Fine-grained parallelism in probabilistic parsing with Habanero Java*

Matthew Francis-Landau*, Bing Xue†, Jason Eisner* and Vivek Sarkar†
Johns Hopkins University
{mfl,jason}@cs.jhu.edu*
Rice University
{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. $A[\mathsf{Sentence}, 0, N]$ 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.

Production	Prob
Sentence → Laugh Smile	0.5
Laugh → Laugh Laugh	0.1
Laugh → *baa*	0.9
Smile → Smile Smile	0.2
Smile → *ba*	0.8
⋮	⋮

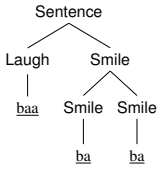

Fig. 1. A toy grammar modified from the SheepNoise grammar [4], with the optimal parse of the sentence "baa ba ba." The set of non-terminals is $T = \{\mathsf{Sentence}, \mathsf{Laugh}, \mathsf{Smile}\}$. The tree contains 5 rules and has probability $A[\mathsf{Sentence}, 0, 3] = 0.5 \times 0.9 \times 0.2 \times 0.8 \times 0.8$. Our experiments used much larger grammars and sentences (for humans, not sheep).

Algorithm 1 Typical CKY algorithm for probabilistic parsing

```
for w ∈ [2, N] do                                   ▷ width of constituent
    for i ∈ [0, N − w] do                           ▷ starting location
        k ← w + i                                   ▷ ending location
        for x ∈ T do                                ▷ nonterminals at location
            A[x, i, k] = max_{i<j<k, y,z∈T} A[y, i, j] · A[z, j, k] · P(x → 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 $A[\mathsf{Sentence}, 0, n] > 0$ (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 $A[\mathsf{Sentence}, 0, n]$ 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.

IA^3 2016; Salt Lake City, Utah, USA; November 2016
978-1-5090-3867-1/16/$31.00 ©2016 IEEE

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 → z y) > 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))
```

our agenda, allowing us to stop early and save work.[1]

Many other structured prediction problems can be regarded as theorem proving, with various proof rules, and computed via an agenda. The Dyna programming language [9] allows easy high-level specification of such schemes via a Prolog-like notation for defining named values in terms of other named values, something like a task graph. Aggregations such as our running maximum are directly supported using a pattern-matching notation. CKY can be expressed declaratively in Dyna as Algorithm 3 (where line 2 covers Algorithm 1), without committing to any serial order or parallelization scheme.

Algorithm 3 CKY expressed in Dyna

```
a(X,I,K)  max=  word(W,I,K) * rule_prob(X,W).
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.

[1] As noted in [6], [7], this is just Dijkstra's shortest-path algorithm, or rather Knuth's generalization of it to weighted *hyper*graphs [8].

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.

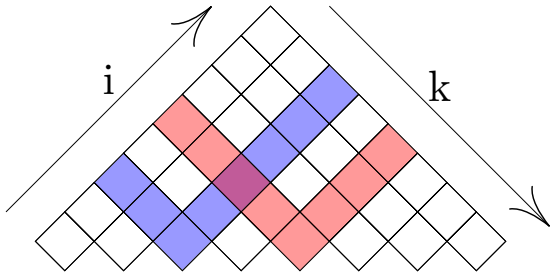

Fig. 2. Interference pattern in parallel parsing.

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

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

```
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) `numTasks`, the number of tasks that should collectively work on the parallel loop
2) `next`, a thread-safe iterator expression that returns the next element to be processed
3) `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`

```
public static <T> void forasyncLazy(. . .) {
  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 naïve `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

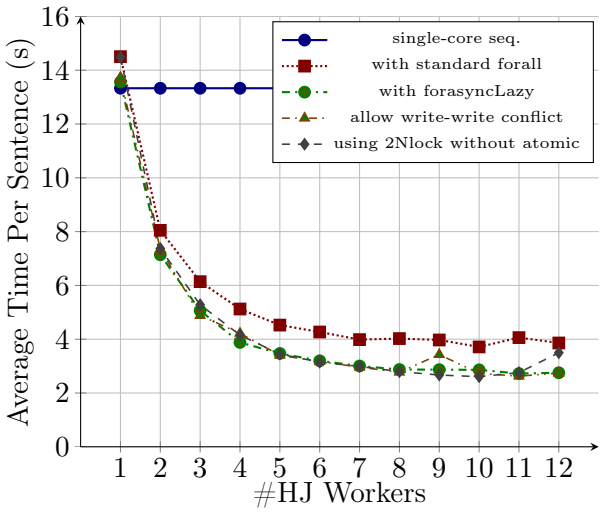

Fig. 3. Results obtained with a 12 processor core, 2.84GHz node with 48GBs of RAM. The y-axis is average execution time of the parser for a single sentence. The x-axis is the number of HJWorkers used in the experiment. The single core sequential plot shows a reference execution time of the original sequential agenda parser on a single thread.

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 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 agenda-based strategy and further optimization opportunities in more flexible strategies [22].

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

[1] F. C. N. Pereira and D. H. D. Warren, "Parsing as deduction," in *Proceedings of the 21st Meeting of the Association for Computational Linguistics*, 1983, pp. 137–144.

[2] J. Eisner, E. Goldlust, and N. A. Smith, "Compiling comp ling: Weighted dynamic programming and the Dyna language," in *Proceedings of HLT-EMNLP*, 2005, pp. 281–290. [Online]. Available: http://cs.jhu.edu/~jason/papers/#emnlp05-dyna

[3] J. K. Kummerfeld, D. Hall, J. R. Curran, and D. Klein, "Parser showdown at the wall street corral: An empirical investigation of error types in parser output," in *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*. Jeju Island, South Korea: Association for Computational Linguistics, July 2012, pp. 1048–1059. [Online]. Available: http://www.aclweb.org/anthology/D12-1096

[4] L. Torczon and K. Cooper, *Engineering A Compiler*, 2nd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

[5] M. Kay, "Algorithm schemata and data structures in syntactic processing," in *Readings in Natural Language Processing*, B. J. Grosz, K. Sparck Jones, and B. L. Webber, Eds. Kaufmann, 1986, pp. 35–70.

[6] D. Klein and C. D. Manning, "Parsing and hypergraphs," in *Proceedings of the International Workshop on Parsing Technologies (IWPT)*, 2001.

[7] M.-J. Nederhof, "Weighted deductive parsing and Knuth's algorithm," *Computational Linguistics*, vol. 29, no. 1, pp. 135–143, 2003.

[8] D. E. Knuth, "A generalization of Dijkstra's algorithm." *Information Processing Letters*, vol. 6, no. 1, pp. 1–5, 1977.

[9] J. Eisner and N. W. Filardo, "Dyna: Extending Datalog for modern AI," in *Datalog Reloaded*, ser. Lecture Notes in Computer Science, O. de Moor, G. Gottlob, T. Furche, and A. Sellers, Eds. Springer, 2011, vol. 6702, pp. 181–220, longer version available as tech report. [Online]. Available: http://cs.jhu.edu/~jason/papers/#eisner-filardo-2011

[10] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar, "Habanero-java: The new adventures of old x10," in *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, ser. PPPJ '11. New York, NY, USA: ACM, 2011, pp. 51–61. [Online]. Available: http://doi.acm.org/10.1145/2093157.2093165

[11] Y. Guo, R. Barik, R. Raman, and V. Sarkar, "Work-first and help-first scheduling policies for async-finish task parallelism," in *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*. Washington, DC, USA: IEEE Computer Society, May 2009, pp. 1–12.

[12] S. Imam, J. Zhao, and V. Sarkar, "A composable deadlock-free approach to object-based isolation," in *Euro-Par 2015: Parallel Processing*, ser. Lecture Notes in Computer Science, J. L. Trff, S. Hunold, and F. Versaci, Eds. Springer, 2015, vol. 9233, pp. 426–437.

[13] A. Chatterjee, B. Gvoka, B. Xue, Z. Budimlic, S. Imam, and V. Sarkar, "A distributed selectors runtime system for java applications," in *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, ser. PPPJ '16. New York, NY, USA: ACM, 2016, pp. 3:1–3:11. [Online]. Available: http://doi.acm.org/10.1145/2972206.2972215

[14] S. M. Imam and V. Sarkar, "Selectors: Actors with multiple guarded mailboxes," in *Proceedings of the 4th International Workshop on Programming Based on Actors Agents and Decentralized Control*, ser. AGERE! '14. New York, NY, USA: ACM, 2014, pp. 1–14. [Online]. Available: http://doi.acm.org/10.1145/2687357.2687360

[15] S. Imam and V. Sarkar, "Habanero-java library: A java 8 framework for multicore programming," in *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, ser. PPPJ '14. ACM, 2014, pp. 75–86. [Online]. Available: http://doi.acm.org/10.1145/2647508.2647514

[16] R. Grishman and M. Chitrao, "Evaluation of a parallel chart parser," Austin-Marriott at the Capitol, Austin, Texas, USA, 1988, pp. 71–76.

[17] A. Dunlop, N. Bodenstab, and B. Roark, "Efficient matrix-encoded grammars and low latency parallelization strategies for cyk," in *In IWPT 11*, 2011.

[18] D. Hall, T. Berg-Kirkpatrick, J. Canny, and D. Klein, "Sparser, better, faster gpu parsing," in *ACL*, 2014.

[19] J. Canny, D. Hall, and D. Klein, "A multi-teraflop constituency parser using GPUs," in *EMNLP*, 2013. [Online]. Available: http://www.aclweb.org/anthology/D13-1195

[20] N. Bodenstab, A. Dunlop, K. Hall, and B. Roark, "Beam-width prediction for efficient context-free parsing," in *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics*. Portland, Oregon: Association for Computational Linguistics, June 2011.

[21] ——, "bubs-parser." [Online]. Available: https://code.google.com/archive/p/bubs-parser/

[22] N. W. Filardo and J. Eisner, "A flexible solver for finite arithmetic circuits," in *Technical Communications of the 28th International Conference on Logic Programming, ICLP 2012*, ser. Leibniz International Proceedings in Informatics (LIPIcs), A. Dovier and V. S. Costa, Eds., vol. 17, Budapest, Sep. 2012, pp. 425–438. [Online]. Available: http://cs.jhu.edu/~jason/papers/#filardo-eisner-2012-iclp