XML on Cell BE

Robert D. Cameron
Dan Lin
School of Computing Science
Simon Fraser University

International Characters, Inc.
http://international-characters.com

Carleton University Cell BE Programming Workshop
May 15-16, 2008
Ottawa, Ontario

The Parabix Approach

- Parallel bit streams for XML.
  - Complete rethinking of XML parsing.
  - Not limited by incremental “performance improvement” approach for existing XML parsers.
  - Present focus on systematic SIMD (SSE, Altivec).
  - Next: leveraging SIMD parallelism for multicore.
- Cell BE is an ideal platform.
  - SPEs have improved SIMD with more registers.

Overview

- Fundamentals (Dan Lin)
  - SIMD notation, || bit streams, s2p algorithm
- Performance Criteria and Expectations
  - order of magnitude improvement on single-core
- Algorithms
  - describe present SIMD implementations
  - discuss data parallel distribution to SPEs
- Wrap-up: Parabix Project Status and Plan

SIMD Library

- Platform
  - MMX
  - SSE
  - PPE
  - SPE
- Endianness
  - Little Endian
  - Big Endian

SIMD operation format

- simd_add_4_h_l (r1, r2)

field width

Half operand
modifier for r1

Half operand
modifier for r2

simd_add_4_h l (r1, r2)

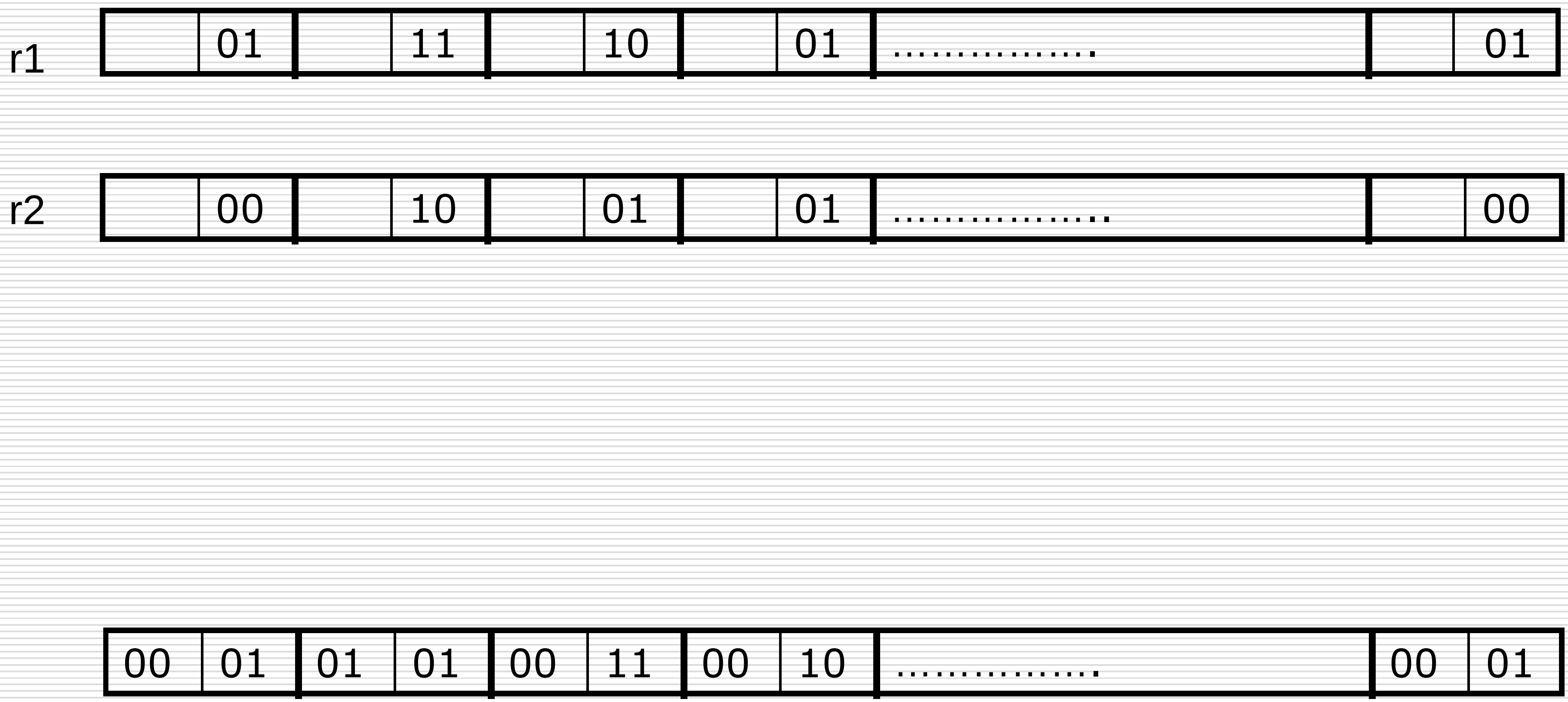
r1	01	10	11	00	10	11	01	00	....	01	11

r2	11	00	10	10	11	01	11	01	....	10	00

simd_add_4_h l (r1, r2)

r1	01		11		10		01		…………….	01	
r2		00		10		01		01	………………		00

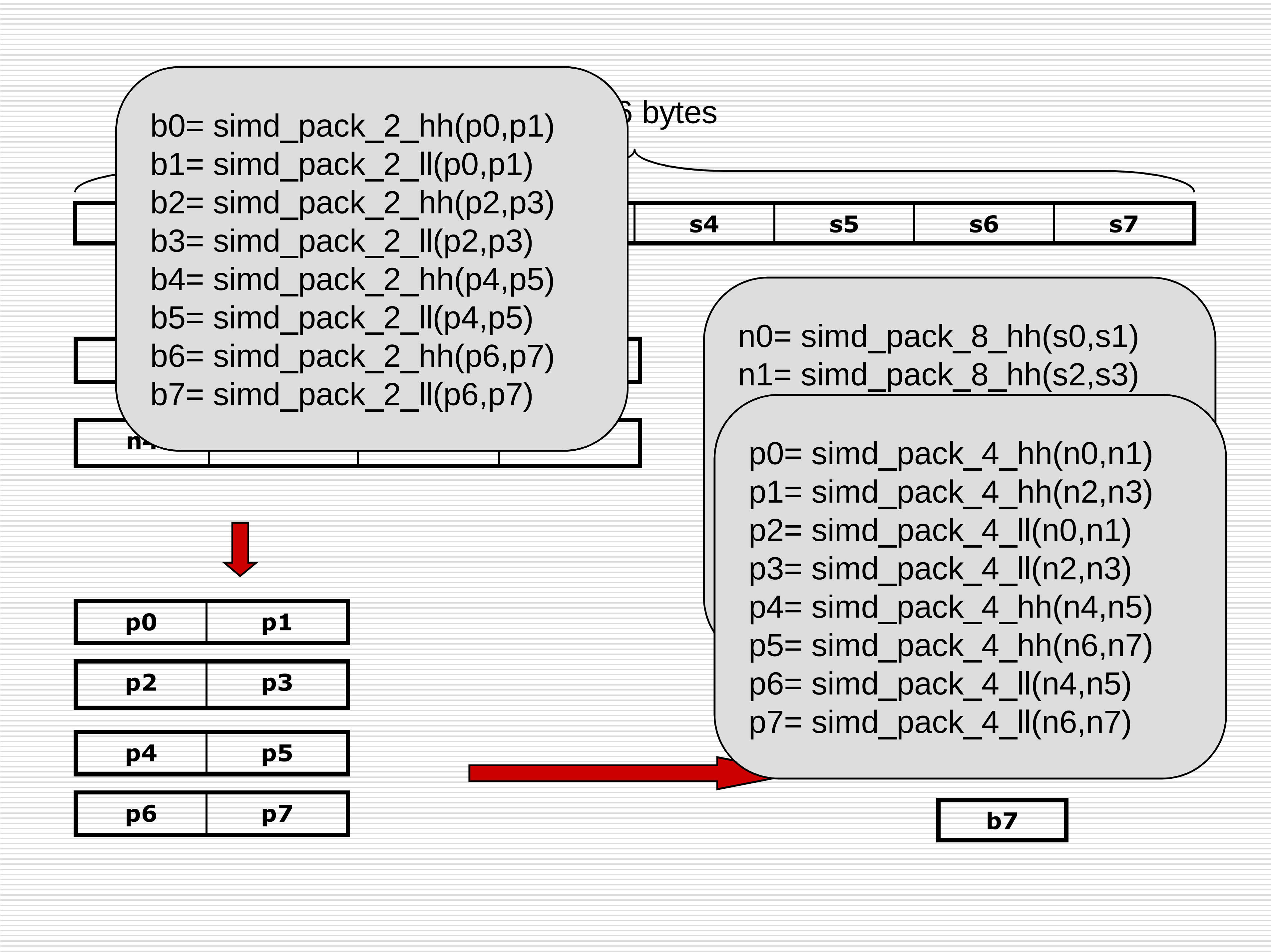
simd_add_4_h l (r1, r2)



Parallel Bit Streams

A								b								**c**								…
0	1	0	0	0	0	0	1	0	1	1	0	0	0	1	0	0	1	1	0	0	0	1	1	…

bit0	0	0	0	…
bit1	1	1	1	…
bit2	0	1	1	…
bit3	0	0	0	…
bit4	0	0	0	…
bit5	0	0	0	…
bit6	0	1	1	…
bit7	1	0	1	…



6 bytes
s4
s5
s6
s7
b0= simd_pack_2_hh(p0,p1)
b1= simd_pack_2_ll(p0,p1)
b2= simd_pack_2_hh(p2,p3)
b3= simd_pack_2_ll(p2,p3)
b4= simd_pack_2_hh(p4,p5)
b5= simd_pack_2_ll(p4,p5)
b6= simd_pack_2_hh(p6,p7)
b7= simd_pack_2_ll(p6,p7)
n0= simd_pack_8_hh(s0,s1)
n1= simd_pack_8_hh(s2,s3)
p0= simd_pack_4_hh(n0,n1)
p1= simd_pack_4_hh(n2,n3)
p2= simd_pack_4_ll(n0,n1)
p3= simd_pack_4_ll(n2,n3)
p4= simd_pack_4_hh(n4,n5)
p5= simd_pack_4_hh(n6,n7)
p6= simd_pack_4_ll(n4,n5)
p7= simd_pack_4_ll(n6,n7)
p0
p1
p2
p3
p4
p5
p6
p7
b7

Performance Criteria, Expectations

- Use perf. counters to calculate CPU cycles/byte
- Compare with byte-at-a-time validating parsers
  - 100 cycles/byte range
- Parabix expectations
  - single-core goal: 10 cycles/byte range
  - multi-core goal: approach linear speed-up by exploiting natural parallelism in SIMD algorithms

Transposition - Performance

- Ideal algorithm/ideal arch: 0.2 ops per byte
- SSE/PPE/SPE: use bytepack variant algorithm
  - PPE/SPE: 0.6 ops per byte
  - SSE: 1.1 cycles per byte
- In all cases, parallel bit stream transposition cost is tiny compared to 100 cycles/byte.
- Allocation to SPEs: no interblock dependencies.

What Do Parallel Bit Streams Give?

- Fast bit scan to next “<”, etc.
  - Replace byte at a time loops.
- Fast character validation: UTF-8, 0xFFFE/F, ...
- Fast UTF-8 to UTF-16 transcoding
- Fast parallel regular expression matching
- Fast hash value computation

Character Class Formation

- Combining 8 bits of a code unit gives a character class stream.
- compile(**[CharDef(LAngle, “<”)]**)

```
temp1 = simd_or(bit[0], bit[1]);
temp2 = simd_and(bit[2], bit[3]);
temp3 = simd_andc(temp2, temp1);
temp4 = simd_and(bit[4], bit[5]);
temp5 = simd_or(bit[6], bit[7]);
temp6 = simd_andc(temp4, temp5);
LAngle = simd_and(temp3, temp6);
```

Character Class Formation

- Multiple definitions have common subexpressions.
- compile(**[CharDef(LAngle, “<”), CharDef(RAngle, “>”)]**)

```
temp1 = simd_or(bit[0], bit[1]);
temp2 = simd_and(bit[2], bit[3]);
temp3 = simd_andc(temp2, temp1);
temp4 = simd_and(bit[4], bit[5]);
temp5 = simd_or(bit[6], bit[7]);
temp6 = simd_andc(temp4, temp5);
LAngle = simd_and(temp3, temp6);
temp7 = simd_andc(bit[6], bit[7]);
temp8 = simd_and(temp4, temp7);
RAngle = simd_and(temp3, temp8);
```

Character Class Formation

- Ranges of characters are often very simple to compute.
- compile([**CharSet('Control', ['\x00-\x1F']),**
  CharSet('Digit', ['0-9'])])

```
temp1 = simd_or(bit[0], bit[1]);
temp2 = simd_or(temp1, bit[2]);
Control = simd_andc(simd_const_1(1), temp2);
temp3 = simd_and(bit[2], bit[3]);
temp4 = simd_andc(temp3, temp1);
temp5 = simd_or(bit[5], bit[6]);
temp6 = simd_and(bit[4], temp5);
Digit = simd_andc(temp4, temp6);
```

Lexical Item Streams

- Using the character class compiler, we define a set of lexical item streams for XML.
  - MarkupStart, NameFollow, WhiteSpace, QuoteScan, Hyphen, Qmark, CDend, Hex, Digit
  - 67 operations in total to classify 128 bytes.
  - 0.5 ops per byte.
- Can define with no interblock dependencies
  - data parallel distribution to SPEs.

UTF-8/XML Char Validation

- Verify that UTF-8 sequences are well-formed.
  - prefix bytes followed by 1, 2 or 3 suffix bytes
  - prefixes C0-C1, F5-FF are ruled out
  - special constraints for prefixes E0, ED, F0, F4
- Rule out illegal XML characters
  - 0x00-0x1F except TAB, LF, CR, 0xFFFE-0xFFFF

UTF-8 Byte Classification

- Form bit streams to classify UTF-8 bytes
  - **`u8prefix = simd_and(bit0, bit1);`**
  - **`u8suffix = simd_andc(bit0, bit1);`**
  - **`pfx2 = simd_andc(u8prefix, bit2);`**
- Compile logic for bad prefix ranges
- Form scope streams to establish suffix expectations.
  - ```
    scope22 = simd_or(sfli(pfx2, 1),
                      sbli(oldpfx2, 127));
    ```
    ```

UTF-8/XML Validation(cont'd)

- Find mismatches with xor:
  - `error_mask = simd_xor(anyscope, u8suffix);`
- Optimize logic based on max UTF-8 sequence length.
  - e.g. bit0 is 0 for entire block: no prefixes or suffixes
- Overall character validation cost is 0.1-0.5 cycles per byte, depending on input.

UTF-8/XML Validation on SPEs

- Distribution of character validation to SPEs:
  - scope streams may cross partition boundaries
- Options:
  - use a 3-, 16- or 128- byte overlap
  - PPE preprocessing
    - pack data from 16 partition boundaries into a single block
    - calculate scope streams at boundaries in advance

Transcoding to UTF-16

- XML files typically stored in UTF-8
  - variable-length byte-oriented encoding
- Applications often use UTF-16 internally
  - fixed 16-bits per character (except rare characters in supplementary plane)
- UTF-8 to UTF-16 transcoding a typical requirement for XML parsers.
- Frequently cited as a major cost: 30% or more.

Transcoding to UTF-16 (cont'd)

- Calculate 16 parallel bit streams using logic and shift operations.
  - About 4 ops per bit stream per block.
- Principal challenge: variable length mapping
  - Every 1, 2 or 3 byte U-8 seq.: 1 UTF-16 value.
  - 4-byte UTF-8 sequences: 2 UTF-16 values.
  - Convention: calculate UTF-16 bit values at position of last byte in sequence (& scope42).
  - Output is only generated for these positions.
  - Mapping is achieved by parallel bit deletion.

Parallel Bit Deletion

- Mark all positions to be deleted.
  - `delmask = simd_or(u8prefix, scope32, scope43)`
- Apply a parallel deletion algorithm.
- Ideal: deletion by central induction.
  - Move bits to center within each field.
  - Solve 4-bit fields, then 8-bit, then 16 ...
  - Use SIMD rotate of PPU/SPU.
  - One rotate per field width per stream.

UTF-8 to UTF-16 on SPEs

- UTF-8 to UTF-16 transcoding has been implemented and tested on PPE.
- Distribution to SPEs involves same transition boundary issues as UTF-8 validation.
- SPE implementation should benefit from greater register availability:
  - eliminate PPE loads and stores for temporary values due to register pressure

UTF-8 to UTF-16 on SPEs

- UTF-8 to UTF-16 transcoding has been implemented and tested on PPE.
- Distribution to SPEs involves same transition boundary issues as UTF-8 validation.
- SPE implementation should benefit from greater register availability:
  - eliminate PPE loads and stores for temporary values due to register pressure

Regular Expression Matching

Parallel Matching of [-+]?[0-9]+ Regular Expression

```
;5.796953 - 6++ 4+ gnorw 17- 421     character stream
00000000001001100100000000010000     [-+] character class
01011111100010001000000001100111     [0-9] character class
00000000101000100100000100010001     c0, initial cursor
10000000010100010010000010001000     end_mask
```

Regular Expression Matching

Parallel Matching of [-+]?[0-9]+ Regular Expression

`;5.796953 - 6++ 4+ gnorw 17- 421`	**character stream**
`00000000001001100100000000010000`	**[-+] character class**
`00000000101000100100000100010001`	**c0, initial cursor**
`00000000110001001000000100100001`	**c1 = c0 + (c0 & [-+])**

Prefilter for max match length of 1

Regular Expression Matching

Parallel Matching of [-+]?[0-9]+ Regular Expression

`;5.796953 - 6++ 4+ gnorw 17- 421`	character stream
`01011111100010001000000001100111`	[0-9] character class
`00000000110001001000000100100001`	**c1 = c0 + (c0 & [-+])**
`00100000000000010000000010001000`	**(c1+[0-9])&~[0-9] &~c1**

Filter digits that did not propagate

Postfilter for min match length of 1

Regular Expression Matching

Parallel Matching of [-+]?[0-9]+ Regular Expression

`;5.796953 - 6++ 4+ gnorw 17- 421`	character stream
`00000000001001100100000000010000`	[-+] character class
`01011111100010001000000000110011 1`	[0-9] character class
`00000000101000100100000010001000 1`	**c0, initial cursor**
`10000000001010001001000000100010 00`	**end_mask**
`00000000110001001000000010010000 1`	**c1 = c0 + (c0 & [-+])**
`00100000000000001000000000100010 00`	**(c1+[0-9])&~[0-9] &~c1**
`00000000000000001000000000100010 00`	**end_mask & c2**

Three complete matches found.

Parabix Project

- Project home: parabix.costar.sfu.ca.
- Open source under OSL 3.0.
- Combined copyleft/patentleft strategy.
- Nearing completion of validating XML 1.0 parser on Intel/SSE and PPC/Altivec.
- Primary focus: additional XML functionality using SIMD.
- Collaboration opportunities: acceleration on multicore: Cell BE is ideal!