Fine-grained parallelism in probabilistic parsing with Habanero Java

Matthew Francis-Landau\textsuperscript{1}, Bing Xue\textsuperscript{2}, Jason Eisner\textsuperscript{1} and Vivek Sarkar\textsuperscript{2}

Johns Hopkins University\textsuperscript{1} and Rice University\textsuperscript{2}

This material is based upon work supported by the National Science Foundation under Collaborative Grants No. 1629564 and 1629459
Probabilistic Parsing

● Core problem in Natural Language Processing (NLP)
  ○ Computationally expensive
  ○ Load-balancing is hard at fine-grained level

● Similar programming patterns appear in many machine learning (ML) algorithms
  ○ Parallelizing probabilistic parsing algorithms can be a proxy task for parallelization of a large set of ML algorithms
Production: a rewrite rule specifying a symbol substitution that can be recursively performed to generate new symbol sequences (from Wikipedia)

<table>
<thead>
<tr>
<th>Production</th>
<th>Prob</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sentence → Laugh Smile</td>
<td>0.5</td>
</tr>
<tr>
<td>Laugh → Laugh Laugh</td>
<td>0.1</td>
</tr>
<tr>
<td>Laugh → baa</td>
<td>0.9</td>
</tr>
<tr>
<td>Smile → Smile Smile</td>
<td>0.2</td>
</tr>
<tr>
<td>Smile → ba</td>
<td>0.8</td>
</tr>
</tbody>
</table>

Stochastic Grammar
Millions of productions!

<table>
<thead>
<tr>
<th>Production</th>
<th>Prob</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sentence $\rightarrow$ Laugh Smile</td>
<td>0.5</td>
</tr>
<tr>
<td>Laugh $\rightarrow$ Laugh Laugh</td>
<td>0.1</td>
</tr>
<tr>
<td>Laugh $\rightarrow$ \textit{baa}</td>
<td>0.9</td>
</tr>
<tr>
<td>Smile $\rightarrow$ Smile Smile</td>
<td>0.2</td>
</tr>
<tr>
<td>Smile $\rightarrow$ \textit{ba}</td>
<td>0.8</td>
</tr>
</tbody>
</table>

Baa ba ba
Probabilistic Parsing

Algorithm 1 Typical CKY algorithm for probabilistic parsing

1: \textbf{for} $w \in [2, N]$ \textbf{do} \hspace{1cm} \triangleright width of constituent
2: \hspace{.5cm} \textbf{for} $i \in [0, N - w]$ \textbf{do} \hspace{1cm} \triangleright starting location
3: \hspace{1.5cm} $k \leftarrow w + i$ \hspace{1cm} \triangleright ending location
4: \hspace{2cm} \textbf{for} $x \in T$ \textbf{do} \hspace{1cm} \triangleright nonterminals at location
5: \hspace{2.5cm} $A[x, i, k] = \max_{i < j < k, y, z \in T} A[y, i, j] \cdot A[z, j, k] \cdot P(x \rightarrow y \ z)$

CKY expressed declaratively in Dyna\cite{1}

\begin{align*}
\text{a}(X,I,K) \text{ max}= & \text{ word}(W,I,K) \times \text{ rule\_prob}(X,W). \\
\text{a}(X,I,K) \text{ max}= & \text{ a}(Y,I,J) \times \text{ a}(Z,J,K) \times \text{ rule\_prob}(X,Y,Z) \\
\text{goal} &= \text{ a}\("\text{Sentence}\", 0, n). \\
\end{align*}

A[x, i, k] is max of all probabilities of substring [i:k] producing non-terminal symbol x from symbols y and z

We want to derive the Sentence symbol

2 nested loops through all productions
Probabilistic Parsing

\[ x = \text{Sentence} \]

\[ A[x, i, k] = \max_{i < j < k, y, z \in T} A[y, i, j] \cdot A[z, j, k] \cdot P(x \rightarrow y \ z) \]

- Fill in each cell for substrings of size 2

2 nested loops through all productions
Probabilistic Parsing

\[ x = \text{Sentence} \]

\[ A[x, i, k] = \max_{i < j < k} A[y, i, j] \cdot A[z, j, k] \cdot P(x \rightarrow y \ z) \]

- Fill in each cell for substrings of size 3 based on values from substrings of size 2 according to the production rules

2 nested loops through all productions
Probabilistic Parsing

\[ x = \text{Sentence} \]

\[ A[x, i, k] = \max_{i < j < k, y, z \in T} A[y, i, j] \cdot A[z, j, k] \cdot P(x \rightarrow y z) \]

- The parse tree with the largest probability ends up at position (Sentence, 0, N)
Probabilistic Parsing

More realistically, probabilistic parsing is an irregular application ...

- Not all cells get filled
  - Some productions do not exist for $x$
  - Lower half of the matrix is unused

- Not all cells take the same amount of time to fill
  - Number of possible productions varies for each substring

- Most work wasted
  - Most cells do not contribute to final result (the upper left corner) because their contributions are ultimately beaten in some “max” operation
Alternative to CKY: Agenda Parsing

- Worklist version of CKY parsing (or an approximation)
  - Each update to a cell is a work item, and put them into an agenda
  - Prioritizes updates with higher probability
- **Stop early** and **save work** given “good enough” parse tree
  - Eliminates much unneeded computation in CKY
  - Reaches “good enough” parse tree faster with its greedy approach
  - If the priority function is an admissible A* heuristic, the algorithm becomes exact
- **A generalized Dijkstra’s Algorithm**
  - Can be applied to machine learning algorithms similar to probabilistic parsing
  - A “meta-algorithm” for dynamic programming schemes
Cell-level Parallel Agenda Parsing

- Need to process multiple agenda items (cell update) in parallel
  - use Java’s `BlockingPriorityQueue` for thread-safe worklist and Habanero Java (HJLib)\(^2\) parallel constructs for asynchronous processing
- Need to ensure total order of execution on agenda
  - Capture top \(m\) items on agenda to process in parallel

\(x = \text{Sentence}\)

\[x = \text{Sentence}\]

---

Write-write conflict happens when two agenda items want to update the same cell.

Need to ensure atomic max operation on cell updates:
- Max operation implemented with CAS (Compare-And-Swap)
Cell-level Parallel Agenda Parsing

- Two serially dependent cells can be updated at the same time
- Need to ensure the most recent maximum value is considered
  - An update to cell value will generate new update with higher priority on agenda
Parallel Agenda Parsing with Habanero Java[2]

```java
class AgendaParser {
    ...
    while(!agenda.isEmpty()){
        Collection<T> taskItems =
            agenda.slice(0,m);
        \textbf{forall} (taskItems, (t)->{
            process(t);
        });
        //implicit finish
    }
    ...
}
```

- Treat all agenda items as individual asynchronous tasks
- Capture top $m$ items on agenda to process in parallel
- HJLib forall construct creates asynchronous tasks for each item in a Collection with an implicit barrier


Parallel Agenda Parsing with Habanero Java[2]

- New API
  ```java
  forasyncLazy(numTasks, taskItems, processBody)
  ○ numTasks - number of async processes to create
  ○ taskItems - an iterator as task generator
  ○ processBody - lambda expression to operate on an task item
  
  - Agenda as taskItems always returns true for hasNext() until parse completes

```
Experimental Results

- Extend the bubs-parser[^3][^4] code base’s agenda parser
- 2.8GHz Westmere-EP computing nodes
  - 12 Intel Xeon X5660 processor cores
  - 48GB RAM per node
- 25 sentences
  - < 30 words per sentence
- Grammar with ~2 Million productions

Conclusion and Future Work

● ~5x performance improvements due to parallelism without impairment on accuracy
● Methods applicable to general dynamic programming schemes
● Dyna language\cite{1} provides high-level specification of DP schemes
● Our long-term goal is to support source-to-source compilation of Dyna programs into parallel HJ programs for multicore and distributed-memory parallelism

CKY expressed declaratively in Dyna\cite{1}

\[
\begin{align*}
    a(X,I,K) &\text{ max}= \text{ word}(W,I,K) \times \text{ rule\_prob}(X,W). \\
    a(X,I,K) &\text{ max}= a(Y,I,J) \times a(Z,J,K) \times \text{ rule\_prob}(X,Y,Z) \quad \text{goal} = a("Sentence", 0, n).
\end{align*}
\]

(Incomplete) HJLib code

```java
Iterator<ChartCell> agendaItems = new Iterator<>(){
    public boolean hasNext() { return !doneParsing; }
    public T next() { return agenda.take(); }
}
finish( ()->{
    forasyncLazy(numTasks, agendaItems, (c) -> {
        chartUpdate(c.i, c.k, c.x);
        agendaUpdate(c, chart);
    }
});
return chart.get("Sentence")[0][N];
```


Thank you for your time!

Questions?
Java Grammar: a set of production rules that describes how valid strings are formed according to a language’s syntax.
Parsing Java Programs

Deterministic grammar
Small number of grammar rules
Parsing Natural Language