Parallel Parsing: The Earley and Packrat Algorithms

Seth Fowler and Joshua Paul

May 19, 2009

1 Introduction

Parsing plays a critical role in our modern computer infrastructure: scripting languages such as Python and JavaScript, layout languages such as HTML, CSS, and Postscript/PDF, and data exchange languages such as XML and JSON are all interpreted, and so require parsing. Moreover, by some estimates, the time spent parsing while producing a rendered page from HTML, CSS, and JavaScript is as much as 40%. Motivated by the importance of parsing, and also by the wide-spread adoption of chip multiprocessors, we are interested in investigating the potential for parallel parsing algorithms.

We begin with one classical and one more modern parsing algorithm, the Earley and the Packrat algorithm, respectively, and consider methods for parallelizing both. The general strategy is the same in both cases: break the string to be parsed into contiguous blocks, process each block in parallel, and finally re-join the blocks. We find that using this general framework, we are able to obtain a meaningful speedup relative to serial implementations of each algorithm. In particular, we obtain a speedup of 5.4 using the parallel Earley implementation on 8 processors, and a speedup of 2.5 using the parallel Packrat algorithm on 5 processors.

Another algorithmic property that we consider is *work efficiency*, a measure of the amount of extra work done by the parallel algorithm relative to the serial algorithm. Poor work efficiency indicates that the parallel algorithm is doing considerably more work than is required, and thereby using more energy. With the relative importance of energy increasing, even a parallel algorithm that achieves significant speedup may not be worthwhile if it exhibits poor work efficiency. Though it is difficult to formulate a precise definition of work efficiency in terms of program operations, we qualitatively demonstrate that our algorithms, with some caveats, exhibit reasonable work efficiency.

Related work: There is a substantial literature related to the hardness of parsing context free grammars (CFG), most related to the result of Valiant[8] that CFG parsing is equivalent to binary matrix multiplication. This equivalence provides, in theory, parallel algorithms for parsing, but these (to the authors' knowledge) have not been put into practice.

There are several published algorithms (see, for example, Hill and Wayne[6]) for parallelizing a variant of the the Earley algorithm, the CYK algorithm. Unfortunately, in practice CYK has a much longer running time than Earley (even though it has the same worst-case complexity of $O(n^3)$), and so it is not typically used.

For the Earley algorithm itself, there are very few parallelization methods published (though there are many optimizations – see, for example, Aycock and Horspool[1]). One such method by Chiang and Fu[3] uses a decomposition similar to the one we develop, but goes on develop the algorithm for use on a specialized VLSI. Similarly, Sandstrom[7] develops an algorithm based on a similar decomposition, but implements it using a message passing architecture. In both cases, the grain of the parallelization is much finer than what we propose, and there is no notion,

of speculation, a central component of our algorithm.

Finally, to the authors' knowledge, there are no published works on parallelizing the Packrat algorithm.

2 Background

Context-free grammars: A *formal grammar* is a set of formation rules that implicitly describe syntactically valid sequences of output symbols from some alphabet. A *context-free grammar* is a specific type of grammar, specified by:

- Σ, a set of *terminal* symbols.
- V, a set of *non-terminal* symbols.
- R, a set of production rules, each taking a single non-terminal to an arbitrary sequence of terminal and non-terminal symbols.
- S, the designated start non-terminal.

Using the production rules, it possible to construct a sequence of terminals, beginning with the start non-terminal. Such a sequence is said to be *derived* from the grammar, and the steps in the construction comprise a *derivation* of the sequence. The parsing problem can then be stated formally as follows: given a sequence of terminals, determine whether it can be derived by the grammar, and if so, produce one or more such derivations.

When referring to a context-free grammar, the following notation will be used hereafter: A, B will be used to denote non-terminals (e.g. $A, B \in V$); a, b will be used to denote terminals (e.g. $a, b \in \Sigma$); α, β, γ will be used to denote arbitrary (possibly empty) sequences of terminals and non-terminals (e.g. $\alpha, \beta, \gamma \in (V \cup \Sigma)^*$).

The Earley algorithm: Given as input a length-n sequence of terminals $T_n = x_1 x_2 \ldots x_n$, the Earley algorithm[4] constructs $n + 1$ *Earley sets*: an initial set S_0, and a set S_i associated with each input terminal x_i. Elements of these sets are *Earley items*, each of which is intuitively an 'in-progress' production rule of the grammar. The Earley sets are constructed in such a way that S_i contains all possible Earley items for terminals $x_1 x_2 \ldots x_i$; upon completion, S_n therefore contains all possible Earley items for the entire input sequence.

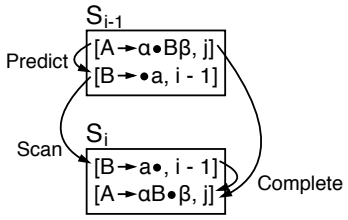

Figure 1: Illustration of the SCAN, PREDICT, and COMPLETE mechanisms, supposing that our grammar contains the rule $B \to a$ and that $x_i = a$.

Formally, an Earley item comprises a production rule, a position in the right-hand side of rule indicating how much of the rule has been seen, and an index to an earlier set indicating where the rule began (we frequently refer to this as the *origin* of the item). Earley items are written:

$$[A \to \alpha \bullet \beta, j],$$

where $\bullet$ denotes the position in the right-hand side of the rule and j indexes the Earley set S_j. Items may be added to the set S_i using the following three mechanisms:

SCAN : If $i > 0$, $[A \to \alpha \bullet a\beta, j]$ is in S_{i-1}, and $a = x_i$, add $[A \to \alpha a \bullet \beta, j]$ to S_i.

PREDICT : If $[A \to \alpha \bullet B\beta, j]$ is in S_i and $B \to \gamma$ is a production rule, add $[B \to \bullet\gamma, i]$ to S_i.

COMPLETE : If $[A \to \alpha\bullet, j]$ is in S_i and $[B \to \beta \bullet A\gamma, k]$ is in S_j, add $[B \to \beta A \bullet \gamma, k]$ to S_i.

See Figure 1 for an illustration of each mechanism, and observe that constructing Earley set S_i depends only on $\{S_j \mid j \leq i\}$. Thus, dynamic programming may be used, with Earley sets constructed in increasing order.

Recalling that S is the start non-terminal, Earley set S_0 is initialized to contain $[S \to \bullet\alpha, 0]$ for all rules $S \to \alpha$. Upon completion, the input sequence can be derived if and only if S_n contains $[S \to \alpha\bullet, 0]$ for some rule $S \to \alpha$; if the sequence can be derived, one or more derivations may be obtained by traversing

Algorithm 1 EARLEY(T_n)

$S_0 \leftarrow \{[S \rightarrow \bullet\alpha, 0] \mid S \rightarrow \alpha \in P\}$
for $i = 0$ to n **do**
  repeat
    apply SCAN, PREDICT, COMPLETE to S_i
  until nothing new added to S_i
end for

if $[S \rightarrow \alpha\bullet, 0] \in S_n$ for some $S \rightarrow \alpha$ **then**
  return true
else
  return false
end if

backwards through the Earley sets. See Algorithm 1 for pseudocode.

Parsing expression grammars: Though their roots lie in *Top-Down Parsing Language*[2], a formal grammar invented in the 1970s to characterize top-down recursive parsers, *parsing expression grammars* (PEGs) are a relatively recent group of grammars which first appear in the literature in 2002[5]. PEGs, as their heritage suggests, describe the process by which a parser may recognize that a string belongs to a language, rather than the process by which a string belonging to a language may be generated. They are defined by a 4-tuple of the same form as that used for CFGs, but the interpretation of the rules and the set of operators that may be used within a rule are different. 1 displays the operators available in a PEG, many of which resemble regular expression operators.

Operator	*Example*
Sequence	$[A \leftarrow BC]$
Ordered choice	$[A \leftarrow B/C]$
Zero or more	$[A \leftarrow B^*]$
One or more	$[A \leftarrow B+]$
Optional	$[A \leftarrow B?]$
Followed by	$[A \leftarrow \&B]$
Not followed by	$[A \leftarrow !B]$

Table 1: PEG operators

In a PEG, each rule is treated as a deterministic algorithm; for example, the simple rule $[A \leftarrow BC]$ instructs the parser that it may recognize an A by first recognizing a B and then, if that is successful, by recognizing a C. This property means that PEGs, unlike CFGs, cannot directly support left-recursion; $[A \leftarrow AB]$, which requires that a parser recognize an A by first recognizing an A, can never match successfully. In practice, this is not a problem, since left-recursive rules can always be rewritten to eliminate left-recursion[5].

The alternation operator |, used either explicitly or implicitly in CFGs when there are multiple ways to construct the same nonterminal, does not have a precise equivalent in PEGs. Instead, PEGs use the ordered choice operator /. From the point of view of the parser, the alternation operator has the effect that all of the alternatives are considered simultaneously. It is possible for more than one of the alternatives to match, in which case both possibilities must be explored further by the parser. In contrast, the ordered choice operator is interpreted in an algorithmic, "if-else" fashion; for example, $[A \leftarrow B/C]$ instructs the parser to recognize an A by first attempting to recognize a B and only if that fails to attempt to recognize a C. Because of the difference between these two operators, CFGs can express ambiguous grammars, while PEGs cannot, even if there is a need. PEGs are therefore better suited for machine-readable languages than natural languages.

Packrat parsing: Just as parsing expression grammars are a modern extension of TDPL, *packrat parsing* extends a theoretical linear-time algorithm for parsing TDPL which also dates from the 1970s[5]. Packrat parsing, like its ancestor, requires constructing a table in memory that grows in size linearly with the string to be parsed; this almost always requires more memory than popular parsing algorithms for LL(1) and LR(1) grammars, which only require memory proportional to the depth of the parse tree for a given input string. This memory requirement made this algorithm impractical for decades, but modern machines have enough memory to make packrat parsing realistic.

The *packrat table* is two-dimensional, with the rows representing nonterminals in the PEG and the columns representing positions in the input string. Each cell $[C_i, j]$ in the table stores the result of an attempt to recognize nonterminal $[i]$ at position $[j]$. If the nonterminal is matched successfully, the cell stores the first position after that match – for example, if the parser consumed two characters, starting at position $[j]$, in the process of matching nonterminal $[i]$, the cell will hold the value $[j + 2]$. If nonterminal $[i]$ is not matched successfully, the cell will hold a special value indicating that the recognition attempt failed.

The rule for matching nonterminal $[i]$ may refer to another nonterminal. If nonterminal $[u]$ is referenced at input string position $[v]$, the packrat parser proceeds by examining the value of cell $[C_u, v]$; it may either find a reference to a later position $[v + n]$, in which case it advances to that position and attempts to match the next part of the rule for $[i]$, or it may find the special failure value, which signals that nonterminal $[u]$ does not exist a position $[v]$. The packrat parser may also find that cell $[C_u, v]$ has not yet been evaluated; in this case, it recursively evaluates $[C_u, v]$ before completing its original computation. Each cell memoizes the result of a potentially expensive recursive computation; by ensuring that no cell is ever evaluated more than once, the packrat parser can evaluate any PEG in linear time with respect to the length of the input string.

To parse a string using the packrat algorithm, the parser begins by evaluating the cell corresponding to the first position in the string and the start symbol of the grammar. Evaluation of cells continues recursively until it is possible to determine whether the start symbol matched or not. The start symbol matches if and only if the parse was a success and the input string belongs to the language specified by the grammar. In the common case, only a fraction of the cells in the packrat table will have been evaluated at this point; the majority are never touched because they could not possibly contribute to matching the start symbol. In this way, the packrat algorithm uses lazy evaluation to perform a minimal amount of work despite the large size of its underlying data structure.

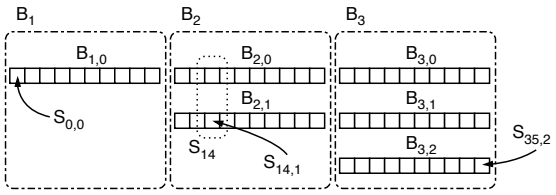

Figure 2: Illustration of the partition of the Earley sets $S_0, S_1, \ldots, S_{35}$ into the blocks B_1, B_2, B_3, and the associated decomposition into Earley subsets and sub-blocks.

3 Parallelizing the Earley and Packrat algorithms

3.1 Parallelizing Earley

We proceed towards a parallel Earley algorithm in two steps: first, we show that the operation of the serial Earley algorithm can be decomposed into sub-blocks that depend upon one another in a very regular way; second, we propose a method for removing a key set of dependencies among the sub-blocks, thereby allowing them to be computed in parallel.

Sub-block decomposition: Recall that the serial Earley algorithm associates an Earley set S_i with each terminal x_i. The first step in the decomposition is to partition the Earley sets into a sequence of m contiguous *blocks*, $B_1, B_2, \ldots, B_m$. An Earley item belonging to an Earley set in block B_i may have origin in any block B_j with $j \leq i$. Thus motivated, the second step in the decomposition is to partition each Earley set S_p in block B_i into *Earley subsets*, $S_{p,0}, S_{p,1}, \ldots S_{p,i-1}$ with:

$$S_{p,j} = \{e \in S_p \mid \text{block origin of } e = i - j\}.$$

Additionally define $B_{i,j} = \{S_{p,j} \mid S_p \in B_i\}$, and refer to this as a *sub-block* of block B_i. An Earley subset $S_{p,j}$ in sub-block $B_{i,j}$ contains only Earley items with an origin in block B_{i-j}. See Figure 2 for an illustration of the decomposition and terminology.

Having characterized the decomposition of Earley sets into sub-blocks, we now describe how to efficiently compute the Earley subsets therein. Suppose

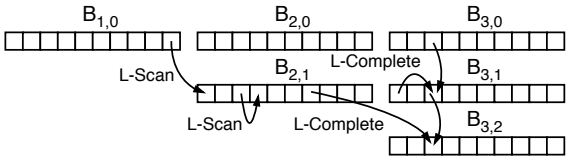

Figure 3: Illustration of the L-SCAN and L-COMPLETE mechanisms for several representative cases. Compare with the ordinary SCAN and COMPLETE mechanisms in Figure 1.

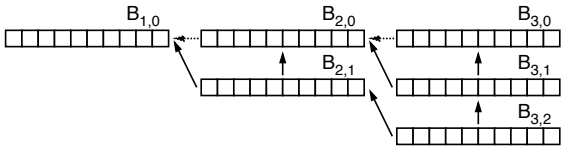

Figure 4: Computational dependencies for sub-blocks. The dotted lines are the dependencies associated with the sub-blocks $B_{i,0}$ under a naïve computation.

that we are interested in computing the Earley subset $S_{p,j}$ in sub-block $B_{i,j}$ (with $j > 0$). There are two mechanisms by which items may be added, both modifications of the mechanisms described for the ordinary Earley algorithm:

L-SCAN : Suppose $S_{p-1} \in B_i$: then if $[A \to \alpha \bullet a\beta, p']$ is in $S_{p-1,j}$ and $a = x_p$, add $[A \to \alpha a \bullet \beta, p']$ to $S_{p,j}$. Otherwise, $S_{p-1} \in B_{i-1}$: if $[A \to \alpha \bullet a\beta, p']$ is in $S_{p-1,j-1}$ and $a = x_p$, add $[A \to \alpha a \bullet \beta, p']$ to $S_{p,j}$.

L-COMPLETE : If $[A \to \alpha \bullet, p']$ is in $S_{p,k}$ with $k \leq j$, and $[B \to \beta \bullet A\gamma, p'']$ is in $S_{p',j-k}$ (in sub-block $B_{i-k,j-k}$), add $[B \to \beta A \bullet \gamma, p'']$ to $S_{p,j}$.

These mechanisms are restricted relative to the analogous mechanisms in the serial Earley algorithm since Earley subsets in sub-block $B_{i,j}$ contain only Earley items with origin in block B_{i-j}. In particular, the scan mechanism may depend only on $S_{p-1,j} \in B_{i,j}$ or $S_{p-1,j-1} \in B_{i-1,j-1}$ since the resulting item must have origin in block $B_{i-j} = B_{(i-1)-(j-1)}$. Similarly, if a completion is made with respect to an Earley item in block B_{i-k} (corresponding to the item in $S_{p,k} \in B_{i,k}$), it must be in sub-block $B_{i-k,j-k}$ so that the resulting item has origin in block $B_{(i-k)-(j-k)} = B_{i-j}$. Finally, there is no predict mechanism, since predict would create Earley items with origin p (and thus in block $B_i \neq B_{i-j}$).

See Figure 3 for an illustration of the mechanisms, and observe that, analogously to the ordinary Earley algorithm, the computation of sub-block $B_{i,j}$ (with $j > 0$) depends only on sub-blocks $\{B_{i',j'} \mid i' - j' \leq i - j\}$. We have thus far avoided discussion of sub-block $B_{i,0}$, as it does not naturally fit into the previous analysis. Indeed, straight-forward computation of the sub-block $B_{i,0}$ depends on sub-block $B_{i-1,0}$ since successfully applying the predict mechanism requires all Earley items, regardless of their origin. This leads to the sub-block dependence graph depicted in Figure 4.

Elimination of key dependencies: We now propose a method for computing the the Earley subset $S_{p,0}$ of sub-block $B_{i,0}$ without relying on the aforementioned dependencies. Intuitively, rather than explicitly depending upon previous sub-blocks, the method assumes that *every* Earley item in previous sub-blocks is present. Such 'speculative' Earley items are represented as ordinary Earley items, but with an unspecified origin:

$$[A \to \alpha \bullet \beta, -1].$$

There are three mechanisms by which items may be added to Earley subset $S_{p,0}$, both modifications of the mechanisms described for the ordinary Earley algorithm:

S-SCAN : Suppose $S_{p-1} \in B_i$: then if $[A \to \alpha \bullet a\beta, p']$ is in $S_{p-1,0}$ and $a = x_p$, add $[A \to \alpha a \bullet \beta, p']$ to $S_{p,0}$. Otherwise, $S_{p-1} \in B_{i-1}$: add speculative item $[A \to \alpha a \bullet \beta, -1]$ to $S_{p,0}$ for all rules $A \to \alpha a \beta$ with $a = x_p$.

S-PREDICT : If $[A \to \alpha \bullet B\beta, p']$ is in $S_{p,0}$ and $B \to \gamma$ is a rule in R, add $[B \to \bullet\gamma, p]$ to $S_{p,0}$.

S-COMPLETE : Suppose $[A \to \alpha \bullet, p']$ is in $S_{p,0}$. If $p' \geq 0$ and $[B \to \beta \bullet A\gamma, p'']$ is in $S_{p',0}$, add $[B \to \beta A \bullet \gamma, p'']$ to $S_{p,0}$. If $p' = -1$, add speculative item $[B \to \beta A \bullet \gamma, -1]$ to $S_{p,0}$ for all rules $B \to \beta A \gamma$.

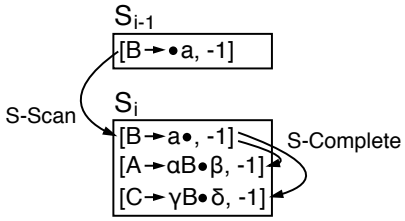

Figure 5: Illustration of the S-SCAN and S-COMPLETE mechanisms, supposing that our grammar contains the rules $A \to \alpha B \beta$ and $C \to \gamma B \delta$, and that $x_i = a$.

As in the ordinary Earley algorithm, $S_{0,0}$ is initialized to contain $[S \to \bullet\alpha, 0]$ for all rules $S \to \alpha$. See Figure 5 for a simple illustration of these mechanisms.

The method described above is potentially computationally costly, since many speculative items may have to be processed. Additionally, it may produce 'excess' Earley items originating in block i, namely those that are not produced by the serial algorithm. Characterizing the number of speculative and excess items and their effect on the work-efficiency and speed-up of the parallelized algorithm will be an objective of the following sections.

Parallel computation: Having eliminated the key dependencies, sub-blocks can be computed in parallel. In particular, sub-blocks $B_{i,0}$ for $1 \leq i \leq m$ do not depend on any prior computation, and sub-block $B_{i,j}$ (for $j > 0$) directly depends only on the computation of sub-block $B_{i-1,j-1}$. Thus, it is natural to express parallelism at the level of blocks, with the associated sub-blocks computed one by one, starting with $B_{i,0}$. Algorithm 2 provides pseudo-code for computing block B_i.

Implementation details Though the parallel Earley algorithm as written above is a complete description, several implementation details help to produce a program that achieves a significant speedup. The most important such implementation details follow:

Memory allocation and caching: Consider the following properties of the parallel Earley algorithm:

Algorithm 2 PARALLELEARLEY(T_n, i)

$s \leftarrow$ start index of block B_i
$e \leftarrow$ end index of block B_i

if $s = 0$ **then**
  $S_{s,0} \leftarrow \{[S \to \bullet\alpha, 0] \mid S \to \alpha \in P\}$
else
  $S_{s,0} \leftarrow \{[A \to \alpha a \bullet \beta, -1] \mid A \to \alpha a \beta \in P,\ a = x_s\}$
end if

for $i = s$ to e **do**
  repeat
    apply S-SCAN, S-PREDICT, S-COMPLETE to $S_{i,0}$
  until nothing new added to $S_{i,0}$
end for

for $j = 1$ to i **do**
  wait for sub-block $B_{i-1,j-1}$
  for $i = s$ to e **do**
    repeat
      apply L-SCAN, L-COMPLETE to $S_{i,j}$
    until nothing new added to $S_{i,j}$
  end for
  notify sub-block $B_{i+1,j+1}$
end for

- After an Earley subset is computed, it can never be written again.
- An Earley subset may be read several times thereafter, and each time the reader is interested exclusively in those Earley items that have 'terminated' or those that are 'in-progress'.

Thus, though a set data type is used to compute an Earley subset, upon completion the set is converted to a semi-ordered array to expedite future accesses. This conversion has the added benefit of saving memory (sets have significant overhead), and improving locality (arrays are contiguous in memory).

Moreover, such a conversion requires an iteration over the set, and so it becomes computationally inexpensive to empty the set in the process. Afterwards, the set may be re-used to compute the next Earley subset, sparing the allocation and initialization of a new set object.

Thread and synchronization model: As described above, the parallel Earley algorithm allocates a single thread to each block, with sub-blocks being cal-

culated one by one beginning with sub-block $B_{i,0}$. As a result, the only communication required between threads is ensuring that, prior to computing $B_{i,j}$ (for $j > 0$), $B_{i-1,j-1}$ has completed (a result that follows from induction and the aforementioned read-only property of computed sub-blocks).

This can be accomplished with a counting semaphore between each pair of neighboring blocks: when the thread associated with block B_{i-1} completes sub-block $B_{i-1,j}$ the semaphore is 'incremented', and before the thread associated with block B_i begins computing $B_{i,j}$ $(j > 0)$, the semaphore is 'decremented'.

Finally, note that it may be worthwhile to partition the input terminal sequence into a number of blocks greater than the number of processors. Intuitively, this allows a processor that completes its assigned block to move on to a new block, increasing hardware utilization and performance. We empirically investigate the consequences of such a partition in the next section.

3.2 Parallelizing Packrat

The naïve approach: In the serial packrat algorithm, a single thread evaluates cells in the table recursively until it is able to determine whether the start symbol matches successfully. A naïve approach to parallelizing this algorithm is to add additional *worker threads* working on the same table. Each additional thread selects cells to work on based upon two criteria: the current position of the main thread, which is moving along the table from left to right, and a heuristic that it uses to choose the cells that are the most likely to be evaluated in the future by the main thread.

In the most general case, each worker thread may evaluate cells at any position. This creates synchronization problems; each time a thread chooses a cell to work on, it must ensure that no other threads are evaluating that cell, or work will be duplicated. Furthermore, all of the threads must constantly observe the position of the main thread; the further to the left of the main thread's position a cell lies, the more backtracking would be required for the main thread to reach that cell, and therefore the less likely it is that the cell will influence the result of the parse if the grammar is well-behaved. This design is even more problematic on NUMA machines or multicore machines which do not share the L2 cache between processors; the constant reads and writes to different parts of the packrat table may cause cache contention that will have a potentially profound effect on performance.

The problems of the general case can be solved by dividing the input string into blocks, and assigning each worker thread to a block. Each worker thread no longer has to observe what all of the other worker threads are doing; they can simply evaluate cells in their block until the main thread reaches the block's left side. When the main thread is about to enter a worker thread's block, it notifies that thread to terminate; this notification is the only synchronized communication that needs to occur. There are still problems, however. Terminating each worker thread as its block is reached means that later blocks have many more cells speculatively evaluated than earlier blocks, so that work efficiency gets worse and worse as we move from left to right in the packrat table. Depending upon the details of the grammar and the placement of the block boundaries, the main thread may enter and exit the same block several times; even if it restarts the block's worker thread when it exits the block, time that could have been spent speculating is wasted and unnecessary thread creation and termination overhead must occur. Finally, we still have not eliminated the requirement that at least two threads must access every block at some point during execution, and the main thread must still access every block in the packrat table; this is potentially costly on NUMA machines or in a cluster.

Start symbol synthesis: The existence of the main thread is desirable because it ensures that the algorithm does not fall too far behind what a serial algorithm would do. However, the main thread makes it necessary to share the entire packrat table between multiple threads and can cause poor behavior as it enters and exits a block. To maintain much of the advantage the main thread provides while eliminating its undesirable properties, we propose a message-

passing model based on *start symbol synthesis*. The input string is divided into blocks, and each block is assigned a worker thread. Each block uses heuristics to speculate until the block to its left has finished. When a block finishes, its final act is to send the block to its right a synthesized start symbol. This new start symbol encapsulates all of the unfinished work of the block – cells that it could not evaluate using only the information within the block. It takes the forum of a PEG rule with the following property: the rule matches successfully if and only if the original start symbol would have matched if its evaluation had continued to the same point. By evaluating this synthesized start symbol, the next block can continue the computation that the original start symbol began.

Preprocessing the grammar is useful for efficient start symbol synthesis. The repetition operators and the optional operator are replaced with the expressions in 2. These expression make use of (), the *empty string*, a special terminal that always matches without consuming any input. Simplifying the operators reduces the complexity of the final algorithm; in addition, simplifying the repetition operators in this way is always necessary to guarantee linear time parsing[5]. Next, the grammar is transformed so that the rule for each nonterminal only references one operator; this again simplifies the final algorithm, although it may involve creating new nonterminals. Sequences in the grammar are also simplified in a manner inspired by lisp: each sequence with more than two elements is replaced with a sequence with the same first a element and a new nonterminal as the second element. The new nonterminal's rule states that it matches when a new sequence, consisting of the remaining elements of the original sequence, matches. If this new sequence contains more than two elements, it is subdivided in the same way. After this transformation is complete, every sequence in the grammar has exactly two elements. By simplifying the sequences in this way, we gain simplicity and efficiency: every time a sequence crosses a block boundary, the part that extends into the next block is always an existing nonterminal or terminal. Terminals can always be evaluated very quickly. Existing nonterminals may already be memoized in the next block; even if they aren't, since a row exists for them in the packrat table they can be memoized if they need to be evaluated more than once. In contrast, the portion of an arbitrary sequence that extends into the next block may be a subsequence that does not correspond to any nonterminal; if it has to be evaluated more than once, it can't be memoized, which will make the algorithm take more than linear time.

Original	*Transformed*
$[A \leftarrow B^*]$	$[A \leftarrow BA/()]$
$[A \leftarrow B+]$	$[A \leftarrow BA/B]$
$[A \leftarrow B?]$	$[A \leftarrow B/()]$

Table 2: PEG operator transformations

The start symbol synthesis process fits naturally into the recursive evaluation of cells in the packrat algorithm. Each cell is expanded to hold two optional fields: a *complete future expression*, and a position in the block corresponding to an *incomplete future expression*. A complete future expression is a portion of the final synthesized start symbol that the block will produce. It is complete because it reaches all of the way to the end of the block, so no more cells in the current block need to be evaluated to finish constructing it. Complete future expressions are combined using the PEG operators to produce the final synthesized start symbol. An incomplete future expression, on the other hand, is suspected to be impossible to evaluate without leaving the block, but its construction is not yet complete. Incomplete future expressions may be proved not to match within the current block, but they cannot be proved to match. They are generated by the ordered choice operator when a choice with higher priority extends outside the block, but a choice with lower priority does not. As each cell in the packrat table is evaluated recursively, it combines all of the complete future expressions of its children and resolves incomplete future expressions as much as it possibly can. The way in which it accomplishes these goals depends upon the operator used by the nonterminal corresponding to that cell:

Ordered choice: Let c be the first of the choices that

either matches successfully within the block, or yields an incomplete future expression position, p, when evaluated. Let f be the set of complete future expressions encountered up to this point. If f is empty, and c was a successful match, yield success. If f is not empty, combine the elements of f using the ordered choice operator to form e. If c was found, yield e as this cell's complete future expression and either p or, if p is not defined, the position following c as this cell's incomplete future expression position. If no c was found, yield e, unless f was empty, in which case yield failure.

Sequence: Let s_1 be the first element of the sequence and s_2 be the second element. If evaluating s_1 yields failure, yield failure. If evaluating s_1 yields a complete future expression c_1, then if s_1 did not also yield an incomplete future expression position p_1, yield a sequence containing c_1 followed by s_2 as this cell's complete future expression. If s_2 yields success and s_1 yielded success, yield success. If s_2 yields success and p_1 is defined, yield c_1 as this cell's complete future expression and the position following s_2 as this cell's incomplete future expression position. If s_2 yields a complete future expression c_2 and a possible incomplete future expression position p_2, then if p_1 is defined, yield c_1 combined with c_2 using the ordered choice operator as this cell's complete future expression and p_2, if defined, as this cell's incomplete future expression position. If c_2 is defined, yield c_2 as this cell's complete future expression and p_2, if defined, as this cell's incomplete future expression position.

Not followed by: If the current position is at the end of the block, yield the current nonterminal as this cell's complete future expression. Otherwise, yield the normal result of evaluating this rule.

Others: If the current position is at the end of the input string, yield failure. If the current position is at the end of the block, yield the current nonterminal as this cell's complete future expression. Otherwise, yield the normal result of evaluating this rule.

Eventually, the recursive cell evaluations of each block will return to the start symbol for that block. At that point, there will be no incomplete future expression position, and for most blocks, the start symbol will either yield failure or a complete future expression that forms the synthesized start symbol for the next block. The rightmost block's start symbol, however, will either indicate success or failure when its evaluation is complete; whichever value it indicates determines the result for the parse as a whole.

The processing window: A final enhancement to the algorithm is the *processing window*, which serves two purposes: it reduces the imbalance in work-efficiency between the blocks near the beginning of the string and the blocks at the end, and it can be used to make the parallel packrat algorithm require an amount of space constant in the size of the input string. With a processing window in use, a fixed size is chosen for each block; ideally, a packrat table of the size chosen will fit entirely in the cache of each processing node in the parallel machine the algorithm is being run on. The ideal size will vary depending on the number of nonterminals in the grammar of interest. As many worker threads are then created as there are processing nodes, unless the input string is short enough that even fewer are required. These worker threads then begin parsing the input string, with the thread corresponding to the leftmost block processing its start symbol and the others working speculatively, as described above. If the input string is long enough, there will be more blocks than worker threads; it is in this situation that the processing window kicks in.

When the leftmost block has been completely evaluated and the synthesized start symbol it created has been passed to the right, the thread that was evaluating that block is reused. It is given the block just to the right of the rightmost block that currently has a thread working on it. The effect is as though there is a window being held in front of the blocks, and only the blocks that show through the window have threads assigned to them; when a block is com-

pleted, the window is moved to the right one block, so that a new block receives a thread to work on it. This process continues until the processing window has moved all of the way to the right and there are no blocks which are not either complete or currently being evaluated.

This technique is especially beneficial for packrat parsing since packrat tables can be so large. To save space, packrat tables can be allocated only for the blocks within the processing window; as the window moves, the old tables can either be wiped clean and returned to a pristine state, or deallocated and replaced with freshly allocated ones. If this is done, the space required by the algorithm becomes constant instead of linear in the input length, which makes packrat parsing an even more compelling alternative to convential parsing algorithms.

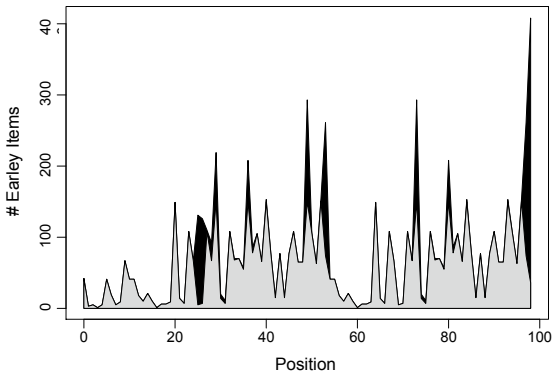

Figure 6: Total number of Earley items in each Earley subset using the serial Earley algorithm (gray) and parallel Earley algorithm using four blocks (black), when run on a typical Java file.

4 Results

4.1 Parallel Earley algorithm results

The above Java implementation was coupled with an 'off-the-shelf' context free grammar for Java. Using the parallel and serial Earley algorithms, the below results are produced deterministically, with the exception of the speedup results. The timing results used to compute reported speedups are the arithmetic mean of the timing results of the final 20 out of 25 consecutive runs.

Cost of speculation: Recall from Section 3.1 that the the final step in creating a parallel version of the Earley algorithm is breaking the dependence of subblock $B_{i,0}$ on $B_{i-1,0}$. As described, this requires incorporating speculative Earley items into the computation. Moreover, in using speculative items it is possible that 'excess' items, non-speculative items not produced by the serial Earley algorithm, are constructed. In this section, we attempt to provide some intuition and quantification of how and when Earley items are constructed as a by-product of speculation.

We first measure the total number of items in each Earley subset using both the serial Earley algorithm and the parallel Earley algorithm with four blocks. The results for a typical (small) Java file are summarized in Figure 6, and for an atypical Java file consisting of 30 (empty) nested classes in Figure 7.

In both of these plots, the beginning of each block is distinguished by a significant elevation in the number of Earley items, nearly all of which are speculative items produced by the S-Scan mechanism. Though the number of speculative and excess items typically returns to 0 very quickly, infrequent spikes not explained by the start of a block remain.

Further analysis has shown that these subsequent spikes occur when a speculative production (namely, a production that did not begin in the current block) terminates, causing a cascade of new speculative items via the S-Complete mechanism. In particular, this frequently occurs at the end of functions and classes, when an 'unmatched' brace is encountered; this last explanation matches the observed large number of speculative items towards the end of both Java files.

To quantify the effect of the number of blocks, we measure the total number items over all Earley subsets produced by the parallel Earley algorithm for various number of blocks. The results for both the typical and atypical Java files, normalized to the number of items produced by the serial algorithm, are summarized in Figure 8.

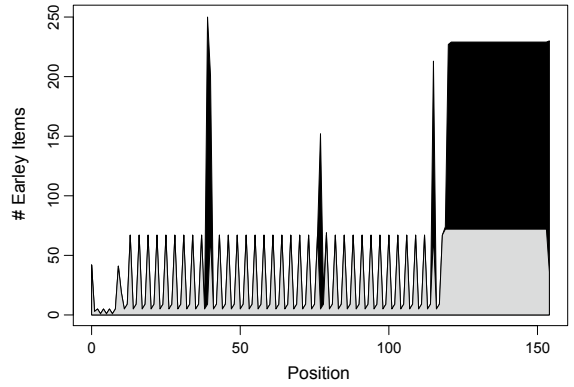

Figure 7: Total number of Earley items in each Earley subset using the serial Earley algorithm (gray) and parallel Earley algorithm using four blocks (black), when run on an atypical, deeply-nested Java file.

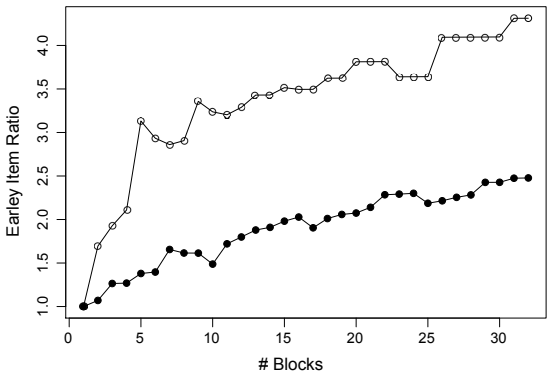

Figure 8: Ratio of the number of Earley items produced by the parallel algorithm to the number of Earley items produced by the serial algorithm for the given block size: a typical Java file (•), and a deeply-nested Java file (○).

Note that, for small numbers of blocks, the normalized value for the deeply-nested Java file grows more rapidly than that for the typical Java file. Once again, this is explained by the previous observation that unmatched braces produce large number of speculative items: in this case, unmatched braces constituting the end of the file (and the resultant large number of speculative items) are encountered even for small numbers of blocks. Once this source for speculative items is saturated, however, both files add speculative items at similar rates as the number of blocks grows.

Observed work-efficiency / speedup: Though the situation portrayed in 8 seems difficult to overcome, consider that nearly all Java files are 'typical', and are also much longer. As a result, occasional spikes as in Figure 6 are amortized over long, well-behaved regions. We consider a more direct measurement of the work-efficiency of the parallel Earley algorithm over a long (12,000 line) Java file, namely the 'speedup' of the parallel algorithm when run using a single thread and various numbers of blocks. Also included in Figure 9 are the results when multiple threads are utilized.

Focusing on the case with a single thread, we see that using multiple blocks produces a speedup of less than one, as expected. Even for 16 blocks, however, the speedup is greater than 0.8, suggesting the work efficiency is actually quite good (a reasonable interpretation is that $> 80\%$ of the time is spent doing useful work). Thus, parallelization is feasible, and observing the speedups associated with multiple threads, is realizable. The plot indicates a roughly linear relationship between speedup and the number of processors (so long as the number of blocks is sufficiently high). In the best case, a speedup of 5.44 was obtained using 8 threads and 8 blocks.

4.2 Parallel Packrat algorithm results

The parallel packrat algorithm described above was implemented completely, except for the constant memory usage feature. The algorithm was evaluated using an off-the-shelf PEG for parsing expression grammars. It was run against two inputs, a small 648 byte grammar for simple arithmetic, (Calc.peg) and a larger 2,916 byte grammar for parsing expression grammars. (PEG.peg) The speculation heuristic used evaluated a nonterminal near the top of the grammar's parse tree, Definition, and assumed that any matches were true matches, skipping over the positions used in the match when performing further speculation. Trials were conducted for worker thread

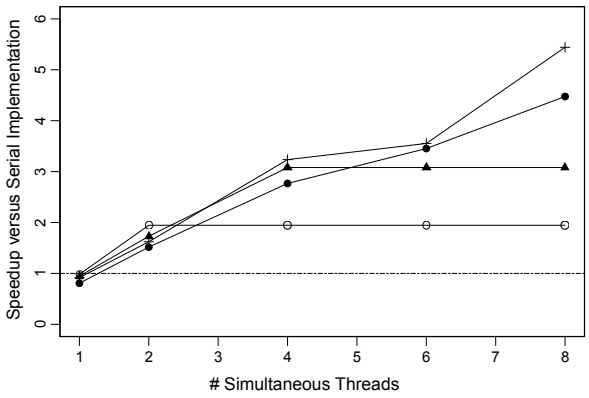

Figure 9: Speedup of the parallel Earley algorithm over the serial Earley algorithm as a function of the number of threads for: 2 blocks (○), 4 blocks (▲), 8 blocks (●), and 16 blocks (+).

counts between 1 and 8 and for block sizes of 250, 500, 750, and 1000. A special 'even' block size, signifying that the input string was to be divided evenly among the worker threads, was also evaluated. For each combination of thread count, block size, and input, 100 trials were conducted; results are based on the arithmetic mean of these trials. The machine used to run these benchmarks was an 8-core Mac Pro.

Work efficiency: The packrat tables used in the worker threads were inspected after execution of the algorithm was complete, and the number of cells evaluated was recorded. The results are depicted in Figures 10 and 11, which present the ratio of the number of cells evaluated by the parallel algorithm to the number that would have been evaluated using the serial algorithm. We can see that smaller block sizes generally result in greater speculation in both cases. The 'even' line in the Calc.peg example particularly stands out; Calc.peg is such a small file that evenly dividing its characters between more than three threads results in tiny block sizes that create extreme inefficiency. At the far right side of the figure, with 8 threads, the block size is only 81 characters; block sizes as small as this cause a great deal of additional speculative work to be done because the threads associated with later blocks have time to evaluate a much higher percentage of the cells they consider, and each thread must wait for all previous blocks to complete before it stops speculating. The 'even' measurement is additionally hurt because, since it divides the entire input string evenly between the workers, it does not take advantage of the processing window to prevent work on later parts of the string from getting out of control. Finally, in both figures, the larger block sizes remain essentially flat after a certain number of threads has been reached; this is because, with a large enough block size, only the first few threads have any work to do.

Speedup: The time taken to execute the parallel parsing algorithm was recorded using the built-in Java function `System.nanoTime()`. The time taken to allocate data structures, such as the packrat tables, was excluded, since these data structures can be reused between invocations of the algorithm and it seems realistic that such an optimization may be employed in the real-world. The results are depicted in Figures 12 and 13. Both graphs demonstrate a peak – for PEG.peg, at 5 threads, and for Calc.peg at 4 threads – but their behavior after the peak is reached is quite different. Calc.peg's speedup plunges precipitously both to the left and the right; it's probable that this peak represents a block size sweet spot at 162 characters per block that has more to do with the specific content of the input at boundaries between blocks than with a consistent trend. PEG.peg displays a better overall speedup – its peak is at about 2.5 instead of 2.3 for Calc.peg – but its results, combined with those of Calc.peg, suggest that the parallel packrat algorithm may be limited in its ability to take advantage of more threads. In both cases, we see speedup climb to a certain level and then remain more or less flat. PEG.peg, which is about 4.5 times as large as Calc.peg, showed an improvement in speedup potential, but speedup didn't increase linearly with the increase in input size. These data suggest that the speedup of the parallel packrat algorithm, while promising, is far from linear either in terms of the number of threads or the size of the in-

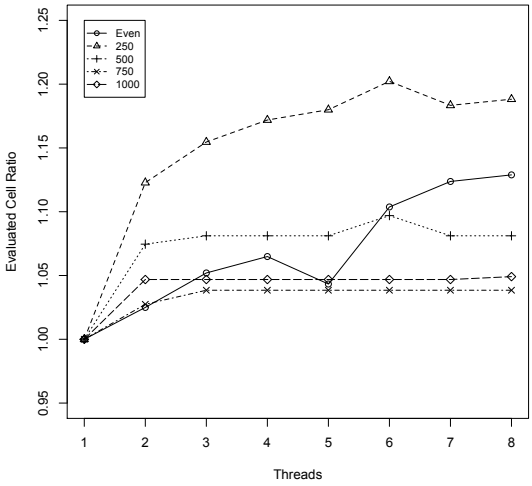

Figure 10: Ratio of the number of cells evaluated by the parallel packrat algorithm with a given number of worker threads to the number of cells evaluated without speculation, for PEG.peg.

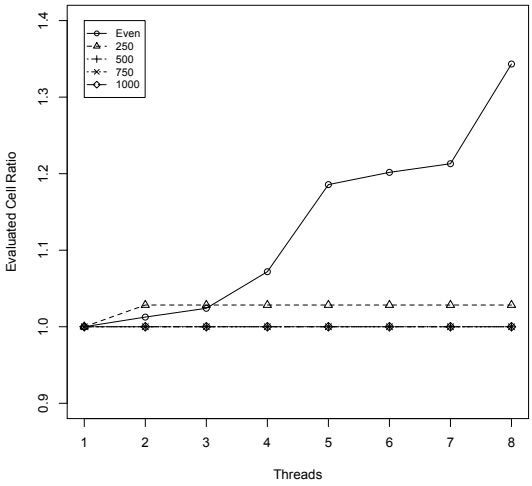

Figure 11: Ratio of the number of cells evaluated by the parallel packrat algorithm with a given number of worker threads to the number of cells evaluated without speculation, for Calc.peg.

put.

5 Conclusion

We have presented parallel versions of two well-known parsing algorithms, Earley and Packrat. Though the algorithms work differently internally, they are both fundamentally based upon the idea that the input to a parsing algorithm can be divided into blocks such that a significant amount of useful work can be done on a block without having parsed the input which preceded it. Our results have confirmed this intuition and shown that parsing is an area with a substantial amount of untapped parallelization potential. Both of our algorithms achieved speedup on realistic inputs; a maximum speedup of 5.44 was observed in the case of Earley, and 2.5 in the case of Packrat. Our algorithms are reasonably work-efficient for real-world inputs, and the parallel Packrat algorithm is capable of achieving better space complexity than the original serial algorithm.

We believe that further improvements to the algorithms and their implementations may yield still better results. Both algorithms would benefit from a heuristic to choose where block divisions occur; as we varied the sizes of the blocks our algorithms used, our results changed in nonlinear ways that suggest opportunities to further improve performance. In addition, the Packrat algorithm depends heavily on the heuristic used to choose cells to speculatively evaluate, and little work has been invested in optimizing this part of the algorithm. Beyond heuristics, neither algorithm currently takes advantage of vector or SIMD instructions, which may be a fruitful area of further research. It may even be possible to formulate new vector-style instructions tailored for parsers that could help these algorithms improve even further.

These parallel algorithms will become increasingly relevant as a new wave of computing devices with a large number of small, simple cores become more and more popular. Even for traditional computers, the ubiquity of the Web and interpreted languages leads us to believe that parallel parsing algorithms like these will prove useful in the future. We are excited to have taken a first step towards the parallel

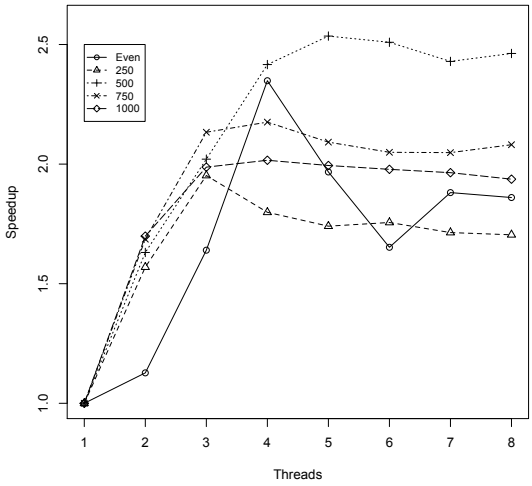

Figure 12: Speedup of the parallel packrat algorithm as compared to the single-threaded case, for PEG.peg.

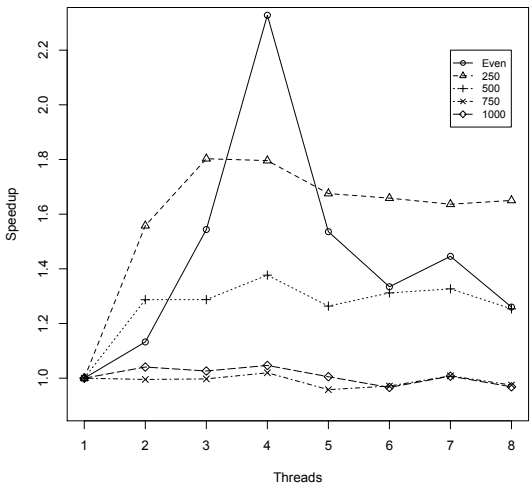

Figure 13: Speedup of the parallel packrat algorithm as compared to the single-threaded case, for Calc.peg.

parsing algorithms of tomorrow.

References

[1] J Aycock and R Horspool. Practical earley parsing. *The Computer Journal*, 46 (6):620–630, 2002.

[2] A. Birman. *The tmg recognition schema*. PhD thesis, Princeton University, Princeton, NJ, USA, 1970.

[3] Y. Chiang and K. Fu. Parallel parsing algorithms and vlsi implementations for syntactic pattern recognition. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, PAMI-6 (3):302–314, 1984.

[4] J. Earley. An efficient context-free parsing algorithm. *Commun. ACM*, 13:94–102, 1970.

[5] B. Ford. Packrat parsing: a practical linear-time algorithm with backtracking. Master's thesis, Massachusetts Institute of Technology, 2002.

[6] J Hill and A Wayne. A cyk approach to parsing in parallel: A case study. *Proceeding of the Twenty-Second SIGCSE Technical Symposium on Computer Science Education*, pages 240–245, 1991.

[7] G Sandstrom. A parallel extension of earley's parsing algorithm. 1994.

[8] L. Valiant. General context-free recognition in less than cubic time. *Journal of Computer Science*, 10:308–315, 1975.