Parsing in Parallel on Multiple Cores and GPUs

Mark Johnson

Centre for Language Sciences and Department of Computing
Macquarie University

ALTA workshop
December 2011
Why parse in parallel?

- The future of computing is parallel processing
  - CPUs are unlikely to get much faster
  - but the number of processing units is likely to increase dramatically
- Can we effectively use parallel processing for parsing?
  - straight-forward approach: divide the sentences amongst the processors
  - but some unsupervised grammar induction procedures require reparsing the training corpus many times and update the grammar after each parse
Outline

Review of parallel architectures

Approaches to parallel parsing

Experimental evaluation

Conclusion and future work
Popular parallel architectures

- **Networked clusters**
  - commodity machines or blade servers
  - communication via network (e.g., Ethernet) (slow)
  - tools: Message-passing Interface (MPI), Map-Reduce

- **Symmetric multi-processor (SMP) machines**
  - multiple processors or cores executing different code
  - communication via *shared memory* (fast)
  - tools: OpenMP, pthreads

- **Graphics Processor Units (GPUs)**
  - Single Instruction Multiple Threads (SIMT) parallelism
  - communication via specialised shared memory (fast)
  - tools: CUDA, OpenCL

- Multi-core SMPs and GPUs are becoming more alike
Parallelisation in CPUs

- Modern CPUs have become increasingly parallel
  - SIMD vectorised floating point arithmetic (SSE)
- Multicore (8 or 12 core) CPUs are now standard
- Highly uniform memory architecture make these easy to program
GPUs have more compute power than CPUs
GPUs are highly parallel

- GPUs can run *hundreds of threads simultaneously*
- Highly data-parallel SIMT operations
- There are *general-purpose programming tools* (CUDA, OpenCL), but programming is hard
  - non-uniform memory architecture
- Standard libraries exist for e.g. *matrix calculations* (CUBLAS)
- The hardware and software are *evolving rapidly*
What’s hard about parallel programming?

- **Copying in parallel is easy**
  
  \[
  \text{for } i \text{ in } 1, \ldots, n:\n  \quad C[i] = A[i] + B[i]
  \]
  
  ▶ runs in constant time (with enough processors)

- **Reduction is parallel is hard**

  \[
  \text{sum } = 0
  \]
  
  \[
  \text{for } i \text{ in } 1, \ldots, n:\n  \quad \text{sum } += A[i] + B[i]
  \]
  
  ▶ standard approach uses a binary tree
  ▶ runs in \(O(\log n)\) time
  ▶ OpenMP can automatically generate code for simple reductions
  ▶ many tutorials on how to do this in CUDA
Outline

Review of parallel architectures

Approaches to parallel parsing

Experimental evaluation

Conclusion and future work
Sentence-level parallelism

- Baseline approach: *to parse a corpus, divide the sentences amongst the processors*
  - standard approach to parsing a corpus on a networked cluster
  - works well on SMP machines too
  - impractical on GPUs (memory, program complexity) (?)
- Not applicable in *real-time applications* or *certain specialised sequential algorithms* (e.g., “collapsed” MCMC samplers)
Why is sub-sentential parallel parsing hard?

- Hierarchical structure \(\Rightarrow\) parsing operations must be ordered
  - assume standard \textit{bottom-up ordering} here
    \(\Rightarrow\) smaller constituents needed to build larger constituents
- Scores of \textit{ambiguous parses} need to be appropriately combined. If different analyses are constructed by different processes, we may need \textit{synchronisation}
- Parallel work units must be \textit{large enough that synchronisation costs don't dominate}
CFGs in Chomsky Normal Form

• Every Context-Free Grammar (CFG) is equivalent to a CFG in Chomsky Normal Form (CNF), where all rules are either:
  ▶ binary rules of the form $A \rightarrow BC$, where $A$, $B$ and $C$ are nonterminal symbols, or
  ▶ unary rules of the form $A \rightarrow w$, where $A$ is a nonterminal symbol and $w$ is a terminal symbol.

• All standard $O(n^3)$ CFG parsing algorithms explicitly or implicitly convert the grammar into CNF
The parsing chart

- **String positions** identify the begin and end of each constituent
- Example: If \( w = \text{the cat chased the dog} \), then the string positions are:
  
  \[
  \begin{array}{cccccc}
  & 0 & \text{the} & 1 & \text{cat} & 2 & \text{chased} & 3 & \text{the} & 4 & \text{dog} & 5
  \end{array}
  \]

  and the substring \( w_{2:5} = \text{chased the dog} \)

- Given a string to parse \( w = w_1 \ldots w_n \), the chart is a table \( \text{Chart}[i, k, A] \) where:
  
  \[
  \text{Chart}[i, k, A] = \text{score of all analyses } A \Rightarrow^+ w_{i+1} \ldots w_k
  \]

- Example (continued): \( \text{Chart}[2, 5, \text{VP}] \) is score of all ways of analysing \( \text{chased the dog} \) as a VP.

- The parse tree can be identified in \( O(n^2) \) time from a complete chart, so constructing the chart is the rate-limiting step
The chart recursion for a CNF PCFG

- **Terminals:** (base case)

\[
\text{Chart}[i-1, i, A] = P(A \rightarrow w_i)
\]

- **Nonterminals:** (recursion)

\[
\text{Chart}[i, k, A] = \sum_{A \rightarrow B C} \sum_{j:i<j<k} P(A \rightarrow B C) \text{Chart}[i, j, B] \text{Chart}[j, k, C]
\]

(For Viterbi parsing, replace sums with max)
Computing the chart

\[
\begin{align*}
&\text{for } i \text{ in } 0, \ldots, n-1: \\
&\quad \text{for } a \text{ in } 0, \ldots, m-1:
\quad \quad \text{Chart}[i,i+1,a] = \text{Terminal}[\text{Word}[i],a]
\end{align*}
\]

\[
\begin{align*}
&\text{for gap in } 2, \ldots, n: \\
&\quad \text{for } i \text{ in } 0, \ldots, n-gap: \\
&\quad \quad k = i + \text{gap} \\
&\quad \quad \text{for } a \text{ in } 0, \ldots, m-1:
\quad \quad \quad \text{Chart}[i,k,a] = 0
\quad \quad \quad \text{for } j \text{ in } i+1, \ldots, k-1: \\
&\quad \quad \quad \quad \text{for } b \text{ in } 0, \ldots, m-1:
\quad \quad \quad \quad \quad \text{for } c \text{ in } 0, \ldots, m-1:
\quad \quad \quad \quad \quad \quad \quad \text{Chart}[i,k,a] += \text{Rule}[a,b,c] \ast \text{Chart}[i,j,b] \ast \text{Chart}[j,k,c]
\end{align*}
\]

- Non-terminal calculation consumes bulk of time
- The blue loops can be freely reordered and computed in parallel
- The red loops can be freely reordered and accumulate in parallel
- Need to synchronise updates to Chart[A, i, k]
Factored CKY parsing

```
for gap in 2, . . ., n:
    for i in 0, . . ., n − gap:
        k = i + gap
        for b in 0, . . ., m − 1:
            for c in 0, . . ., m − 1:
                BC[b, c] = 0
                for j in i + 1, . . ., k − 1:
                    BC[b, c] += Chart[i, j, b] * Chart[j, k, c]
    for a in 0, . . ., m − 1:
        Chart[i, k, a] = 0
        for b in 0, . . ., m − 1:
            for c in 0, . . ., m − 1:
                Chart[i, k, a] += Rule[a, b, c] * BC[b, c]
```

- Proposed by Dunlop, Bodenstab and Roark (2010)
  - reduces “grammar constant” by reducing the degree of loop nesting
Multi-core SMP parallelism for PCFG parsing

- Experimented with a parallel matrix algebra package, but results were disappointing
- *OpenMP programs* are C++ programs with *pragmas* that indicate which loops can be parallelised, and how
- Synchronisation constructs used:
  - thread-private variables
  - parallel “for” reductions
  - atomic updates (for reductions)
- Experimented with various loop reorderings and parallelisation
- Here we report results for parallelising:
  - the *outermost loops* (over $i$ and $a$)
  - the *innermost loops* (over $j$, $b$ and $c$)
  - *all loops*
A CUDA GPU kernel for PCFG parsing

- Using CUBLAS ran $100 \times$ slower than unparallelised CPU version
- Direct translation into CUDA ran $200 \times$ slower than unparallelised CPU version
- Recoded algorithm to exploit:
  - global memory (slow but accessible to all blocks; stores Chart)
  - texture memory (faster but read-only; stores Rule)
  - shared memory (accessible to all threads in block; stores BC)
  - thread-local memory (to accumulate intermediate results)
- Computes all diagonals in chart in parallel
- Used a custom algorithm to perform reduction in parallel:
  \[ BC[b,c] += Chart[i,j,b] \times Chart[j,k,c] \]
  - code used depends on whether it can be done in one block
Outline

Review of parallel architectures

Approaches to parallel parsing

Experimental evaluation

Conclusion and future work
Experimental set-up

- Experimented on a range of different dense PCFGs
  - a PCFG is dense iff $P(A \rightarrow B C) > 0$ for most $A, B, C$
  - dense grammars arise in unsupervised grammar learning
  - report results for a PCFG with 32 nonterminals, 32,768 binary rules with random rule probabilities (as typical in unsupervised grammar learning)

- Experiments run on dual quad-core 3.0GHz Intel Harpertown CPUs and a NVIDIA Fermi s2050 GPU with 448 CUDA cores running at 1.15GHz

- Software: CUDA 3.2 toolkit and gcc 4.4.4 with SSE3 SIMD floating-point vector subsystem

- All experiments run twice in succession; variation < 1%
### Average parse times

<table>
<thead>
<tr>
<th>Parser</th>
<th>Sentences/sec</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>11</td>
<td>1.0</td>
</tr>
<tr>
<td>(i) outer parallel</td>
<td>84</td>
<td>7.5</td>
</tr>
<tr>
<td>(ii) inner parallel</td>
<td>11</td>
<td>1.0</td>
</tr>
<tr>
<td>(iii) both parallel</td>
<td>29</td>
<td>2.6</td>
</tr>
<tr>
<td>Factored</td>
<td>122</td>
<td>11.0</td>
</tr>
<tr>
<td>(i) outer parallel</td>
<td>649</td>
<td>60.0</td>
</tr>
<tr>
<td>(ii) inner parallel</td>
<td>27</td>
<td>2.4</td>
</tr>
<tr>
<td>(iii) both parallel</td>
<td>64</td>
<td>5.7</td>
</tr>
<tr>
<td>CUDA</td>
<td>206</td>
<td>18.4</td>
</tr>
</tbody>
</table>

- Parsing speeds of the various algorithms on 1,345 sentences from section 24 of the Penn WSJ treebank.
- Speed-up is relative to the baseline parser.
Parse times as a function of sentence length

Parser
- baseline
- factored
- baseline+outer SMP
- factored+outer SMP
- CUDA

Sentence length vs. Parsing time (seconds)
Speedups as a function of sentence length

![Graph showing speedups as a function of sentence length. The x-axis represents sentence length, while the y-axis represents speedup relative to the baseline parser. There are multiple lines for different parsers: baseline, factored, baseline+outer SMP, factored+outer SMP, and CUDA. The graph illustrates how each parser performs under varying sentence lengths.]
Outline

Review of parallel architectures

Approaches to parallel parsing

Experimental evaluation

Conclusion and future work
Conclusion

- Large speedups with both SMP and CUDA parallelism
  - SMP speedup close to theoretical maximum (×8)
  - *parallelising inner loops hurts rather than helps*
    perhaps this destroys SSE SIMD vectorisation?
- SMP implementation was faster than CUDA implementation
  - CUDA is 18 × faster than baseline
  - CUDA is *comparatively slower on short sentences* (initialisation costs?)
- The Dunlop, Bodenstab and Roark (2010) factorisation is very useful!
Future work

- Repeat these experiments on newer hardware
  - 24-core SMP machines now available
  - new GPUs are more powerful and easier to program
- Experiment with other GPU-based parsing algorithms
  - non-uniform architecture $\Rightarrow$ many variations to try
  - parse multiple (short) sentences at once
- Extend this work to other kinds of grammars
  - sparse PCFGs
  - dependency grammars
PCFG parsing as matrix arithmetic

# (2) build larger constituents from smaller
for gap = 2, ..., n:
    for i = 0, ..., n–gap:
        k = i + gap
        for A in Nonterminals:
            for j = i+1, ..., k–1:
                $\text{Chart}[i,k,A] += \text{Chart}[i,j,\cdot]^T \times \mathbf{R}[A] \times \text{Chart}[j,k,\cdot]$

where $\mathbf{R}$ is a vector of matrices

\[ \mathbf{R}[A](B,C) = \mathbf{P}(A \rightarrow B \ C) \]

- Our matrices are often small $\Rightarrow$ not much parallelism gain (?)
- Other matrix formulations may be more efficient
  - accumulating results one at a time is inefficient
  - would be nice to parallelise more loops
Sparse grammars

- Many realistic grammars are *sparse*, so dense matrix-based approaches are inefficient in time and memory
  - Converting a grammar into CNF may introduce many new nonterminals
    - Example: left binarisation replaces $VP \rightarrow VB \ NP \ PP$ with the pair of rules
      
      \[
      VP \rightarrow VB_\ NP \ PP \\
      VB_\ NP \rightarrow VB \ NP
      \]
    - new nonterminals (e.g., $VB_\ NP$) appear in few rules

- The sparsity pattern depends heavily on the grammar involved
  - *fastest parsing algorithm may depend on grammar*

- Hash tables are a standard uniprocessor implementation technique for sparse grammars
  - For SMP, parallel hash tables seem practical
  - For GPUs, other techniques (e.g., sort and reduce) may be more effective