Changbo Chen\textsuperscript{1}  Svyatoslav Covano\textsuperscript{2,3}  Farnam Mansouri\textsuperscript{2}
Marc Moreno Maza\textsuperscript{2}  Ning Xie\textsuperscript{2}  Yuzhen Xie\textsuperscript{2}

\textsuperscript{1}Chinese Academy of Sciences, China
\textsuperscript{2}University of Western Ontario, Canada
\textsuperscript{3}École Polytechnique, France

ICMS, August 8, 2014
Plan

1. Overview
2. Code organization and user interface
3. Core algorithms
4. Applications
No symbolic computation software dedicated to *sequential polynomial arithmetic* managed to play the unification role that the BLAS play in numerical linear algebra.

Could this work in the case of hardware accelerators?

How to benefit from other successful projects related to polynomial arithmetic, like FFTW, SPIRAL and GMP?
Overview: the Basic Polynomial Algebra Subprograms

Driving observation

▷ Polynomial multiplication and matrix multiplication are at the core of many algorithms in symbolic computation.

▷ Algebraic complexity is often estimated in terms of multiplication time. At the software level, this reduction is also common (Magma, NTL, FLINT, ...)

▷ BPAS design follows the principle reducing everything to multiplication.

Targeted functionalities

**Level 1**: core routines specific to a coefficient ring or a polynomial representation: multi-dimensional FFTs, SLP operations, ...

**Level 2**: basic arithmetic operations for dense or sparse polynomials with coefficients in \( \mathbb{Z}, \mathbb{Q} \) or \( \mathbb{Z}/p\mathbb{Z} \): polynomial multiplication, Taylor shift, ...

**Level 3**: advanced arithmetic operations taking as input a zero-dimensional regular chains: normal form of a polynomial, multivariate real root isolation, ...

Programs on multi-core processors can be written in CilkPlus or OpenMP. Our Meta_Fork framework http://www.metafork.org performs automatic translation between the two as well as conversions to C/C++.

Graphics Processing Units (GPUs) with code written in CUDA, provided by the CUMODP library http://www.cumodp.org.

Unifying code for both multi-core processors and GPUs is conceivable (see the SPIRAL project) but highly complex (multi-core processors enforce memory consistency while GPUs do not, etc.)
Overview: implementation techniques

Level 1: core routines
- code is highly optimized in terms of work, data locality and parallelism,
- code generation is used at library installation time.

Level 2: basic arithmetic operations
- functions provide a variety of algorithmic solutions for a given operation,
- the user can choose either algorithms minimizing work or those maximizing parallelism.
- Example: Schönaghe-Strassen, divide-and-conquer plain, $k$-way Toom-Cook and the two-convolution method for integer polynomial multiplication.

Level 3: advanced arithmetic operations
- functions combine several Level 2 algorithms for achieving a given task,
- this leads to adaptive algorithms that select appropriate Level 2 functions depending on available resources (number of cores, input data size).
Plan

1. Overview
2. Code organization and user interface
3. Core algorithms
4. Applications
Subprojects

- Polynomial types with specified coefficient ring: ModularPolynomial/, IntegerPolynomial/ and RationalNumberPolynomial/.
- Polynomial types with unspecified coefficient ring (template classes): Polynomial/.
- ModularPolynomial/ is based on the Modpn library and includes our FFT code generator, which is inspired by FFTW and SPIRAL.
- IntegerPolynomial/ relies on the GMP library.

User interface

- The UI currently exposes part of the polynomial types (the univariate ones and sparse multivariate polynomials)
- Exposing the other ones is work in progress.
- But the entire project is freely available in source at www.bpaslib.org.
The above is a snapshot of the BPAS classes.
This shows two multivariate polynomial concrete classes, namely
\texttt{DistributedDenseMultivariateModularPolynomial\langle Field\rangle} and \texttt{SMQP}, and three univariate polynomial ones, namely \texttt{DUQP}, \texttt{DUQP} and \texttt{SparseUnivariatePolynomial\langle Ring\rangle}.
Many other classes are provided like \texttt{Intervals}, \texttt{RegularChains}, ...
The BPAS classes \texttt{Integer} and \texttt{RationalNumber} are BPAS wrappers for GMP’s \texttt{mpz} and \texttt{mpq} classes.

#include <bpas.h>
using namespace std;

int main (int argc, char *argv[]) {
    DUZP a (4096), b(4096);    // Initializing space
    for (int i = 0; i < 4096; ++i) { a.setCoefficient(i, rand() % 1000 + 1); }
    for (int i = 0; i < 4096; ++i) { b.setCoefficient(i, rand() % 1000 + 1); }
    DUZP c = a * a - b * b, d = a * a * a - b * b * b;
    DUZP g = c.gcd(d);     // Gcd computation, g = a - b
    c /= g;                 // Exact division, c = a + b
    cout << "g = " << g << endl;

    DUQP p; // Initializing as a zero polynomial
    p = (p + mpq_class(1) << 4095) + mpq_class(4095);  // p = x^{4095} + 4095
    Intervals boxes = p.realRootIsolation(0.5);
    cout << "boxes = " << boxes << endl;

    SMQP f(3), g(2); // Initializing with number of variables
    SMQP h = (f^2) + f * g * mpq_class(2) + (g^2);
    SparseUnivariatePolynomial<SMQP> s = h.convertToSUP("x");
    SMQP z (s);
    if (z != h) { cout << z << " & " << h << endl; }

    return 0;
}
Overview

Code organization and user interface

Core algorithms

Applications
Three core subprograms

- One-dimensional modular FFTs
- Parallel integer polynomial multiplication
- Parallel Taylor shift computation \( f(x) \quad \leftrightarrow \quad f(x + 1) \)
1-D FFTs: classical cache friendly algorithm

If the input vector does not fit in cache, a recursive algorithm is applied. Once the vector fits in cache, an iterative algorithm (not requiring shuffling) takes over.

On an ideal cache of $Z$ words with $L$ words per cache line this yeilds a cache complexity of $\Omega\left(\frac{n}{L}(\log_2(n) - \log_2(Z))\right)$ which is not optimal.

\[
\text{FFT}([a_0, a_1, \cdots, a_{n-1}, \omega]) \\
\text{if } n \leq \text{HThreshold} \text{ then} \\
\quad \text{ArrayBitReversal}(a_0, a_1, \cdots, a_{n-1}) \\
\quad \text{return FFT\_iterative\_in\_cache}([a_0, a_1, \cdots, a_{n-1}, \omega]) \\
\text{end if} \\
\text{Shuffle}(a_0, a_1, \cdots, a_{n-1}) \\
[a_0, a_1, \cdots, a_{n/2-1}] = \text{FFT}([a_0, a_1, \cdots, a_{n/2-1}, \omega^2]) \\
[a_{n/2}, a_{n/2+1}, \cdots, a_{n-1}] = \text{FFT}([a_{n/2}, a_{n/2+1}, \cdots, a_{n-1}, \omega^2]) \\
\text{return } [a_0 + a_{n/2}, a_1 + \omega \cdot a_{n/2+1}, \cdots, a_{n/2-1} - \omega^{n/2-1} \cdot a_{n-1}]\]
Cache optimal 1-D FFT

- Instead of processing row-by-row, one computes as deep as possible while staying in cache (resp. registers): this yields a blocking strategy.
- On the left picture, assuming $Z = 4$, on the first (resp, last) two rows, we successively compute red, green, blue, orange 4-point blocks.
- On an ideal cache of $Z$ words with $L$ words per cache line the cache complexity drops to $O(n/L(\log_2(n)/\log_2(Z)))$ which is optimal.
Cache optimal 1-D FFT

- Modifying the previous blocking strategy such that each block is an FFT on $2^K$ points, for a given $K$, and
- choosing a sparse radix prime $p$ (like $p = r^4 + 1$, for $r = 2^{16} - 2^8$) such that multiplying by the twiddle factors is cheap enough,
- the algebraic complexity drops from $O(n \log_2 (n))$ to $O(n \log_K (n))$, which is optimal on today’s desktop computers for a well chosen $K$. 
Figure: 1-D modular FFTs: Modpn vs BPAS.

- BPAS 1-D FFTs code is automatically generated using Python scripts.
- In addition to the above optimal blocking strategy, instruction level parallelism (ILP) is carefully considered: vectorized instructions are explicitly used (SSE2, SSE4) and instruction pipeline usage is highly optimized.
Schönaghe-Strassen

1. **Evaluation**: \( Z_f = \sum_{i=0}^{n} f_i 2^{i,\ell} \) and \( Z_g = \sum_{i=0}^{m} g_i 2^{i,\ell} \);
2. **Multiplying**: \( Z_h = Z_f \times Z_g \), using GMP library;
3. **Unpacking**: \( h_i \) from \( Z_h = \sum_{i=0}^{n+m-1} h_i 2^{i,\ell} \).

- its complexity in terms of bit operations is \( O(\ell \log(\ell) \log(\log(\ell))) \), where \( \ell \) is the maximum bit-size of a coefficient;
- optimal work, purely serial due to the difficulties of parallelizing 1D FFTs on multicore processors.
Divide-and-conquer algorithm

1 **Division**: \( f(x) = f_0(x) + f_1(x) \cdot x^{n/2} \) and \( g(x) = g_0(x) + g_1(x) \cdot x^{n/2}; \)
2 **Execute recursively**:
   - Store \( f_0 \times g_0 \) \& \( f_1 \times g_1 \) in the result array;
   - Store \( f_0 \times g_1 \) \& \( f_1 \times g_0 \) in the auxiliary arrays;
3 **Addition**: add the auxiliary arrays to the result one.

- use (one or) two DnC levels, then use the Schönaghe-Strassen algorithm;
- its complexity in terms of bit operations is \( O(\ell \log(\ell) \log(\log(\ell))) \), where \( \ell \) is the maximum bit-size of a coefficient, but the constant has increased approximately by 4;
- not work-efficient, static parallelism (close to 16).
Core algorithm: parallel integer polynomial multiplication

**k-way Toom-Cook**

1. **Division:** \( f(x) = f_0(x) + f_1(x) x^{n/k} + \cdots + f_{k-1}(x) x^{(k-1)n/k} \) and \( g(x) = g_0(x) + g_1(x) x^{n/k} + \cdots + g_{k-1}(x) x^{(k-1)n/k} \);

2. **Conversion:** Set \( X = x^{n/k} \) and obtain \( F(X) = Zf_0 + Zf_1 X + \cdots + Zf_{k-1} X^{k-1} \) and \( G(X) = Zg_0 + Zg_1 X + \cdots + Zg_{k-1} X^{k-1} \);

3. **Evaluation:** Evaluate \( f, g \) at \( 2k-1 \) points: \((0, X_1, \ldots, X_{2k-3}, \infty)\);

4. **Multiplying:** \((w_0, \ldots, w_{2k-2}) = (F(0) \cdot G(0), \ldots, F(\infty) \cdot G(\infty))\);

5. **Interpolation:** Recover \((Z_{h_0}, Z_{h_1}, \ldots, Z_{h_{2k-2}})\) where \( H(X) = f(X)g(X) = Z_{h_0} + Z_{h_1} X + \cdots + Z_{h_{2k-2}} X^{2k-2} \);

6. **Conversion:** Recover polynomial coefficients from \( Z_{h_0}, \ldots, Z_{h_{2k-2}} \), obtaining \( h(x) = h_0(x) + h_1(x) x^{n/k} + \cdots + h_{2k-2}(x) x^{(2k-2)n/k} \).

- Complexity in terms of bit operations is \( O(\ell \log(\ell) \log(\log(\ell))) \), where \( \ell \) is the maximum bit-size of a coefficient;
- 4-way & 8-way Toom-Cook are available;
- Static parallelism (about 7 and 13 when \( k = 4 \) and \( k = 8 \), resp), suffer from overheads in data conversions.
Core algorithm: parallel integer polynomial multiplication

A new algorithm: the two-convolution method
Core algorithm: parallel integer polynomial multiplication

1. Convert \( a(y), b(y) \) to bivariate \( A(x, y), B(x, y) \) s. t. \( a(y) = A(\beta, y) \) and \( b(y) = B(\beta, y) \) hold at \( \beta = 2^M \), \( K = \deg(A, x) = \deg(B, x) \), where \( KM \) is essentially the maximum bit size of a coefficient in \( a, b \).

2. Consider \( C^+(x, y) \equiv A(x, y) B(x, y) \mod <x^K + 1> \) and \( C^-(x, y) \equiv A(x, y) B(x, y) \mod <x^K - 1> \), then compute \( C^+(x, y) \) and \( C^-(x, y) \) modulo machine-word primes so as to use efficient 2D FFTs.

3. Consider \( C(x, y) = \frac{C^+(x, y)}{2}(x^K - 1) + \frac{C^-(x, y)}{2}(x^K + 1) \), then evaluate \( C(x, y) \) at \( x = \beta \), which finally gives \( c(y) = a(y) b(y) \).
Our experimental results were obtained on an 48-core AMD Opteron 6168, running at 900Mhz with 256 GB of RAM and 512KB of L2 cache.

<table>
<thead>
<tr>
<th>Size</th>
<th>Work(KS)*</th>
<th>Work(CVL₂)*</th>
<th>Span(CVL₂)*</th>
<th>Work(CVL₂)/Span(CVL₂)</th>
<th>Work(CVL₂)/Work(KS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2048</td>
<td>795,549,545</td>
<td>1,364,160,088</td>
<td>41,143,119</td>
<td>33.16</td>
<td>1.715</td>
</tr>
<tr>
<td>4096</td>
<td>4,302,927,423</td>
<td>5,663,423,709</td>
<td>96,032,325</td>
<td>58.97</td>
<td>1.316</td>
</tr>
<tr>
<td>8192</td>
<td>16,782,031,611</td>
<td>23,827,123,688</td>
<td>292,735,521</td>
<td>81.39</td>
<td>1.420</td>
</tr>
<tr>
<td>16384</td>
<td>63,573,232,166</td>
<td>100,488,072,711</td>
<td>1,017,726,160</td>
<td>98.93</td>
<td>1.584</td>
</tr>
<tr>
<td>32768</td>
<td>269,887,534,779</td>
<td>425,149,529,176</td>
<td>3,804,178,563</td>
<td>111.76</td>
<td>1.575</td>
</tr>
</tbody>
</table>

Table: Cilkview analysis of CVL₂ and KS (short for Schönaghe-Strassen). (* shows the number of instructions)
Core algorithm: parallel integer polynomial multiplication

![Graph showing the relationship between degree and time for different libraries: BPAS, FLINT, and Maple. The x-axis represents the degree, and the y-axis represents the time in seconds. The graph shows an upward trend, indicating increased time with higher degrees for all libraries.]
Core algorithm: parallel integer polynomial multiplication

Degree << Maximum bit size of a coefficient

Time (sec)

BPAS
FLINT
Maple
The adaptive algorithm based on the input size and available resources

- Very small: Plain multiplication
- Small or Single-core: Schönaghe-Strassen algorithm
- Big but a few cores: 4-way Toom-Cook
- Big: 8-way Toom-Cook
- Very big: Two-convolution method
Core algorithm: parallel Taylor shift $f(x) \longmapsto f(x + 1)$

### Parallel Pascals triangle by blocking

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>...</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>↓</td>
<td>↓</td>
<td>↓</td>
<td>↓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$f_d$</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>...</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>↓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$f_{d-1}$</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>...</td>
<td>+</td>
<td>$g_{d-1}$</td>
</tr>
<tr>
<td>↓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$f_1$</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td></td>
<td>$g_1$</td>
<td></td>
</tr>
<tr>
<td>↓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$f_0$</td>
<td>+</td>
<td></td>
<td></td>
<td></td>
<td>$g_0$</td>
<td></td>
</tr>
</tbody>
</table>

- Let $n$ be the degree and $\ell$ be the maximum bit-size of a coefficient, the complexity in terms of bit operations: $O(n^2(n + \ell))$;
- highly effective when both the input data size and the number of available cores are small due to optimal cache complexity.
Core algorithm: parallel Taylor shift $f(x) \mapsto f(x + 1)$

Algorithm E in [2]: a divide-and-conquer procedure, relying on polynomial multiplication

\[
(f_0 + f_1(x + 1)) + (f_2 + f_3(x + 1)) \times (x + 1)^2
\]

\[
f_0 + f_1(x + 1)
\]

\[
f_2 + f_3(x + 1)
\]

\[
f_0
\]

\[
f_1
\]

\[
f_2
\]

\[
f_3
\]

- Let $n$ be the degree and $\ell$ be the maximum bit-size of a coefficient, the complexity in terms of bit operations: $O(M(n^2 + n\ell) \log n)$;
- effective when the two-convolution multiplication dominates its counterparts.

Core algorithm: parallel Taylor shift $f(x) \mapsto f(x + 1)$

The adaptive algorithm based on the input size

- Small: Parallel Pascals triangle
- Big: Algorithm E in [2], but for multiplication in small degree, using parallel Pascals triangle as the base case

Plan

1. Overview
2. Code organization and user interface
3. Core algorithms
4. Applications
Applications

- Parallel univariate real root isolation
- Parallel multivariate real root isolation
- Symbolic integration
Application: parallel univariate real root isolation

**Input:** A univariate squarefree polynomial \( f(x) = c_d x^d + \cdots + c_1 x + c_0 \) with rational number coefficients

**Output:** A list of pairwise disjoint intervals \([a_1, b_1], \ldots, [a_e, b_e]\) with rational endpoints such that

- each \([a_i, b_i]\) contains one and only one real root of \( f(x) \);
- if \( a_i = b_i \), the real root \( x_i = a_i(b_i) \); otherwise, the real root \( a_i < x_i < b_i \) (\( f(x) \) doesn’t vanish at either endpoint).
The most costly operation is the Taylor Shift operation, that is, the map $f(x) \mapsto f(x + 1)$. 

Vincent-Collins-Akritas (VCA, 1976)
We run two parallel real root algorithms, BPAS and CMY [3], which are both implemented in CilkPlus, against Maple 18 serial \texttt{realroot} command, which implements a state-of-the-art algorithm.

<table>
<thead>
<tr>
<th>Size</th>
<th>BPAS (Parallel)</th>
<th>CMY [3] (Parallel)</th>
<th>\texttt{realroot} (Serial)</th>
<th>$\frac{T_{CMY}}{T_{BPAS}}$</th>
<th>$\frac{T_{realroot}}{T_{BPAS}}$</th>
<th># Roots</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cnd</td>
<td>32768</td>
<td>18.141</td>
<td>125.902</td>
<td>816.134</td>
<td>6.94</td>
<td>44.99</td>
</tr>
<tr>
<td></td>
<td>65536</td>
<td>66.436</td>
<td>664.438</td>
<td>7,526.428</td>
<td>10.0</td>
<td>113.29</td>
</tr>
<tr>
<td>Chebycheff</td>
<td>2048</td>
<td>608.738</td>
<td>594.82</td>
<td>1,378.444</td>
<td>0.98</td>
<td>2.26</td>
</tr>
<tr>
<td></td>
<td>4096</td>
<td>8,194.06</td>
<td>10,014</td>
<td>35,880.069</td>
<td>1.22</td>
<td>4.38</td>
</tr>
<tr>
<td>Laguerre</td>
<td>2048</td>
<td>1,336.14</td>
<td>1,324.33</td>
<td>3,706.749</td>
<td>0.99</td>
<td>2.77</td>
</tr>
<tr>
<td></td>
<td>4096</td>
<td>20,727.9</td>
<td>23,605.7</td>
<td>91,668.577</td>
<td>1.14</td>
<td>4.42</td>
</tr>
<tr>
<td>Wilkinson</td>
<td>2048</td>
<td>630.481</td>
<td>614.94</td>
<td>1,031.36</td>
<td>0.98</td>
<td>1.64</td>
</tr>
<tr>
<td></td>
<td>4096</td>
<td>9,359.25</td>
<td>10,733.3</td>
<td>26,496.979</td>
<td>1.15</td>
<td>2.83</td>
</tr>
</tbody>
</table>

Table: Running time (in sec.) on a 48-core AMD Opteron 6168 node for four examples.

## Application: parallel multivariate real root isolation

<table>
<thead>
<tr>
<th>Example</th>
<th>BPAS (CilkPlus)</th>
<th>RealRootIsolate (Serial Maple)</th>
<th>Isolate (Serial C)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>4-Body-Homog</td>
<td>0.402</td>
<td>0.608</td>
<td>0.382</td>
<td></td>
</tr>
<tr>
<td>Arnborg-Lazard</td>
<td>0.146</td>
<td>0.299</td>
<td>0.066</td>
<td></td>
</tr>
<tr>
<td>Caprassse</td>
<td>0.018</td>
<td>0.14</td>
<td>0.154</td>
<td>7.778</td>
</tr>
<tr>
<td>Circles</td>
<td>0.051</td>
<td>0.894</td>
<td>0.814</td>
<td>15.961</td>
</tr>
<tr>
<td>Cyclic-5</td>
<td>0.021</td>
<td>0.147</td>
<td>0.206</td>
<td>9.810</td>
</tr>
<tr>
<td>Czapor-Geddes-Wang</td>
<td>0.2</td>
<td>0.135</td>
<td>0.184</td>
<td></td>
</tr>
<tr>
<td>D2v10</td>
<td>0.029</td>
<td>0.075</td>
<td>177.999</td>
<td>2.586</td>
</tr>
<tr>
<td>D4v5</td>
<td>0.037</td>
<td>0.044</td>
<td>49.09</td>
<td>1.189</td>
</tr>
<tr>
<td>Fabfaux</td>
<td>0.192</td>
<td>0.231</td>
<td>0.071</td>
<td></td>
</tr>
<tr>
<td>Katsura-4</td>
<td>0.171</td>
<td>0.416</td>
<td>0.044</td>
<td></td>
</tr>
<tr>
<td>L-3</td>
<td>0.02</td>
<td>0.252</td>
<td>0.12</td>
<td>6.0</td>
</tr>
<tr>
<td>Neural-Network</td>
<td>0.029</td>
<td>0.332</td>
<td>0.131</td>
<td>4.517</td>
</tr>
<tr>
<td>R-6</td>
<td>0.014</td>
<td>0.048</td>
<td>20.612</td>
<td>3.429</td>
</tr>
<tr>
<td>Rose</td>
<td>0.026</td>
<td>0.336</td>
<td>0.599</td>
<td>12.923</td>
</tr>
<tr>
<td>Takeuchi-Lu</td>
<td>0.027</td>
<td>0.16</td>
<td>0.031</td>
<td>1.148</td>
</tr>
<tr>
<td>Wilkinsonxy</td>
<td>0.023</td>
<td>0.165</td>
<td>0.046</td>
<td>2.0</td>
</tr>
<tr>
<td>Nld-10-3</td>
<td>1.249</td>
<td>8.993</td>
<td>707.334</td>
<td>7.20</td>
</tr>
</tbody>
</table>

Table: Running time (in sec.) on a 12-core Intel Xeon 5650 node for BPAS vs. Maple 17 RealRootIsolate vs. C (with Maple 17 interface) Isolate.
Application: symbolic integration

R. H. C. Moir, R. M. Corless, and D. J. Jeffrey (2014, July) present an implementation based on the BPAS library, computing

\[ F(x) = \int f(x) \, dx. \]

For instance, it evaluates \( \int \frac{x^4 - 3x^2 + 6}{x^6 - 5x^4 + 5x^2 + 4} \, dx = \text{invtan}(x^3 - 3x, x^2 - 2). \)

```
/  6-3*x^2+1*x^4
| ----------------- dx =
/ 4+5*x^2-5*x^4+1*x^6
-  
    a*log((-2) + (6*a)*x + (1)*x^2 + (-2*a)*x^3)
/  
-  
a|1/4+1*a^2=0
```
Concluding remarks

- The BPAS library is the first polynomial algebra library which emphasizes performance aspects (cache complexity, parallelism) on multi-core architectures.
- Its core operations (dense integer polynomial multiplication, real root isolation) outperform their counterparts in recognized computer algebra software (FLINT, Maple).
- Its companion library CUDA Modular Polynomial (CUMODP) has similar goals on GPGPUs [www.cumodp.org](http://www.cumodp.org).
- Together, they are designed to support the implementation of polynomial system solvers on hardware accelerators.
- The BPAS library is available in source at [www.bpaslib.org](http://www.bpaslib.org).