Basic Polynomial Algebra Subprograms

Changbo Chen[1] Svyatoslav Covanov[2,3] Farnam Mansouri[2]
Marc Moreno Maza[2] Ning Xie[2] Yuzhen Xie[2]

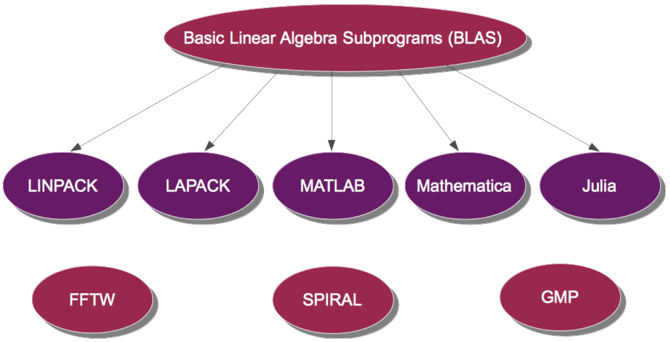
[1]Chinese Academy of Sciences, China
[2]University of Western Ontario, Canada
[3]École Polytechnique, France

ICMS, August 8 2014

Plan

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

Overview: building blocks in scientific software



- 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 to multiplication 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, . . .

Overview: targeted architectures

- The BPAS library http://www.bpaslib.org is written in C++ with CilkPlus http://www.cilkplus.org/ extension targeting multi-cores.
- 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,
- automatic 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 between algorithms minimizing work or algorithms maximizing parallelism.
- Example: Schönaghe-Strassen, divide-and-conquer, 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

Code organization

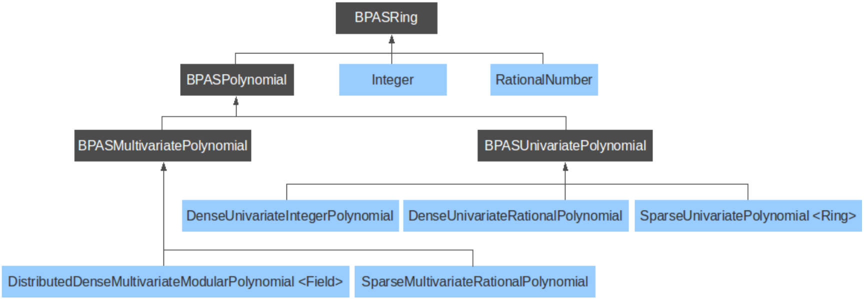
Subprojects

- 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`.

User interface



- The above is a snapshot of the BPAS ring classes
- This shows two multivariate polynomial concrete classes, namely `DistributedDenseMultivariateModularPolynomial<Field>` and `SMQP`, and three univariate polynomial ones, namely `DUZP`, `DUQP` and `SparseUnivariatePolynomial<Ring>`.
- The BPAS classes `Integer` and `RationalNumber` are BPAS wrappers for GMP's `mpz` and `mpq` classes.
- Many other classes are provided like `Intervals`, `RegularChains`, ...

User interface: code example

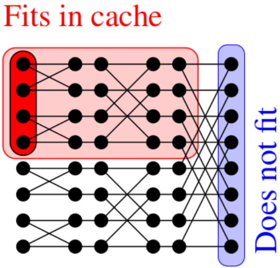
```
#include <bpas.h>

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^2) - (b^2),  d = (a^3) - (b^3);
    DUZP g = c.gcd(d); // Gcd computation, g = a − b
    c /= g; // Exact division, c = a + b
    std::cout << "g = " << g << std::endl;

    DUQP p; // Initializing as a zero polyomial
    p = (p + mpq_class(1) << 4095) + mpq_class(4095); // p = x^4095 + 4095
    Intervals boxes = p.realRootIsolation(0.5);
    std::cout << "boxes = " << boxes << std::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) { std::cout << z << " & " << h << " should not differ " << std::endl; }

    return 0;
}
```

Plan

Three core subprograms

- One-dimensional modular FFTs
- Parallel dense integer polynomial multiplication
- Parallel Taylor shift computation $f(x) \longmapsto f(x+1)$

1-D FFTs: classical cache friendly algorithm



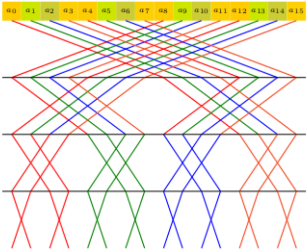
FFT($[a_0, a_1, \cdots, a_{n-1}], \omega$)

- **if** $n \leq$ HTHRESHOLD **then**
  - ArrayBitReversal($a_0, a_1, \cdots, a_{n-1}$)
  - **return** FFT_iterative_in_cache($[a_0, a_1, \cdots, a_{n-1}], \omega$)
- **end if**
- Shuffle($a_0, a_1, \cdots, a_{n-1}$)
- $[a_0, a_1, \cdots, a_{n/2-1}]$=FFT($[a_0, a_1, \cdots, a_{n/2-1}], \omega^2$)
- $[a_{n/2}, a_{n/2+1}, \cdots, a_{n-1}]$=FFT($[a_{n/2}, a_{n/2+1}, \cdots, a_{n-1}], \omega^2$)
- **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 friendly 1-D FFT

- 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 yields a cache complexity of $\Omega(n/L(\log_2(n) - \log_2(Z)))$ which is not optimal.

1-D FFTs: cache complexity optimal algorithm



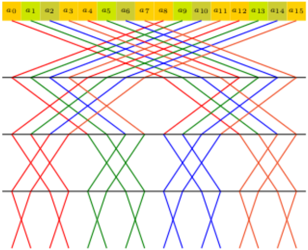
Fast Fourier Transform in R

```
procedure FFT((α_0 α_1 ... α_{N-1}), ω, N = J·K, Ω = ω^{N/K})
  for 0 ≤ k < K − 1 do                                  ▷ Inner transforms
    for 0 ≤ k' < J − 1 do
      γ[k][k'] = α_{k'K+k}
    end for
    c[k] = FFT(γ[k], ω^K, J, Ω)
  end for
  for 0 ≤ j < J − 1 do                                  ▷ Outer transforms
    for 0 ≤ k < K − 1 do
      δ[j][k] = c[k][j] * ω^{jk}                        ▷ Computation of coefficients
    end for
    d[j] = FFT(δ[j], ω^J, K, Ω)
    for 0 ≤ j' < K − 1 do
      β_{j'J+j} = d[j][j']
    end for
  end for
  return b = (β_0, ..., β_{N-1})
end procedure
```

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 the 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 $\boxed{O(n/L(\log_2(n)/\log_2(Z)))\ \text{which is optimal.}}$

1-D FFTs in BPAS: putting Fürer's algorithm into practice



Second Strategy FFT

```
procedure FFT((α_0 α_1 ... α_{N-1}), ω, N = 2^{k-j} · 2^j, Ω = ω^{2^{k-j}})
    for 0 ≤ l < 2^j − 1 do                              ▷ Inner transforms
        for 0 ≤ l' < 2^{k-j} − 1 do
            γ[l][l'] = α_{l' 2^j + l}
        end for
        c[l] = FFT(γ[l], ω^K, 2^{k-j}, Ω)
    end for
    for 0 ≤ i < 2^{k-j} − 1 do                          ▷ Outer transforms
        for 0 ≤ l < 2^j − 1 do
            δ[i][l] = c[l][i] * ω^{il}                  ▷ Computation of coefficients
        end for
        d[i] = FFT(δ[i], ω^{2^{k-j}}, 2^j, Ω)
        for 0 ≤ i' < 2^j − 1 do
            β_{i' 2^{k-j} + i} = d[i][i']
        end for
    end for
    return b = (β_0, ..., β_{N-1})
end procedure
```

Cache-and-work optimal 1-D FFT

- Modifying the previous blocking strategy such that each block is an FFT on 2^K points, for a given K (small in practice), 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 $\boxed{O(n \log_K(n)) \text{ which is optimal on today's desktop computers.}}$

1-D FFTs in BPAS

- 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.
- BPAS 1-D FFT code automatically generated by configurable Python scripts.

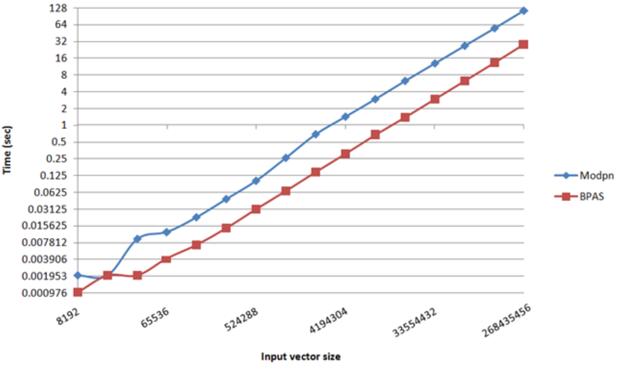


Figure: 1-D modular FFTs: Modpn (**serial**) vs BPAS (**serial**).

Parallel dense integer polynomial multiplication

Reducing to Schönaghe-Strassen algorithm via Kronecker's substitution (KS+SS)

0 **Input**: $f = \sum_{i=0}^{n} f_i\, x^i$ and $g = \sum_{i=0}^{m} g_i\, x^i$

1 **Choose**: $2^\ell \geq ||f||_\infty + ||g||_\infty + \max(n, m) + 1$

2 **Evaluation**: $Z_f = \sum_{i=0}^{n} f_i\, 2^{i\ell}$ and $Z_g = \sum_{i=0}^{m} g_i\, 2^{i\ell}$;

3 **Multiplying**: $Z_h = Z_f \times Z_g$, using GMP library;

4 **Unpacking**: h_i from $Z_h = \sum_{i=0}^{n+m} h_i\, 2^{i\ell}$.

5 **Return**: $f\, g = \sum_{i=0}^{n+m} h_i\, x^i$

- its work in terms of bit operations is $O(s \log_2(s) \log_2(\log_2(s)))$, where s is the maximum bit-size of f or g;
- purely serial due to the difficulties of parallelizing 1-D FFTs on multicore processors.

Parallel dense integer polynomial multiplication

Divide-and-conquer algorithm with reduction to GMP's integer multiplication

1. **Division**: $f(x) = f_0(x) + f_1(x)\,x^{n/2}$ and $g(x) = g_0(x) + g_1(x)\,x^{n/2}$;
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 levels of recursion, then use the KS+SS algorithm;
- its work in terms of bit operations is $O(s \log_2(s) \log_2(\log_2(s)))$, where s is the maximum bit-size of f or g, but the constant has been multiplied approximately by 4;
- static parallelism (close to 16).

Parallel dense integer polynomial multiplication

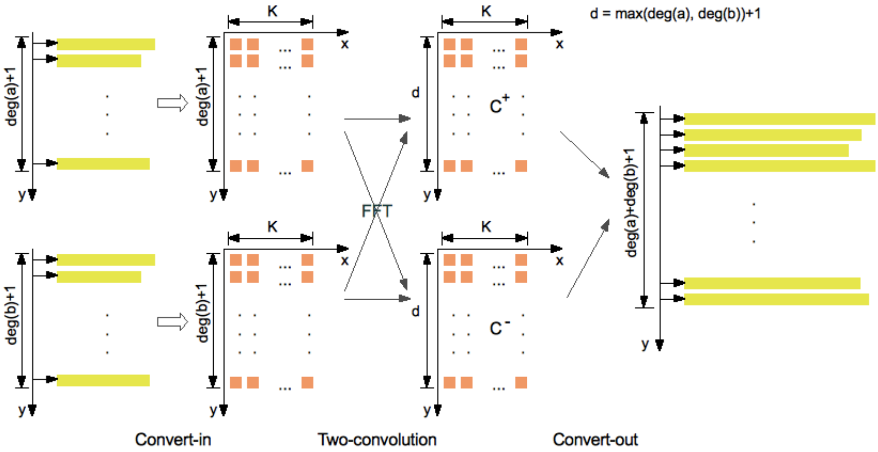
k-way Toom-Cook algorithm

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 $F(X) = Z_{f_0} + Z_{f_1}\,X + \cdots + Z_{f_{k-1}}\,X^{k-1}$ and $G(X) = Z_{g_0} + Z_{g_1}\,X + \cdots + Z_{g_{k-1}}\,X^{k-1}$;
3. **Evaluation**: Evaluate f, g at $2\,k-1$ points: $(0, X_1, \ldots, X_{2\,k-3}, \infty)$;
4. **Multiplying**: $(w_0, \ldots, w_{2\,k-2}) = (F(0)\cdot G(0), \ldots, F(\infty)\cdot G(\infty))$;
5. **Interpolation**: Recover $(Z_{h_0}, Z_{h_1}, \ldots, Z_{h_{2\,k-2}})$ where $H(X) = f(X)\,g(X) = Z_{h_0} + Z_{h_1}\,X + \cdots + Z_{h_{2\,k-2}}\,X^{2\,k-2}$
6. **Conversion**: Recover polynomial coefficients from $Z_{h_0}, \ldots, Z_{h_{2\,k-2}}$, obtaining $h(x) = h_0(x) + h_1(x)\,x^{n/k} + \cdots + h_{2\,k-2}(x)\,x^{(2\,k-2)\,n/k}$.

- work in terms of bit operations is $O(s \log_2(s) \log_2(\log_2(s)))$, where s is the maximum bit-size of f or g, but the constant has been multiplied approximately by 2 for $k = 8$;
- 4-way & 8-way Toom-Cook are available;
- static parallelism (about 7 and 13 when $k = 4$ and $k = 8$, resp).

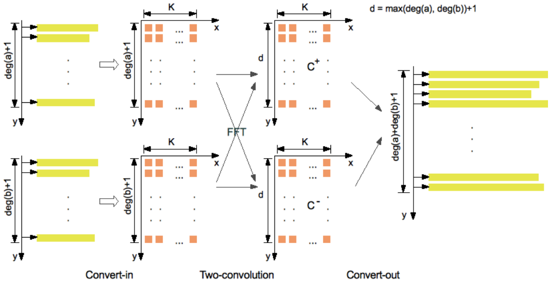
Parallel dense integer polynomial multiplication

A new algorithm: the two-convolution method



- work is $O(s \log_K(s))$, where s is the maximum bit-size of an input;
- parallelism is $O(\frac{\sqrt{s}}{\log_2(s)})$.

Parallel dense 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 $K\,M$ is essentially the maximum bit size of a coefficient in a, b.
2. Consider $C^+(x,y) \equiv A(x,y)\,B(x,y) \bmod < x^K + 1 >$ and $C^-(x,y) \equiv A(x,y)\,B(x,y) \bmod < x^K - 1 >$, then compute $C^+(x,y)$ and $C^-(x,y)$ modulo machine-word primes so as to use efficient 2-D 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.

Size	Work(KS+SS)*	Work(CVL_2)*	Span(CVL_2)*	$\frac{\mathrm{Work}(\mathrm{CVL}_2)}{\mathrm{Span}(\mathrm{CVL}_2)}$	$\frac{\mathrm{Work}(\mathrm{CVL}_2)}{\mathrm{Work}(\mathrm{KS+SS})}$
2048	795,549,545	1,364,160,088	41,143,119	33.16	1.715
4096	4,302,927,423	5,663,423,709	96,032,325	58.97	1.316
8192	16,782,031,611	23,827,123,688	292,735,521	81.39	1.420
16384	63,573,232,166	100,688,072,711	1,017,726,160	98.93	1.584
32768	269,887,534,779	425,149,529,176	3,804,178,563	111.76	1.575

Table: `Cilkview` analysis of CVL_2 and KS+SS. (* shows the number of instructions)

Parallel dense integer polynomial multiplication

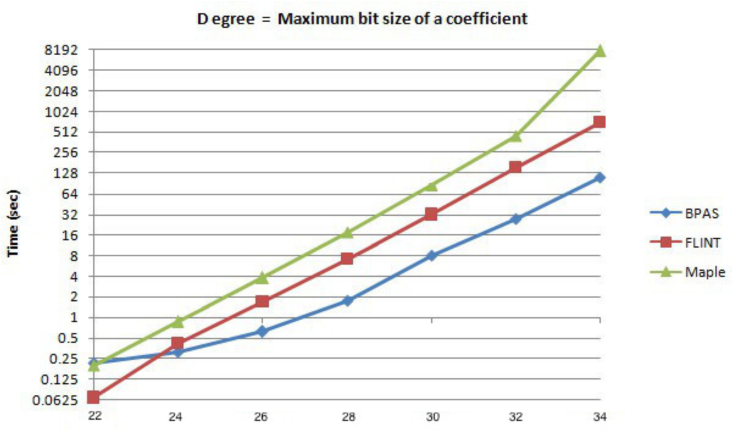


Figure: BPAS (parallel) vs FLINT (serial) vs Maple 18 (serial) with the logarithmic scale in radix 2 of the maximum bit-size of an input polynomial as the horizontal axis.

Parallel dense integer polynomial multiplication

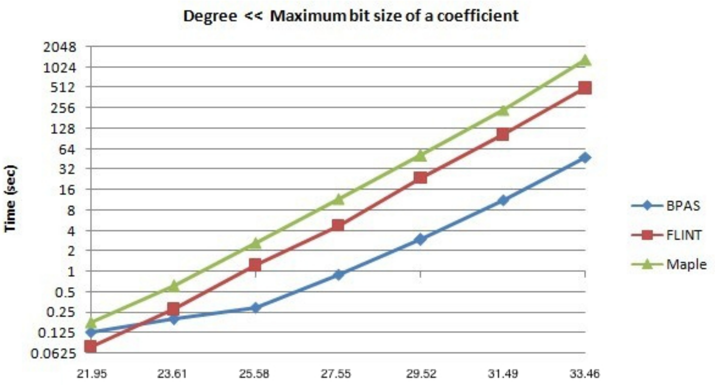


Figure: BPAS (parallel) vs FLINT (serial) vs Maple18 (serial) with the logarithmic scale in radix 2 of the maximum bit-size of an input polynomial as the horizontal axis.

Parallel dense integer polynomial multiplication

The adaptive algorithm based on the input size and available resources

- Very small: Plain multiplication
- Small or Single-core: KS+SS algorithm
- Big but a few cores: 4-way Toom-Cook
- Big: 8-way Toom-Cook
- Very big: Two-convolution method

Parallel Taylor shift $f(x) \longmapsto f(x+1)$

Parallel Pascals triangle by blocking

$$
\begin{array}{ccccccccccccc}
 & & 0 & & 0 & & \dots & & 0 & & 0 & & \\
 & & \downarrow & & \downarrow & & & & \downarrow & & \downarrow & & \\
f_d & \rightarrow & + & \rightarrow & + & \rightarrow & \dots & \rightarrow & + & \rightarrow & + & \rightarrow & g_d \\
 & & \downarrow & & \downarrow & & & & \downarrow & & & & \\
f_{d-1} & \rightarrow & + & \rightarrow & + & \rightarrow & \dots & \rightarrow & + & \rightarrow & g_{d-1} & & \\
 & & & \vdots & & & & & & & & & \\
 & & \downarrow & & \downarrow & & & & & & & & \\
f_1 & \rightarrow & + & \rightarrow & + & \rightarrow & g_1 & & & & & & \\
 & & \downarrow & & & & & & & & & & \\
f_0 & \rightarrow & + & \rightarrow & g_0 & & & & & & & &
\end{array}
$$

- Let n be the degree and ℓ 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.

Parallel Taylor shift $f(x) \longmapsto f(x+1)$

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

$$
\begin{array}{ccccccccccccc}
 & (f_0 & + & f_1(x+1)) & & + & (f_2 & + & f_3(x+1)) & & \times & (x+1)^2 & \\
 & & & \nearrow & & & & & & & \nwarrow & & \\
 & f_0 & + & f_1(x+1) & & & & & & f_2 & + & f_3(x+1) & \\
 & \nearrow & & \nwarrow & & & & & & & \nearrow & \nwarrow & \\
f_0 & & & & f_1 & & & & & f_2 & & & f_3
\end{array}
$$

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

[2] J. von zur Gathen and J. Gerhard. Fast algorithms for Taylor shifts and certain difference equations. In ISSAC, pages 40–47, 1997.

Parallel Taylor shift $f(x) \longmapsto f(x+1)$

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

A third alternative algorithm is work in progress.

[2] J. von zur Gathen and J. Gerhard. Fast algorithms for Taylor shifts and certain difference equations. In ISSAC, pages 40–47, 1997.

Plan

Applications

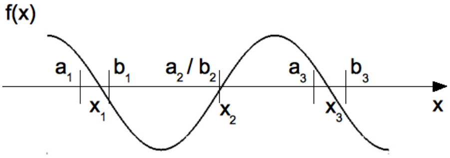
- Parallel univariate real root isolation
- Parallel multivariate real root isolation
- Symbolic integration

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 real root of $f(x)$ is contained in one and only one $[a_i, b_i]$;
- 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).



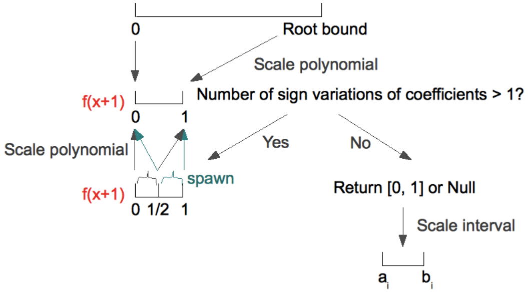


Figure: Parallel Vincent-Collins-Akritas (VCA, 1976)

- The most costly operation is the Taylor Shift operation, that is, the map $f(x) \longmapsto f(x+1)$.

Parallel univariate real root isolation

We run two parallel real root algorithms, BPAS and CMY [3], which are both implemented in CilkPlus, against Maple 18 serial *realroot* command (interface of the RUR-based code implemented in C by F. Rouillier) which implements a state-of-the-art algorithm.

	Size	BPAS (Parallel)	CMY [3] (Parallel)	*realroot* (Serial)	$\frac{T_{\mathrm{CMY}}}{T_{\mathrm{BPAS}}}$	$\frac{T_{realroot}}{T_{\mathrm{BPAS}}}$	#Roots
Cnd	32768	18.141	125.902	816.134	6.94	44.99	1
	65536	66.436	664.438	7,526.428	10.0	113.29	1
Chebycheff	2048	608.738	594.82	1,378.444	0.98	2.26	2047
	4096	8,194.06	10,014	35,880.069	1.22	4.38	4095
Laguerre	2048	1,336.14	1,324.33	3,706.749	0.99	2.77	2047
	4096	20,727.9	23,605.7	91,668.577	1.14	4.42	4095
Wilkinson	2048	630.481	614.94	1,031.36	0.98	1.64	2047
	4096	9,359.25	10,733.3	26,496.979	1.15	2.83	4095

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

[3] C. Chen, M. Moreno Maza, and Y. Xie. Cache complexity and multicore implementation for univariate real root isolation. J. of Physics: Conf. Series, 341, 2011.

Parallel multivariate real root isolation

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

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.

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 $\int \frac{x^4 - 3x^2 + 6}{x^6 - 5x^4 + 5x^2 + 4}\,dx = \mathrm{invtan}(x^3 - 3x, x^2 - 2)$.

```
  -
 /     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
- 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

Basic Polynomial Algebra Subprograms