and C.E. Leiserson, “Scheduling Multithreaded Computations by Work Stealing”, in *Journal of the ACM (JACM)*, vol. 46, issue 5 pp. 720-748, 1999.

[14] D. Chase and Y. Lev, “Dynamic Circular Work-Stealing Deque”, in *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA’05)*, pp. 21-28, 2005.

[15] A. Williams, “What’s New in Boost Threads?”, *Dr. Dobb’s Journal*, November 2008 issue.

[16] http://www.xperience.org/.

[17] D. Long and M. Fox, “The 3rd international Planning Competition: Results and Analysis”, *Journal of Artificial Intelligence Research*, vol. 20, pp. 1-59, 2003.

[18] M. Michael, “Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 15 issue 6, pp. 491-504, 2004.