# Basic Polynomial Algebra Subprograms

Changbo Chen[1] Sviatoslav Covanov[2] Farnam Mansouri[2]
Marc Moreno Maza[2] Ning Xie[2] Yuzhen Xie[2]

[1]Chinese Academy of Sciences
[2]University of Western Ontario

ISSAC 2014 Software demo

# Plan

# 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.

# BPAS Mandate (2/2)

## 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.

# User interface (1/2)

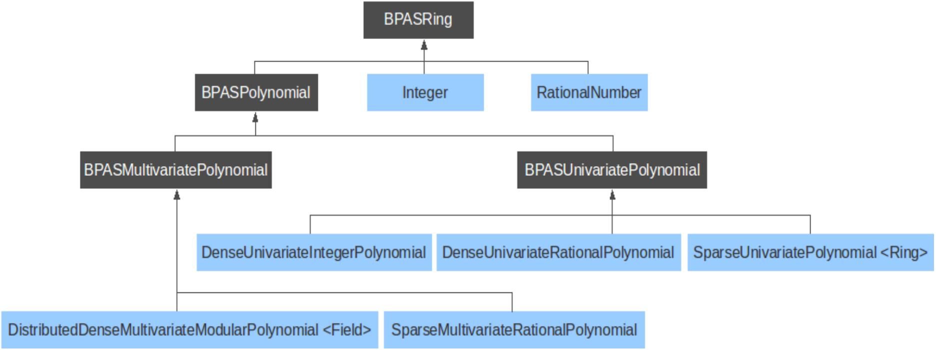



Figure: A snapshot of BPAS algebraic data structures.

# User interface (2/2)

```
#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

# 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 `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.

# 1-D FFT Benchmarks

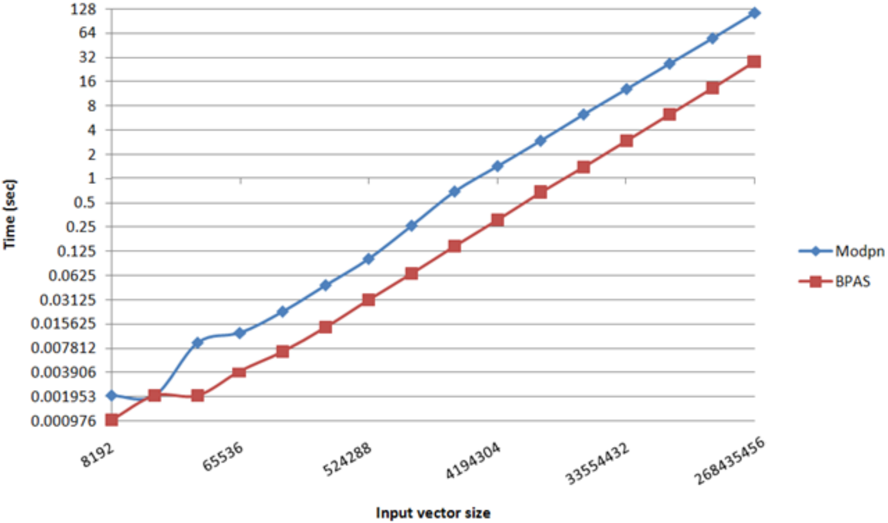



Figure: One-dimensional modular FFTs: Modpn vs BPAS.

# Plan

# 2-D TFT versus 2-D FFT

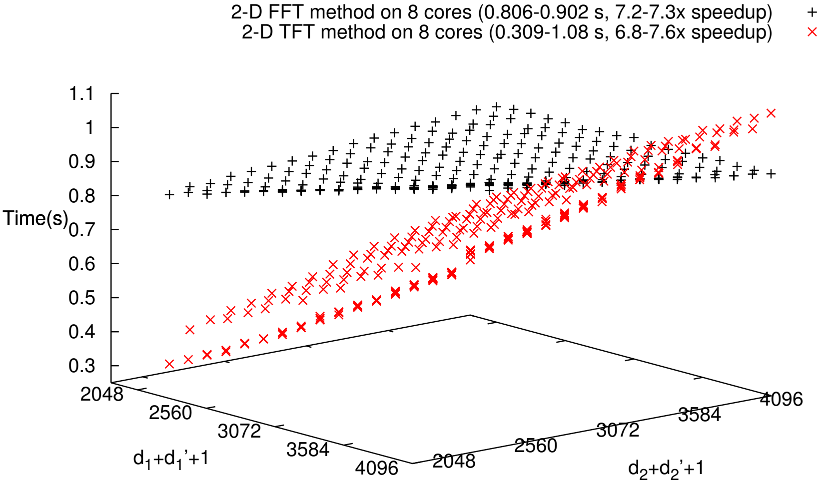
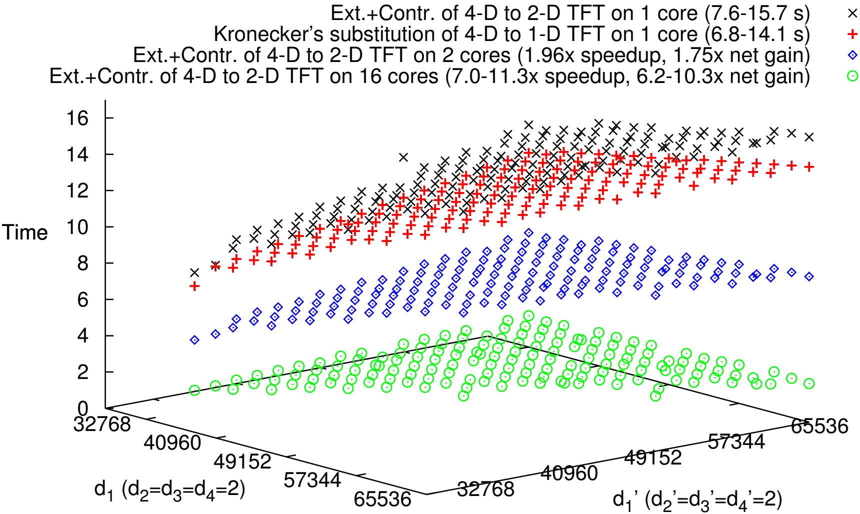



Figure: Timing of bivariate multiplication for input degree range of $[1024, 2048)$ on 8 cores.

# Balanced Dense Multiplication over $\mathbb{Z}/p\mathbb{Z}$



# Plan

# Integer polynomial multiplication

- 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.

# DnC+KS+GMP: GMP Reduction via KS and distributivity

## 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 $F(X) = Z_{f_0} + Z_{f_1} X + \ldots + Z_{f_{k-1}} X^{k-1}$ and $G(X) = Z_{g_0} + Z_{g_1} X + \ldots + Z_{g_{k-1}} X^{k-1}$.
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.

# $k$-way Toom-Cook and reduction to GMP (2/4)

## Parallelization

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

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

Calculations are distributed among workers.

# $k$-way Toom-Cook and reduction to GMP (3/4)

## 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**

| $\sqrt{s}$ | Div. & Conv. | Eval. | Mul. | Inter. | Con. & Merge |
|---|---|---|---|---|---|
| 16384 | 10% | 8% | 44% | 25% | 9% |
| 32768 | 9% | 9% | 43% | 27% | 10% |
| 65536 | 8% | 8% | 45% | 28% | 9% |

**Toom-4 Profiled results**

| $\sqrt{s}$ | Div. & Conv. | Eval. | Mul. | Inter. | Con. & Merge |
|---|---|---|---|---|---|
| 16384 | 13% | 3% | 57% | 11% | 13% |
| 32768 | 12% | 3% | 61% | 11% | 12% |
| 65536 | 8% | 2% | 66% | 10% | 11% |

# $k$-way Toom-Cook and reduction to GMP (4/4)

| Algorithm | Timing | Work | Span | Work/serial | Span/serial |
|---|---|---|---|---|---|
| KS (Ser.) | 1 | 16781990263 | 16781990263 | 1 | 1 |
| DnC | 15.86 | 67222430575 | 4237755216 | 4 | 0.25 |
| Toom-4 | 6.42 | 28289430113 | 4382572841 | 1.68 | 0.26 |
| Toom-8 | 11.26 | 24449014227 | 2023790230 | 1.45 | 0.12 |

## Notes

- Span of Toom-8 is the best, 12 % of the KS
  $\rightarrow$ 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.

# 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. $\max(\mathrm{bitsize}(a), \mathrm{bitsize}(b)) \in \Theta(\max(\mathrm{bitsize}(a), \mathrm{bitsize}(b)))$

   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)

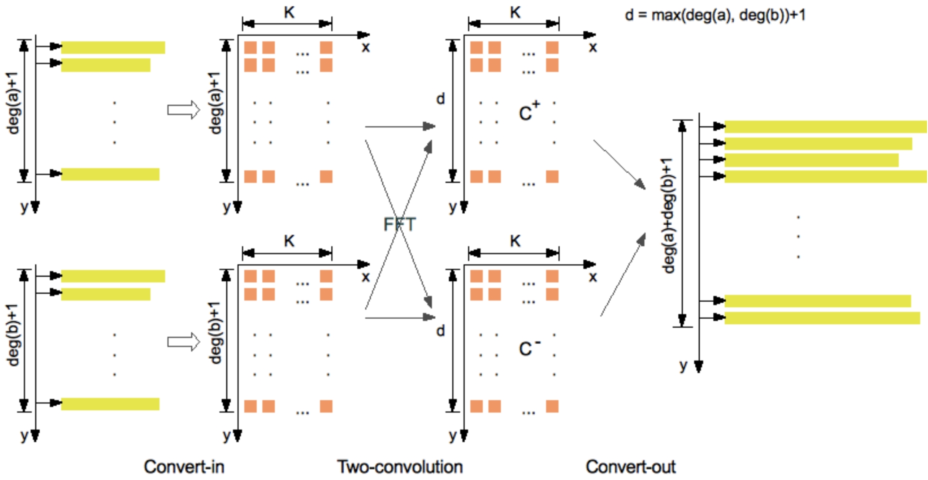



Figure: Multiplication scheme for dense univariate integer polynomials.

# 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.

# Two-convolution method (5/7)

| $\sqrt{s}$ | ConvertIn | TwoConvolution | ConvertOut | Total |
|---|---|---|---|---|
| 2048 | 0.038 | 0.106 | 0.042 | 0.195 |
| 4096 | 0.039 | 0.23 | 0.107 | 0.39 |
| 8192 | 0.119 | 0.895 | 0.267 | 1.298 |
| 16384 | 0.248 | 3.705 | 0.665 | 4.643 |
| 32768 | 0.943 | 16.272 | 4.217 | 21.496 |

Table: Using 2 primes on a 48-core AMD Opteron node.

| $\sqrt{s}$ | ConvertIn | TwoConvolution | ConvertOut | Total |
|---|---|---|---|---|
| 2048 | 0.037 | 0.113 | 0.052 | 0.227 |
| 4096 | 0.028 | 0.206 | 0.103 | 0.364 |
| 8192 | 0.07 | 0.652 | 0.307 | 1.059 |
| 16384 | 0.224 | 2.71 | 0.73 | 3.698 |
| 32768 | 0.943 | 11.796 | 4.174 | 16.978 |

Table: Using 3 primes on a 48-core AMD Opteron node.

This about four faster than Toom-8 at $\sqrt{s} = 8192$.

# Two-convolution method (6/7)


Figure: Dense integer polynomial multiplication: BPAS vs FLINT vs Maple 18.

# Two-convolution method (7/7)

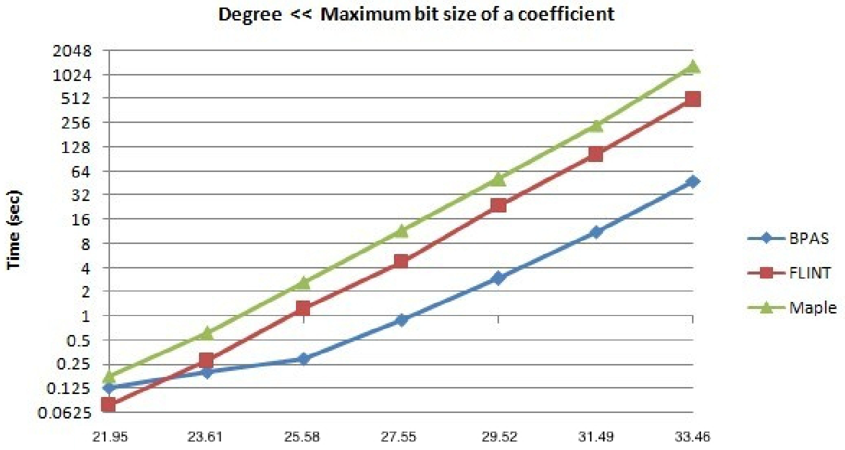



Figure: Dense integer polynomial multiplication: BPAS vs FLINT vs Maple 18.

# 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).

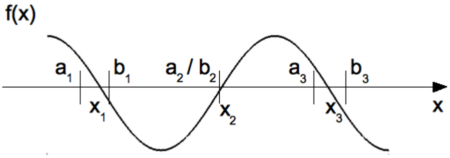



Figure: An example of input / output

# 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**
  $p_1 := x^n p(1/x)$;  $p_2 := p_1(x+1)$
  let $d$ be the number of sign variations of the coefficients of $p_2$
  **if** $d \leq 1$ **then** return $d$
  $p_1 := 2^n p(x/2)$;  $p_2 := p_1(x+1)$
  **if** $x \mid p_2$ **then** $m := 1$ **else** $m := 0$
  $m' :=$ NumberInZeroOne$(p_1)$
  $m := m +$ NumberInZeroOne$(p_2)$
  return $m + m'$
**end**

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

# Univariate Taylor Shift: Parallel Horner's Method

## Horner's method

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

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

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.

$$\begin{array}{ccccccccccc}
 & & 0 & & 0 & & 0 & & 0 & & \\
 & & \downarrow & & \downarrow & & \downarrow & & \downarrow & & \\
a_3 & \rightarrow & + & \rightarrow & + & \rightarrow & + & \rightarrow & + & \rightarrow & c_3 \\
 & & \downarrow & & \downarrow & & \downarrow & & & & \\
a_2 & \rightarrow & + & \rightarrow & + & \rightarrow & + & \rightarrow & c_2 & & \\
 & & \downarrow & & \downarrow & & & & & & \\
a_1 & \rightarrow & + & \rightarrow & + & \rightarrow & c_1 & & & & \\
 & & \downarrow & & & & & & & & \\
a_0 & \rightarrow & + & \rightarrow & c_0 & & & & & &
\end{array}$$

Thus, $g(x) = f(x+1) = c_3\,x^3 + c_2\,x^2 + c_1\,x_1 + c_0$. 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

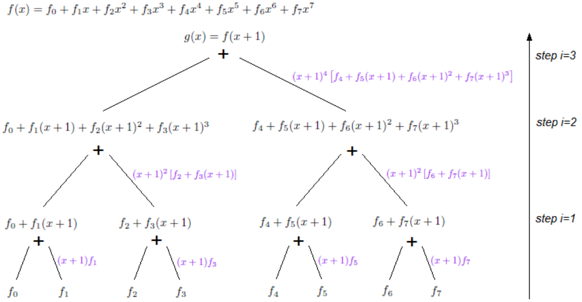
## Divide & conquer method

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,



# Benchmarks (1/2)

| Degree | BPAS | CMX-2010 | *realroot* | #Roots |
|---|---|---|---|---|
| 4095 | 3.31 | 2.622 | 7.137 | 1 |
| 8191 | 13.036 | 17.11 | 40.554 | 1 |
| 16383 | 61.425 | 124.798 | 274.706 | 1 |
| 32767 | 286.46 | 970.165 | 2059.147 | 1 |
| 65535 | 1311.07 | 7960.57 | *Err. | 1 |

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

| Degree | BPAS | CMX-2010 | *realroot* | #Roots |
|---|---|---|---|---|
| 4095 | 1.93 | 1.917 | 3.941 | 1 |
| 8191 | 6.467 | 7.611 | 29.448 | 1 |
| 16383 | 25.291 | 34.198 | 239.771 | 1 |
| 32767 | 96.507 | 172.432 | 1863.125 | 1 |
| 65535 | 375.678 | 995.358 | *Err. | 1 |

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

# Benchmarks (2/2)

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

Table: Running times on AMD 48-core

# Plan

# Summary

Basic Polynomial Algebra Subprograms

www.bpaslib.org