# Bitwise Data Parallelism in Regular Expression Matching

Rob Cameron, Tom Shermer, Arrvindh Shriraman, Ken Herdy, Dan Lin, Ben Hull, Meng Lin

School of Computing Science
Simon Fraser University

August 25, 2014

# Outline

# Acceleration of Regular Expression Matching

- Example: quickly find instances of `(^|[ ])\p{Lu}\p{Ll}+[.!?]($|[ ])` in text.
- Sequential algorithms use finite automata or backtracking.
- Parallelizing these approaches is difficult.
    - Finite state machines are the 13th (and hardest) "dwarf" in the Berkeley Landscape of Parallel Computing Research.
    - Embarassingly sequential?
    - Some success in parallel application of FSMs to multiple input streams.
    - Recent work shows some promise using techniques such as coalesced FSMs and principled speculation.

# Our Approach

- A new algorithm family for regular expression matching based on bitwise data parallelism.

# Our Approach

- A new algorithm family for regular expression matching based on bitwise data parallelism.
- A fundamentally parallel approach in contrast to approaches that parallelize existing DFA or NFA algorithms.

# Our Approach

- A new algorithm family for regular expression matching based on bitwise data parallelism.
- A fundamentally parallel approach in contrast to approaches that parallelize existing DFA or NFA algorithms.
- Builds on the Parabix methods that have been used for XML parsing and Unicode transcoding.

# Our Approach

- A new algorithm family for regular expression matching based on bitwise data parallelism.
- A fundamentally parallel approach in contrast to approaches that parallelize existing DFA or NFA algorithms.
- Builds on the Parabix methods that have been used for XML parsing and Unicode transcoding.
- Uses bitstream addition for simultaneous nondeterministic matching of character class repetitions (MatchStar primitive).

# Our Approach

- A new algorithm family for regular expression matching based on bitwise data parallelism.
- A fundamentally parallel approach in contrast to approaches that parallelize existing DFA or NFA algorithms.
- Builds on the Parabix methods that have been used for XML parsing and Unicode transcoding.
- Uses bitstream addition for simultaneous nondeterministic matching of character class repetitions (MatchStar primitive).
- Compilation technologies for regular expressions (new), character classes (existing), unbounded bitstreams (existing).

# Our Approach

- A new algorithm family for regular expression matching based on bitwise data parallelism.
- A fundamentally parallel approach in contrast to approaches that parallelize existing DFA or NFA algorithms.
- Builds on the Parabix methods that have been used for XML parsing and Unicode transcoding.
- Uses bitstream addition for simultaneous nondeterministic matching of character class repetitions (MatchStar primitive).
- Compilation technologies for regular expressions (new), character classes (existing), unbounded bitstreams (existing).
- Recent work: all compilers integrated together with LLVM for fully dynamic regular expression matching.

# Bitwise Data Parallelism

- Parabix methods use a transform representation of text.
- Bitstreams are formed using one bit per input byte.
- Eight basis bit streams are defined for bit 0, bit 1, ... bit 7 of each byte.
- Perform bitwise processing with wide SIMD registers.
    - Process 128 bytes at a time with SSE2, Neon, Altivec.
    - Process 256 bytes at a time with AVX2.
- Transposition supported efficiently with SIMD pack operations.

# Impressive Results in Full Unicode Matching

- Find capitalized words at ends of sentences.
- Use Unicode upper/lower case categories.
- Match `(^|[ ])\p{Lu}\p{Ll}+[.!?]($|[ ])` against 110 MB Arabic file.
- `pcregrep` 14,772,797,548 CPU cycles.
- `egrep` 45,951,194,784 CPU cycles.
- `icgrep` (Parabix) 653,530,064 CPU cycles.
- 20X acceleration over `pcgregrep`, 70X over GNU `egrep`.

# Parallel Bit Streams: A Transform Representation of Text

- Given a byte-oriented character stream $T$, e.g., "`Ab17;`".
- Transpose to 8 parallel bit streams $b_0$, $b_1$, ..., $b_7$.
- Each stream $b_k$ comprises bit $k$ of each byte of $T$.

| $T$ | A | b | 1 | 7 | ; |
|---|---|---|---|---|---|
| ASCII | 01000001 | 01100010 | 00110001 | 00110111 | 00111011 |

# Parallel Bit Streams: A Transform Representation of Text

- Given a byte-oriented character stream $T$, e.g., "`Ab17;`".
- Transpose to 8 parallel bit streams $b_0$, $b_1$, ..., $b_7$.
- Each stream $b_k$ comprises bit $k$ of each byte of $T$.

| $T$ | A | b | 1 | 7 | ; |
|---|---|---|---|---|---|
| ASCII | 01000001 | 01100010 | 00110001 | 00110111 | 00111011 |
| $b_0$ | 0 | 0 | 0 | 0 | 0 |

# Parallel Bit Streams: A Transform Representation of Text

- Given a byte-oriented character stream $T$, e.g., "`Ab17;`".
- Transpose to 8 parallel bit streams $b_0$, $b_1$, ..., $b_7$.
- Each stream $b_k$ comprises bit $k$ of each byte of $T$.

| $T$ | A | b | 1 | 7 | ; |
|---|---|---|---|---|---|
| ASCII | 01000001 | 01100010 | 00110001 | 00110111 | 00111011 |
| $b_0$ | 0 | 0 | 0 | 0 | 0 |
| $b_1$ | 1 | 1 | 0 | 0 | 0 |

# Parallel Bit Streams: A Transform Representation of Text

- Given a byte-oriented character stream $T$, e.g., "Ab17;".
- Transpose to 8 parallel bit streams $b_0$, $b_1$, ..., $b_7$.
- Each stream $b_k$ comprises bit $k$ of each byte of $T$.

| $T$ | A | b | 1 | 7 | ; |
|---|---|---|---|---|---|
| ASCII | 01000001 | 01100010 | 00110001 | 00110111 | 00111011 |
| $b_0$ | 0 | 0 | 0 | 0 | 0 |
| $b_1$ | 1 | 1 | 0 | 0 | 0 |
| $b_2$ | 0 | 1 | 1 | 1 | 1 |
| $b_3$ | 0 | 0 | 1 | 1 | 1 |
| $b_4$ | 0 | 0 | 0 | 0 | 1 |
| $b_5$ | 0 | 0 | 0 | 1 | 0 |
| $b_6$ | 0 | 1 | 0 | 1 | 1 |
| $b_7$ | 1 | 0 | 1 | 1 | 1 |

# Parabix Programming

- Parabix programs written as unbounded bit stream operations.

# Parabix Programming

- Parabix programs written as unbounded bit stream operations.
- Unbounded bit streams considered as arbitrarily large integers.

# Parabix Programming

- Parabix programs written as unbounded bit stream operations.
- Unbounded bit streams considered as arbitrarily large integers.
- Fundamental operations: bitwise logic, bit-stream shifting and long-stream addition.

# Parabix Programming

- Parabix programs written as unbounded bit stream operations.
- Unbounded bit streams considered as arbitrarily large integers.
- Fundamental operations: bitwise logic, bit-stream shifting and long-stream addition.
- Parabix tool chain has three components:

# Parabix Programming

- Parabix programs written as unbounded bit stream operations.
- Unbounded bit streams considered as arbitrarily large integers.
- Fundamental operations: bitwise logic, bit-stream shifting and long-stream addition.
- Parabix tool chain has three components:
  - Character Class Compiler (CCC) produces stream equations from character classes.

# Parabix Programming

- Parabix programs written as unbounded bit stream operations.
- Unbounded bit streams considered as arbitrarily large integers.
- Fundamental operations: bitwise logic, bit-stream shifting and long-stream addition.
- Parabix tool chain has three components:
  - Character Class Compiler (CCC) produces stream equations from character classes.
  - Parallel Block Compiler (Pablo) converts unbounded stream programs to C++/SIMD.

# Parabix Programming

- Parabix programs written as unbounded bit stream operations.
- Unbounded bit streams considered as arbitrarily large integers.
- Fundamental operations: bitwise logic, bit-stream shifting and long-stream addition.
- Parabix tool chain has three components:
  - Character Class Compiler (CCC) produces stream equations from character classes.
  - Parallel Block Compiler (Pablo) converts unbounded stream programs to C++/SIMD.
  - Portable SIMD library for multiple architectures.

# Character Class Formation

- Combining 8 bits of a code unit gives a character class stream.

# Character Class Formation

- Combining 8 bits of a code unit gives a character class stream.
- CCC(`cc_a = [a]`)

# Character Class Formation

- Combining 8 bits of a code unit gives a character class stream.
- CCC(`cc_a = [a]`)
- ```
  temp1 = (bit[1] &~ bit[0])
  temp2 = (bit[2] &~ bit[3])
  temp3 = (temp1 & temp2)
  temp4 = (bit[4] | bit[5])
  temp5 = (bit[7] &~ bit[6])
  temp6 = (temp5 &~ temp4)
  cc_a = (temp3 & temp6)
  ```
  ```

# Character Class Ranges

- Ranges of characters are often very simple to compute.

# Character Class Ranges

- Ranges of characters are often very simple to compute.
- CCC(cc_0_9 = [0-9])

# Character Class Ranges

- Ranges of characters are often very simple to compute.
- CCC(`cc_0_9 = [0-9]`)
- ```
  temp7 = (bit[0] | bit[1])
  temp8 = (bit[2] & bit[3])
  temp9 = (temp8 &~ temp7)
  temp10 = (bit[5] | bit[6])
  temp11 = (bit[4] & temp10)
  cc_0_9 = (temp9 &~ temp11)
  ```
  ```

# Character Class Common Subexpressions

- Multiple definitions use common subexpressions.
- CCC(`cc_z9 = [z9]`)
- ```
  temp12 = (bit[4] &~ bit[5])
  temp13 = (temp12 & temp5)
  temp14 = (temp9 & temp13)
  temp15 = (temp1 & temp8)
  temp16 = (bit[6] &~ bit[7])
  temp17 = (temp12 & temp16)
  temp18 = (temp15 & temp17)
  cc_z9 = (temp14 | temp18)
  ```
  ```

# Marker Streams

- Marker stream $M_i$ indicates the positions that are reachable after item $i$ in the regular expression.

# Marker Streams

- Marker stream $M_i$ indicates the positions that are reachable after item $i$ in the regular expression.
- Each marker stream $M_i$ has one bit for every input byte in the input file.

# Marker Streams

- Marker stream $M_i$ indicates the positions that are reachable after item $i$ in the regular expression.
- Each marker stream $M_i$ has one bit for every input byte in the input file.
- $M_i[j] = 1$ if and only if a match to the regular expression up to and including item $i$ in the expression occurs at position $j - 1$ in the input stream.

# Marker Streams

- Marker stream $M_i$ indicates the positions that are reachable after item $i$ in the regular expression.
- Each marker stream $M_i$ has one bit for every input byte in the input file.
- $M_i[j] = 1$ if and only if a match to the regular expression up to and including item $i$ in the expression occurs at position $j - 1$ in the input stream.
- Conceptually, marker streams are computed in parallel for all positions in the file at once (bitwise data parallelism).

# Marker Streams

- Marker stream $M_i$ indicates the positions that are reachable after item $i$ in the regular expression.
- Each marker stream $M_i$ has one bit for every input byte in the input file.
- $M_i[j] = 1$ if and only if a match to the regular expression up to and including item $i$ in the expression occurs at position $j - 1$ in the input stream.
- Conceptually, marker streams are computed in parallel for all positions in the file at once (bitwise data parallelism).
- In practice, marker streams are computed block-by-block, where the block size is the size of a SIMD register in bits.

# Marker Stream Example

- Consider matching regular expression a[0-9]*[z9] against the input text below.

input data `a453z--b3z--az--a12949z--ca22z7--`

# Marker Stream Example

- Consider matching regular expression `a[0-9]*[z9]` against the input text below.
- $M_1$ marks positions after occurrences of a.

| | |
|---|---|
| input data | `a453z--b3z--az--a12949z--ca22z7--` |
| $M_1$ | `.1...........1...1..........1.....` |

# Marker Stream Example

- Consider matching regular expression `a[0-9]*[z9]` against the input text below.
- $M_1$ marks positions after occurrences of `a`.
- $M_2$ marks positions after occurrences of `a[0-9]*`.

| | |
|---|---|
| input data | `a453z--b3z--az--a12949z--ca22z7--` |
| $M_1$ | `.1...........1...1.........1.....` |
| $M_2$ | `.1111........1...111111....111...` |

# Marker Stream Example

- Consider matching regular expression `a[0-9]*[z9]` against the input text below.
- $M_1$ marks positions after occurrences of `a`.
- $M_2$ marks positions after occurrences of `a[0-9]*`.
- $M_3$ marks positions after occurrences of `a[0-9]*[z9]`.

```
input data  a453z--b3z--az--a12949z--ca22z7--
M1          .1...........1...1.........1.....
M2          .1111........1...111111....111...
M3          .....1........1.....1.11......1..
```

# Matching Character Class Repetitions with MatchStar

# Matching Character Class Repetitions with MatchStar

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

# Matching Character Class Repetitions with MatchStar

- $\text{MatchStar}(M, C) = (((M \wedge C) + C) \oplus C) \vee M$
- Consider $M_2 = \text{MatchStar}(M_1, C)$

| | |
|---|---|
| input data | `a453z--b3z--az--a12949z--ca22z7--` |
| $M_1$ | `.1...........1...1.........1.....` |
| $C = [0-9]$ | `.111....1.......11111.....11.1..` |

# Matching Character Class Repetitions with MatchStar

- $\text{MatchStar}(M, C) = (((M \wedge C) + C) \oplus C) \vee M$
- Consider $M_2 = \text{MatchStar}(M_1, C)$
- Use addition to scan each marker through the class.

| | |
|---|---|
| input data | `a453z--b3z--az--a12949z--ca22z7--` |
| $M_1$ | `.1...........1...1.........1.....` |
| $C = [0\text{-}9]$ | `.111....1........11111.....11.1..` |
| $T_0 = M_1 \wedge C$ | `.1...............1.........1.....` |
| $T_1 = T_0 + C$ | `....1...1.............1......11..` |

# Matching Character Class Repetitions with MatchStar

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

| | |
|---|---|
| input data | `a453z--b3z--az--a12949z--ca22z7--` |
| $M_1$ | `.1...........1...1.........1.....` |
| $C = [0\text{–}9]$ | `.111....1........11111.....11.1..` |
| $T_0 = M_1 \wedge C$ | `.1...............1.........1.....` |
| $T_1 = T_0 + C$ | `....1...1.............1......11..` |
| $T_2 = T_1 \oplus C$ | `.1111............111111....111...` |

# Matching Character Class Repetitions with MatchStar

- $\text{MatchStar}(M, C) = (((M \wedge C) + C) \oplus C) \vee M$
- Consider $M_2 = \text{MatchStar}(M_1, C)$
- 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 | `a453z--b3z--az--a12949z--ca22z7--` |
| $M_1$ | `.1...........1...1.........1.....` |
| $C = [0\text{–}9]$ | `.111....1........11111.....11.1..` |
| $T_0 = M_1 \wedge C$ | `.1...............1.........1.....` |
| $T_1 = T_0 + C$ | `....1...1.............1......11..` |
| $T_2 = T_1 \oplus C$ | `.1111............111111....111...` |
| $M_2 = T_2 \vee M_1$ | `.1111........1...111111....111...` |

# Regular Expression Compilation

- Our regular expression compiler produces unbounded Pablo code.
- RE_compile(a[0-9]*[z9])
- ```
  m0 = ~0
  m1 = pablo.Advance(m0 & cc_a)
  m2 = pablo.MatchStar(m1, cc_0_9)
  m3 = pablo.Advance(m2, cc_z9)
  ```
  ```

# Alternations and Optional Terms

- Most RE features are handled naturally.
- RE_compile(`a(b?|cd)`)
- 
```
m0 = ~0
m1 = pablo.Advance(m0 & cc_a)
m2 = pablo.MatchStar(m1, cc_b)
m3 = m1 | m2       # b is optional
m4 = pablo.Advance(m1, cc_c)
m5 = pablo.Advance(m2, cc_d)
m6 = m3 | m5       # two alternatives
```

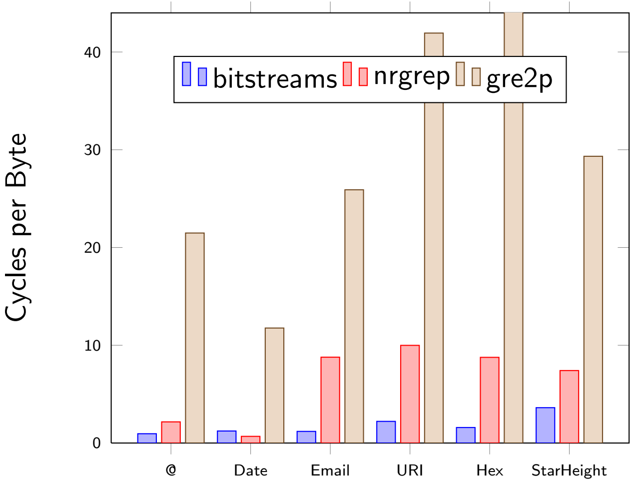
# Nested Repetitions Use While Loops

- While loops are used for complex or nested repetitions.
- RE_compile(`(a[0-9]*[z9])*`)
- 
```
m0 = ~0
t = m0   # while test variable
a = m0   # while result accumulator
while t:
   m1 = pablo.Advance(t & cc_a)
   m2 = pablo.MatchStar(m1, cc_0_9)
   m3 = pablo.Advance(m2, cc_z9)
   t = m3 &~ a   # iterate only for new matches
   a = a | m3
```

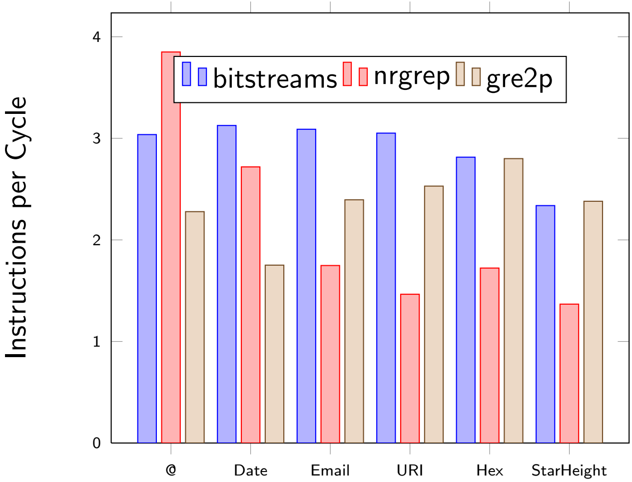
# Test Expressions

| Name | Expression |
|---|---|
| @ | `@` |
| Date | `([0-9][0-9]?)/([0-9][0-9]?)/([0-9][0-9]([0-9][0-9])?)` |
| Email | `([^ @]+)@([^ @]+)` |
| URI | `(([a-zA-Z][a-zA-Z0-9]*)://\|mailto:)([^ /]+)(/[^ ]*)?\|([^ @]+)@([^ @]+)` |
| Hex | `[ ](0x)?([a-fA-F0-9][a-fA-F0-9])+[.:,?! ]` |
| StarHeight | `[A-Z]((([a-zA-Z]*a[a-zA-Z]*[ ])*[a-zA-Z]*e[a-zA-Z]*[ ])*[a-zA-Z]*s[a-zA-Z]*[ ])*[.?!]` |

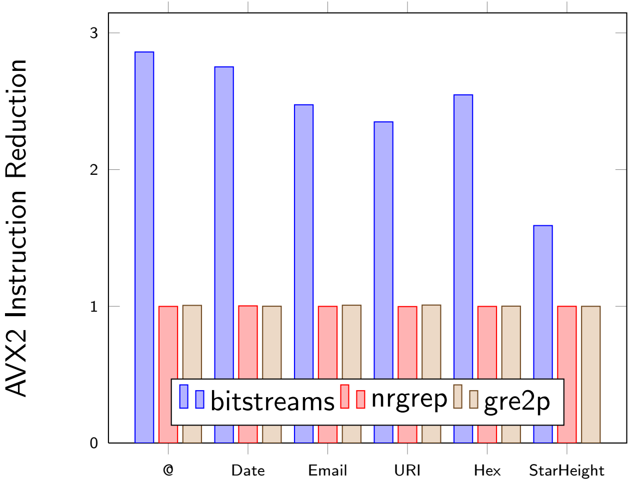
# SSE2 Performance



# IPC


Instructions per Cycle
bitstreams
nrgrep
gre2p
0
1
2
3
4
@
Date
Email
URI
Hex
StarHeight

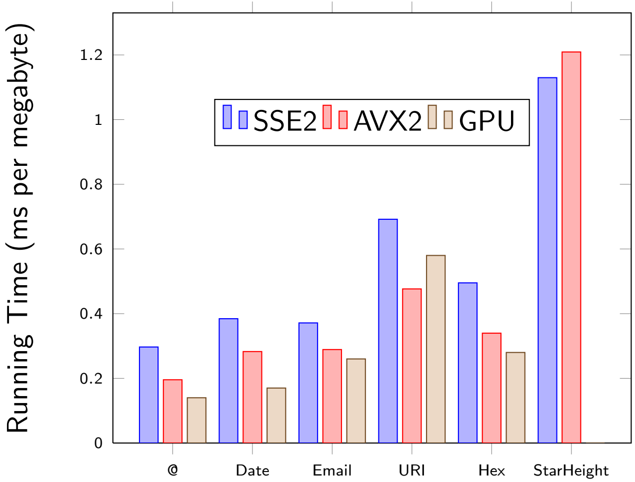
# SIMD Scalability



# Speedups Achieved

| Expression | Bitstream/AVX2 grep Speedup | | |
|---|---|---|---|
| | vs. nrgrep | vs. gre2p | vs. GNU grep -e |
| At | 3.5X | 34X | 1.6X |
| Date | 0.76X | 13X | 48X |
| Email | 9.5X | 28X | 12X |
| URI | 6.6X | 27X | 518X |
| Hex | 8.1X | 105X | 267X |
| StarHeight | 1.9X | 7.6X | 97X |

# GPU Performance



# Results

- A new class of parallel regular expression algorithms has been introduced based on the concept of bitwise data parallelism and MatchStar.
- Single core acceleration over sequential implementations can be dramatic.
- A long-stream addition technique has been developed to allow MatchStar to scale directly with SIMD instruction width.
- Perfect scaling in instruction count was observed with 256-bit AVX2 technology versus 128-bit SIMD technology except for nested repetition.
- GPU implementations show promise, but need additional work.

# Ongoing/Future Work

- The prototype technologies have now been re-implemented in a single C++ executable combining 4 compilers.
  - CCC: Character class compiler
  - RE_compile: regular expression compiler
  - Pablo: Block-at-a-time compiler
  - LLVM: Fully dynamic code generation.
- Compilation overhead is high, but tolerable for large files.
- Unicode support has been added, including additional MatchStar algorithms for variable-length Unicode character classes.
- Open source implementation available: `http://parabix.costar.sfu.ca/svn/icGREP/`