Changbo Chen¹  Sviatoslav Covanov²  Farnam Mansouri²
Marc Moreno Maza²  Ning Xie²  Yuzhen Xie²

¹Chinese Academy of Sciences
²University of Western Ontario

ISSAC 2014 Software demo
Plan

1. Overview
2. Fast Fourier Transform
3. ModularPolynomial
4. IntegerPolynomial
5. RationalNumberPolynomial
6. Conclusion
Background

Reducing everything to multiplication

- 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).
- Can we do the same for fork-join-multithreaded algorithms?

Building blocks in scientific software

- The Basic Linear Algebra Subprograms (BLAS) is an inspiring and successful project providing low-level kernels in linear algebra, used by LINPACK, LAPACK, MATLAB, Mathematica, Julia (among others).
- Other BB’s successful projects: FFTW, SPIRAL (among others).
- The GNU Multiple Precision Arithmetic Library project plays a similar role for rational numbers and floating-point numbers.
- No symbolic computation software dedicated to sequential polynomial arithmetic managed to play the unification role of the BLAS.
- Could this work in the case of hardware accelerators?
BPAS Mandate (1/2)

## Functionalities

- **Level 1**: basic arithmetic operations that are specific to a polynomial representation or a coefficient ring: multi-dimensional FFTs/TFTs, univariate real root isolation
- **Level 2**: basic arithmetic operations for dense or sparse polynomials with coefficients in \( \mathbb{Z} \), \( \mathbb{Q} \) or \( \mathbb{Z}/p\mathbb{Z} \).
- **Level 3**: advanced arithmetic operations taking as input a zero-dimensional regular chains: normal form of a polynomial, multivariate real root isolation.
Targeted architectures

- Multi-core processors with code written in CilkPlus or OpenMP. Our Meta_Fork framework performs automatic translation between the two as well as conversions to C/C++.
- Graphics Processing Units (GPUs) with code written in CUDA, provided by CUMODP.
- 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.)
## Design

### Algorithm choice
- Level 1 functions (n-D FFTs/TFTs) are highly optimized in terms of locality and parallelism.
- Level 2 functions provide a variety of algorithmic solutions for a given operation (polynomial multiplication, Taylor shift, etc.)
- Level 3 functions combine several Level 2 algorithms for achieving a given task.

### Implementation techniques
- At Level 1, code generation at installation time (auto-tuning) is used.
- At Level 2, the user can choose between algorithms minimizing work and algorithms maximizing parallelism.
- At Level 3, this leads to adaptive algorithms that select appropriate Level 2 functions depending on available resources (number of cores, input data size).
Organization

Developer view point

- Source tree has two branches: 32bit and 64bit arithmetic with large bodies of common code
- Four sub-projects: ModularPolynomial, IntegerPolynomial, RationalNumberPolynomial and Polynomial.
- For the former three, the base ring is known at compile time while the latter provides polynomial arithmetic over an arbitrary BPAS ring.
- Python scripts generate 1-D FFT code at installation time.
- Dense representations, SLPs and sparse representations are available.

User view point today

- Only the 64bit arithmetic branch, but full access to each sub-project.
- Regression tests and benchmark scripts are also distributed.
- Documentation is generated by doxygen.
- A manually written documentation is work in progress.
Figure: A snapshot of BPAS algebraic data structures.
User interface (2/2)

```c
#include <bpas.h>

int main(int argc, char *argv[]) {
  /* Univariate Integer Polynomial Multiplication */
  DUZP a(128), b(128);
  a.read("a_input.dat"); b.read("b_input.dat");
  DUZP c = a * b;

  /* Real Root Isolation */
  mpq_class width(1, 20);
  RegularChains rcs;
  rcs.read("rcs_input.dat");
  Intervals boxes = realRootIsolation(rcs, width);

  return 0;
}
```

**Figure:** A snapshot of BPAS code.
Plan

1. Overview

2. Fast Fourier Transform

3. ModularPolynomial

4. IntegerPolynomial

5. RationalNumberPolynomial

6. Conclusion
BPAS 1-D FFT

- BPAS 1-D FFTs code is optimized in terms of cache complexity and register usage, following principles introduced by FFTW and SPIRAL.

- The FFT of a vector of size $n$ is computed in a divide-and-conquer manner until the vector size is smaller than a threshold, at which point FFTs are computed using a tiling strategy.

- At compile time, this threshold is used to generate and optimize the code. For instance, the code of all FFTs of size less or equal to $\text{HTHRESHOLD}$ are decomposed into blocks (typically performing FFTs on 8 or 16 points) for which straight-line program (SLP) machine code is generated by python scripts.

- Instruction level parallelism (ILP) is carefully considered: vectorized instructions are explicitly used (SSE2, SSE4) and instruction pipeline usage is highly optimized.

- Other environment variables are available for the user to control different parameters in the code generation.
Figure: One-dimensional modular FFTs: Modpn vs BPAS.
Plan

1. Overview
2. Fast Fourier Transform
3. ModularPolynomial
4. IntegerPolynomial
5. RationalNumberPolynomial
6. Conclusion
**2-D TFT versus 2-D FFT**

2-D FFT method on 8 cores (0.806-0.902 s, 7.2-7.3x speedup)
2-D TFT method on 8 cores (0.309-1.08 s, 6.8-7.6x speedup)

**Figure:** Timing of bivariate multiplication for input degree range of [1024, 2048) on 8 cores.
Balanced Dense Multiplication over $\mathbb{Z}/p\mathbb{Z}$

Ext.+Contr. of 4-D to 2-D TFT on 1 core (7.6-15.7 s)
Kronecker's substitution of 4-D to 1-D TFT on 1 core (6.8-14.1 s)
Ext.+Contr. of 4-D to 2-D TFT on 2 cores (1.96x speedup, 1.75x net gain)
Ext.+Contr. of 4-D to 2-D TFT on 16 cores (7.0-11.3x speedup, 6.2-10.3x net gain)
Plan

1. Overview

2. Fast Fourier Transform

3. ModularPolynomial

4. IntegerPolynomial

5. RationalNumberPolynomial

6. Conclusion
Five different integer polynomial multiplication algorithms are available:

- Schönhage-Strassen,
- 8-way Toom-Cook,
- 4-way Toom-Cook,
- divide-and-conquer plain multiplication
- and the two-convolution method.

The first one has optimal work (among the five) but is purely serial due to the difficulties of parallelizing 1D FFTs on multicore processors.

The next three algorithms are parallelized but their parallelism is static, that is, independent of the input data size; these algorithms are practically efficient when both the input data size and the number of available cores are small, for details.

The fifth algorithm relies on modular 2D FFTs which are computed by means of the row-column scheme; this algorithm delivers a high scalability and can fully utilize fat computer nodes.
KS+GMP: Reduction to GMP via Kronecker substitution

\[ f(x) := \sum_{i=0}^{n} f_i x^i, \quad g(x) := \sum_{i=0}^{m} g_i x^i, \quad h(x) = f(x) \times g(x) = \sum_{i=0}^{n+m-1} h_i x^i \]

Assume \( 0 \leq f_i < H_f \) and \( 0 \leq j < m \) for all \( f_i \)'s and \( g_j \)'s.

Then we have: \( 0 \leq h_k < \min(n, m)H_f H_g + 1 \), for \( 0 \leq k < n + m - 1 \).

Thus \( B := \lceil \log_2 \min(n, m)H_f H_g + 1 \rceil \) is the maximum number of bits required for representing a coefficient of the result.

If \( f \) or \( g \) has a negative coefficient we use a “two-complement”-trick.

**Steps**

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

**Analysis**

Work is in \( O(s\log(s)\log(\log(s))) \). \( s \) is the maximum bit size of \( f \) or \( g \) (Schönhage & Strassen). But no parallelism and modest data locality.
**Steps**

1. **Divide:** \( f(x) = f_0(x) + x^{n/2}f_1(x), \quad g(x) = g_0(x) + x^{n/2}g_1(x) \)

2. **Execute 4 sub-problems recursively.**
   - Store \( f_0 \times g_0 \) and \( f_1 \times g_1 \) in the result array.
   - Store \( f_0 \times g_1 \) and \( f_1 \times g_0 \) in auxiliary arrays.

3. **Add the auxiliary arrays to the result.**

4. **Use (one or) two DnC levels, then use the KS+GMP algorithm.**

**Analysis**

Work is in \( O(s \log(s) \log(\log(s))) \). But, w.r.t. pure KS+GMP, the constant has increased by approximately by 4. However, parallelism is close to 16.
**k-way Toom-Cook and reduction to GMP (1/4)**

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

2. **Conversion:** Set \( X = x^{n/k} \) and apply KS to the \( f_i \)'s and \( g_j \)'s. Obtaining \( F(X) = Zf_0 + Zf_1X + \ldots + Zf_{k-1}X^{k-1} \) and \( G(X) = Zg_0 + Zg_1X + \ldots + Zg_{k-1}X^{k-1} \).

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: \( H(X) = f(X)g(X) = Z_{h_0} + Z_{h_1}X + \ldots + Z_{h_{2k-2}}X^{2k-2} \).

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

7. **Merge:** Add intermediate results to the final result.
Parallelization

All steps must be parallelized. For the evaluation and interpolation, we use the fact that these steps can be done via linear algebra.

\[
\begin{bmatrix}
Z_{h_0} \\
Z_{h_1} \\
. \\
. \\
. \\
Z_{h_{14}}
\end{bmatrix}
= 
\begin{bmatrix}
1 & 0 & 0 & 0 & \ldots & 0 \\
X_1^0 & X_1^1 & X_1^2 & X_1^3 & \ldots & X_1^{14} \\
. & . & . & . & \ldots & . \\
. & . & . & . & \ldots & . \\
X_{13}^0 & X_{13}^1 & X_{13}^2 & X_{13}^3 & \ldots & X_{13}^{14} \\
0 & 0 & 0 & 0 & \ldots & 1
\end{bmatrix}^{-1} \times
\begin{bmatrix}
w_0 \\
w_1 \\
w_2 \\
w_3 \\
w_4 \\
w_5 \\
w_6 \\
w_7 \\
w_{14}
\end{bmatrix}
\]

Calculations are distributed among workers.
Analysis

- Work in $O(s \log(s) \log(\log(s)))$. W.r.t. pure KS+GMP, for $k = 8$, the constant has increased approximately by 2 for $s = 2^{24}$.
- However, parallelism is about 7 and 13 when $k = 4$ and $k = 8$, resp.
- In the 3 tables below, a 12-core (Intel Xeon) is used.
- In the last one, we have $\sqrt{s} = 8192$.

### Toom-8 Profiled results

<table>
<thead>
<tr>
<th>$\sqrt{s}$</th>
<th>Div. &amp; Conv.</th>
<th>Eval.</th>
<th>Mul.</th>
<th>Inter.</th>
<th>Con. &amp; Merge</th>
</tr>
</thead>
<tbody>
<tr>
<td>16384</td>
<td>10%</td>
<td>8%</td>
<td>44%</td>
<td>25%</td>
<td>9%</td>
</tr>
<tr>
<td>32768</td>
<td>9%</td>
<td>9%</td>
<td>43%</td>
<td>27%</td>
<td>10%</td>
</tr>
<tr>
<td>65536</td>
<td>8%</td>
<td>8%</td>
<td>45%</td>
<td>28%</td>
<td>9%</td>
</tr>
</tbody>
</table>

### Toom-4 Profiled results

<table>
<thead>
<tr>
<th>$\sqrt{s}$</th>
<th>Div. &amp; Conv.</th>
<th>Eval.</th>
<th>Mul.</th>
<th>Inter.</th>
<th>Con. &amp; Merge</th>
</tr>
</thead>
<tbody>
<tr>
<td>16384</td>
<td>13%</td>
<td>3%</td>
<td>57%</td>
<td>11%</td>
<td>13%</td>
</tr>
<tr>
<td>32768</td>
<td>12%</td>
<td>3%</td>
<td>61%</td>
<td>11%</td>
<td>12%</td>
</tr>
<tr>
<td>65536</td>
<td>8%</td>
<td>2%</td>
<td>66%</td>
<td>10%</td>
<td>11%</td>
</tr>
</tbody>
</table>
### Notes

- Span of Toom-8 is the best, 12% of the KS
  → It should be much better on better machines.
- Work of the Toom-8 & Toom-4 are more than KS, but much better than DnC.
- But for Toom-8 and Toom-4, GMP-multiplication is not the dominant part.
- Hence, we need an algorithm which can scale on fatter nodes.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Timing</th>
<th>Work</th>
<th>Span</th>
<th>Work/serial</th>
<th>Span/serial</th>
</tr>
</thead>
<tbody>
<tr>
<td>KS (Ser.)</td>
<td>1</td>
<td>16781990263</td>
<td>16781990263</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>DnC</td>
<td>15.86</td>
<td>67222430575</td>
<td>4237755216</td>
<td>4</td>
<td>0.25</td>
</tr>
<tr>
<td>Toom-4</td>
<td>6.42</td>
<td>28289430113</td>
<td>4382572841</td>
<td>1.68</td>
<td>0.26</td>
</tr>
<tr>
<td>Toom-8</td>
<td>11.26</td>
<td>24449014227</td>
<td>2023790230</td>
<td>1.45</td>
<td>0.12</td>
</tr>
</tbody>
</table>
Two-convolution method (1/7)

Specifications

- **Input**: $a(y), b(y) \in \mathbb{Z}[y]$ with $\max(\deg(a), \deg(b)) < d$ and $N_0$ the maximum bit size of a coefficient among $a(y)$ and $b(y)$.
- **Intention**: $a(y), b(y)$ are dense in the sense that most coefficients have a size in the order of $N_0$.
- **Output**: $c(y) = a(y)b(y)$.

Theorem

Let $w$ be the number of bits of a machine word. There exist positive integers $N, K, M$, with $N = K M$ and $M \leq w$, such that the integer $N$ is $w$-smooth (and so is $K$), we have $N_0 < N \leq N_0 + \sqrt{N_0}$ and the following algorithm for multiplying $a(y), b(y)$ has

- a work of $O(d K \log_2(d K)(\log_2(d K) + 2M))$ word operations,
- a span of $O(K \log_2(d) \log_2(d K))$ word operations and,
- incurs $O(\lceil dN/wL \rceil + \lceil (\log_2(d K) + 2M) \rceil dK/L)$ cache misses,
Two-convolution method (2/7)

**Principle**

1. **Convert-in**: convert the integer coefficients of \(a(y), b(y)\) to \(\mathbb{Z}[x]\), thus converting \(a(y), b(y)\) to \(A(x, y), B(x, y)\) s.t. for some \(\beta \in \mathbb{Z}\):
   1. \(a(y) = A(\beta, y)\) and \(b(y) = B(\beta, y)\),
   2. \(\deg(A, y) = \deg(a)\) and \(\deg(B, y) = \deg(b)\),
   3. \(\max(\text{bitsize}(a), \text{bitsize}(b)) \in \Theta(\max(\text{bitsize}(a), \text{bitsize}(b)))\)

Let \(m > 4H\) be an integer, where \(H\) is the maximum absolute value of a coefficient of the integer polynomial \(C(x, y) := A(x, y)B(x, y)\).

2. **Compute**: \(m\) and \(A(x, y), B(x, y)\) are constructed such that
   1. the polynomials \(C^+(x, y) := A(x, y)B(x, y) \mod \langle x^K + 1 \rangle\) and \(C^-(x, y) := A(x, y)B(x, y) \mod \langle x^K - 1 \rangle\) are computed over \(\mathbb{Z}/m\mathbb{Z}\) via FFT techniques
   2. meanwhile the following equation holds over \(\mathbb{Z}\):
   \[
   C(x, y) = \frac{C^+(x, y)}{2}(x^K - 1) + \frac{C^-(x, y)}{2}(x^K + 1).
   \]

3. **Convert-out**: Compute \(u(y) := C^+(\beta, y)\) and \(v(y) := C^-(\beta, y)\) over \(\mathbb{Z}\). Then, deduce \(c(y) := \frac{u(y) + v(y)}{2} + \frac{-u(y) + v(y)}{2}2^N\) over \(\mathbb{Z}\).
Two-convolution method (3/7)

Figure: Multiplication scheme for dense univariate integer polynomials.
Two-convolution method (4/7)

Principle (recall)

1. **Convert-in**: convert \( a(y), b(y) \) to \( A(x, y), B(x, y) \) s.t. for some \( \beta \in \mathbb{Z} \) we have \( a(y) = A(\beta, y) \) and \( b(y) = B(\beta, y) \).

2. Let \( m > 4H \) where \( H := \| C(x, y) \|_\infty \) with \( C(x, y) := A(x, y)B(x, y) \).

3. **Compute**: \( C^+(x, y) := A(x, y)B(x, y) \mod \langle x^K + 1 \rangle \) and \( C^-(x, y) := A(x, y)B(x, y) \mod \langle x^K - 1 \rangle \) over \( \mathbb{Z}/m\mathbb{Z} \).

4. **Convert-out**: Compute \( u(y) := C^+(\beta, y) \) and \( v(y) := C^-(\beta, y) \) over \( \mathbb{Z} \). Then, deduce \( c(y) := \frac{u(y)+v(y)}{2} + \frac{-u(y)+v(y)}{2} 2^N \) over \( \mathbb{Z} \).

Remarks

- For polynomials with size in the Giga-bytes, we can choose \( m < 2w \). Thus each of \( C^+(x, y) \) and \( C^-(x, y) \) requires at most two 2-D FFT/TFT over a prime field with characteristic of machine word size.

- **Convert-in** and **Convert-out** are done only with addition and shift operations on byte vectors: GMP is not used.

- The cache complexity of this process is proved to be optimal.
This about four faster than Toom-8 at $\sqrt{s} = 8192$. 

<table>
<thead>
<tr>
<th>$\sqrt{s}$</th>
<th>ConvertIn</th>
<th>TwoConvolution</th>
<th>ConvertOut</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>2048</td>
<td>0.037</td>
<td>0.113</td>
<td>0.052</td>
<td>0.227</td>
</tr>
<tr>
<td>4096</td>
<td>0.028</td>
<td>0.206</td>
<td>0.103</td>
<td>0.364</td>
</tr>
<tr>
<td>8192</td>
<td>0.07</td>
<td>0.652</td>
<td>0.307</td>
<td>1.059</td>
</tr>
<tr>
<td>16384</td>
<td>0.224</td>
<td>2.71</td>
<td>0.73</td>
<td>3.698</td>
</tr>
<tr>
<td>32768</td>
<td>0.943</td>
<td>11.796</td>
<td>4.174</td>
<td>16.978</td>
</tr>
</tbody>
</table>

**Table:** Using 3 primes on a 48-core AMD Opteron node.
Figure: Dense integer polynomial multiplication: BPAS vs FLINT vs Maple 18.
Figure: Dense integer polynomial multiplication: BPAS vs FLINT vs Maple 18.
Plan

1. Overview
2. Fast Fourier Transform
3. ModularPolynomial
4. IntegerPolynomial
5. RationalNumberPolynomial
6. Conclusion
Univariate Real Root Isolation Algorithm: Find Roots

**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).

![Diagram of a polynomial function with intervals labeled](image)

**Figure:** An example of input / output
Real Root Isolation (Collins-Vincent-Akritas Algorithm)

**Algorithm 1: NumberInZeroOne(p)**

**Input:** a squarefree univariate polynomial $p$

**Output:** number of real roots of $p$ in $(0,1)$

**begin**

\[ p_1 := x^n p(1/x); \quad p_2 := p_1(x + 1) \]

let $d$ be the number of sign variations of the coefficients of $p_2$

if $d \leq 1$ then return $d$

\[ p_1 := 2^n p(x/2); \quad p_2 := p_1(x + 1) \]

if $x \mid p_2$ then $m := 1$ else $m := 0$

$m' := \text{NumberInZeroOne}(p_1)$

$m := m + \text{NumberInZeroOne}(p_2)$

return $m + m'$

**end**

The Taylor shift $f(x) \mapsto f(x + 1)$ operation is at the core of the above algorithm for real root isolation (counting).
Horner’s method

We compute \( g(x) = f_0 + (x + 1)(f_1 + \cdots + (x + 1)f_d) \) in n steps

\[
g^{(0)} = f_d, \quad g^{(i)} = (x + 1)g^{(i-1)} + f_{d-i} \quad \text{for} \quad 1 \leq i \leq d,
\]

and obtain \( g = g^{(d)} \).

One can represent this computation in a Pascal’s Triangle.

Given an example \( f(x) = a_3 x^3 + a_2 x^2 + a_1 x + a_0 \), we have, in Horner’s rule as follows.

\[
\begin{array}{ccccccc}
0 & 0 & 0 & 0 & 0 & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & c_3 \\
0 & 0 & 0 & 0 & 0 & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & c_2 \\
0 & 0 & 0 & 0 & 0 & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & c_1 \\
0 & 0 & 0 & 0 & 0 & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & c_0 \\
\end{array}
\]

Thus, \( g(x) = f(x + 1) = c_3 x^3 + c_2 x^2 + c_1 x + c_0 \). For instance, we can parallelize the addition \( a_1 + a_2, \ a_2 + a_3 \) and \( a_3 + 0 \), and so on.
Univariate Taylor Shift: Parallel Divide & Conquer Method

Divide & conquer method

We assume that \( d + 1 = 2^\ell \) is a power of two. In a precomputation stage, we compute \((1 + x)^{2^i}\) for \(0 \leq i < \ell\). In main stage, with polynomials \( f^{(0)}, f^{(1)} \in \mathbb{Q}[x] \) of degree less than \( \frac{d+1}{2} \), we compute

\[
g(x) = f^{(0)}(x + 1) + (x + 1)^{(d+1)/2} f^{(1)}(x + 1),
\]

where we compute \( f^{(0)}(x + 1) \) and \( f^{(1)}(x + 1) \) recursively.

We parallelize each univariate polynomial multiplication by a DnC method. For example,

\[
f(x) = f_0 + f_1x + f_2x^2 + f_3x^3 + f_4x^4 + f_5x^5 + f_6x^6 + f_7x^7
\]
\[
g(x) = f(x + 1)
\]
Benchmarks (1/2)

<table>
<thead>
<tr>
<th>Degree</th>
<th>BPAS</th>
<th>CMX-2010</th>
<th>realroot</th>
<th>#Roots</th>
</tr>
</thead>
<tbody>
<tr>
<td>4095</td>
<td>3.31</td>
<td>2.622</td>
<td>7.137</td>
<td>1</td>
</tr>
<tr>
<td>8191</td>
<td>13.036</td>
<td>17.11</td>
<td>40.554</td>
<td>1</td>
</tr>
<tr>
<td>16383</td>
<td>61.425</td>
<td>124.798</td>
<td>274.706</td>
<td>1</td>
</tr>
<tr>
<td>32767</td>
<td>286.46</td>
<td>970.165</td>
<td>2059.147</td>
<td>1</td>
</tr>
<tr>
<td>65535</td>
<td>1311.07</td>
<td>7960.57</td>
<td>*Err.</td>
<td>1</td>
</tr>
</tbody>
</table>

Table: Running time of $Bn, d(x) = 2^d x^d + .. + 2^d$ on Intel 12-core

<table>
<thead>
<tr>
<th>Degree</th>
<th>BPAS</th>
<th>CMX-2010</th>
<th>realroot</th>
<th>#Roots</th>
</tr>
</thead>
<tbody>
<tr>
<td>4095</td>
<td>1.93</td>
<td>1.917</td>
<td>3.941</td>
<td>1</td>
</tr>
<tr>
<td>8191</td>
<td>6.467</td>
<td>7.611</td>
<td>29.448</td>
<td>1</td>
</tr>
<tr>
<td>16383</td>
<td>25.291</td>
<td>34.198</td>
<td>239.771</td>
<td>1</td>
</tr>
<tr>
<td>32767</td>
<td>96.507</td>
<td>172.432</td>
<td>1863.125</td>
<td>1</td>
</tr>
<tr>
<td>65535</td>
<td>375.678</td>
<td>995.358</td>
<td>*Err.</td>
<td>1</td>
</tr>
</tbody>
</table>

Table: Running time of $Cn, d(x) = x^d + d$ on Intel 12-core
## Benchmarks (2/2)

<table>
<thead>
<tr>
<th></th>
<th>Size</th>
<th>BPAS</th>
<th>CMX-2010</th>
<th>realroot</th>
<th>#Roots</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bnd</strong></td>
<td>4096</td>
<td>4.003</td>
<td>5.125</td>
<td>4.955</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>8192</td>
<td>11.025</td>
<td>25.228</td>
<td>23.754</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>16384</td>
<td>43.498</td>
<td>127.412</td>
<td>159.245</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>32768</td>
<td>176.351</td>
<td>609.513</td>
<td>1,011.872</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>65536</td>
<td>701.682</td>
<td>4,695.31</td>
<td>9,741.249</td>
<td>1</td>
</tr>
<tr>
<td><strong>Cnd</strong></td>
<td>4096</td>
<td>0.704</td>
<td>1.209</td>
<td>1.699</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>8192</td>
<td>1.533</td>
<td>5.228</td>
<td>10.899</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>16384</td>
<td>5.086</td>
<td>25.296</td>
<td>109.420</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>32768</td>
<td>18.141</td>
<td>125.902</td>
<td>816.134</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>65536</td>
<td>66.436</td>
<td>664.438</td>
<td>7,526.428</td>
<td>1</td>
</tr>
<tr>
<td><strong>Chebycheff</strong></td>
<td>2048</td>
<td>608.738</td>
<td>594.82</td>
<td>1,378.444</td>
<td>2047</td>
</tr>
<tr>
<td></td>
<td>4096</td>
<td>8,194.06</td>
<td>10,014</td>
<td>35,880.069</td>
<td>4095</td>
</tr>
<tr>
<td><strong>Laguerre</strong></td>
<td>2048</td>
<td>1,336.14</td>
<td>1,324.33</td>
<td>3,706.749</td>
<td>2047</td>
</tr>
<tr>
<td></td>
<td>4096</td>
<td>20,727.9</td>
<td>23,605.7</td>
<td>91,668.577</td>
<td>4095</td>
</tr>
<tr>
<td><strong>Wilkinson</strong></td>
<td>2048</td>
<td>630.481</td>
<td>614.94</td>
<td>1,031.36</td>
<td>2047</td>
</tr>
<tr>
<td></td>
<td>4096</td>
<td>9,359.25</td>
<td>10,733.3</td>
<td>26,496.979</td>
<td>4095</td>
</tr>
</tbody>
</table>

**Table:** Running times on AMD 48-core
Plan

1. Overview

2. Fast Fourier Transform

3. ModularPolynomial

4. IntegerPolynomial

5. RationalNumberPolynomial

6. Conclusion
www.bpaslib.org