Parsing in Parallel on Multiple Cores and GPUs

Mark Johnson

Centre for Language Sciences
and
Department of Computing
Macquarie University

ALTA workshop
December 2011

MACQUARIE UNIVERSITY

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

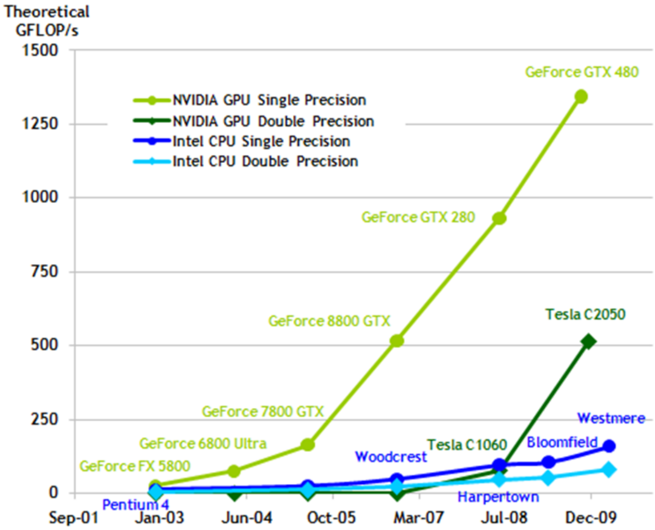
Popular parallel architectures

- *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

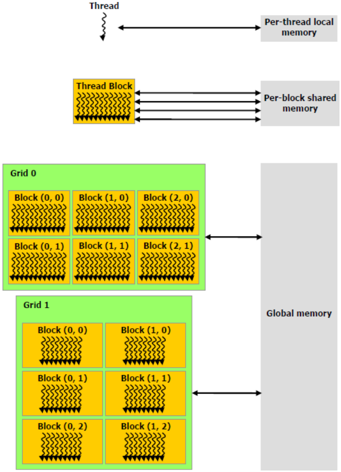
GPUs have more compute power than CPUs



MACQUARIE UNIVERSITY

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*



MACQUARIE UNIVERSITY

What's hard about parallel programming?

- *Copying in parallel is easy*

  for i **in** 1, ..., n:
  $C[i] = A[i] + B[i]$

  - runs in constant time (with enough processors)
- *Reduction is parallel is hard*

  $sum = 0$
  for i **in** 1, ..., n:
  $sum \mathrel{+}= A[i] + B[i]$

  - 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

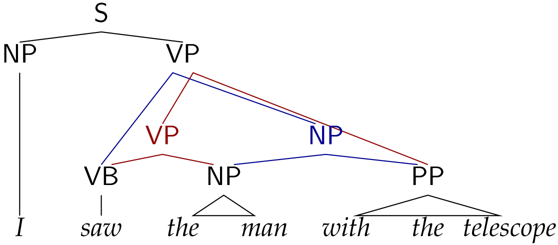
Conclusion and future work

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 ⇒ parsing operations must be *ordered*
  - assume standard *bottom-up ordering* here
  - ⇒ smaller constituents needed to build larger constituents
- Scores of *ambiguous parses* need to be appropriately combined. If different analyses are constructed by different processes, we may need *synchronisation*
- Parallel work units must be *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 $A \rightarrow B\,C$, 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

MACQUARIE UNIVERSITY

The parsing chart

- *String positions* identify the begin and end of each constituent
- Example: If $\boldsymbol{w}$ = *the cat chased the dog*, then the string positions are:

$$_0 \textit{ the } _1 \textit{ cat } _2 \textit{ chased } _3 \textit{ the } _4 \textit{ dog } _5$$

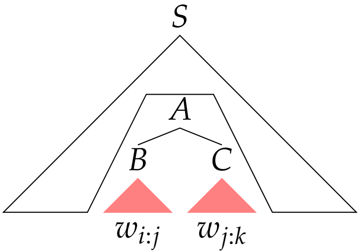
  and the substring $\boldsymbol{w}_{2:5}$ = *chased the dog*
- Given a string to parse $\boldsymbol{w} = w_1 \ldots w_n$, the *chart* is a table $Chart[i, k, A]$ where:

$$Chart[i, k, A] \quad = \quad \text{score of all analyses } A \Rightarrow^{+} w_{i+1} \ldots w_k$$

- Example (continued): $Chart[2, 5, \mathrm{VP}]$ is score of all ways of analysing *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*

MACQUARIE UNIVERSITY

The chart recursion for a CNF PCFG



- *Terminals:* (base case)

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

- *Nonterminals:* (recursion)

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

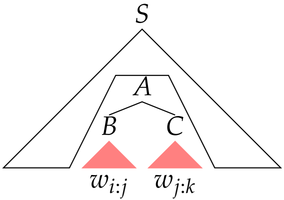
(For Viterbi parsing, replace sums with max)

MACQUARIE UNIVERSITY

Computing the chart

```
for i in 0, ..., n−1:
  for a in 0, ..., m−1:
    Chart[i,i+1,a] = Terminal[Word[i],a]

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

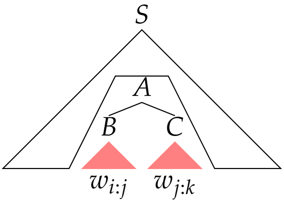


- Non-terminal calculation consumes bulk of time
- *The blue loops can be freely reordered and computed in parallel*
- *The red loops can be freely reordered and accumulate in parallel*
- Need to *synchronise updates to* $Chart[A, i, k]$

MACQUARIE UNIVERSITY

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*

MACQUARIE UNIVERSITY

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 × *slower* than unparallelised CPU version
- Direct translation into CUDA ran 200 × *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:

$$BC[b,c] \mathrel{+}= Chart[i,j,b] * Chart[j,k,c]$$

    - code used depends on whether it can be done in one block

MACQUARIE UNIVERSITY

Outline

MACQUARIE UNIVERSITY

Experimental set-up

- Experimented on a range of different *dense PCFGs*
  - a PCFG is dense iff $\mathrm{P}(A \to B\ C) > 0$ 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 $< 1\%$

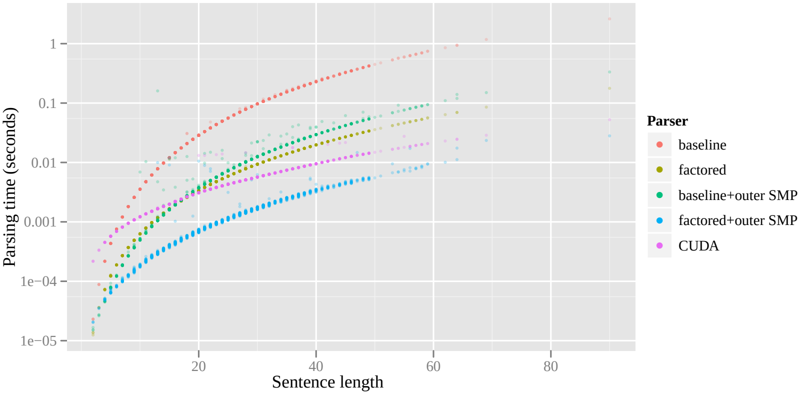
MACQUARIE UNIVERSITY

Average parse times

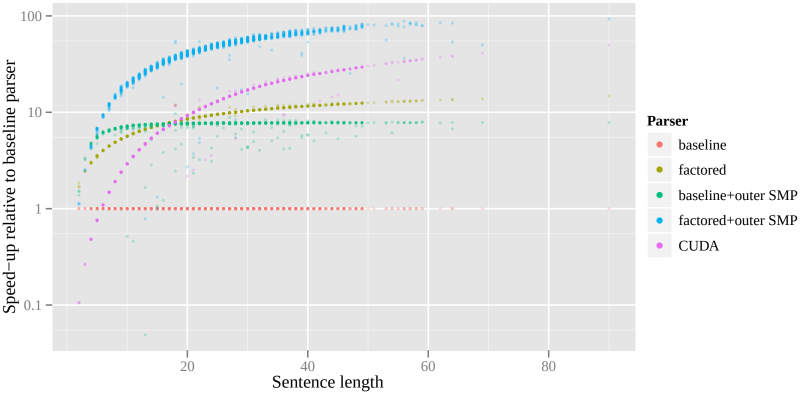
Parser	Sentences/sec	Speed-up
Baseline	11	1.0
(i) outer parallel	84	7.5
(ii) inner parallel	11	1.0
(iii) both parallel	29	2.6
Factored	122	11.0
(i) outer parallel	**649**	**60.0**
(ii) inner parallel	27	2.4
(iii) both parallel	64	5.7
CUDA	206	18.4

- 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



Speedups as a function of sentence length



Outline

Conclusion

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

MACQUARIE UNIVERSITY

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:
                Chart[i,k,A] += Chart[i,j,·]^T × R[A] × Chart[j,k,·]
```

where **R** is a *vector of matrices*

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

- Our matrices are often small ⇒ 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

MACQUARIE UNIVERSITY

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 → VB NP PP with the pair of rules

$$\begin{aligned} \mathsf{VP} &\rightarrow \mathsf{VB\_NP}\ \mathsf{PP} \\ \mathsf{VB\_NP} &\rightarrow \mathsf{VB}\ \mathsf{NP} \end{aligned}$$

    - 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