Bitwise Data Parallelism with LLVM: The ICgrep Case Study

Robert D. Cameron✉, Nigel Medforth, Dan Lin, Dale Denis, and William N. Sumner

School of Computing Science, Simon Fraser University, Surrey, B.C., Canada
{cameron,nmedfort,lindanl,daled,wsumner}@sfu.ca

Abstract. Bitwise data parallelism using short vector (SIMD) instructions has recently been shown to have considerable promise as the basis for a new, fundamentally parallel, style of regular expression processing. This paper examines the application of this approach to the development a full-featured Unicode-capable open-source grep implementation. Constructed using a layered architecture combining Parabix and LLVM compiler technologies, icGrep is the first instance of a potentially large class of text processing applications that achieve high performance text processing through the combination of dynamic compilation and bitwise data parallelism. In performance comparisons with several contemporary alternatives, 10× or better speedups are often observed.

This is the authors' version of the paper published in Algorithms and Architectures for Parallel Processing, Wang, Guojun and Zomaya, Albert and Perez, Gregorio Martinez and Li, Kenli (eds), *Lecture Notes in Computer Science* 9529, Nov. 2015, pp 373-387, http://dx.doi.org/10.1007/978-3-319-27122-4_26. The final publication is available at link.springer.com.

Keywords: Bitwise Data Parallelism; Dynamic Compilation; ICgrep; MatchStar; Parabix Transform; Regular Expression; SIMD

1 Introduction

Although well-established technical standards exist for Unicode regular expressions [4], most of today's regular expression processing toolsets fail to support the full set of processing features even at the most basic level [11]. One of the fundamental issues is performance and so it makes good sense to consider the ways in which parallel processing approaches can help address the gap.

Efforts to improve the performance of regular expression matching through parallelization have generally concentrated on the use of SIMD, multicore or GPU technologies to accelerate multiple instances of independent matching problems. Scarpazza [10] used SIMD and multicore parallelism to accelerate small ruleset tokenization applications on the Cell Broadband Engine. Salapura [9] built on these techniques to accelerate business analytics applications using SSE

instructions on commodity processors. Zu et al. [13] use GPU technology to implement NFA-based regular expression matching with parallelism devoted both to processing a compressed active state array as well as to handling matching of multiple packet instances.

Using parallel methods to accelerate matching of a single pattern on a single input stream is more difficult. Indeed, of the 13 dwarfs identified in the Berkeley overview of parallel computing research, finite state machines (FSMs) are considered the hardest to parallelize (embarrassingly sequential) [1]. However, some success has been reported recently along two independent lines of research. The first focuses on data-parallel division of an input stream into separately processed segments on multiple cores. In this case, the challenge is to identify the appropriate starting state or states that need to be considered for all but the first segment. Mytkowicz et al. [8] use dynamic convergence and range coalescing to dramatically reduce the number of states in play at any point, while Zhao et al. [12] address the problem using principled speculation. The second line of research is based on Parabix technology: the use of short vector SIMD instructions (e.g., Intel SSE or AVX instructions) to process bit streams in one-to-one correspondence with input character streams. Following this approach, Cameron et al. [3] have recently prototyped a new bitwise data parallel algorithm that shows significant acceleration of regular expression matching performance compared to sequential implementations even on a single core.

These previous works in applying parallel methods to regular expression matching have generally focused on traditional ASCII-based matching problems. However, documents and data formats increasingly use Unicode, particularly in the form of UTF-8. W3Techs.com reports that the percentage of websites reporting UTF-8 as the character encoding has reached 84% as of May 2015. But regular expression matching over UTF-8 introduces complexities of variable-length encodings for characters. To process UTF-8 documents directly, Unicode regular expressions must generally undergo considerable state expansion to equivalent regular expressions over byte sequences. Alternatively, if the price is paid to first transcode a UTF-8 document to UTF-16, the problem of very large transition tables arises.

In this paper, we report on the use of the implementation of a full UTF-8 regular expression search tool, building on the bitwise data parallel methods of the Parabix framework combined with the dynamic compilation capabilities of LLVM [5]. The result is icGrep, a high-performance, full-featured open-source grep implementation with systematic support for Unicode regular expressions, fully implementing Unicode level 1 requirements of Unicode Technical Standard #18[4]. As an alternative to classical grep implementations, icGrep offers dramatic performance acceleration in Unicode regular expression matching.

The main contributions reported here are the extension of the bitwise data parallel methods for regular expression matching to handle UTF-8 natively, validating the bitwise data parallel method with the first complete implementation in a useful tool, and introducing icGrep itself as research artifact. From the latter perspective, icGrep is significant in several ways. First, it allows the per-

formance results reported here to be independently replicated. Secondly, it is a platform for further research into regular expression matching using bitwise methods and is indeed being used to investigate Unicode level 2 requirements in a project funded by Google. Finally, it is working example demonstrating the Parabix+LLVM framework, and useful for guiding further research into bitwise data parallel algorithms generally and how they may take advantage of evolving architectural features.

The remainder of this paper is organized as follows. Section 2 presents background material dealing with Unicode regular expressions, the Parabix framework and regular expression matching techniques using bitwise data parallelism. Section 3 addresses the issues and performance challenges associated with meeting Unicode regular expression requirements and presents extensions to the Parabix techniques that we have developed to address them. Section 4 describes the overall architecture of the icGrep implementation with a focus on the integration of Parabix and LLVM technologies. Section 5 evaluates the performance of icGrep on several types of matching problems with two contemporary competitors, pcre2grep and ugrep of the ICU (International Component for Unicode) software library. Section 6 concludes the paper with remarks on developing the Parabix+LLVM framework for other applications as well as identifying further research questions in Unicode regular expression matching with bitwise data parallelism.

2 Background

Unicode Regular Expressions. Traditional regular expression syntax is oriented towards string search using regular expressions over ASCII or extended-ASCII byte sequences. A grep search for a line beginning with a capitalized word might use the pattern "`^[A-Z][a-z]+`" ("extended" syntax). Here, "`^`" is a zero-width assertion matching only at the start of a line, "`[A-Z]`" is a character class that matches any single character in the contiguous range of characters from A through Z, while the plus operator in "`[a-z]+`" denotes repetition of one or more lower case ASCII letters.

While explicit listing of characters of interest is practical with ASCII, it is less so with Unicode. In the Unicode 7.0 database, there are 1490 characters categorized as upper case and 1841 categorized as lower case. Rather than explicitly listing the individual characters, then, it is more practical to use named character classes, such as `Lu` for upper case letters and `Ll` for lower case letters. Using these names, our search might be rewritten to find capitalized words in any language as "`^\p{Lu}\p{Ll}+`" (Perl-compatible syntax). The Unicode consortium has defined an extensive list of named properties that can be used in regular expressions.

Beyond named properties, Unicode Technical Standard #18 defines additional requirements for Unicode regular expressions, at three levels of complexity [4]. Level 1 generally relates to properties expressed in terms of individual Unicode codepoints, while level 2 introduces complexities due to codepoint se-

quences that form grapheme clusters, and level 3 relates to tailored locale-specific support. We consider only Unicode level 1 requirements in this paper, as most grep implementations are incomplete with respect the requirements even at this level. The additional level 1 regular expression requirements primarily relate to larger classes of characters that are used in identifying line breaks, word breaks and case-insensitive matching. Beyond this, there is one important syntactic extension: the ability to refine character class specifications using set intersection and subtraction. For example, `[\p{Greek}&&\p{Lu}]` denotes the class of upper case Greek letters, while `[\p{Ll}--\p{ASCII}]` denotes the class of all non-ASCII lower case letters.

Bitwise Data Parallel Matching. Regular expression search using bitwise data parallelism has been recently introduced and shown to considerably outperform methods based on DFAs or NFAs [3]. In essence, the method is 100% data parallel, considering all input positions in a file simultaneously. A set of parallel bit streams is computed, with each bit position corresponding to one code-unit position within input character stream. Each bit stream may be considered an L-bit integer, where L is the length of the input stream. Matching is performed using bitwise logic and addition on these long integers.

Consider the following simplified definition of regular expressions. A regular expression may be a character class C or formed by composition. If R and S are regular expressions, then the concatenation RS is the regular expression standing for the set of all strings formed by concatenating one string from R and one string from S, $R|S$ is the alternation standing for the union of the sets of strings from R and S, and $R*$ is the Kleene-star closure denoting the set of strings formed from 0 or more occurrences of strings from R.

Character class streams, such as CharClass(`[d]`) for the stream that marks the position of "d" characters and CharClass(`[a-z]`) for the stream of lower case ASCII alphabetics are first computed in a fully data-parallel manner. Then the search process proceeds by computing *marker streams* that mark the positions of matches so far. Each bit in a marker stream indicates that all characters prior to the given position have been matched. The initial marker stream m_0 consists of all ones, i.e., $m_0 = 2^L - 1$, indicating that all positions are in play. Matching then proceeds in accord with the following equations.

$$\begin{aligned}
\text{Match}(m, C) &= \text{Advance}(\text{CharClass}(C) \wedge m) \\
\text{Match}(m, RS) &= \text{Match}(\text{Match}(m, R), S) \\
\text{Match}(m, R|S) &= \text{Match}(m, R) \vee \text{Match}(m, S)) \\
\text{Match}(m, C*) &= \text{MatchStar}(m, \text{CharClass}(C)) \\
\text{Match}(m, R*) &= m \vee \text{Match}(\text{Match}(m, R), R*)
\end{aligned}$$

Here, Advance is an operation that advances all markers by a single position.

$$\text{Advance}(m) = m + m$$

MatchStar finds all matches of character class repetitions in a surprisingly simple manner [3].

$$\text{MatchStar}(M, C) = (((M \wedge C) + C) \oplus C) \vee M$$

Interestingly, the MatchStar operation also has application to the parallelized long-stream addition itself [3], as well as the bit-parallel edit distance algorithm of Myers[7]. For repetitions of other regular expressions, the final recursive equation accounts for the repeated extension of current matches; this equation is applied until no new match positions result.

For example, Figure 1 shows how the regular expression `d[a-z]*ed` is matched against some input text using bitwise methods. In this diagram we use periods to denote 0 bits so that the 1 bits stand out. In the first step the character class stream `[d]` is matched and the results shifted one position (Advance) to produce marker bitstream M_1. Five matches indicated by marker bits are now in play simultaneously. The next step applies the MatchStar operation to find all the matches that may then be reached with the Kleene-* repetition `[a-z]*` (M_2). This produces pending matches at many positions. However, there is no need to consider these matches one at a time using lazy or greedy matching strategies. Rather, the full marker stream M_3 of remaining possibilities after matching `[e]` is easily computed using bitwise logic and shift. The final step produces marker stream M_4 indicating the single position at which the entire regular expression is matched.

input data	`dead dreams defeated.`
$M_1 = \text{Advance}(\texttt{[d]})$	`.1..1.1......1......1`
$M_2 = \text{MatchStar}(M_1, \texttt{[a-z]})$	`.1111.111111.11111111`
$M_3 = \text{Advance}(M_2 \wedge \texttt{[e]})$	`..1.....1.....1.1..1.`
$M_4 = \text{Advance}(M_3 \wedge \texttt{[d]})$	`....................1`

Fig. 1: Matching `d[a-z]*ed` using bitwise data parallelism

The Parabix toolchain [6] provides a set of compilers and run-time libraries that target the SIMD instructions of commodity processors (e.g., SSE or AVX instructions on x86-64 architecture). Input is processed in blocks of code units equal to the size in bits of the SIMD registers, for example, 128 bytes at a time using 128-bit registers. Using the Parabix facilities, the bitwise data parallel approach to regular expression search was shown to deliver substantial performance acceleration for traditional ASCII regular expression matching tasks, often 5× or better [3].

3 Bitwise Methods for UTF-8

As described in the following section, icGrep is a reimplementation of the bitwise data parallel method implemented on top of LLVM infrastructure and adapted

for Unicode regular expression search through data streams represented in UTF-8. In this section, we present the techniques we have used to extend the bitwise matching techniques to the variable-length encodings of UTF-8.

The first requirement in implementing a regular expression processor over UTF-8 data streams is to translate Unicode regular expressions over codepoints to corresponding regular expressions over sequences of UTF-8 bytes or *code units*. The `toUTF8` function does the work, transforming input expressions such as '`\u{244}[\u{2030}-\u{2137}]`' to the corresponding UTF-8 representation: `\xE2((\x84[\x80-\xB7])|(([\x81-\x83][\x80-\xBF])|(\x80[\xB0-\xBF])))`

UTF-8 Byte Classification and Validation. In UTF-8, bytes are classified as (1) individual ASCII bytes, (2) prefixes of two-, three-, or four-byte sequences, or (3) suffix bytes. In addition, we say that the *scope* bytes of a prefix are the immediately following byte positions at which a suffix byte is expected. Mismatches between scope expectations and occurrences of suffix bytes indicate errors (we omit other error equations for brevity). Two helper streams are also useful. The Initial stream marks ASCII bytes and prefixes of multibyte sequences, while the NonFinal stream marks any position that is not the final position of a Unicode character.

$$
\begin{aligned}
\text{ASCII} &= \text{CharClass}(\texttt{[\textbackslash x00-\textbackslash x7F]}) \\
\text{Prefix} &= \text{CharClass}(\texttt{[\textbackslash xC2-\textbackslash F4]}) \\
\text{Prefix3or4} &= \text{CharClass}(\texttt{[\textbackslash xE0-\textbackslash xF4]}) \\
\text{Prefix4} &= \text{CharClass}(\texttt{[\textbackslash xF0-\textbackslash xF4]}) \\
\text{Suffix} &= \text{CharClass}(\texttt{[\textbackslash x80-\textbackslash xBF]}) \\
\text{Scope} &= \text{Advance}(\text{Prefix}) \vee \text{Advance}(\text{Prefix3or4}, 2) \vee \text{Advance}(\text{Prefix4}, 3) \\
\text{Mismatch} &= \text{Scope} \oplus \text{Suffix} \\
\text{Initial} &= \text{ASCII} \vee \text{Prefix} \\
\text{NonFinal} &= \text{Prefix} \vee \text{Advance}(\text{Prefix3or4}) \vee \text{Advance}(\text{Prefix4}, 2)
\end{aligned}
$$

Unicode Character Classes. Whereas ASCII character classes can be determined by simple bitwise logic at a single code unit position, the UnicodeClass stream for a given class involves logic for up to four positions. By convention, we define UnicodeClass(U) for a given Unicode character class U to be the stream marking the *final* position of any characters in the class.

Using these definitions, it is then possible to extend the matching equations to operate with Unicode character classes as follows.

$$
\begin{aligned}
\text{Match}(m, U) &= \text{Advance}(\text{ScanThru}(m, \text{NonFinal}) \wedge \text{UnicodeClass}(U)) \\
\text{Match}(m, U*) &= \text{MatchStar}(m, \text{UnicodeClass}(U) \vee \text{NonFinal}) \wedge \text{Initial}
\end{aligned}
$$

Here, we use the ScanThru [2] operation to move a set of markers each through the nonfinal bytes of UTF-8 sequences to the final byte position.

$$ScanThru(m, c) = (m + c) \wedge \neg c$$

Figure 2 shows this technique in operation in the case of advancing through byte sequences (each 3 bytes in length) corresponding to Chinese characters. To better demonstrate the process, we use `ni3`, `hao` and `men` to represent these characters. $CC_{\texttt{ni3}}$ is the bitstream that marks character `ni3` and $CC_{\texttt{hao}}$ is the bitstream that marks character `hao`. We start with the marker stream m_0 initialized to Initial, indicating all positions are in play. Using ScanThru, we move to the final position of each character t_1. Applying bitwise `AND` with $CC_{\texttt{ni3}}$ and advancing gives the two matches m_1 for ni3. Applying ScanThru once more advances to the final position of the character after `ni3`. The final result stream m_2 shows the lone match for the multibyte sequence `ni3hao`.

input data	`ni3hao(Hello),ni3men(You),`
CC_{ni3}	`..1.....1.....1.....`
CC_{hao}	`.....1.....`
$m_0 =$ Initial	`1..1..111111111..1..111111`
NonFinal	`11.11.....11.11.....`
$t_1 = \text{ScanThru}(m_0, \text{NonFinal})$	`..1..111111111..1..1111111`
$m_1 = \text{Advance}(t_1 \wedge \text{CC}_{\text{ni3}})$	`...1.....1.....1.....`
$t_2 = \text{ScanThru}(m_1, \text{NonFinal})$	`.....1.....1.....1.....`
$m_2 = \text{Advance}(t_2 \wedge CC_{\text{hao}})$	`......1.....`

Fig. 2: Processing of a multibyte sequence ni3hao

Unicode MatchStar. The $\text{MatchStar}(M, C)$ operation directly implements the operation of finding all positions reachable from a marker bit in M through a character class repetition of an ASCII byte class C. In UTF-8 matching, however, the character class byte streams are marked at their *final* positions. Thus the one bits of a Unicode character class stream are not necessarily contiguous. This in turn means that carry propagation within the MatchStar operation may terminate prematurely.

In order to remedy this problem, icGrep again uses the NonFinal stream to "fill in the gaps" in the $\text{UnicodeClass}(U)$ bitstream so that the MatchStar addition can move through a contiguous sequence of one bits. In this way, matching of an arbitrary Unicode character class U can be implemented using $\text{MatchStar}(m, \text{UnicodeClass}(U) \vee \text{NonFinal})$.

Predefined Unicode Classes. icGrep employs a set of bitstreams that are precompiled into the executable. These include all bitstreams corresponding to Unicode property expressions such as `\p{Greek}`. Each property potentially contains

many code points, so we further embed the calculations within an if hierarchy. Each if-statement within the hierarchy determines whether the current input block contains any codepoints at all in a given Unicode range. At the outer level, the ranges are quite coarse, becoming successively refined at deeper levels. This technique works well when input documents contain long runs of text confined to one or a few ranges.

4 Architecture

Regular Expression Preprocessing. As shown in Fig. 3, compilation in icGrep comprises three logical layers: RegEx, Parabix and the LLVM layer, each with their own intermediate representation (IR), transformation and compilation modules. As we traverse the layers, the IR becomes more complex as it begins to mirror the final machine code. The layering enables further optimization based on information available at each stage. The RegEx Parser validates and transforms the input RegEx into an abstract syntax tree (AST). Successive RegEx Transformations exploit domain knowledge to optimize the regular expressions. The aforementioned `toUTF8` transformation also applies during this phase to generate code unit classes.

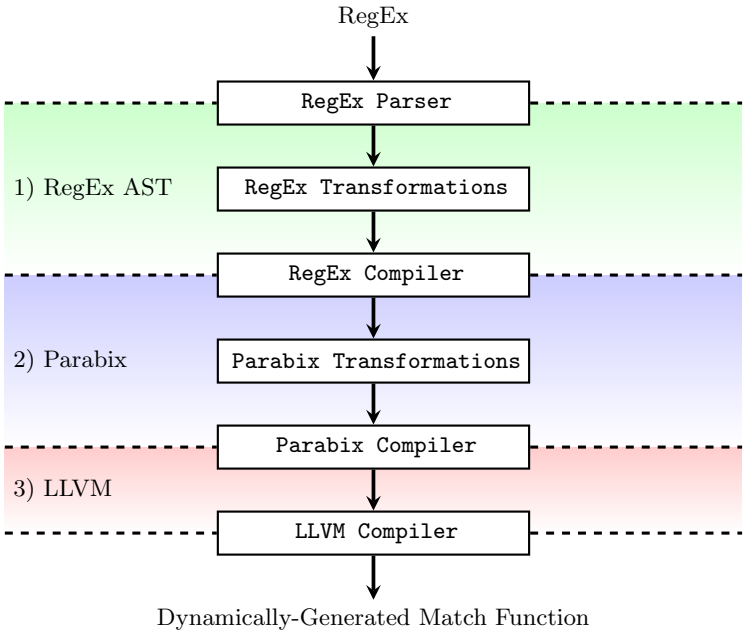

Fig. 3: icGrep compilation architecture

The next layer transforms this AST into the instructions in the Parabix IR. Recall that the Parabix layer assumes a transposed view of the input data. The *RegEx Compiler* first transforms all input code unit classes, analogous to

non-Unicode character classes, into a series of equations over these transposed bitstreams. It next transforms the AST into Parabix instructions that use the results of these equations. For instance, it converts alternations into a sequence of calculations that are merged with `OR`s. The results of these passes are combined and transformed through typical optimization passes including dead code elimination (DCE), common subexpression elimination (CSE) and constant folding. These optimizations exploit redundancies that are harder to recognize in the RegEx AST itself.

The Parabix Compiler then directly converts the Parabix IR into LLVM IR. The LLVM Compiler framework provides flexible APIs for compilation and linking. Using these, icGrep dynamically generates a match function for identifying occurrences of the RegEx.

Dynamic Grep Engine. Figure 4 shows the structure of the icGrep matching engine. The input data is transposed into 8 parallel bit streams through the Transposition module. Using the 8 basis bits streams, the Required Streams Generator computes the line break streams, UTF-8 validation streams and the Initial and NonFinal streams needed to support ScanThru and MatchStar with UTF-8 data. The Dynamic Matcher, dynamically compiled via LLVM, retrieves the 8 basis bits and the required streams from their memory addresses and starts the matching process. During the matching process, any references to named Unicode properties generate calls to the appropriate routine in the Named Property Library. The Dynamic Matcher returns one bitstream that marks all the positions that fully match the compiled regular expression. Finally, a Match Scanner scans through the returned bitstream to select the matching lines and generate the normal grep output.

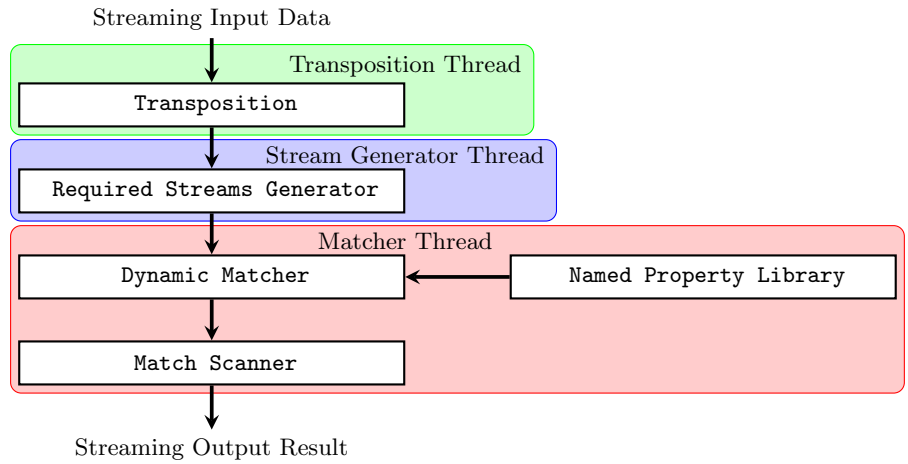

Fig. 4: Data flow in an icGrep execution

We can also apply a pipeline parallelism strategy to further speed up the process of icGrep. Transposition and the Required Streams Generator can be

performed in a separate thread which can start even before the dynamic compilation starts. The output of Transposition and the Required Streams Generator, that is the 8 basis bits streams and the required streams, are stored in a shared memory buffer for subsequent processing by the Dynamic Matcher once compilation is complete. A single thread performs both compilation and matching using the computed basis and required streams. To avoid L2 cache contention, we allocate only a limited amount of space for the shared data in a circular buffer. The performance is dependent on the slowest thread. In the case that the cost of transposition and required stream generation is more than the matching process, we can further divide up the work and assign two threads for Transposition and the Required Streams Generator.

5 Evaluation

In this section, we report on the evaluation of icGrep performance, looking at three aspects. First, we discuss some performance aspects of icGrep internal methods, looking at the impact of optimizations discussed previously. Then we move on to a systematic performance study of icGrep with named Unicode property searches in comparison to two contemporary competitors, namely, pcre2grep released in January 2015 and ugrep of the ICU 54.1 software distribution. Finally, we examine complex expressions and the impact of multithreading icGrep on an Intel i7-2600 (3.4GHz) and i7-4700MQ (2.4GHz) processor.

5.1 Optimizations of Bitwise Methods

In order to support evaluation of bitwise methods, as well as to support the teaching of those methods and ongoing research, icGrep has an array of command-line options. This makes it straightforward to report on certain performance aspects of icGrep, while others require special builds.

For example, the command-line switch `-disable-matchstar` can be used to eliminate the use of the MatchStar operation for handling Kleene-* repetition of character classes. In this case, icGrep substitutes a while loop that iteratively extends match results. Surprisingly, this does not change performance much in many practical cases. In each block, the maximum iteration count is the maximum length run encountered; the overall performance is based on the average of these maxima throughout the file. But when search for XML tags using the regular expression `<[^!?][^>]*>`, a slowdown of more than 2× may be found in files with many long tags.

In order to short-circuit processing when no remaining matches are possible in a block, our regular expression compiler periodically inserts if-statements to check whether there are any marker bits still in play. To control this feature in dynamically generated code, the number of pattern elements between each if-test can be selected with the `-if-insertion-gap=` option. The default value in icGrep is 3; setting the gap to 100 effectively turns off if-insertion. Eliminating if-insertion sometimes improves performance by avoiding the extra if tests and

branch mispredictions. For patterns with long strings, however, there can be a substantial slowdown.

The precompiled calculations of the various Unicode properties are each placed in if-hierarchies as described previously. To assess the impact of this strategy, we built a version of icGrep without such if-hierarchies. In this case, when a Unicode property class is defined, bitwise logic equations are applied for all members of the class independent of the Unicode blocks represented in the input document. For the classes covering the largest numbers of codepoints, we observed slowdowns of up to 5×.

5.2 Simple Property Expressions

A key feature of Unicode level 1 support in regular expression engines the support that they provide for property expressions and combinations of property expressions using set union, intersection and difference operators. Both `ugrep` and `icgrep` provide systematic support for all property expressions at Unicode Level 1 as well as set union, intersection and difference. However, in order to implement these operators with `pcre2grep`, we translated them into an equivalent form using lookbehind assertions.

We generated a set of regular expressions involving all Unicode values of the Unicode general category property (`gc`) and all values of the Unicode script property (`sc`). We then generated expressions involving random pairs of `gc` and `sc` values combined with a random set operator chosen from union, intersection and difference. All property values are represented at least once. A small number of expressions were removed because they involved properties not supported by pcre2grep. In the end 246 test expressions were constructed in this process.

We selected a set of Wikimedia XML files in several major languages representing most of the world's major language families as a test corpus. For each program under test, we performed searches for each regular expression against each XML document. Test cases were ranked by the percentage of matching lines found in the XML document and grouped in 5% increments. Performance is reported in CPU cycles per byte on an Intel i7-2600 machine. The results are presented in Fig. 5.

When comparing the three programs, icGrep exhibits dramatically better performance, particularly when searching for rare items. The performance of both pcre2grep and ugrep improves (CPU cycles per byte decreases) as the percentage of matching lines increases. This occurs because each match allows them to bypass processing the rest of the line. On the other hand, icGrep shows a slight drop-off in performance with the number of matches found. This is primarily due to property classes that include large numbers of codepoints. These classes require more bitstream equations for calculation and also have a greater probability of matching. Nevertheless, the performance of icGrep in matching the defined property expressions is stable and well ahead of the competitors in all cases.

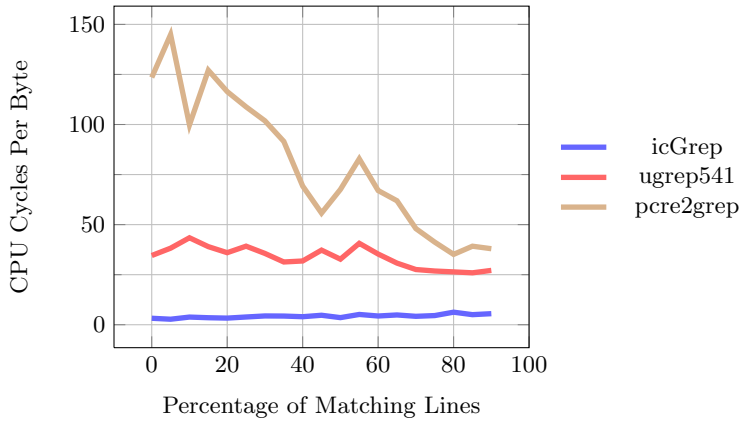

Fig. 5: Matching Performance for Simple Property Expressions

5.3 Complex Expressions

This study evaluates the comparative performance of the matching engines on a set of more complex expressions, shown in Table 1. The first two are alphanumeric (A.N.) expressions, differing only in that the first one is anchored to match the entire line. The third searches for lines consisting of text in Arabic script. The fourth expression is a published currency expression taken from Stewart and Uckelman[11]. An expression matching runs of six or more Cyrillic script characters enclosed in initial/opening and final/ending punctuation is fifth in the list. The last expression matches internationalized email names.

Table 1: Regular expressions

Name	**Regular Expression**
A.N. #1	`^[\p{L}\p{N}]*((\p{L}\p{N})|(\p{N}\p{L}))[\p{L}\p{N}]*$`
A.N. #2	`[\p{L}\p{N}]*((\p{L}\p{N})|(\p{N}\p{L}))[\p{L}\p{N}]*`
Arabic	`^[\p{Arabic}\p{Common}]*\p{Arabic}[\p{Arabic}\p{Common}]*$`
Currency	`(\p{Sc}\s*(\d*|(\d{1,3}([,.]\d{3})*))([,.]\d{2}?)?)|` `((\d*|(\d{1,3}([,.]\d{3})*))([,.]\d{2}?)?\s*\p{Sc})`
Cyrillic	`[\p{Pi}\p{Po}]\p{Cyrillic}{6,}[\p{Pf}\p{Pe}]`
Email	`([^\p{Z}<]+@[\p{L}\p{M}\p{N}.-]+\.(\p{L}\p{M}*){2,6})` `(>|\p{Z}|$)`

Table 2 shows the performance results on our Intel i7-2600 test machine, reporting seconds taken per GB of input averaged over 10 runs each on our Wikimedia document collection.

Table 2: Matching times for complex expressions (s/GB)

	icGrep			
Expression	**SEQ**	**MT**	**pcre2grep**	**ugrep541**
A.N. #1	2.4 – 5.0	2.1 – 4.4	8.2 – 11.3	8.8 – 11.3
A.N. #2	2.3 – 4.9	2.0 – 4.1	209.9 – 563.5	182.3 – 457.9
Arabic	1.5 – 3.4	1.2 – 2.6	7.5 – 270.8	8.9 – 327.8
Currency	0.7 – 2.1	0.4 – 1.4	188.4 – 352.3	52.8 – 152.8
Cyrillic	1.6 – 3.9	1.3 – 2.8	30.5 – 49.7	11.2 – 20.1
Email	3.0 – 6.9	2.7 – 6.4	67.2 – 1442.0	108.8 – 1022.3

The most striking aspect of Table 2 is that both ugrep and pcregrep show dramatic slowdowns with ambiguities in regular expressions. This is most clearly illustrated in the different performance figures for the two Alphanumeric test expressions but is also evident in the Arabic, Currency and Email expressions. Contrastingly, icGrep maintains consistently fast performance in all test scenarios. The multithreaded icGrep shows speedup in every case but balancing of the workload across multiple cores is clearly an area for further work. Nevertheless, our three-thread system shows up to a 40% speedup.

Table 3 shows the speedups obtained with icGrep on a newer Intel i7-4700MQ machine, considering three SIMD ISA alternatives and both single-threaded and multi-threaded versions. All speedups are relative to the base single-threaded SSE2 performance on this machine, which is given in seconds per GB in the first column. The SSE2 results are again using the generic binaries compiled for compatibility with all 64-bit processors. The AVX1 results are for Intel AVX instructions in 128-bit mode. The main advantage of AVX1 over SSE2 is its support for 3-operand form, which helps reduce register pressure. The AVX2 results are for icGrep compiled to use the 256-bit AVX2 instructions, processing blocks of 256 bytes at a time.

Table 3: Speedup of complex expressions on i7-4700MQ (σ)

	Base	**SEQ**		**MT**		
Expression	**s/GB**	**AVX1**	**AVX2**	**SSE2**	**AVX1**	**AVX2**
A.N. #1	2.76 (.65)	1.05 (.03)	1.25 (.08)	1.18 (.02)	1.19 (.03)	1.59 (.10)
A.N. #2	2.69 (.66)	1.05 (.02)	1.36 (.09)	1.20 (.03)	1.19 (.04)	1.80 (.11)
Arabic	1.82 (.39)	1.05 (.03)	1.15 (.08)	1.37 (.03)	1.37 (.04)	1.66 (.10)
Currency	1.04 (.30)	1.03 (.02)	1.04 (.06)	1.59 (.15)	1.61 (.14)	1.78 (.21)
Cyrillic	2.10 (.44)	1.06 (.02)	0.96 (.06)	1.27 (.02)	1.33 (.04)	1.23 (.09)
Email	3.57 (.87)	1.05 (.03)	1.37 (.14)	1.13 (.03)	1.16 (.04)	1.67 (.18)
Geomean	–	1.04	1.18	1.28	1.30	1.61

In each case, the use of three-operand form with AVX1 confers a slight speedup. The change to use 256 bits with AVX2 gives a further overall improvement, but some mixed results due to the limitations of 256 bit addition. Combining the AVX2 ISA with multithreading gives and average overall 61% speedup compared to base.

6 Conclusion

icGrep demonstrates that predictable high-performance Unicode regular expression search can be achieved using a systematically parallel approach based on bitwise data parallelism. On modern commodity processors with SSE2 or better SIMD instruction sets, performance is dramatically better than that achievable using sequential state-transition methods based on DFAs, NFAs or backtracking. Multithread parallelism further enhances performance using a pipeline parallelism model.

Future research includes the investigation of regular expression matching techniques to handle Unicode level 2 and 3 requirements as well as the extension of optimization techniques to take advantage of MatchStar for more complex repetitions. Beyond regular expression matching, investigation of the bitwise data parallel model for other demanding Unicode processing tasks also seems worthwhile.

References

1. Asanovic, K., Bodik, R., Catanzaro, B.C., Gebis, J.J., Husbands, P., Keutzer, K., Patterson, D.A., Plishker, W.L., Shalf, J., Williams, S.W., et al.: The landscape of parallel computing research: A view from Berkeley. Tech. Rep. UCB/EECS-2006-183, EECS Department, University of California, Berkeley (2006)
2. Cameron, R.D., Amiri, E., Herdy, K.S., Lin, D., Shermer, T.C., Popowich, F.P.: Parallel scanning with bitstream addition: An XML case study. In: Euro-Par, pp. 2–13. Springer (2011)
3. Cameron, R.D., Shermer, T.C., Shriraman, A., Herdy, K.S., Lin, D., Hull, B.R., Lin, M.: Bitwise data parallelism in regular expression matching. In: PACT. pp. 139–150. ACM, New York, NY, USA (2014)
4. Davis, M., Heninger, A.: Unicode technical standard 18, Unicode regular expressions. The Unicode Consortium (2012)
5. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis & transformation. In: Code Generation and Optimization 2004. pp. 75–86. IEEE (2004)
6. Lin, D., Medforth, N., Herdy, K.S., Shriraman, A., Cameron, R.: Parabix: Boosting the efficiency of text processing on commodity processors. In: High Performance Computer Architecture. pp. 1–12. IEEE (2012)
7. Myers, G.: A fast bit-vector algorithm for approximate string matching based on dynamic programming. Journal of the ACM 46(3), 395–415 (1999)
8. Mytkowicz, T., Musuvathi, M., Schulte, W.: Data-parallel finite-state machines. In: ASPLOS. pp. 529–542. ACM (2014)

9. Salapura, V., Karkhanis, T., Nagpurkar, P., Moreira, J.: Accelerating business analytics applications. In: HPCA. pp. 1–10. IEEE (2012)
10. Scarpazza, D.P.: Top-performance tokenization and small-ruleset regular expression matching. International Journal of Parallel Programming 39(1), 3–32 (2011)
11. Stewart, J., Uckelman, J.: Unicode search of dirty data, or: How i learned to stop worrying and love Unicode technical standard # 18. Digital Investigation 10, S116–S125 (2013)
12. Zhao, Z., Wu, B., Shen, X.: Challenging the embarrassingly sequential: parallelizing finite state machine-based computations through principled speculation. In: ASPLOS. pp. 543–558. ACM (2014)
13. Zu, Y., Yang, M., Xu, Z., Wang, L., Tian, X., Peng, K., Dong, Q.: GPU-based NFA implementation for memory efficient high speed regular expression matching. In: PPoPP. pp. 129–140. ACM (2012)