Parallelizing the CKY and Earley Parsing Algorithms
Silas Boyd-Wickizer, Austin Clements, Neha Narula
May 18, 2009

1 INTRODUCTION

Context-free parsing algorithms are one of the oldest and most well-understood aspects of natural language processing. Efforts to reduce the time complexity of these algorithms have produced two particularly popular algorithms: the Cocke-Kasami-Younger (CKY) bottom-up parsing algorithm [5, 9], and the Earley top-down parsing algorithm [2, 3]. However, despite these efforts, parsing remains a time-consuming process because typical natural language grammars are very large and human language tends to produce highly ambiguous sentences with many possible parses, even for seemingly straightforward sentences.

While ambiguity is the bane of parsing performance for these algorithms, it represents a perfect opportunity to take advantage of recent developments in multicore hardware. As of this writing, many general-purpose 16 processor machines exist and the number of processors is rapidly increasing. In order to take advantage of such hardware, however, algorithms must be redesigned to divide work into largely independent parts.

We examine both the CKY algorithm and the Earley algorithm in the context of modern multi-processor hardware and modify both algorithms to take advantage of the parallelism available with such machines. We demonstrate the CKY is highly amenable to parallelization because the dependencies between operations are very sparse and well-defined. Earley proved more difficult to parallelize; in literature some parallelized Earley by making the entries in the Earley chart independent, and predicting all possible rules per entry instead of working only with a subset of states from the entry that came before [6]. We chose to instead maintain the top-down nature of Earley for contrast with CKY.

Intuitively, parallelism helps most when there are many subproblems that can be computed independently. Larger grammars tend to introduce greater ambiguity,

2 BACKGROUND

There is a large body of work on parallelizing algorithms to achieve better performance in multi-processor settings. In this section, we introduce important aspects of parallel algorithms, briefly overview their application to parsing, and describe the parallel programming model that we use in our implementation and throughout the rest of the paper.

2.1 Measuring Parallelism

When parallelizing any algorithm, it is important to consider the algorithm’s work and span. Work is the total amount of computation performed by the algorithm, across all processors (or, equivalently, the total time taken when run on one processor). Span is a theoretical measure of the fastest time an algorithm could execute given an infinite number of processors. An algorithm’s span is limited by its critical path, the longest sequence of computations that depend on each other’s results. The ratio of an algorithm’s work to its span limits its speed-up, or the performance it can achieve relative to its single processor performance. An ideal parallel algorithm achieves linear speed-up; that is, given \( P \) processors, it will execute \( P \) times faster than it would on one processor.

Many things can limit parallelism to sub-linear speed-up. For example, when one computation depends on the results of another, the two cannot execute in parallel, and they have the potential to become part of the critical path, increasing the algorithm’s span. Even lock-free synchronization between computations, which does not permit computations to block and wait for the completion of other computations, can introduce severe limits to scalability because it requires communication.

2.2 Parallelizing Parsing

We detail our methods of parallelizing CKY and Earley in Section 3. Both CKY and Earley are instances of dynamic programming algorithms and both divide the parsing problem into smaller subproblems that can be solved separately. This structure makes them amenable to parallelization because the separate subproblems can often execute in parallel.

CKY parallelizes very well because the dependencies between operations are very sparse and well-defined. Earley proved more difficult to parallelize; in literature some parallelized Earley by making the entries in the Earley chart independent, and predicting all possible rules per entry instead of working only with a subset of states from the entry that came before [6]. We chose to instead maintain the top-down nature of Earley for contrast with CKY.

Intuitively, parallelism helps most when there are many subproblems that can be computed independently. Larger grammars tend to introduce greater ambiguity,
and thus larger numbers of independent subproblems. Likewise, longer sentences generally produce more ambiguous parses and thus larger numbers of independent subproblems. Both factors allow the parsing algorithms to take advantage of greater parallelism: indeed, the relative performance of parallel parsing over sequential parsing improves as grammar size or sentence lengths increase.

2.3 The Cilk Language
We developed our parsers using the Cilk parallel programming model and language [1, 4]. This Cilk language is based on C, but eases the burden on the developer for programming large, highly asynchronous parallel programs.

The Cilk model adds two important operations to the C language: spawn and sync. When a function call is preceded by the Cilk keyword spawn, that function call executes in a separate thread, allowing the caller to continue executing. The return values of spawned children are not available until the calling function executes a Cilk sync statement, which causes the caller to block until all spawned children have completed.

Notably, Cilk does not provide special mechanisms for data synchronization. Developers are expected to use regular locks and atomic operation to prevent concurrent modification to shared data. However, even this simple spawn-sync interface significantly eases low-level parallel programming.

While regular operating system-level threads are extremely high overhead, Cilk is designed for expressing fine-grained parallelism and, thus, its spawn operation is only a few times slower than a regular C function call. The Cilk runtime system implements scheduling of these spawned functions to processors and guarantees that no processors are idle when functions are waiting to execute. The scheduler implements a low-overhead work-stealing protocol, which allows processors with no outstanding work to migrate pending computations away from busy processors. This work-stealing scheduler allows Cilk to efficiently execute algorithms with fine-grained parallelism.

3 Parsing Algorithms
In this section, we present the details of our parallel CKY and Earley algorithms. We begin with an overview of regular, sequential CKY parsing and incrementally develop it into our full parallel version. We then explore the limited exploitable parallelism in the Earley algorithm.

3.1 CKY Parsing
The CKY algorithm parses sentences by constructing an $n \times n$ chart, where each cell $(i, j)$ corresponds to the sentence fragment spanning from word $i$ to word $n_j$. It fills each cell with the parse trees for its corresponding fragment, a process that culminates in a set of parse trees that span the entire sentence. The insight behind the algorithm is that the parse trees in each cell are binary combinations of the parse trees for each way of bisecting the cell’s sentence fragment on word boundaries.

At the core of CKY, the algorithm for computing the contents of a single cell consists of three nested loops, as shown in the pseudocode in Figure 1. The outer-most loop bisects the sentence fragment spanned by the new cell on each word boundary, examining each pair $(l, r)$ of sentence fragments that can be concatenated to form the new cell’s sentence fragment. Each of these sentence fragment pairs corresponds to a pair of neighboring cells as shown in Figure 2: the parses of $l$ are found in a cell to the left of the new cell and the parses of $r$ are found in a cell below it. We refer to this outer loop as a *dot product* because it combines these neighboring cells in a pair-wise fashion, collecting parse trees that can be constructed as binary combinations of parse trees from some pair of cells. The inner two loops of CKY perform a *join*...
operation between each pair of neighboring cells, pair- 
ing up each possible parse of \( l \) with each possible parse of \( r \) and constructing new parse trees out of any pair of 
parse trees that matches the right hand side of any gram- 
mar rule. For example, if \( l \) can be parsed as an NP or a 
JJ and \( r \) can be parsed as a VP or a PP, the algorithm 
will try each combination,

\[
\begin{array}{c|c|c|c}
\text{NP} & \text{S} \rightarrow \text{NP VP} & \text{NP} \rightarrow \text{NP PP} & \text{NPPRD} \rightarrow \text{NP PP} \\
\text{JJ} & \text{VP} & \text{ADJP} \rightarrow \text{JJ PP} & \text{PP}
\end{array}
\]

yielding parse trees for \( S, NP, NPPRD, \) and ADJP in 
the new cell.

The traversal order of the CKY chart itself is under- 
constrained, a fact we exploit when introducing paral- 
lelism later. Any traversal where a cell is computed only 
after all cells to its left and all cells below it are com- 
pleted will work.

The pair-wise approach of the CKY algorithm lim- 
its it to binary context-free grammars. Fortunately, any 
gramar can be reduced to a binary grammar, for ex- 
ample, using Chomsky normal form [7]. However, we 
avoid full Chomsky normal form, as the transformation 
looses information about unary rules that makes it dif- 
ficult to rewrite parse trees produced by CKY in terms 
of the original grammar. Instead, we modified the CKY 
algorithm to support unary rules directly using a unit ex- 
pansion process. Whenever the algorithm adds a parse 
tree to a cell, we expand its non-terminal by also adding 
other parses that can be produced by unary rules from the 
original’s non-terminal. For example, if \( VP \rightarrow VB \) and 
the algorithm adds a VB parse to a cell, we also add a 
VP parse to the cell if an equivalent parse is not already 
present. This process repeats on the newly added parses 
until no more new expansions can be added.

Because so many of the computations involved in 
CKY parsing are independent of each other, CKY is rife 
with opportunities for parallel computation. We begin 
with a simple approach where independent cells are com- 
puted in parallel and build up to our full approach that 
achieves a high degree of parallelism throughout the enti- 
tire parsing process. However, while many of the com- 
putations performed by CKY are independent, the chart 
data structure itself is shared. For all modifications made 

to the chart, we either prove that they cannot interfere, or 
use lock-free algorithms to synchronize modifications.

3.1.1 Cell-level Parallelism

We can parallelize the computation of cell contents us- 
ing a simple wavefront algorithm where we compute all 
of the cells on the first diagonal in parallel, wait for these 
to complete, then compute the cells on the second di- 
agonal in parallel, and so forth. None of the cells on a 
diagonal contribute to the contents of any other cells on 
the same diagonal, so this traversal guarantees that all 
cells will be complete before they are consumed, avoid- 
ing the need for any synchronization when reading cell 
contents. This property is crucial to achieving good scal- 
ability and we maintain it as we introduce more and more 
parallelism. Unfortunately, this naive approach achieves 
almost no speed-up from parallelism. As we progress 
up the diagonals, we can compute fewer and fewer cells 
in parallel. Furthermore, because the cells on a partic- 
ular diagonal will complete in different lengths of time, 
threads are frequently left idle even when there is other 
work that could be done.

Thus, rather than hard-coding a particular cell traversal 
order, we directly track cell dependencies to dynam- 
ically traverse the chart, beginning a new cell com- 
putation as soon as all cells it depends on complete. The 
structure of this dependency graph, shown in Figure 3a, 
mirrors the constraints on the serial chart traversal order 
described in Section 3.1 because each cell depends on all 
of the cells to its left and all of the cells below it. These 
dependencies guarantee that no synchronization is ever 
necessary for reading the contents of other cells because 
every cell will have completed fully before being read.

Many of these dependencies are redundant, however. 
For example, the top right cell in Figure 3a depends on 
all three cells to its left. However, simply depending on 
the single cell to its immediate left suffices because this 
cell, in turn, depends on the other two. If we take the 
transitive reduction of the dependency graph, we arrive 
at a much simpler, equivalent graph where each cell de- 
pends only on the cell immediately to its left and the cell 
immediately below, as shown in Figure 3b.

In traditional concurrent programs, threads block on 
other computations, waiting for them to complete be- 
fore using their results. However, because this pull-based 
model does not scale well to large numbers of short-lived 
threads, we instead use a push-based model. Cells do not 
wait on other cells to complete. Instead, when a cell com- 
pletes, it check if it is the last dependency of any other 
cells and, if so, spawns the computations of those cells. 
The simple structure of the dependency graph lends itself 
well to this approach. We keep one counter per cell that 
tracks the number of completed dependencies. When 
a cell completes, it atomically increments the counters 
of the cell above it and to its right. If either counter 
reaches 2, it spawns that cell’s computation.

This approach uses minimal synchronization to move 
the wavefront of cell computation across the chart dy- 
namically, taking advantage of computational resources 
whenever any cell can be computed.
3.1.2 *Intra-Cell Parallelism*

Even with a dynamic wavefront, the parallelism of a cell-level approach is severely limited by the size of the chart and, especially, the length of the chart’s diagonal. Thus, in order to achieve greater parallelism, we must divide the work done within each cell.

Fortunately, all of the operations performed by the inner-most loop of \texttt{FILLCELL} (from Figure 1) are independent of each other and, thus, can be executed in parallel. Unfortunately, the overhead of spawning these computations relative to the actual work performed by them trumps any gains from increased parallelism.

We take an intermediate approach: we always spawn separate computations for each pair of neighboring cells (each term of the dot product), but we process the pairings of parse trees from neighboring cells in larger batches. The join operation can easily be divided up into essentially equal work units by flattening the two inner loops and spawning a separate computation for the first $b$ pairings, another for the next $b$ pairings, and so on.

Like cell-level parallelism, this approach still never requires synchronization for reading another cell’s contents because all source cells will be completed before we begin computing a new cell. Unlike cell-level parallelism, where each cell was filled by a single thread and thus no synchronization was required to add new contents to a cell, with the introduction of intra-cell parallelism, multiple threads may fill the same cell at the same time, so writing to cells now requires synchronization. The simplest solution to this, which we improve upon below, is for each cell to maintain an atomic counter that records the index of the next free slot in the cell’s list of entries.

### Unit Expansion

With cell-level parallelism, unit expansion could be performed without any need to avoid races between the addition of duplicate parse trees because unit expansion occurs on a per-cell basis. Under intra-cell parallelism, we must revisit this in order to prove that no cross-thread synchronization is necessary. Two parse trees are equivalent if and only if they have the same non-terminal at their root and point to the same two child parse trees from neighboring cells. Because each thread (indeed, each individual parse tree pairing operation) considers non-overlapping pairs of child parse trees, there is no danger of two threads generating identical unit expansions. Therefore, no synchronization of unit expansion is necessary under intra-cell parallelism.

### Sub-Allocation

While maintaining a single atomically updated index for each cell’s next free entry is a simple and effective way to synchronize concurrent modifications to the same cell, because cell modifications occur constantly and each individual modification takes so little time, even this shared variable introduces severe scaling
limitations. Thus, we use a sub-allocation scheme to reduce contention on these index variables. Rather than allocating individual entry slots from a cell, worker threads allocate slot groups, each consisting of, e.g., 16 slots, and maintain per-thread information about the current slot group and the next free slot within that group. This introduces minor fragmentation into each cell’s entry list because slots may be left empty when a cell completes, but the reduced contention more than compensates for the penalty of fragmentation.

Cache-Aware Partitioning We can further improve the performance of the join operation by cache-aware partitioning of the contents of neighboring cells. On modern hardware, accessing cache memory is an order of magnitude faster than accessing main memory, so any workload whose memory working set fits in the cache will run significantly faster than a workload that does not fit in the cache. Multiprocessor systems amplify this effect: each individual CPU has local cache memory and, thus, by partitioning the working set well between CPU’s, a workload can take advantage of the larger aggregate cache size and even achieve super-linear speed-up as the number of CPU’s increases.

Rather than flatten the two inner loops of the CKY algorithm and process each b pairings separately, we divide the contents of each neighboring cell into bands, each containing $\sqrt{b}$ entries, as shown in Figure 4. We then spawn computations for each pair of bands and each computation, in turn, joins the entries from its respective bands. By setting the band size to fit in the cache, this minimizes the number of accesses to main memory. This simple partitioning optimization demonstrates nearly double the performance of the naive approach and allows our approach to achieve super-linear speed-up, as we discuss further in Section 4.2.2.

3.1.3 Fine-Grained Dependencies

Intra-cell parallelism creates enough fine-grained parallelism to keep processors busy while there are cells whose dependencies are complete. However, even its scalability is limited by the size of the chart. In particular, as the algorithm approaches the top-right corner of the chart, where there are fewer cells on each diagonal, it encounters a bottleneck where it must wait for all of the work in dependent cells to complete before spawning any of the work in a new cell.

We can improve this situation by tracking dependencies at a finer, sub-cell level. As shown in Figure 5, rather than spawning the computations to fill a cell only after all of its source cells are complete, we can instead track the dependencies of each individual join operation. Like before, once all of the join operations in a cell are complete, the cell is complete. However, once a cell is complete, instead of checking cells that depend on it, it checks all of the sub-cells that depend on it and can immediately spawn these join operations directly, even if other dependencies of that cell are not yet complete.

This reduces the critical path length from the start of the algorithm to the completion of the last join operation in the top-right cell by allowing individual processors to more eagerly progress up the chart, thus reducing the overall time required to complete the parse. Furthermore, fine-grained dependencies eliminate the bottleneck as the algorithm approaches diagonals with fewer cells. Indeed, if we ignore the amount of parallelism from within each join operation, which varies with the ambiguity of the sentence, this approach exploits the availability of at least

Figure 4: Intra-cell parallelism. Each cell is divided into bands and each pair of bands from each pair of neighboring cells is processed in parallel.

Figure 5: Sub-cell dependencies, showing each join operation within each cell. These join operations, in turn, spawn multiple computations to join the bands of the two source cells.
Adding states to entries, so, as we did for CKY, we use separate computations. This introduces contention for all prediction, scanning, and completion computations as shown in the pseudocode in Figure 3.2. First, we spawn only infinite recursion in some grammars, entries must keep try. In order to prevent not only wasted computation, but can generate duplicates of states already present in an entry currently being processed. Furthermore, these operations add states not only to the next entry, but also to the entry currently being processed. Consequently, these operations can add states not only to the next entry, but also to the entry currently being processed. The algorithm iterates over each entry in the chart, filling it completely before moving to the next entry. Within each entry, it iterates over each state in the entry, predicting when it encounters a non-terminal state, scanning when it encounters a terminal state, and completing when it encounters the end of a non-terminal state. While existing states are never modified, these operations can add states not only to the next entry, but also to the entry currently being processed. Furthermore, these operations can generate duplicates of states already present in an entry. In order to prevent not only wasted computation, but infinite recursion in some grammars, entries must keep only unique states.

We parallelize two aspects of the Earley algorithm, as shown in the pseudocode in Figure 3.2. First, we spawn all prediction, scanning, and completion computations as separate computations. This introduces contention for adding states to entries, so, as we did for CKY, we use shared state in each entry to track the next free state slot. Executing the completer in parallel is particularly important, as this operation must scan all existing states and is therefore expensive. Second, because states are only added to entries and never modified once added, we can safely parallelize the traversal of the states within an entry, with the caveat that, once we have traversed all of the states known to be in the entry, we must wait for all outstanding computations to complete and then process any additional states that were added to the current entry. This introduces a bottleneck similar to the one found towards the top-right of the CKY algorithm, where processors may sit idle before more work can be generated.

Duplicate detection presents a particularly difficult problem because it requires fine-grained synchronization on entry contents. Instead of performing complete duplicate detection, we use an approximate solution that reduces contention, but at the cost of a very slight probability of adding a duplicate state. Adding a state scans all existing states in an entry for duplicates without any synchronization and, if it finds none, it inserts the new state into the next free slot. However, there is a very short window of time between when this scan completes and when the new state is added where an undetected duplicate may appear. This does not affect the correctness of the algorithm, and the benefit of reduced synchronization outweighs the cost of processing a state twice later.

For comparison, we also parallelized the Earley parsing algorithm. While Earley typically outperforms CKY parsing because it generates fewer intermediate parse trees that do not contribute to the final parse tree, it is also much harder to parallelize because the operations performed by the algorithm are less independent than the operations performed by CKY. Thus, while Earley may outperform CKY running on a single processor, the limitations the algorithm places on parallelism allow parallel CKY to surpass it given enough processors.

The Earley algorithm operates by constructing a chart with $n+1$ entries, for an $n$-word sentence. It fills each entry with states representing a partial parse tree that has been generated thus far. Each partial subtree is represented only once.

The algorithm iterates over each entry in the chart, filling it completely before moving to the next entry. Within each entry, it iterates over each state in the entry, predicting when it encounters a non-terminal state, scanning when it encounters a terminal state, and completing when it encounters the end of a non-terminal state. While existing states are never modified, these operations can add states not only to the next entry, but also to the entry currently being processed. Furthermore, these operations can generate duplicates of states already present in an entry. In order to prevent not only wasted computation, but infinite recursion in some grammars, entries must keep only unique states.

We parallelize two aspects of the Earley algorithm, as shown in the pseudocode in Figure 3.2. First, we spawn all prediction, scanning, and completion computations as separate computations. This introduces contention for adding states to entries, so, as we did for CKY, we use shared state in each entry to track the next free state slot. Executing the completer in parallel is particularly important, as this operation must scan all existing states and is therefore expensive. Second, because states are only added to entries and never modified once added, we can safely parallelize the traversal of the states within an entry, with the caveat that, once we have traversed all of the states known to be in the entry, we must wait for all outstanding computations to complete and then process any additional states that were added to the current entry. This introduces a bottleneck similar to the one found towards the top-right of the CKY algorithm, where processors may sit idle before more work can be generated.

In Section 3.1, we introduced the concept of parallelism and showed how it can be used to improve the performance of the Earley algorithm. In this section, we describe the implementation details of our parallelized Earley parser.

4 RESULTS

We implemented the CKY and Earley algorithms in Cilk++ 4.2.4 (a commercialization of MIT Cilk-5 [8]), utilizing Cilk’s fine-grained work scheduling algorithm. We ran all experiments on an AMD 16-core system running Linux with 64 Gbytes of memory. We used the Penn Treeback Wall Street Journal grammar, as well as the WSJ grammar samplings from from 6,863 Laboratory 2.

For many of our experiments, we compare both the absolute times required by the parser implementations, as well as the their speed-up relative to single processor performance. Considering both is important, as the added complexity of parallelism can negatively impact the absolute performance of an algorithm, even if it improves its relative performance.

4.1 CKY Performance

The final CKY algorithm presented at the end of Section 3.1 achieves near-perfect scaling up to 16 cores, the number of cores available in our test system. However, parallelism allows the parser to scale better not only with the number of available processors, but also with the sentence length and grammar size. Increasing either of these factors increases the ambiguity of the final parse, which, in turn, increases the opportunities for parallelism.
4.2 CKY Parallelism

Section 3.1 introduced increasingly more advanced ways to partition CKY into parallel tasks. Here, we demonstrate that the additional complexity introduced by more advanced partitioning doesn’t overtake the respective parallelism gains.

First, we compare the performance of the three main partitioning variations we introduced earlier: cell-level parallelism, intra-cell parallelism, and intra-cell parallelism with fine-grained dependencies. Figure 9 summarizes the comparison of these parsers as we vary the number of cores (and thus the achievable parallelism).

As predicted, cell-level parallelism achieves some speed-up, but levels out at ~1.4x. Given that the cell-level algorithm can fill about $n/2$ cells concurrently, on average, one might expect better speed-up for a sentence of length $n = 10$. However, the sizes of cells are highly unbalanced—24% of the cells in the example sentence contain no entries at all, while some contain as many as 106,000—so the achievable parallelism is effectively much less. However, cell-level parallelism’s baseline single core performance is 19% better than intra-cell parallelism.
parallelism, because intra-cell parallelism requires synchronization for adding cell contents, which introduces contention between threads.

Intra-cell parallelism scales substantially better than cell-level parallelism. While cell-level parallelism spawns a total of only 56 computations for the sample sentence, intra-cell parallelism spawns 694, enough to keep many more processors active. However, even with this many threads, intra-cell parallelism fails to scale linearly because it reaches a bottleneck as it approaches the top-right corner of the chart.

Adding fine-grained dependencies to intra-cell parallelism brings the algorithm to near-linear scaling, achieving 15.2x speed-up at 16 processors. Even though this does not introduce any additional spawns over intra-cell parallelism with cell-level dependencies, the critical final boost comes from removing the bottleneck at the top-right of the chart.

4.2.1 Sub-Allocator Performance

Though the sub-allocator is a seemingly minor modification to the process of adding entries to cells, reducing contention on the shared index of the next free entry in each cell is critical for scalability. Figure 10 compares the performance of the full algorithm with and without the sub-allocator. While the sub-allocator introduces an 11% overall performance overhead, as indicated by the single core performance, it nearly doubles the performance and scalability at 16 processors.

4.2.2 Cache-Aware Partitioning Performance

Figure 10 also compares the performance of the full algorithm with and without cache-aware partitioning. With cache-aware partitioning, parse trees from neighboring cells are joined in fixed-size bands (we found bands of 100 entries to be optimal, though this will vary heavily with cache size and implementation details). Without cache-aware partitioning, the two join loops are flattened and the process broken up linearly. While the scalability of the algorithm with cache-awareness is not significantly better than without, there are two important differences. First, the absolute performance doubles over the entire range of processor count. This is a per-processor effect of reducing the number of expensive trips of main memory. Second, with cache-aware partitioning, the algorithm achieves super-linear scaling between two and twelve processors. As the algorithm runs on more processors, it has access to more aggregate cache. By partitioning the work so it takes advantage of its local cache, the algorithm as a whole can take advantage of this increased cache size, essentially increasing its effective memory bandwidth as the number of processors increases. We expect this trend would continue above twelve processors, but is impeded by other scalability limitations.

4.3 Earley Performance

Figure 11 compares the performance of a parallelized Earley algorithm with a non-parallelized Earley algorithm. Since we maintained Earley as a top-down parser, each entry in the chart is dependent upon the work done in the entry before it. This dependency means we could not fully parallelize the work for different entries. This speedup represents the gain that can be achieved by proceeding through the entries linearly, but parallelizing the work within an entry.

Figure 12 shows the performance of our Earley parser for sentences of various lengths. Again, we compare single processor performance against 8 processor and 16 processor performance. Here the performance for short sentences is different for a different number of processors, but as we increase the sentence length and thus the ambiguity, the gap between single processor performance and multi-processor performance does not widen as dramatically as with the CKY parser, only increasing from 3x to 6x. This is because as the number of words increases, the number of entries in the chart increases, and the Earley parser must process these sequentially.
Figure 11: The parse time and speedup of Earley parsing versus the number of processors for “The companies are very very likely to go under.”

Figure 12: Performance of Earley with various processor counts, parsing “John is very \{0,3\} happy .” using a 30% sample of the WSJ grammar.

Figure 13: Performance of Earley with various processor counts, parsing “John is very happy .” at various grammar sample sizes.

Figure 13 shows how Earley scales as grammar size increases. As for sentence length, the number of processors has a far greater impact on the speed-up of the algorithm than the size of the grammar.

5 CONCLUSION

Natural language parsing is a natural application for parallelization, though achieving linear speedup requires carefully understanding the algorithm and modifying it to create fine-grained units of work that can be solved independently, avoid points of synchronization, and size data to fit within a processor’s cache. Furthermore, while some algorithms may achieve higher performance in traditional, single processor settings, in the parallel realm, the scalability of an algorithm has the greatest impact on its performance.

As processing power grows with the number of cores on a chip, utilizing gains in computational power to better understand and process natural language will require rethinking existing single-threaded algorithms to take advantage of highly scalable multicore hardware architectures.

The full source code for our parallel parser implementations can be found at: http://pdos.csail.mit.edu/~amdragon/mcchart.tar.gz.

REFERENCES


