Bitwise Data Parallelism with LLVM: The ICgrep Case Study

Anonymous

No Institute Given

Abstract. Bitwise data parallelism 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, 10X or better speedups are often observed.

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. Valaspura [9] built on these techniques to accelerate business analytics applications using SSE instructions on commodity processors. Zu et al [12] 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. These works have not generally tackled Unicode matching problems.

Using parallel methods to accelerate matching of a single pattern on a single input stream is more difficult. Indeed, of the 13 dwarves identified in the Berkeley overview of parallel computing research, finite state machines (FSMs) are considered the hardest to parallelize (embarassingly sequential) [1]. However, some success has been reported recently along two independent lines of research. Mytkowicz et al [8] use SIMD shuffle operations to implement composable DFA transitions using dynamic convergence to reduce the number of states

in play at any one time and range coalescing to compact the transition tables. Unfortunately, the method seems unlikely to apply well to Unicode regular expression matching problems, which routinely require thousands of DFA states for named Unicode properties. Building on the Parabix framework, Cameron et al. [3] introduce regular expression matching using a new bitwise data parallel approach.

In this paper, we report on the use of the implementation of a full Unicode 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. As an alternative to classical grep implementations, icGrep offers dramatic performance acceleration in Unicode regular expression matching.

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 the 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 contemporary competitors, including the latest versions of GNU grep, pcregrep, ugrep of the ICU (International Component for Unicode) and re2grep. 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 explicit listing of all characters of interest, 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 con-

sortium 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 sequences 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. *Character class streams*, such as `[d]` for the stream that marks the position of "d" characters and `[a-z]` for the stream of lower case ASCII alphabetics are first computed in a fully data-parallel manner. Then the matching process proper begins taking advance of bitwise logic and shifting operations as well as an operation for finding all matches to a character class repetition known as MatchStar. At each step of the process a *marker* bit stream identifies the full set of positions within the input data stream that match the regular expression to this point.

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

Fig. 1: Matching `d[a-z]*ed` Using Bitwise Data Parallelism

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 possibilites 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.

The MatchStar operation turns out to be surprisingly simple [3].

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

A single bit stream addition operation suffices to find all reachable positions from marker stream M through character class stream C. 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].

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 5X 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. The `toUTF8` transformation performs this as a regular expression transformation, transforming input expressions such as '`\u{244}[\u{2030}-\u{2137}]`' to the corresponding UTF-8 regular expression consisting of the series of sequences and alternations shown below:

```
\xE2((\x84[\x80-\xB7])|(([\x81-\x83][\x80-\xBF])|(\x80[\xB0-\xBF])))
```

Unicode Advance. As illustrated in Section 2, a bitwise shift (Advance) operation is frequently used in shifting a marker stream from positions matched by one regular expression element to the next. However, characters are represented by

variable-length byte sequences in UTF-8, so that a simple shift operation is inadequate to implement the operation of advancing bit stream positions from one Unicode character to the next.

In order to address the requirements of Unicode advance, 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. 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_{ni3} is the bitstream that marks character *ni3* and CC_{hao} is the bitstream that marks character stream *hao*. To match a two UTF-8 character sequence *ni3hao*, we first create an *Initial* steam that marks the first byte of all the valid characters. We also produce a *NonFinal* stream that marks every byte of all the multibyte characters except for the last byte. Using *Initial* to ScanThru *NonFinal*, we get the bitstream M_2, which marks the positions of the last byte of every character. An overlap between M_2 and CC_{ni3} gives the start position for matching the next character. As illustrated by *Adv*, we find two matches for *ni3* and from these two positions we can start the matching process for the next character *hao*. The final result stream shows 1 match for the multibyte sequence ni3hao.

input data	`ni3hao(Hello),ni3men(You),`
CC_{ni3}	`..1.............1.........`
CC_{hao}	`.....1....................`
Initial	`1..1..111111111..1..111111`
NonFinal	`11.11.........11.11.......`
$M_1 = \mathit{ScanThru}(\mathit{Initial}, \mathit{NonFinal})$	`..1..111111111..1..1111111`
$\mathit{Adv} = \mathit{Advance}(M_1 \wedge CC_{\text{ni3}})$	`...1.............1........`
$M_2 = \mathit{ScanThru}(\mathit{Initial} \wedge \mathit{Adv}, \mathit{NonFinal})$	`.....1.............1......`
$\mathit{match} = M_2 \wedge CC_{\text{hao}}$	`.....1....................`

Fig. 2: Processing of a Multibyte Sequence ni3hao

Unicode MatchStar. The $\mathit{MatchStar}(M, C)$ 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 the carry propagation within the MatchStar operation may terminate prematurely.

In order to remedy this problem, icGrep again uses the two helper bitstreams *Initial* and *NonFinal*. Any full match to a multibyte sequence must reach the initial position of the next character. The *NonFinal* bitstream consists of all

positions except those that are final positions of UTF-8 sequences. It is used to "fill in the gaps" in the CC 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 C (with a 1 bit set at final positions of any members of the class), can be implemented using $MatchStar(M, C|NonFinal)$.

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 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

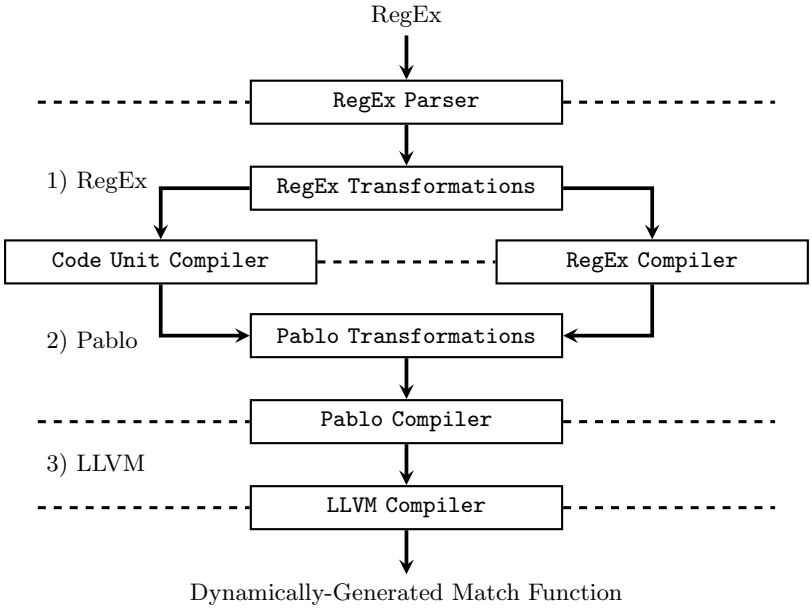

Fig. 3: icGrep Architectural Diagram

Regular Expression Preprocessing. As shown in Figure 3, icGrep comprises three logical layers: RegEx, Pablo and the LLVM layer, each with their own inter-

mediate representation (IR), transformation and compilation modules. As we traverse the layers, the IR becomes significantly more complex as it begins to mirror the final machine code. The RegEx Parser validates and transforms the input RegEx into an abstract syntax tree (AST). The initial *Nullable* pass, determines whether the RegEx contains any prefixes or suffixes that may be removed or modified whilst still providing the same number of matches as the original expression. For example, "`a*bc+`" is equivalent to "`bc`" because the Kleene Star (Plus) operator matches zero (one) or more instances of a specific character. The *toUTF8* pass converts the Unicode character classes in the input RegEx into the equivalent expression(s) that represent the sequences of 8-bit code units necessary to identify occurrences of the class. A final *Simplification* pass flattens nested structures into their simplest legal form. For example, "`a(b((c|d)|e))`" becomes "`ab(c|d|e)`" and "`([0-9]{3,5}){3,5}`" becomes "`[0-9]{9,25}`".

The RegEx layer has two compilers: the Code Unit Compiler and RegEx Compiler, both of which produce Pablo IR. Recall that the Pablo layer assumes a transposed view of the input data. The *Code Unit Compiler* transforms the input code unit classes, either extracted from the RegEx or produced by the *toUTF8* transformation, into a series of bit stream equations. The *RegEx Compiler* assumes that these have been calculated and transforms the RegEx AST into a sequence of instructions. For instance, it would convert any alternations into a sequence of calculations that are merged with `OR`s. The results of these passes are combined and transformed through a series of typical optimization passes, including dead code elimination (DCE), common subexpression elimination (CSE), and constant folding. These are necessary at this stage because the RegEx AST may include common subsequences that are costly to recognize in that form. Similarly, to keep the Code Unit Compiler a linear time function, it may introduce redundant IR instructions as it applies traditional Boolean algebra transformations, such as de Morgan's law, to the computed streams. An intended side-effect of these passes is that they eliminate the need to analyze the data-dependencies inherent in the carry-bit logic, which is necessary for some Pablo instructions but problematic for optimizers to reason about non-conservatively. The Pablo Compiler then converts the Pablo IR into LLVM IR. This is a relatively straightforward conversion: the only complexities it introduces is the generation of Phi nodes, linking of statically-compiled functions, and assignment of carry variables. It produces the dynamically-generated match function used by the icGrep.

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

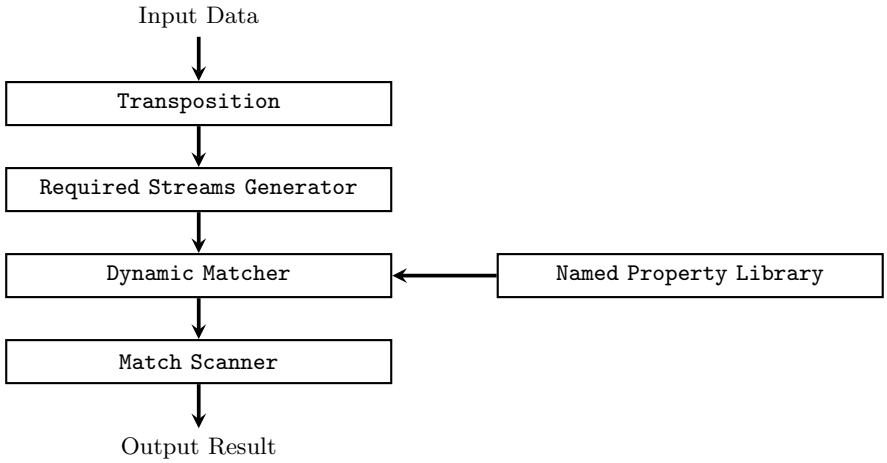

Fig. 4: icGrep Execution Diagram

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.

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 susequent 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 Required Streams Generator.

5 Evaluation

In this section, we report on the evaluation of ICgrep performance, looking at three aspects. First we consider a performance studies in a series of Unicode regular expression search problems in comparison to the contemporary competitors, including pcre2grep released in January 2015 and ugrep of the ICU 54.1 software distribution. Then we move on to investigate some performance as-

pects of ICgrep internal methods, looking at the impact of optimizations and multithreading.

5.1 Simple Property Expressions

A key feature of Unicode level 1 support in regular expression engines is how 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. On the other hand, `pcre2grep` does not support the set intersection and difference operators directly. However, these operators can instead be expressed using a regular expression feature known as a lookbehind assertion. Set intersection involves a regular expression formed with a one of the property expressions and a positive lookbehind assertion on the other, while set difference uses a negative lookbehind assertion.

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 perform searches for each regular expression against each XML document. Results are presented in Figure 5. Performance is reported in CPU cycles per byte on an Intel Core i7 machine. The results were grouped by the percentage of matching lines found in the XML document, grouped in 5% increments. ICgrep shows dramatically better performance, particularly when searching for rare items. As shown in the figure, pcre2grep and ugrep both show increased performance (reduced CPU cycles per byte) with increasing percentage of matches found. In essence, each match found allows these programs to skip the full processing of 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.

5.2 Complex Expressions

We also comparative performance of the matching engines on a series of more complex expressions as shown in Table 1.

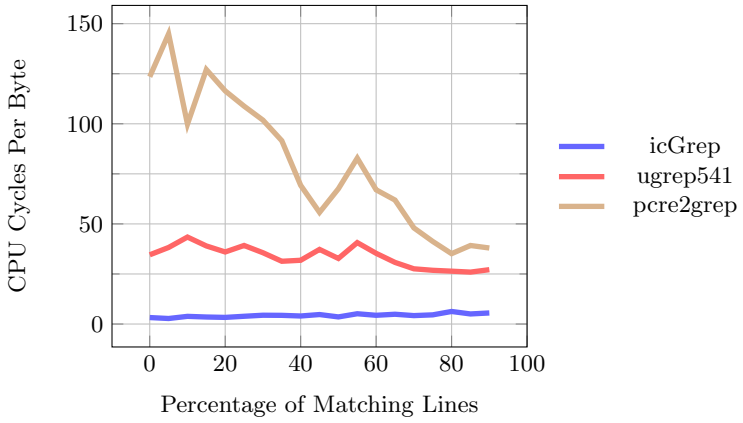

Fig. 5: Matching Performance for Simple Property Expressions

Regular	CPU cycles per byte		
Expression	icGrep	pcre2grep	ugrep
blah	1	1000	100

Table 1: Matching Times for Complex Expressions

5.3 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 relatively 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 maximums throughout the file. But when search for XML tags using the regular expression `<[^!?][^>]*>`, a slowdown of more than 2X may be found in files with many long tags.

The `-disable-log2-bounded-repetition` flag allows these effectiveness of the special techniques for bounded repetition of byte classes to be assessed. A slowdown of 30% was observed with the searches using the regular expression `(^|[ ])[a-zA-Z]{11,33}([.!? ]|$)`, for example.

To control the insertion of if-statements into dynamically generated code, the number of non-nullable pattern elements between the if-tests can be set with the `-if-insertion-gap=` option. The default value in icGrep is 3, setting

the gap to 100 effectively turns of if-insertion. Eliminating if-insertion sometimes improves performance by avoiding the extra if tests and branch mispredications. For patterns with long strings, however, there can be a substantial slowdown; searching for a pattern of length 40 slows down by more than 50% without the if-statement short-circuiting.

ICgrep also provides options that allow various internal representations to be printed out. These can aid in understanding and/or debugging performance issues. For example, the option `-print-REs` show the parsed regular expression as it goes through various transformations. The internal Pablo code generated may be displayed with `-print-pablo`. This can be quite useful in helping understand the match process. It also possible to print out the generated LLVM IR code (`-dump-generated-IR`), but this may be less useful as it includes many details of low-level carry-handling that obscures the core logic.

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 5X.

5.4 Single vs. Multithreaded Performance

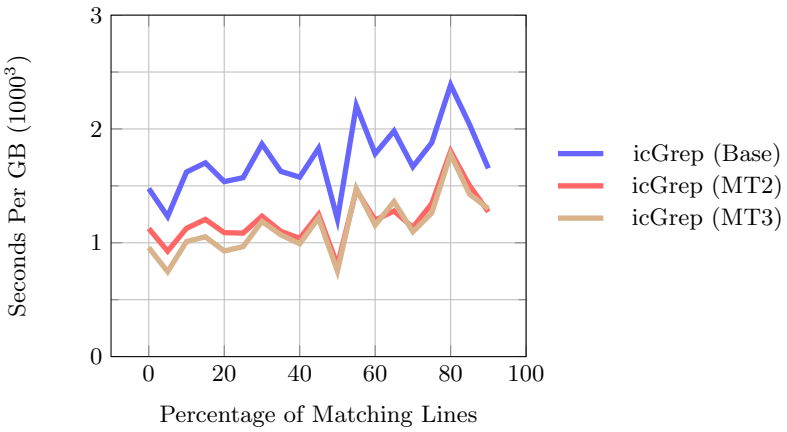

Fig. 6: Multithreading Performance

6 Conclusion

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., Technical Report 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 2011 Parallel Processing, 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: Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT). pp. 139–150. PACT '14, ACM, New York, NY, USA (2014), http://doi.acm.org/10.1145/2628071.2628079
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. CGO 2004. International Symposium on. 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 (HPCA), 2012 IEEE 18th International Symposium on. 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 (JACM) 46(3), 395–415 (1999)
8. Mytkowicz, T., Musuvathi, M., Schulte, W.: Data-parallel finite-state machines. In: Proceedings of the 19th international conference on Architectural support for programming languages and operating systems. pp. 529–542. ACM (2014)
9. Salapura, V., Karkhanis, T., Nagpurkar, P., Moreira, J.: Accelerating business analytics applications. In: High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on. 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. 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 '12 - Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming. pp. 129–140. ACM (2012)