Parallel Parsing:
The Earley and Packrat Algorithms

Seth Fowler and Joshua Paul

May 21, 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 widespread 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.

Related work: There is a substantial literature related to the difficulty 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 the CYK has a much longer running time than the 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 parsing of parsing expression grammars generally or the packrat algorithm specifically.

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:

- \(\Sigma\), 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 is 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. $a, b \in \Sigma$); $\alpha, \beta, \gamma$ 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 $T_n = x_1x_2\ldots x_n$, the Earley algorithm\cite{Earley} 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 
gramar. The Earley sets are constructed in such a way 
that $S_i$ contains all possible Earley items for terminals $x_1x_2\ldots x_i$; upon completion, $S_n$ therefore contains all 
possible Earley items for the entire input sequence.

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:

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

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

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

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

Observe that constructing Earley set $S_i$ depends only on 
$\{S_j | j \leq i\}$. 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 $[S \rightarrow \alpha, 0]$ for all rules $S \rightarrow \alpha$. 
Upon completion, the input sequence can be derived if 
and only if $S_n$ contains $[S \rightarrow \alpha, 0]$ for some rule $S \rightarrow \alpha$; 
if the sequence can be derived, one or more derivations 
may be obtained by traversing backwards through the 
Earley sets. See Algorithm 1 for pseudocode.

 Parsing expression grammars: Though their roots 
lie in Top-Down Parsing Language\cite{Top-Down}, 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\cite{PEG}. 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. Table 1 displays the oper-
ators available in a PEG, many of which resemble regular 
expression operators.

In a PEG, each rule is treated as a deterministic al-
gorithm; 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\cite{PEG}. The determin-
istic nature of PEGs is also manifested in the ordered 
choice operator / which is used when there is more than 
one way to recognize the same nonterminal. The ordered 
choice operator acts like an “if-else” construct in pro-
cedural languages; the first choice to match is selected 
and no further choices are considered. This means that 
PEGs cannot express ambiguous languages; they are nat-
urally suited for languages meant to be read by a machine 
rather than a person.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>$A \leftarrow BC$</td>
</tr>
<tr>
<td>Ordered choice</td>
<td>$A \leftarrow B/C$</td>
</tr>
<tr>
<td>Zero or more</td>
<td>$A \leftarrow B^*$</td>
</tr>
<tr>
<td>One or more</td>
<td>$A \leftarrow B+$</td>
</tr>
<tr>
<td>Optional</td>
<td>$A \leftarrow B?$</td>
</tr>
<tr>
<td>Followed by</td>
<td>$A \leftarrow &amp;B$</td>
</tr>
<tr>
<td>Not followed by</td>
<td>$A \leftarrow !B$</td>
</tr>
</tbody>
</table>

Table 1: PEG operators
**Packrat parsing:** Packrat parsing[5] is a table-based parsing algorithm which requires space linear in the size of the input string; this was considered impractical in the 1970s, but is no longer a problem today. 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 1:** 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 1 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 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 \([A → α \bullet aβ, p']\) is in \(S_{p−1,j}\) and \(a = x_p\), add \([A → aα \bullet β, p']\) to \(S_{p,j}\). Otherwise, \(S_{p−1} \in B_{i−j}\); if \([A → α \bullet aβ, p']\) is in \(S_{p−1,j−1}\) and \(a = x_p\), add \([A → aα \bullet β, p']\) to \(S_{p,j}\).
**Figure 2:** Computational dependencies for sub-blocks. The dotted lines are the dependencies associated with the sub-blocks \( B_{i,0} \) under a naïve computation.

**L-COMPLETE:** If \( [A \rightarrow \alpha \bullet \beta] \) is in \( S_{p,k} \) with \( k \leq j \), and \( [B \rightarrow \beta \bullet A_{\gamma}, p'] \) is in \( S_{p',j-k} \) (in sub-block \( B_{i-k,j-k} \)), add \( [B \rightarrow \beta A_{\gamma}, p'] \) 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 prediction mechanism, since prediction would create Earley items with origin \( p \) (and thus in block \( B_i \neq B_{i-j} \)).

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 2.

**Elimination of key dependencies:** We now propose a method for computing 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:

\[
[A \rightarrow \alpha \bullet \beta, -1].
\]

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 \( [A \rightarrow \alpha \bullet a\beta, p'] \) is in \( S_{p-1,0} \) and \( a = x_p \), add \( [A \rightarrow a\alpha \bullet \beta, p'] \) to \( S_{p,0} \).

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

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

As in the ordinary Earley algorithm, \( S_{0,0} \) is initialized to contain \( [S \rightarrow \bullet \alpha, 0] \) for all rules \( S \rightarrow \alpha \).

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 \( B_{i,0} \) (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 \( B_{1,0} \). Algorithm 2 provides pseudo-code for computing block \( B_{1,0} \).

### 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 pipeline 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.

The start symbol synthesis process fits naturally into the recursive evaluation of cells in the packrat algorithm. Before the algorithm begins, the grammar is transformed for efficiency; expressions are substituted for operators as described in Table 2, each rule is subdivided until it only uses one operator, and sequences are further subdivided until each sequence contains only two elements. Each cell is expanded to hold two optional fields: a future expression, (FE) and a position in the block corresponding to an incomplete future expression. (IFE) An FE is a portion of the final synthesized start symbol that the block will produce; matching any particular FE means that a particular path through the packrat table matched, with several of them potentially needed to account for backtracking that would have occurred in the serial algorithm. Future expressions are combined us-

<table>
<thead>
<tr>
<th>Original</th>
<th>Transformed</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A \rightarrow B^*$</td>
<td>$A \rightarrow BA(/()$</td>
</tr>
<tr>
<td>$A \rightarrow B+$</td>
<td>$A \rightarrow BA/B$</td>
</tr>
<tr>
<td>$A \rightarrow B?$</td>
<td>$A \rightarrow B(/()$</td>
</tr>
</tbody>
</table>

### Table 2: PEG operator transformations

---

*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.

The start symbol synthesis process fits naturally into the recursive evaluation of cells in the packrat algorithm. Before the algorithm begins, the grammar is transformed for efficiency; expressions are substituted for operators as described in Table 2, each rule is subdivided until it only uses one operator, and sequences are further subdivided until each sequence contains only two elements. Each cell is expanded to hold two optional fields: a future expression, (FE) and a position in the block corresponding to an incomplete future expression. (IFE) An FE is a portion of the final synthesized start symbol that the block will produce; matching any particular FE means that a particular path through the packrat table matched, with several of them potentially needed to account for backtracking that would have occurred in the serial algorithm. Future expressions are combined us-
ing the PEG operators to produce the final synthesized start symbol. IfEs are currently being constructed and have matched so far, but have not yet reached the edge of the block; they are potential FEs since they will be evaluated if an existing FE fails, but they may yet fail to match within the current block. As each cell in the packrat table is evaluated recursively, it combines all of the FEs of its children and resolves IFEs as much as it possibly can. The details depend on the PEG operator used by the rule for the cell’s nonterminal (terminals, for the algorithm’s sake, are considered to use the “terminal” operator):

**Ordered Choice**: The FEs of all children that aren’t eliminated within the current block are combined using the ordered choice operator and recorded in the cell. The process stops when either a child produces an IFE, or a child matches successfully within the block. In either case, since PEG matching is greedy, no further options need be considered, and the position after the IFE or matching child is recorded in the cell.

**Sequence**: Each child in the sequence is matched until one child either fails, in which case the whole sequence fails, or produces an FE, in which case the rest of the sequence is appended to the FE using the sequence operator. If, when the FE is produced, an IFE is also produced, matching of the rest of the sequence must continue down this alternate path. After all such alternate paths are considered, they are combined using the ordered choice operator, and the result is recorded in the cell. A single IFE may remain which the sequence did not finish matching; if so, the position following it is also recorded in the cell.

**Terminal and Others**: If the current position is at the end of the input string, record that the cell failed to match unless using the not-followed-by-operator. If the current position is at the end of the block, record the current nonterminal in this cell’s FE.

In all cases, if no FEs or IFEs were generated, the rule matches or doesn’t match within the block as usual, and the FE and IFE fields in the cell are left empty.

Eventually, the recursive cell evaluations of each block will return to the start symbol for that block. At that point, there will be no IFE, and for most blocks, the start symbol will either yield failure or an FE 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; this value 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. When the leftmost block has been completely evaluated and the synthesized start symbol it created has been passed to the right, the corresponding thread is reused to work on the leftmost block that has not yet been assigned. 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 completed, the window is moved to the right one block. 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. The space required by the packrat algorithm can be made constant by maintaining a block’s packrat table in memory only if the block is within the processing window.

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 sub-block $B_{i,0}$ 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 3, and for an atypical Java file consisting of 30 (empty) nested classes in Figure 4.

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 5.

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 Figure 5 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 3 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 6 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 to the start symbol in the PEG grammar, Definition, and assumed that any matches were true matches, not speculating any further within the matched region. Trials were conducted for worker thread 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 with 16GB of RAM.

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 7 and 8, 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 with such small blocks it may not be possible to match a single Definition, which makes the heuristic in use a poor choice. 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. The results are depicted in Figures 9 and 10. 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 suggest that the current implementation of the parallel packrat algorithm may have flaws hampering full parallelization. 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, suggesting that the algorithm reaches its full potential with larger inputs. We suspect, based upon our experience with the parallel Earley algorithm, that excessive memory allocation in the inner loop may be limiting the parallel packrat algorithm’s performance; we hope to eliminate this problem in future implementations.

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 have 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 parsing algorithms of tomorrow.

References


