Further Advances in Parabix Technology for High-Performance Unicode

Robert D. Cameron

School of Computing Science
Simon Fraser University

November 3, 2016

Outline

Outline

Parabix Technology

Parabix Concept

- Parabix employs *bitwise data parallelism* to achieve high-performance text processing.
- XML parsing [HPCA 2012].
- Regular expression matching [PACT 2014].
- Process 128 bytes at a time using 128-bit SSE2 registers on all Intel/AMD 64-bit processors.
- Process 256 bytes at a time using 256-bit AVX2 technology.

Parabix Technology

Parabix Concept

- Parabix employs *bitwise data parallelism* to achieve high-performance text processing.
- XML parsing [HPCA 2012].
- Regular expression matching [PACT 2014].
- Process 128 bytes at a time using 128-bit SSE2 registers on all Intel/AMD 64-bit processors.
- Process 256 bytes at a time using 256-bit AVX2 technology.

Parabix Regular Expression Software

- `icgrep 1.0` employs Parabix methods in a full Unicode Level 1 "grep" search tool [IUC39, ICAPP2015].
- Gigabyte/sec regular expression search.

Recent Advances

Parabix Toolchain

- 100% dynamic compilation to LLVM IR.
- Dynamic processor detection for AVX2.
- Can target NVPTX back end (Nvidia GPUs).
- Application construction using stream-processing kernels.
- Multicore processing using segmented pipeline parallelism.

Recent Advances

Parabix Toolchain

- 100% dynamic compilation to LLVM IR.
- Dynamic processor detection for AVX2.
- Can target NVPTX back end (Nvidia GPUs).
- Application construction using stream-processing kernels.
- Multicore processing using segmented pipeline parallelism.

New Regular Expression Features

- Unicode Name Reflection
- Property Boundary Experessions

Technology Roadmap: Parabix OS

Parabix Shell plus Core Utilities

- Parabix versions of grep, sed, awk, cut, wc, head, tail, join, ...
- Parabix shell: dynamic pipelining using pipeline parallelism.
- Goal: high performance OS for big data applications.
- Compression, transcoding, etc., built-in.
- Design for use with Linux or Darwin kernel.

Technology Roadmap: Parabix OS

Parabix Shell plus Core Utilities

- Parabix versions of grep, sed, awk, cut, wc, head, tail, join, ...
- Parabix shell: dynamic pipelining using pipeline parallelism.
- Goal: high performance OS for big data applications.
- Compression, transcoding, etc., built-in.
- Design for use with Linux or Darwin kernel.

Parabix Components for Unicode

- Integrate high-level Unicode awareness into all core utilities.
- Unicode regular expression support throughout.
- What else? (A call for advice/collaboration!)

Outline

Bitwise Data Parallelism: Character Classes

Consider matching regexp `A[a-z]*e;` against the input text below. First we make some character class bit streams.

input data `Axe; Apples; A badApple; Accede; Ate!`

Bitwise Data Parallelism: Character Classes

Consider matching regexp `A[a-z]*e;` against the input text below. First we make some character class bit streams.

- `[A]`

```
input data   Axe;  Apples; A badApple;  Accede; Ate!
   [A]     1....1.......1...1.......1.......1...
```

Bitwise Data Parallelism: Character Classes

Consider matching regexp `A[a-z]*e;` against the input text below. First we make some character class bit streams.

- `[A]`
- `[a-z]`

```
input data   Axe;  Apples; A badApple;  Accede; Ate!
   [A]     1....1.......1...1.......1.......1...
  [a-z]    .11....11111....111.1111....11111...11.
```

Bitwise Data Parallelism: Character Classes

Consider matching regexp `A[a-z]*e;` against the input text below. First we make some character class bit streams.

- `[A]`
- `[a-z]`
- `[e]`

```
input data   Axe;  Apples; A badApple;  Accede; Ate!
   [A]     1....1.......1...1.......1.......1...
  [a-z]    .11....11111....111.1111....11111...11.
   [e]     ..1.......1.............1......1.1....1.
```

Bitwise Data Parallelism: Character Classes

Consider matching regexp `A[a-z]*e;` against the input text below. First we make some character class bit streams.

- `[A]`
- `[a-z]`
- `[e]`
- `[;]`

```
input data   Axe;  Apples; A badApple;  Accede; Ate!
   [A]     1....1.......1...1.......1.......1...
  [a-z]    .11....11111....111.1111....11111...11.
   [e]     ..1.......1.............1......1.1....1.
   [;]     ...1........1...........1........1.....
```

Regular Expression Matching Equations

Matching equations for `A[a-z]*e`, produce *marker bit streams* indicating positions after a match.

input data `Axe;  Apples; A badApple;  Accede; Ate!`

Regular Expression Matching Equations

Matching equations for `A[a-z]*e`, produce *marker bit streams* indicating positions after a match.

- $M_1 =$ Advance(`[A]`) — *positions after* `[A]`

```
input data   Axe;  Apples; A badApple;  Accede; Ate!
   M1        .1....1......1...1......1......1...
```

Regular Expression Matching Equations

Matching equations for `A[a-z]*e`, produce *marker bit streams* indicating positions after a match.

- $M_1 = \text{Advance}(\texttt{[A]})$ — *positions after* `[A]`
- $M_2 = \text{MatchStar}(M_1, \texttt{[a-z]})$ — *all positions matching* `A[a-z]*`

```
input data   Axe;  Apples; A badApple;  Accede; Ate!
   M1        .1.....1.......1....1.......1.......1...
   M2        .111...111111..1....11111...111111..111.
```

Regular Expression Matching Equations

Matching equations for `A[a-z]*e`, produce *marker bit streams* indicating positions after a match.

- $M_1 = \text{Advance}(\texttt{[A]})$ — *positions after* `[A]`
- $M_2 = \text{MatchStar}(M_1, \texttt{[a-z]})$ — *all positions matching* `A[a-z]*`
- $M_3 = \text{Advance}(M_2 \wedge \texttt{[e]})$ — *all positions matching* `A[a-z]*e`

```
input data   Axe;  Apples; A badApple;  Accede; Ate!
   M1        .1....1.......1...1.......1.......1...
   M2        .111...111111..1....11111...111111..111.
   M3        ...1.......1.............1......1.1....1.
```

Regular Expression Matching Equations

Matching equations for `A[a-z]*e`, produce *marker bit streams* indicating positions after a match.

- $M_1 = \text{Advance}(\texttt{[A]})$ — *positions after* `[A]`
- $M_2 = \text{MatchStar}(M_1, \texttt{[a-z]})$ — *all positions matching* `A[a-z]*`
- $M_3 = \text{Advance}(M_2 \wedge \texttt{[e]})$ — *all positions matching* `A[a-z]*e`
- $M_4 = \text{Advance}(M_3 \wedge \texttt{[;]})$ — *all positions matching* `A[a-z]*e;`

```
input data   Axe;  Apples; A badApple;  Accede; Ate!
   M1        .1....1......1...1......1......1...
   M2        .111...111111..1....11111...111111..111.
   M3        ...1......1...........1.....1.1....1.
   M4        ....1..................1.......1.....
```

MatchStar Definition

- MatchStar$(M, C) = (((M \wedge C) + C) \oplus C) \vee M$

MatchStar Definition

- MatchStar$(M, C) = (((M \wedge C) + C) \oplus C) \vee M$
- Consider $M_2 =$ MatchStar$(M_1,$ [a-z])

input data	`Axe;  Apples; A badApple;  Accede; Ate!`
M_1	`.1.....1.......1....1.......1.......1...`
$C =$ [a-z]	`.11....11111....111.1111....11111...11..`

MatchStar Definition

- MatchStar$(M, C) = (((M \wedge C) + C) \oplus C) \vee M$
- Consider $M_2 =$ MatchStar$(M_1,$ [a-z] $)$
- Use addition to scan each marker through the class.

input data	`Axe;  Apples; A badApple;  Accede; Ate!`
M_1	`.1....1.....1...1.....1.....1...`
$C =$ [a-z]	`.11....11111....111.1111....11111...11..`
$T_0 = M_1 \wedge C$	`.1....1.....1.....1.....1...`
$T_1 = T_0 + C$	`...1.....1...111....1.....1....1.`

MatchStar Definition

- MatchStar$(M, C) = (((M \wedge C) + C) \oplus C) \vee M$
- Consider $M_2 =$ MatchStar$(M_1,$ [a-z] $)$
- Use addition to scan each marker through the class.
- Bits that change represent matches.

input data	`Axe;  Apples; A badApple;  Accede; Ate!`
M_1	`.1.....1.......1....1.......1.......1...`
$C =$ [a-z]	`.11....11111....111.1111....11111...11..`
$T_0 = M_1 \wedge C$	`.1.....1............1.......1.......1...`
$T_1 = T_0 + C$	`...1........1...111.....1........1....1.`
$T_2 = T_1 \oplus C$	`.111...111111.......11111...111111..111.`

MatchStar Definition

- MatchStar$(M, C) = (((M \wedge C) + C) \oplus C) \vee M$
- Consider $M_2 =$ MatchStar$(M_1,$ [a-z] $)$
- Use addition to scan each marker through the class.
- Bits that change represent matches.
- We also have matches at start positions in M_1.

input data	`Axe;  Apples; A badApple;  Accede; Ate!`
M_1	`.1.....1.......1....1.......1.......1...`
$C =$ [a-z]	`.11....11111....111.1111....11111...11..`
$T_0 = M_1 \wedge C$	`.1.....1.............1.......1.......1...`
$T_1 = T_0 + C$	`...1........1...111.....1........1....1.`
$T_2 = T_1 \oplus C$	`.111...111111.......11111...111111..111.`
$M_2 = T_2 \vee M_1$	`.111...111111..1....11111...111111..111.`

Beyond Byte-At-A-Time

- Byte-oriented data is first transformed to 8 parallel bit streams (Parabix transform).
  - Bit stream j consists of bit j of each byte.
  - Uses SIMD pack operations on SSE2, AVX2, etc.
  - About 0.5 CPU cycles/byte amortized cost (AVX2).
- Compute character class bit streams with bitwise logic.
  - `[A]`, `[a-z]`, `[e]`, `[;]` about 0.2 CPU cycles/byte (AVX2).
- Implement matching using bitwise logic, shifting and addition.
  - `A[a-z]*e;` about 0.3 CPU cycles/byte (AVX2).
- Performance beyond the best achievable by byte-at-a-time methods.

Unbounded Stream Abstraction

- Program operations as if *all positions in the file are to be processed simultaneously*.
- Unbounded bitwise parallelism.
- Pablo compiler technology maps to block-by-block processing.
- Information flows between blocks using carry bits.
- LLVM compiler infrastructure for Just-In-Time compilation.
- Custom LLVM improvements further accelerate processing.

icgrep Features

Unicode Level 1 (UTS #18)

- General category (e.g., `\p{Lu}`), script (e.g. `\p{Greek}`), and other core Unicode properties.
- Simple Unicode line breaks, word boundaries.
- Set operations, e.g., `[\p{Greek}&&\p{Lu}]`
- Simple Unicode case-insensitive matching.
- Complete implementation of UTS #18 level 1 requirements.

icgrep Features

Unicode Level 1 (UTS #18)

- General category (e.g., `\p{Lu}`), script (e.g. `\p{Greek}`), and other core Unicode properties.
- Simple Unicode line breaks, word boundaries.
- Set operations, e.g., `[\p{Greek}&&\p{Lu}]`
- Simple Unicode case-insensitive matching.
- Complete implementation of UTS #18 level 1 requirements.

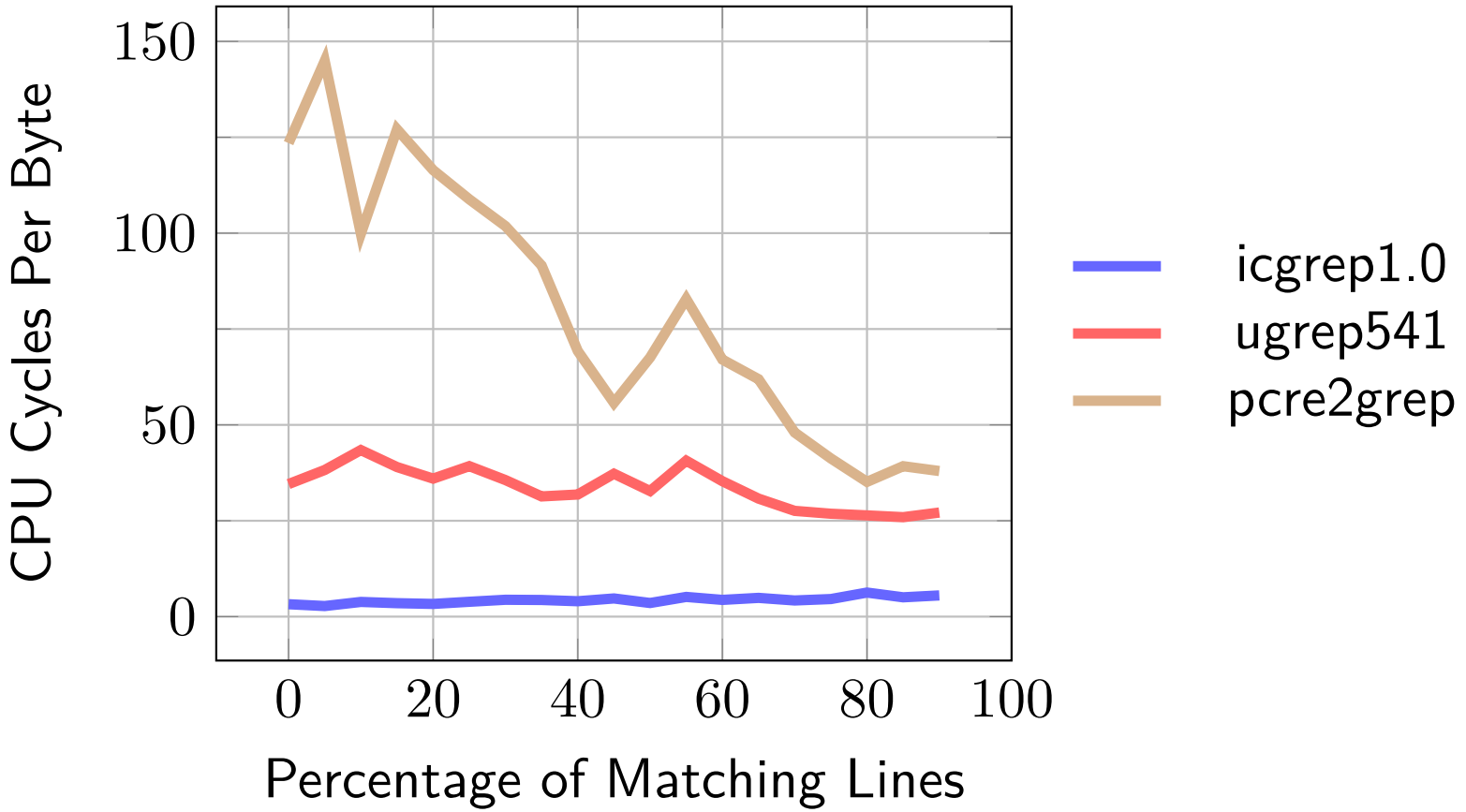
Unicode Level 2

- Grapheme clusters and grapheme cluster mode.
- Name property with regexp values `\N{SMIL(E|ING)}` (emoji search!)
- Broad set of Unicode general, numeric, identifier, case, normalization, shaping, bidirectional, CJK and other properties.

Performance Study

- Search for lines containing characters in computed sets.
- Ex: `[\p{Greek}&&\p{upper}]`
- Set expressions combine general category and script properties.
- Set intersection, union, difference, negation.
- Cover all general categories and scripts.
- 246 test expressions in all.
- Evaluate icgrep 1.0 vs. ugrep541 and pcre2grep.
- Measure performance against a large collection of Wikimedia documents.

Search Time Comparison



Outline

Parabix As a Platform

The Questions

- Is icgrep an interesting one-off application or just one example of a general approach?
- How can Parabix techniques be assembled into a programming framework for high-performance applications?
- How do we produce high-quality high-peformance software without tedious, error-prone and non-portable use of SIMD intrinsics?
- Can the legacy conventions of Posix systems be systematically updated to create an OS with first-class Unicode support?

Parabix As a Platform

The Questions

- Is icgrep an interesting one-off application or just one example of a general approach?
- How can Parabix techniques be assembled into a programming framework for high-performance applications?
- How do we produce high-quality high-peformance software without tedious, error-prone and non-portable use of SIMD intrinsics?
- Can the legacy conventions of Posix systems be systematically updated to create an OS with first-class Unicode support?

Human Resource Issues

- Where can we find software engineers capable of Parabix development?
- Unfortunately, open source is not the answer.
- Graduate students can only provide part of the solution.

Lessons Learned

Avoid Sequential APIs

- Traditional regexp engines provide programmability through APIs.
- But APIs are sequential, fundamentally limiting performance.
- Do not repeat the mistake of icXML.
  - Heroic SIMD/Parabix acceleration of Xerces C++ XML parser.
  - Fighting Amdahl's Law all the way.
  - 2X speedup achieved.
  - Fundamentally limited by Xerces C++ APIs.

Lessons Learned

Avoid Sequential APIs

- Traditional regexp engines provide programmability through APIs.
- But APIs are sequential, fundamentally limiting performance.
- Do not repeat the mistake of icXML.
  - Heroic SIMD/Parabix acceleration of Xerces C++ XML parser.
  - Fighting Amdahl's Law all the way.
  - 2X speedup achieved.
  - Fundamentally limited by Xerces C++ APIs.

Core Utilities

- Consider core utilities, composed with Unix pipes.
- Dynamic compilation allows efficiencies in pipe stage composition.
- Fundamentally parallel core utilities avoid sequential bottlenecks.

Parabix Languages and Compilers

Languages

- Grammars: regexps, character classes, Unicode properties.
- Pablo stream language: operations on arbitrary-length streams.
- LLVM IR: high-level assembly language for stream processing *kernels*.

Parabix Languages and Compilers

Languages

- Grammars: regexps, character classes, Unicode properties.
- Pablo stream language: operations on arbitrary-length streams.
- LLVM IR: high-level assembly language for stream processing *kernels*.

Compilers

- Character class compiler: produce Pablo code for any explicit character class.
- Unicode property compiler: Pablo code for any Unicode property.
- Regexp compiler: produce Pablo code for any regular expression.
- Pablo compiler: produce Pablo Kernels in LLVM IR.
- Kernel Pipeline Compilers: produce IR from a chain of kernels.
- LLVM JIT code generators translate IR to executable code.

Kernels

Kernel Structure

- Kernels are computational abstractions for text stream processing.
- Kernels process input stream sets, producing output stream sets.

Kernels

Kernel Structure

- Kernels are computational abstractions for text stream processing.
- Kernels process input stream sets, producing output stream sets.

Transposition Kernel

- Input: $1 \times$ `i8`: a single stream of 8-bit code units (e.g., UTF-8).
- Output: $8 \times$ `i1`: a set 8 of parallel bit streams (basis bit streams).

Kernels

Kernel Structure

- Kernels are computational abstractions for text stream processing.
- Kernels process input stream sets, producing output stream sets.

Transposition Kernel

- Input: $1 \times \texttt{i8}$: a single stream of 8-bit code units (e.g., UTF-8).
- Output: $8 \times \texttt{i1}$: a set 8 of parallel bit streams (basis bit streams).

Transposition Subkernels

- Transposition can actually be divided into 3 stages.
- Stage 1: $1 \times \texttt{i8}$: to $2 \times \texttt{i4}$ (2 streams of nybbles).
- Stage 2: $2 \times \texttt{i4}$: to $4 \times \texttt{i2}$ (4 streams of bit-pairs).
- Stage 3: $4 \times \texttt{i2}$: to $8 \times \texttt{i1}$ (basis bit streams).

Regular Expression Kernels

Character Class Kernels

- Kernel for the character classes of a regexp: e.g., `a[0-9]*[z9]`
- Input: $8 \times$ `i1`: the 8 basis bit streams.
- Output: $3 \times$ `i1`: 3 bit streams for `[a]`, `[0-9]`, `[z9]`
- Dynamically generated by the Parabix Character Class compiler.

Regular Expression Kernels

Character Class Kernels

- Kernel for the character classes of a regexp: e.g., `a[0-9]*[z9]`
- Input: $8 \times$ `i1`: the 8 basis bit streams.
- Output: $3 \times$ `i1`: 3 bit streams for `[a]`, `[0-9]`, `[z9]`
- Dynamically generated by the Parabix Character Class compiler.

Matching Logic Kernels

- Kernel for the matching logic: e.g., `a[0-9]*[z9]`
- Input: $3 \times$ `i1`: character class streams
- Output: $1 \times$ `i1`: a bit stream of matches found.
- Dynamically generated by the Parabix Regular Expression compiler.

The icgrep Kernels

Line Break Kernel

- Kernel for Unicode line breaks
- Input: $8 \times$ `i1`: the 8 basis bit streams.
- Output: $1 \times$ `i1`: line breaks for any of LF, CR, CRLF, LS, PS, $\cdots$

The icgrep Kernels

Line Break Kernel

- Kernel for Unicode line breaks
- Input: $8 \times$ `i1`: the 8 basis bit streams.
- Output: $1 \times$ `i1`: line breaks for any of LF, CR, CRLF, LS, PS, $\cdots$

Match Scanning Kernel

- Kernel to generate matched lines.
- Three inputs:
  - $1 \times$ `i8`: source byte stream
  - $1 \times$ `i1`: matches bit stream
  - $1 \times$ `i1`: line break bit stream
- Output: $1 \times$ `i8` matched line output stream.

Parallel Deletion Kernels

Bit Stream Compression Kernel

- Two inputs:
  - `Nxi1`: bit streams to compress
  - $1 \times$ `i1`: deletion mask stream
- Output: `Nxi1`: compressed output streams
- Example:

input[1]	10101000101010101011
input[2]	11100111100000110001
deletion mask	00111000001101000110
output[1]	100001011011
output[2]	111111001101

- Provides a general approach to stream filtering.

The icgrep pipeline

A 5-Stage Pipeline

- BasisBits = Transpose(ByteData)
- LineBreaks = UnicodeLineBreaks(BasiBits)
- CharacterClasses = CC_compiler<regexp>(BasisBits)
- MatchPositions = RE_compiler<regexp>(CharacterClasses)
- FinalOutput = MatchScanner(ByteData, LineBreaks, MatchPositions)

The icgrep pipeline

A 5-Stage Pipeline

- BasisBits = Transpose(ByteData)
- LineBreaks = UnicodeLineBreaks(BasiBits)
- CharacterClasses = CC_compiler<regexp>(BasisBits)
- MatchPositions = RE_compiler<regexp>(CharacterClasses)
- FinalOutput = MatchScanner(ByteData, LineBreaks, MatchPositions)

Pipeline Compilation

- The Parabix pipeline compiler builds the complete icgrep runtime.
- Buffers are allocated for all streams.
- Internal states allocated for all kernels.
- Kernels are compiled to process data in defined buffers.
- icgrep, word count, u8u16 now compile with this structure.

Experimental Pipeline Compilers

Pipeline Parallel Compiler

- Each kernel is compiled to a separate thread function.
- Kernels are synchronized through producer/consumer positions.
- Lock-free synchronization through monotonic positions.
- Functional, but balance between pipeline stages is problematic.

Experimental Pipeline Compilers

Pipeline Parallel Compiler

- Each kernel is compiled to a separate thread function.
- Kernels are synchronized through producer/consumer positions.
- Lock-free synchronization through monotonic positions.
- Functional, but balance between pipeline stages is problematic.

NVPTX Pipeline Compiler

- Kernels compiled to PTX code to run on NVidia GPUs.
- Can now compile first 4 icgrep stages to GPU.
- Currently only a single workgroup of 64 threads: 4096 position SIMT.
- MatchedLineScanner compiles to CPU.

Segmented Pipeline Parallelism

Combined Data and Pipeline Parallelism

- Input divided into logical segments.
- Allocate segments to P cores in round robin fashion.
- Core i responsible for all segments n such that $n \mod P = i$.
- Each core executes a full pipeline for its segment.
- For any pipeline stage s and segment $i+1$, core $(i+1) \mod P$ can proceed as soon as core $i \mod P$ completes stage i.
- Workload balanced between cores as long as no stage requires more than $1/P$ of the total time to process a segment.
- In progress $\cdots$

Outline

Proposal 1: Unicode Name Reflection

Loose Matches: Case Insensitivity

- Simple caseless matching: `(?i:*Viagara*)`
- Matches `*viagara*`, `*Viagara*`, `*ViAgArA*`, ···

Proposal 1: Unicode Name Reflection

Loose Matches: Case Insensitivity

- Simple caseless matching: `(?i:*Viagara*)`
- Matches `*viagara*`, `*Viagara*`, `*ViAgArA*`, ···

Loose Matches: Name Reflection

- Use `N` mode flag: `(?N:*Viagara*)`
- Each literal regexp character in `N` mode is replaced by its name pattern expression within word boundaries.
- `(?N:V)` = `\p{name=/\bLATIN CAPITAL LETTER V\b/}`, matching:
  - `CIRCLED LATIN CAPITAL LETTER V`: Ⓥ
  - `LATIN CAPITAL LETTER V WITH HOOK`: Ʋ
  - `LATIN CAPITAL LETTER V WITH TILDE`: Ṽ
  - `NEGATIVE SQUARED LATIN CAPITAL LETTER V`: 🆅
  - as well as V and 7 other codepoints.

Name Reflection

Caseless Name Reflection

- Combine `(?i)` and `(?N)` modes, e.g., `(?iN:*Viagara*)`
- Matches *viAgara*, ✺Ṽ(ⁱ)ⓐ𝔤ạ🅁a✻, ···

Name Reflection

Caseless Name Reflection

- Combine (?i) and (?N) modes, e.g., (?iN:*Viagara*)
- Matches *viAgara*, ✺Ṽ⒤ⓐǥậ🅁a✻, · · ·

Level 2 Name Reflection

- Combined with grapheme cluster mode, (?gN) matches to include named character sequences from NamedSequences.txt.
- (?N:R) can also match the sequence: LATIN CAPITAL LETTER R WITH TILDE;0052 0303
- Also consider (?K) compatibility mode flag.

Future Extensions

Named Substring or Edited Name Reflection

- Potential use case: extended loose name reflection ignoring the word `LATIN`.
- Use `///` match/replace syntax, possibly.
- For example `(?N/LATIN//:a)` matches additional codepoints.
  - 0430 CYRILLIC SMALL LETTER A
  - 04D1 CYRILLIC SMALL LETTER A WITH BREVE
  - 04D3 CYRILLIC SMALL LETTER A WITH DIAERESIS
  - 2090 LATIN SUBSCRIPT SMALL LETTER A
  - AB70 CHEROKEE SMALL LETTER A
  - 104D8 OSAGE SMALL LETTER A
  - 10CC0 OLD HUNGARIAN SMALL LETTER A
  - 118C1 WARANG CITI SMALL LETTER A

Proposal 2: Property Boundary Expressions

Simple Boundary Expressions

- For any binary property `X`, `\b{X}` is equivalent to:
  `(?<\p{X})(?=\P{X})|(?<\P{X})(?=\p{X})`.
- Similarly, for any property-value combination `X=a`, `\b{X=a}` ≡ `(?<\p{X=a})(?=\P{X=a})|(?<\P{X=a})(?=\p{X=a})`.
- As usual, `sc=` may be omitted for any script value, e.g. `\b{Common}`.
- Also, `gc=` may be omitted, except for `\b{gc=l}` and `\b{gc=s}` (conflict with UTS #18 line and sentence boundaries).

Proposal 2: Property Boundary Expressions

Simple Boundary Expressions

- For any binary property `X`, `\b{X}` is equivalent to:
  `(?<\p{X})(?=\P{X})|(?<\P{X})(?=\p{X})`.
- Similarly, for any property-value combination `X=a`, `\b{X=a}` ≡ `(?<\p{X=a})(?=\P{X=a})|(?<\P{X=a})(?=\p{X=a})`.
- As usual, `sc=` may be omitted for any script value, e.g. `\b{Common}`.
- Also, `gc=` may be omitted, except for `\b{gc=l}` and `\b{gc=s}` (conflict with UTS #18 line and sentence boundaries).

Character Class Boundary Expressions

- The concept extends to character classes, e.g., `\b{[a-f]}` ≡ `(?<[a-f])(?=[^a-f])|(?<[^a-f])(?=[a-f])`.
- The character class expression may involve properties, intersection, difference ...

Property Boundary Expressions

Level 2: Generalized Boundary Expressions

- Now consider the generalization for any property, for example script boundaries `\b{Script}`.
- Simple concept: the boundary assertion is satisfied if the property values of the characters on either side of the boundary are *different*.
- Equivalent regexp: a union of property boundary assertions for *all* values of the property.
- Use case: search for spoofing: `\p{alpha}\b{Script}\p{alpha}`
- Parabix implementation straightforward:
  - Encode N property values with $log_2 N$ bitstreams v_i.
  - Compute

$$\bigvee_{i=0}^{\lceil \log_2 N \rceil} v_i \oplus \text{Advance}(v_i)$$

Outline

Unicode Property Support

Binary Property Kernels

- Produced by Unicode property compiler.
- Input: $8 \times$ `i1`: basis bit streams (UTF-8) or `16 x i1` (UTF-16) for input text.
- Output: $1 \times$ `i1`: text positions (final code unit) for codepoints having the property.

Unicode Property Support

Binary Property Kernels

- Produced by Unicode property compiler.
- Input: $8 \times$ `i1`: basis bit streams (UTF-8) or `16 x i1` (UTF-16) for input text.
- Output: $1 \times$ `i1`: text positions (final code unit) for codepoints having the property.

Enumerated Property Kernels

- Produced by Unicode property compiler.
- For property=value syntax, similar to binary property: $1 \times$ `i1` output.
- Property=/regexp/ syntax, $1 \times$ `i1` output.
- Full property information for enumeration of N possible values.
  - Can produce $N \times$ `i1` separate streams.
  - Multiplexed representation: $\lceil \log_2 N \rceil \times$ `i1` streams.

Unicode Property Support

Codepoint Property Kernels

- Codepoint properties: the value of a given property for an input codepoint c is some other codepoint $c' = P(c)$.
- Output: up to $21 \times$ `i1:` streams.
- Each stream P_i indicates whether bit i changes between input and output codepoint.
- Often fewer than 21 streams required.
- Can be used for parallel substitution: $y_i = x_i \oplus P_i$.
- Efficient simple-case folding.

Transcoding Kernels

UTF-8 to UTF-16 Logic Kernel

- Input: $8 \times$ `i1`: the 8 basis bit streams.
- Three outputs:
  - $16 \times$ `i1`: UTF-16 parallel bit streams
  - $1 \times$ `i1`: deletion mask stream
  - $1 \times$ `i1`: UTF-8 error stream
- Only one logical output code unit position for 2 or 3 byte UTF-8 sequence, 2 positions for 4-byte sequences.
- Deletion mask marks positions to be removed from output stream.

Transformation Kernels (Future)

Case Conversion Notes

- Apply simple case conversion using parallel logic.
- Compute bit stream marking positions for multicharacter conversions.
- Bit stream to position index conversion using bit scan instructions.
- Sequential application of multicharacter conversions only at required positions.

Transformation Kernels (Future)

Case Conversion Notes

- Apply simple case conversion using parallel logic.
- Compute bit stream marking positions for multicharacter conversions.
- Bit stream to position index conversion using bit scan instructions.
- Sequential application of multicharacter conversions only at required positions.

Normalization Notes

- Parallel calculation of quick-check properties.
- Skip further processing when all positions within a block yield Yes.
- Compute compressed ccc property value streams (57 possible values: 6 streams).
- Sequential application of normalization where required.

Outline

Conclusion

Performance

- Parabix methods deliver excellent performance for Unicode regular expression matching.
- Performance improves with SIMD instruction set advances: AVX2, AVX-512.
- Multicore acceleration can use segmented pipeline parallelism.

Conclusion

Performance

- Parabix methods deliver excellent performance for Unicode regular expression matching.
- Performance improves with SIMD instruction set advances: AVX2, AVX-512.
- Multicore acceleration can use segmented pipeline parallelism.

Programming Framework

- Programming interface based on kernels, pipelines and compilers.
- Kernels are function on arbitrary-length stream sets.
- Kernels are composed to into pipelines to yield applications.
- Pipeline compilers provide for buffer management and synchronization.
- Compilers put it all together.

Ongoing/Future Work

Towards Parabix OS

- Use Unix pipes and files model, with Parabix under the hood.
- Parabix shell can dynamically compile pipelines.
  - Avoid costs of inverse transposition followed by transposition.
  - Only compute needed bit streams, e.g., decompression followed by grep.
  - Unicode support can be built-in to all new tools.

Ongoing/Future Work

Towards Parabix OS

- Use Unix pipes and files model, with Parabix under the hood.
- Parabix shell can dynamically compile pipelines.
  - Avoid costs of inverse transposition followed by transposition.
  - Only compute needed bit streams, e.g., decompression followed by grep.
  - Unicode support can be built-in to all new tools.

Collaboration Opportunities

- Parabix Technology is open source: `parabix.costar.sfu.ca`.
- cameron@sfu.ca.
- 2017 Sabbatical: seeking invitations.