Fine-grained parallelism in probabilistic parsing with Habanero Java∗

Matthew Francis-Landau∗, Bing Xue†, Jason Eisner† and Vivek Sarkar†
Johns Hopkins University Rice University
{mfl,jason}@cs.jhu.edu† {bx3,vsarkar}@rice.edu†

Abstract—Structured prediction algorithms—used when applying machine learning to tasks like natural language parsing and image understanding—present some opportunities for fine-grained parallelism, but also have problem-specific serial dependencies. Most implementations exploit only simple opportunities such as parallel BLAS, or embarrassing parallelism over input examples. In this work we explore an orthogonal direction: using the fact that these algorithms can be described as specialized forward-chaining theorem provers [1], [2], and implementing fine-grained parallelization of the forward-chaining mechanism. We study context-free parsing as a simple canonical example, but the approach is more general.

I. INTRODUCTION

Structured prediction algorithms in machine learning often exhibit tangled computation graphs, and tasks can be as fine-grained as filling in individual nodes of the graph. Rather than requiring the programmer to manually identify parallelizable procedures of individual algorithms, we explore a general-purpose strategy for filling in the computation graph. In this work, we consider parallelizing a canonical algorithm of this sort. By using fine-grained parallelism, we parallel-process one input example at a time and thus better utilize fast cache memory, as well as achieving better latency per example.

In Section II-C we describe Habanero-Java (HJLib), as a parallelization library which provides powerful primitives for managing complicated computation graphs. In Sections II-B and V we mention the Dyna language, as a framework that can specify the computation graphs for many abstract algorithms.

II. BACKGROUND

A. Probabilistic parsing

Probabilistic parsing is considered one of the core NLP tasks, and a probabilistic CKY parser exemplifies the sort of algorithm we are interested in. Given an input sentence, a probabilistic parser finds its most probable derivation tree, where a tree’s probability is the product of the probabilities of the grammar rules in that tree. We use $0 \leq i < j < k \leq n$ to represent positions in a length-$n$ sentence, and $x, y, z \in T$ to represent non-terminal symbols, so that most grammar rules have the form $x \rightarrow y z$. The basic serial algorithm is an $O(n^3)$ dynamic programming algorithm that finds intermediate quantities $A[x, i, k]$ that are associated with derivations of substrings. After the entries $A[x, k-1, k]$ are initialized based on the words of the sentence, Algorithm 1 fills in $A[x, i, k]$ for increasing values of $w = k-i$, which corresponds to combining short phrases into longer phrases. $A[\text{Sentence}, 0, N']$ ends up holding the probability of the best derivation, whose tree can then be recovered with a little extra bookkeeping (not shown). Many serial speedups to this basic algorithm are used in practice [3], including heuristic methods to skip cells in $A$.

Algorithm 1 Typical CKY algorithm for probabilistic parsing

1: for $w \in [2, N']$ do ▷ width of constituent
2: for $i \in [0, N - w]$ do ▷ starting location
3: \hspace{1em} $k \leftarrow w + i$ ▷ ending location
4: for $x \in T$ do ▷ nonterminals at location
5: \hspace{2em} $A[x, i, k] = \max_{y, z \in T} A[y, i, j] \cdot A[z, j, k] \cdot P(x \rightarrow y z)$

B. Agenda parsing

In this paper we focus on an alternate work-list approach to parsing, which focuses on prioritization of constituents. Our main motivation for introducing this alternate technique is that it more directly maps onto general inference procedures.

Agenda parsing [5] schedules all its work through a priority queue (Algorithm 2), rather than visiting cells of $A$ in a fixed order (Algorithm 1). This approach is based on a view of our parser as a forward-chaining theorem prover [1] that is trying to prove $A[\text{Sentence}, 0, n] > 0$ (meaning that the sentence has a derivation). Discovering $A[x, i, k] > 0$ proves a lemma, and we try to derive new conclusions first from the highest-probability lemmas. It turns out that the actual value of $A[\text{Sentence}, 0, n]$ has converged when we first pop it from

∗An expanded version of this paper with additional discussion can be found at http://cs.jhu.edu/~mfl/papers/fl.xue.sarkar.eisner.2016.pdf.
Algorithm 2 Agenda Parsing Algorithm

1: while not isEmpty(agenda) do
2:   (x, i, k, score) ← pop(agenda) ↩ prioritized by score
3:   if score > A[x, i, k] then
4:     A[x, i, k] ← score
5:     expandFrontier(x, i, k, score)
6:   if (x, i, k) = (Sentence, 0, n) then ↩ built a complete parse
7:     return score ↩ return early
8: return 0 ↩ no positive-probability parse was found
9: function expandFrontier(y, i, j, childScore)
10:   Combine this popped constituent with previously popped constituents to left and right
11:   for all x, z ∈ T s.t. P(x → y z) > 0 do
12:     for all k ∈ {j, n} do
13:       if (siblingscore) A[z, j, k]) > 0 then
14:         parentScore ← childScore · siblingScore · P(x → y z)
15:         PUSH(agenda, (x, i, k, parentScore))
16:     for all x, z ∈ T s.t. P(x → y z) > 0 do
17:       for all h ∈ [0, i) do
18:         if (siblingscore) A[z, h, i]) > 0 then
19:           parentScore ← siblingScore · childScore · P(x → z y)
20:           PUSH(agenda, (x, h, j, parentScore))

Algorithm 3 CKY expressed in Dyna

a(x, i, K) max = word(W, i, K) * rule_prob(x, K).
a(x, i, K) max = a(y, i, J) * a(z, J, K) * rule_prob(x, y, z).
goal = a("Sentence", 0, n).

C. Habanero Java

Habanero-Java (HJLib), developed at Rice University, implements the Habanero execution model [10]. HJLib implements the Async-Finish model [11], in which async represents a general primitive for creating asynchronous computation and data transfer tasks. HJLib APIs include async, forasync, and finish as general primitives for creating and awaiting the completion of asynchronous computation and data transfer tasks. These Async-Finish primitives enable any (block) statement to be executed as a parallel task, including for-loop iterations and method calls [10]. HJLib also supports object-based isolation [12] and distributed program execution [13] for actors and selectors [14]. HJLib implements a priority-based lock-free work-stealing algorithm [15] with multiple thread pools to support priority scheduling of tasks. In this paper, we will mainly focus on using the async and finish primitives for creating and coordinating asynchronous tasks, with fine-grained atomic synchronization among tasks.

III. PARALLELIZATION APPROACHES

In this paper, we are mainly concerned with using multi-core single-node parallelism to reduce the latency of parsing a single sentence in agenda-based parsers. The irregular nature of probabilistic parsing makes this more challenging than simply parsing multiple sentences at once (embarrassing parallelism) to improve throughput. To maintain parser accuracy while processing multiple items on the agenda in parallel, we must consider the impact of read-write and write-write conflicts. We refer to both types of conflicts as interference.

A general interference pattern for a parser is as shown in Figure 2: i is the starting location in Algorithm 1, k is the ending location in Algorithm 1, each increases in the direction of their respective arrow. Each V shape highlights entries that are dependent on the base of the V. The cell shared by both V’s could be concurrently computed and thus subject to interference.

In the scenario shown, a write-write conflict can occur for the overlapping entry, since two scheduled updates can try to concurrently update the cached score. A read-write conflict can occur when the bottom blue entry has been processed and queues the purple entry to the agenda.

Fig. 2. Interference pattern in parallel parsing.

A. Parallel agenda parsing

Our overall approach to parallelization of agenda parsing involves asynchronous execution of multiple agenda items, while addressing two fundamental issues to maintain parser accuracy:

1) Prevention of write-write conflicts in asynchronous constituent updates

2) Maintenance of overall execution order in the agenda

To ensure the correctness of the probabilistic parsing algorithms, we must respect the max operation used to update a given A[x, i, k] cell. We accomplish this by replacing our maxing operation with an atomic compare-and-swap max which only performs the update if the value is guaranteed to increase. We manage each update as an agenda item inside of a BlockingPriorityQueue which provides a concurrent push and pop access and allows us to use a number of threads for executing the agenda parsing main loop as seen in algorithm 2. To achieve parallelism without forfeiting the overall execution order in the agenda, we pick a limited number of tasks off the agenda and execute them in parallel. The agenda preserves global order while the parallel execution in the top items is not necessarily prioritized according to the agenda. In the

As noted in [6], [7], this is just Dijkstra’s shortest-path algorithm, or rather Knuth’s generalization of it to weighted hypergraphs [8].
following sections we discuss the two approaches we studied to parallelize the agenda parser.

B. Task-based work-sharing forall construct

The HJLib forall API provides an easy way to treat all items in a collection as individual asynchronous tasks scheduled with an implicit finish barrier at the end. The implementation creates an async task for each work item before scheduling for execution. To maintain an approximate order of the agenda, we capture the top \( m \) entries of the agenda as shown in Listing 1. The naive usage of forall brings both overheads from creating an async task for each work item and synchronizing at each implicit finish barrier for every top \( m \) elements.

Listing 1: Habanero forall usage

```java
while(!agenda.isEmpty()){
    Collection<T> taskItems = agenda.slice(0, m);
    forall(taskItems, (t)->{ process(t); });
    // Implicit barrier
}
```

C. Parallelization with forasyncLazy

To reduce the synchronization overhead, we propose a new forasyncLazy API to reduce the overheads, and provide a more versatile execution of the agenda. A simplified implementation of forasyncLazy is shown in Listing 2. This API takes three parameters:

1) \( \text{numTasks} \), the number of tasks that should collectively work on the parallel loop
2) \( \text{next} \), a thread-safe iterator expression that returns the next element to be processed
3) \( \text{body} \), a lambda expression that performs the desired computation on the element

Unlike the forall API in which one task is created for every iteration, the forasyncLazy API creates a fixed number of async tasks each of which repeatedly performs the computation body on different tasks returned by the task generator. In this way, we can approximate the execution order based on the order returned by task generator by blocking an async task at the call for next element, while avoiding the cost of synchronization associated with repeated calls to finish.

Listing 2: Implementation of forasyncLazy

```java
public static<T> void forasyncLazy(\ldots) {
    finish() -> {
        for (int i=0; i < numTasks; i++) {
            async() -> {
                while(taskItems.hasNext())
                    body.apply(taskItems.next());
            }
        }
    }
}
```

D. Alternative approaches

A similarly priority-based approach for fine-grained parallelism found in [16] also processes individual items on the agenda in parallel, but treats each step of expanding the frontier as a separate agenda item, and has many more pushes and pops on the shared priority queue structure. The finely defined task items push more duplicate constituents onto the agenda, and the multiple copies are consolidated before pops to reduce interference. This work differs from ours in that it attempts to consolidate more finely defined items on the single shared agenda, and involves more synchronization for an update on the agenda than our strategies described above.

Past work from on fine-grained parallelism also focuses on partitioning the CKY workload to hardware thread mapping that involve efficient representation of the probability matrix and the context-free grammars matrix encoding to allow data locality [17].

Some modern parsing algorithms use pruning heuristics to intelligently avoid computing some cells of \( A[x, i, k] \). This ends up introducing irregularities which on GPUs becomes problematic when trying to schedule similarly sized work. However, by using queuing mechanisms, similar non-terminal expressions can be collected and executed in parallel on a warp [18], [19].

IV. Experimental Results

Our experiments were all performed by extending the Bubs [20] [21] code base which consists of multiple implementations for the parser’s \( A \)-matrix and parsing algorithm (such as CKY and agenda). We ran the experiments on a 2.8GHz Westmere-EP computing node with 12 Intel Xeon X5660 processor cores and 48 GB of RAM per node with RHEL 6.5. Each experiment was performed on 25 sentences with length less than 30 words each, on a grammar with \( \sim 2 \) million productions. All of our experiments only processed one sentence at a time using \( N \) processor cores concurrently.

Figure 3 shows the average time to parse a single sentence for different cases of agenda parsing, as a function of the number of cores used. The first curve (horizontal line) shows the sequential execution time of the original agenda parser implementation that we started with.

The “standard forall” curve shows the average execution time to parse a single sentence with the naive forall approach. The experiment shows promising scalability benefits before leveling off at around 8 workers. The forasyncLazy approach shows overall smaller execution times than for a single sentence, while displaying similar trends in scalability. The forasyncLazy approach improves performance relative to the “standard forall” by eliminating the synchronization overhead from finish barriers during parsing. Both approaches display the same level of accuracy as the serial reference implementation.

V. Future Work

In this paper we have explored different approaches to fine-grained parallelism in agenda based parsing by extending
Habanero-Java, and shows substantial performance improvements in experimental results. The methods in this paper are not limited to probabilistic context free grammar parsing, but can also be applied to other dynamic programming schemes. The Dyna programming language [9] allows easy high-level specification of such schemes via a Prolog-like pattern-matching notation for defining named values in terms of other named values. This work serves as a preamble to our long-term specification of such schemes via a Prolog-like pattern-matching notation for defining named values in terms of other named values. This work serves as a preamble to our long-term goal of building on the Habanero infrastructure to support the compilation of arbitrary Dyna programs (see Section II-B) into parallel Java bytecode using both the parallelization of the compilation of arbitrary Dyna programs (see Section II-B) named values. This work serves as a preamble to our long-term sequential agenda parser on a single thread.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Collaborative Grants No. 1629564 and 1629459 and in part supported by the Data Analysis and Visualization Cyberinfrastructure funded by NSF under grant OCI-0959097 and Rice University. We thank Tim Vieira, Nathaniel Filardo and Rishi Surendran for their assistance with this project.

REFERENCES


