Parsing in Parallel on Multiple Cores and GPUs

Mark Johnson
Centre for Language Sciences and Department of Computing
Macquarie University
Sydney, Australia
Mark.Johnson@MQ.edu.au

Abstract

This paper examines the ways in which parallelism can be used to speed the parsing of dense PCFGs. We focus on two kinds of parallelism here: Symmetric Multi-Processing (SMP) parallelism on shared-memory multicore CPUs, and Single-Instruction Multiple-Thread (SIMT) parallelism on GPUs. We describe how to achieve speed-ups over an already very efficient baseline parser using both kinds of technology. For our dense PCFG parsing task we obtained a 60×speed-up using SMP and SSE parallelism coupled with a cache-sensitive algorithm design, parsing section 24 of the Penn WSJ treebank in a little over 2 secs.

1 Introduction

Performance improvements in computing come increasingly through greater parallelism. This paper studies ways in which this parallelism can be used to improve the speed of PCFG parsing in computational linguistics. Although we focus on a particular task here (constructing the inside chart for dense PCFGs), we expect the insights to be generally applicable.

There are three major ways in which computers are becoming more parallel. At the broadest level, it is now common to network large numbers of computers together into clusters, which are controlled by software such as the Message-Passing Interface (MPI) (Gropp et al., 1999) or Map-Reduce (Lin and Dyer, 2010).

At a lower level, even commodity computers typically have multiple processors or cores, which are connected by a high-speed bus to a shared memory, enabling Symmetric Multi-Processor (SMP) parallelism. SMP parallelism is typically controlled by software such as OpenMP (Chapman et al., 2007) or pThreads. Commodity computers also possess on-chip parallel floating point vectorised arithmetic units. The CPUs we used here have SSE (Streaming SIMD Extensions) vectorised arithmetic, where SIMD abbreviates "Single-Instruction Multiple-Data". SSE is enabled by appropriate compiler flags.

Finally, Graphics Processor Units (GPUs) are increasingly becoming both less specialised and more powerful (on many commodity machines they can perform more floating-point operations per second than the CPU); as we will see here, a GPU can yield quite respectable parsing performance. GPUs are designed for massively parallel Single-Instruction Multiple-Thread (SIMT) programs; each GPU thread is comparatively slow, but the GPU can execute hundreds or thousands of threads in parallel. GPUs are typically programmed using tools such as OpenCL or CUDA (Sanders and Kandrot, 2011).

We concentrate on SMP multi-core and GPU parallelism in this paper because we expect that communication latencies with conventional networking hardware make parallel parsing with networked clusters impractically inefficient. Communication latency is much less of a problem with shared memory SMP and GPU parallelism as communication takes place over the machine's high-speed bus.

A base-line approach for exploiting parallelism in parsing is simply to parse different sentences in parallel on separate instances of the parser. This

Mark Johnson. 2011. Parsing in Parallel on Multiple Cores and GPUs. In *Proceedings of Australasian Language Technology Association Workshop*, pages 29−37

is likely to be the best way to exploit parallelism with networked clusters and SMP multi-core machines when parsing a large corpus of sentences off-line. However, there are situations where parsing must be on-line; e.g., when parsing is a component of a system that interacts with users, or with machine-learning algorithms such as Metropolis-Hastings Sampling that update after each sentence is parsed (Johnson et al., 2007).

2 Previous work

Parsing in parallel has been studied for several decades, and space constraints prevent anything but a cursory summary here. Hill and Wayne (1991) identified the basic data dependencies between the entries in chart cells, and discussed their implications for parallel parsing. Nijholt (1994) also studied the order in which chart cells can be filled, and discusses its implications for a variety of shift-reduce and chart-based parsing algorithms. Thompson (1994) pointed out that the close relationship between CKY parsing and matrix multiplication can be exploited for parsing in parallel; we rely on similiar observations below. Ninomiya et al. (1997) described an agenda-based approach for parallelising CKY parsing on large SMP machines, while Bordim et al. (2002) describes the implementation of a CKY parser on Field-Programmable Gate Arrays (FPGAs). Sandstrom (2004) describes a parallel implementation of Earley's parsing algorithm. Dunlop et al. (2010) stresses importance of minimising cache misses in the design of efficient parsing algorithms and described how to restructure the CKY algorithm to reduce grammar constant,[1] which as we show below has a dramatic impact on parallel SMP parsing.

3 The CKY algorithm for dense PCFGs

This section introduces the basic CKY parsing algorithm used below. Here we're assuming that the grammar is in Chomsky-Normal Form (CNF), i.e., all rules in the grammar are of the form $A \rightarrow B\,C$

[1]The "grammar constant" refers to the variation in parsing time as a function of grammar size. Standard analyses study how parsing time varies as a function of sentence length while the grammar is held constant; the choice of grammar affects parsing time in such analyses via a "grammar constant".

```
for i in 0, . . ., n−1:
  for a in 0, . . ., m−1:
    C[i,i+1,a] = T[W[i],a]

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

where:

m	number of nonterminals in grammar
n	length of input string
$W[i]$	word in input string at position i
$T[w, A]$	probability of $A \rightarrow w$
$R[A, B, C]$	probability of $A \rightarrow B\,C$
$C[i, k, A]$	inside probability of A spanning (i, k)

Figure 1: Pseudo-code for baseline CKY algorithm for dense PCFG parsing and an explanation of the variables used in the algorithm. All arrays are in row-major order, except that the inside chart C uses specialised indexing to take advantage of the fact that the first string position index is always less than the second.

```
for i in 0, ..., n−1:
  for a in 0, ..., m−1:
    C[i,i+1,a] = T[W[i],a]

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

where:

m	number of nonterminals in grammar
n	length of input string
$W[i]$	word in input string at position i
$T[w, A]$	probability of $A \rightarrow w$
$R[A, B, C]$	probability of $A \rightarrow B\,C$
$C[i, k, A]$	inside probability of A spanning (i, k)
BC	an $m \times m$ scratch array
Zero	an $m \times m$ array of zeros.

Figure 2: Pseudo-code for the factored CKY algorithm for dense PCFG parsing.

or $A \rightarrow w$, where A, B and C are nonterminals and w is a terminal (Aho and Ullman, 1972). In this paper we focus on *dense* PCFGs, i.e., where most of the possible rules have positive probability. Dense grammars with these properties occur in applications such as unsupervised grammar induction. While sparse grammars have many important applications, there are many different possible patterns of sparsity, and the optimal parsing algorithm may depend on the particular sparsity pattern the grammar instantiates. Moreover, it is extremely difficult to develop effective search procedures (such as heuristic $A^\star$ search) for dense grammars in which most rules have approximately equal probabilities, so this is a situation where a brute-force exhaustive calculation of the kind that the algorithms discussed below may well be the preferred approach.

We focus on CKY-style pure bottom-up parsing algorithms here because of their simplicity, and with dense grammars their performance often equals or exceeds that of more complex parsing algorithms: if every possible chart cell will be filled with a non-trivial probability, a predictive parsing algorithm (such as the Earley algorithm) will have to instantiate every cell anyway.

We also focus on the construction of the "inside chart" here, i.e., $\mathrm{P}(A \Rightarrow^+ w_i, \ldots, w_{j-1})$ for each nonterminal A and $0 \leq i < j < n$, where n is the number of words in the input string. Constructing the inside chart is the crucial $O(n^3)$ step of the Inside-Outside algorithm for estimating PCFGs (Charniak, 1993), and this computation is typically the rate-limiting computation in PCFG sampling algorithms (Johnson et al., 2007) as well. By replacing a sum with a max, the same algorithms can be used to constuct the Viterbi chart, from which a most probable parse tree can be extracted in $O(n^2)$ time, so again the Inside computation is the rate-limiting step. Because our grammar is dense we used pre-allocated fixed-sized arrays to hold the grammar rules and the inside chart, thus minimising expensive memory management and pointer arithmetic (in our experience unless great care is taken while coding, these costs can dominate parsing time).

Figure 1 presents pseudo-code for the baseline CKY parsing algorithm. The main part of the algorithm consists of six nested loops. All these loops

except the outermost (over the *gap* variable) can be freely reordered without affecting correctness. We experimented with a large number of reorderings of these variables; in preliminary experiments we found that the order presented here resulted in fastest parsing.[2]

Dunlop et al. (2010) point out that the algorithm in Figure 1 requires a grammar rule retrieval for each mid-point *j* of each (i, k) span (as well as each combination of nonterminals *a, b* and *c*), and show how to reduce this by factoring the algorithm as shown in Figure 2. This changes the "grammar constant" as mentioned above. They point out that this also improves the cache efficiency on modern CPUs. As we experimentally confirm below, the improvement that factoring brings can be dramatic.

4 Multi-core SMP parallelism using OpenMP

It is straight-forward to parallelise both the baseline and factored algorithms for multi-core SMP machines using OpenMP (Chapman et al., 2007). OpenMP programs are C++ programs with pragmas that indicate which loops should be parallelised. We experimented with several alternative reorderings of the loops and using an optimised matrix-algebra package (Guennebaud et al., 2010), but these did not improve parsing speed.

Developing OpenMP versions of the baseline and factored CKY algorithms is relatively straightforward. The main technical challenges in parallelising the CKY algorithm are synchronising the parallel threads and ensuring that different parallel threads do not interfere with each other. This is achieved by using synchronisation constructs with implicit barriers, *thread-private* temporary variables and constructs that ensure that updates to shared variables occur as *atomic* operations.

For the baseline CKY algorithm we constructed three parallel variants by parallelising (i) the outermost two for loops (over the *i* and *a* variables) using the OpenMP *parallel for* construct, (ii) the inner three loops (over *j*, *b* and *c*) using a *parallel for reduction* into a temporary variable, and (iii) a variant in which all loops (except the one involving the *gap* variable) are parallelised.

For the factored CKY algorithm we constructed three parallel variants by parallelising (i) the outermost for loop (involving the *i* variable), (ii) the innermost variables (involving the *j*, *b*, *c* and *a* variables), and (iii) a variant in which all loops (except the one involving the *gap* variable) are parallelised. Multiple *thread-private* instances of the *BC* variable are required when the outermost loops are parallelised, and we used the OpenMP *atomic* construct to synchronise updates to *BC* when the innermost loops were parallelised.

5 A CUDA GPU kernel for PCFG parsing

We experimented with several approaches to GPU parsing based on standard GPU matrix algebra packages (NVIDIA Corporation, 2010) but results were extremely disappointing; the resulting code ran orders of magnitude slower than the baseline CPU-based parser above. In order to obtain results competitive with the multi-core SMP algorithms described above we developed custom GPU programs. Our GPU subroutines or *kernels* were written in CUDA, which is a C++ dialect for specifying programs consisting of CPU code and GPU kernels (Sanders and Kandrot, 2011).

We focused on developing a CUDA implementation of the factored CKY algorithm here. CUDA programming is considerably more complicated than OpenMP programming, and we don't claim to have produced an optimal program here; additional experimentation could yield further speed improvements.[3]

A straight-forward translation into CUDA kernels of the baseline and factored algorithms above produced disappointing results: it ran approximately *200 times slower* than the factored CKY parser described above. A quick survey of the CUDA devel-

[2]These reorderings do not affect the theoretical complexity of the CFG parsing algorithm; it is still $O(n^3)$, where n is the length of the sentence being parsed. However, the loop ordering may affect the opportunities for SIMD optimisation and memory cache efficiency, since reordering the loops affects locality of memory access.

[3]We also experimented with CUBLAS, a CUDA implementation of BLAS (Basic Linear Algebra Subprograms), which we found yielded performance one to two orders of magnitude slower than our custom CUDA kernels. However, a new version of CUBLAS was released after this paper was submitted; this new version has several technical improvements that may enable it to be effective for PCFG parsing.

oper message boards showed that direct translations of CPU-based programs often perform poorly, and for good performance one needs to redesign the algorithms to take advantage of the specialised GPU hardware.

Computation on NVIDIA GPUs is organised into *blocks* of up to 1,024 parallel threads. A single CUDA *launch* starts up to hundreds of thousands of blocks; modern GPUs can execute several hundred thread blocks in parallel (the remainder are queued). Just as with SMP programming, the chief technical challenges in CUDA programming are synchronising the parallel threads and ensuring that different parallel threads do not interfer with each other. CUDA programming is more difficult than SMP programming because each individual GPU processor is much less capable than a CPU (CUDA programming is done using a restricted subset of C++), and data access must follow a very tightly prescribed set of rules if it is to perform reasonably efficiently.

Unlike on a regular CPU, the memory on a GPU has a complex organisation which the CUDA programmer must be aware of; the following sketch omits many details. *Global memory* is comparatively slow but accessible to all threads of all blocks; it is used to store persistent information and communicate between threads in different blocks; we store the inside chart C in global memory. *Texture memory* is a kind of global memory that permits more efficient cached read-only access; we stored the grammar rules R and the terminal probabilities T in texture memory. *Shared memory* is local to and accessible to all threads in the same block and is much faster than global memory; we stored (a local copy of) the *BC* array in shared memory. In addition, we also use *thread-local variables* to maintain local state and accumulate intermediate results within a single thread.

Our CUDA kernels consist of over 500 lines of code, so we only sketch them here. Our central data structures are the chart C, the rule probabilities R and the lexical probabilities T. Our CUDA implementation starts by launching a kernel that copies the terminal probabilities T for each of the words W into the chart C; this is easily and completely parallelised, and takes very little time.

Then it computes the chart one diagonal at a time in parallel. It launches one or more kernels for each value of *gap* in $2, \ldots, n$. If $n - \mathit{gap}$ is small enough then all of the chart entries $C[i, k, \cdot]$ (where $k = i + \mathit{gap}$) can be computed by a single thread block, and only one kernel launch is required. But for larger values of *gap* we decompose the computation into multiple thread blocks based on the mid string position j and store intermediate results in global memory; a second kernel launch is used to reduce these into the chart entries $C[i, k, \cdot]$.

A major goal in designing the CUDA kernel was to perform the sum

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

in the factored CKY algorithm as efficiently as possible. In order to achieve this we first copy all of the relevant chart entries $C[i, j, \cdot]$ and $C[j, k, \cdot]$ from global memory into the faster shared memory (this can be done in parallel), and then accumulate the results into BC, which is also stored in shared memory. This step can also be done completely in parallel.

Finally, we compute the chart entries $C[i, k, a]$ for all $k = i + \mathit{gap}$ and all a in parallel. If $n - \mathit{gap}$ is small enough that the computation can be done in one thread group then this is done only using shared memory, otherwise temporary results are stored in global memory so they are visible to other thread blocks. The reduction

$$C[i,k,a] \mathrel{+}= R[a,b,c]*BC[b,c]$$

in the factored CKY algorithm is also tricky, as it requires a double sum over b and c. In order to do this we generalised the parallel tree-based reduction algorithm presented in Harris (2010) to compute all of the chart entries $C[i, k, \cdot]$ as a parallel reduction.

6 Evaluation on a dense PCFG

We experimented with a number of different grammars, but because the results were generally similiar, we only describe one experiment here. The strings we parsed consist of the yields of the 1,345 trees in section 24 of the Penn WSJ treebank. Any word that did not appear 5 or more times in sections 2–21 was replaced with ⋆UNC⋆. We constructed a dense PCFG with 32 non-terminals (i.e., 32,768 binary rules) and random rule probabilities, which might be typical of the initial grammar in an unsupervised

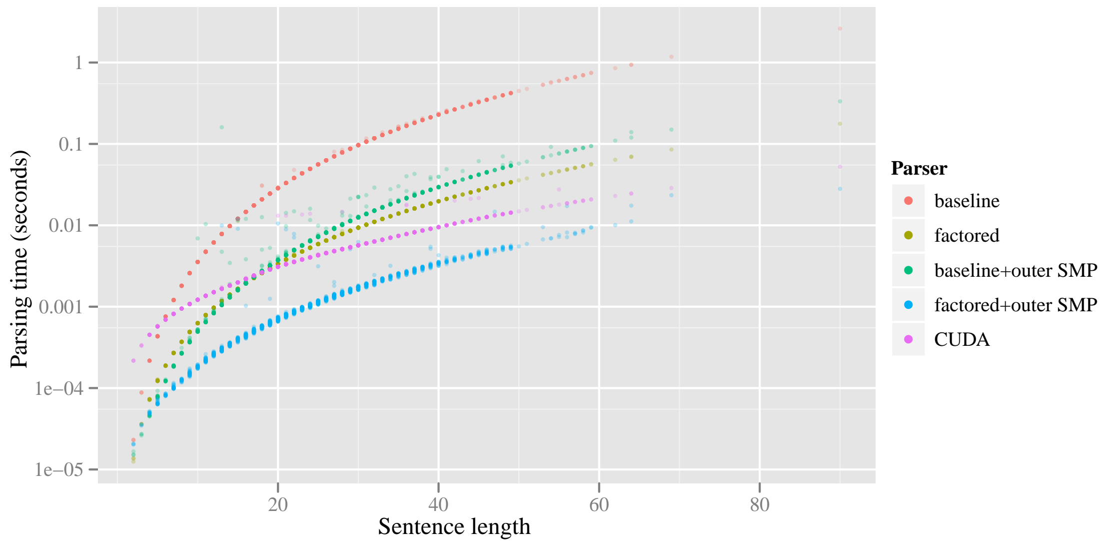

Figure 3: Parsing times as a function of sentence length on 1,345 sentences from section 24 of the Penn WSJ treebank for the baseline CYK parser, the baseline parser with SMP parallelism (outer loops parallelised), the factored CKY parser, the factored CKY parser (outer loops parallelised), and the CUDA implementation of the factored CKY parser.

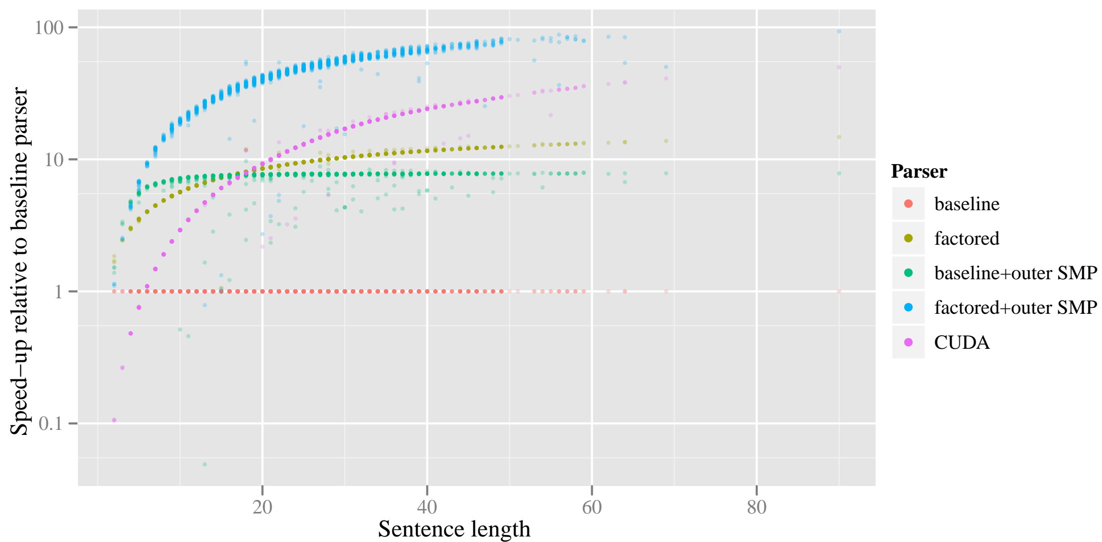

Figure 4: Speed-up relative to the baseline CKY parser as a function of sentence length on 1,345 sentences from section 24 of the Penn WSJ treebank for the baseline parser with SMP parallelism (outer loops parallelised), the factored CKY parser, the factored CKY parser (outer loops parallelised), and the CUDA implementation of the factored CKY parser.

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

Table 1: 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.

PCFG induction system using the Inside-Outside algorithm or a Metropolis-Hastings sampler. We used single-precision floating-point arithmetic in all experiments, and multiplied the terminal rule probabilities $A \to w$ by 10^4 to avoid underflow.

We ran our experiments on a single node of an SGI Altix XE 320 cluster with two quad-core 3.0GHz Intel Harpertown CPUs, a 1600MHz front side bus, 16GB DDR2-800 memory and two NVIDIA Fermi s2050 GPUs, each with 448 CUDA cores running at 1.15GHz (we only used one GPU here). We used the CUDA 3.2 toolkit and gcc 4.4.4. We selected compiler flags that enabled full optimisation, including enabling SSE3 SIMD floating-point vector subsystem, as prior experiments showed that this significantly speeds all calculations.

Table 1 presents the results of our experiments. We repeated all of our experiments twice in succession and report the time of the second run here; however, run-times varied by less than 1% between the two runs. Figures 3 and 4 depict parsing time and speed-up as a function of sentence length respectively (in order to avoid overloading the graphs, they only show a subset of the results). It's important to recognise that even our baseline parser is very fast (averaging 11 sentences/second), and both our SMP and GPU implementations were significantly faster.

7 Conclusions

We obtained large speedups over an already very fast baseline parser using both multi-core SMP and CUDA parallelism. Parallelising the outer loops in the multi-core SMP algorithms seems to be extremely effective; we see speed-ups close to the theoretical maximum of 8 times for both the baseline and factored algorithms. Parallelising the inner loops is devastating to performance, perhaps because it interfers with cache optimisation and SSE3 SIMD vectorisation (turning off the SSE SIMD vectorisation in these cases did not improve performace). The factored algorithm with parallelised outer loops performed fastest in our experiments, but the CUDA implementation was next best, parsing faster than all of the parallelised baseline algorithms.

As Figure 4 makes clear, the speed-up obtained by both the CUDA and factored algorithm with parallelised outer loops relative to the baseline increases with sentence length (with the CUDA speed-up increasining fastest), which suggests that parallelisation helps most where it is most needed, i.e., on longer sentences.

It is surprising that the CUDA implementation did not outperform the best SMP implementation. Perhaps this is because our SMP implementation uses highly-optimised, OpenMP/SSE3-parallelised code and can exploit the powerful Xeon CPUs. It is also possible that our dense PCFG parsing task is "too easy" to take full advantage of the power of the GPU; the entire corpus of 1,345 sentences took just a few seconds to parse, and it's possible that initialisation and data-transfer from the host machine to the GPU imposed a significant overhead. It would be interesting to repeat the experiments described here with a grammar that is one or two orders of magnitude larger.

In fact, as Figures 3 and 4 make clear, the CUDA implementation is comparatively slow on short sentences; for sentences of length 5 or less, the CUDA implementation is slower than even the baseline parser, which is consistent with the hypothesis that initialisation and data-transfer are imposing significant performance costs. It would be interesting to repeat these experiments on a larger corpus with larger and perhaps sparser grammars. It also might be more efficient to parse more than one sentence in parallel on a single GPU, which might keep more of the CUDA cores busy more of the time, although we did not try this here.

There are several lessons to draw from these results. First, parallelisation does not always produce

speed-ups; indeed parallelising the inner loops did not improve performance on either the baseline or factored algorithms. Second, parsing algorithms that perform well on conventional CPUs may need considerable redesign in order to produce good results on GPUs. Third, as the impressive performance of the factored algorithm shows, good algorithm design is of crucial importance.

Finally, this is an area where both the hardware and software are still rapidly improving. The number of cores in a single multi-core processor is likely to increase rapidly; already it is possible to obtain commodity machines with 24 cores. The improvement in GPU technology is if anything even more dramatic: as well as increasing the number of processors, new GPUs are equipped with more flexible buses that permit more complex kinds of data parallelism and ease programming. On the software side, up-coming versions of OpenMP will permit a greater range of efficient reduction constructs, which may permit us to avoid using the relatively expensive *atomic* synchronisation primitive. For GPUs, upcoming versions of CUDA will provide a variety of parallel programming libraries (including for sparse matrix algebra), which may make it easier to write considerably more efficient parallel parsing algorithms. Thus it is reasonable to expect a dramatic improvement in parallel parsing in the near future.

Acknowledgments

I'd like to thank the reviewers for their thoughtful comments and suggestions. This work was supported was supported under the Australian Research Councils Discovery Projects funding scheme (project number DP110102593).

References

Alfred V. Aho and Jeffery D. Ullman. 1972. *The Theory of Parsing, Translation and Compiling; Volume 1: Parsing*. Prentice-Hall, Englewood Cliffs, New Jersey.

Jacir Bordim, Yasuaki Ito, and Koji Nakano. 2002. Accelerating the CKY parsing algorithm using FPGAs. In Sartaj Sahni, Viktor Prasanna, and Uday Shukla, editors, *High Performance Computing HiPC 2002*, volume 2552 of *Lecture Notes in Computer Science*, pages 41–51. Springer Berlin / Heidelberg.

Barbara Chapman, Gabriele Jost, and Ruud van der Pas. 2007. *Using OpenMP: Portable Shared Memory Parallel Programming*. The MIT Press, Cambridge, Massachusetts.

Eugene Charniak. 1993. *Statistical Language Learning*. The MIT Press, Cambridge, Massachusetts.

Aaron Dunlop, Nathan Bodenstab, and Brian Roark. 2010. Reducing the grammar constant: an analysis of CKY parsing efficiency. Technical Report CSLU-2010-02, Oregon Health and Science University.

William Gropp, Ewing Lusk, and Anthony Skjellum. 1999. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. The MIT Press, Cambridge, Massachusetts.

Gaël Guennebaud, Benoît Jacob, et al. 2010. Eigen v3.0beta2. http://eigen.tuxfamily.org.

Mark Harris. 2010. Optimizing parallel reduction in CUDA. Technical report, NVIDIA Corporation.

Jane C. Hill and Andrew Wayne. 1991. A CYK approach to parsing in parallel: a case study. In *Proceedings of the twenty-second SIGCSE technical symposium on Computer science education*, SIGCSE '91, pages 240–245, New York, NY, USA. ACM.

Mark Johnson, Thomas Griffiths, and Sharon Goldwater. 2007. Bayesian inference for PCFGs via Markov chain Monte Carlo. In *Human Language Technologies 2007: The Conference of the North American Chapter of the Association for Computational Linguistics; Proceedings of the Main Conference*, pages 139–146, Rochester, New York, April. Association for Computational Linguistics.

Jimmy Lin and Chris Dyer. 2010. *Data-Intensive Text Processing with MapReduce*. Morgan and Claypool.

Anton Nijholt. 1994. Parallel approaches to context-free language parsing. In Geert Adriaens and Udo Hahn, editors, *Parallel Natural Language Processing*, pages 135–167. Ablex Publishing Corporation.

Takashi Ninomiya, Kentaro Torisawa, Kenjiro Taura, and Jun'ichi Tsujii. 1997. A parallel CKY parsing algorithm on large-scale distributed-memory parallel machines. In *The Proceedings of the Pacific Association for Computational Linguistics (PACLING 97)*, pages

223–231, Tokyo. Department of Informatics, Meisei University.

NVIDIA Corporation, 2010. *CUDA CUBLAS Library*, PG-05326-032 v02 edition.

Jason Sanders and Edward Kandrot. 2011. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley, Upper Saddle River, New Jersey.

Greg Sandstrom. 2004. A parallel extension of Earleys parsing algorithm. Technical report, Earlham College.

Henry Thompson. 1994. Parallel parsers for context-free grammars: Two actual implementations compared. In Geert Adriaens and Udo Hahn, editors, *Parallel Natural Language Processing*, pages 168–187. Ablex Publishing Corporation.