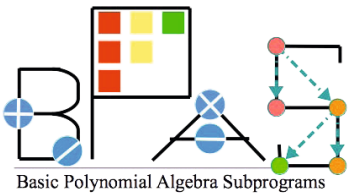
The Basic Polynomial Algebra Subprograms

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

[1] Chongqing Key Laboratory of Automated Reasoning and Cognition,
Chongqing Institute of Green and Intelligent Technology,
Chinese Academy of Sciences, China
changbo.chen@hotmail.com
[2] University of Western Ontario, Canada
{moreno,covanov,fmansou3,nxie6,yxie}@csd.uwo.ca

Abstract. The Basic Polynomial Algebra Subprograms (BPAS) provides arithmetic operations (multiplication, division, root isolation, etc.) for univariate and multivariate polynomials over prime fields or with integer coefficients. The code is mainly written in CilkPlus [10] targeting multicore processors. The current distribution focuses on dense polynomials and the sparse case is work in progress. A strong emphasis is put on adaptive algorithms as the library aims at supporting a wide variety of situations in terms of problem sizes and available computing resources. One of the purposes of the BPAS project is to take advantage of hardware accelerators in the development of polynomial systems solvers. The BPAS library is publicly available in source at www.bpaslib.org.

Keywords: Polynomial arithmetic, parallel processing, multi-core processors, Fast Fourier Transforms (FFTs).



1 Design and Specification

Inspired by the Basic Linear Algebra Subprograms (BLAS), BPAS functionalities are organized into three levels. At Level 1, one finds basic arithmetic operations that are specific to a polynomial representation or specific to a coefficient ring. Examples of Level-1 operations are multi-dimensional FFTs/TFTs and univariate real root isolation. At Level 2, arithmetic operations are implemented for all types of coefficients rings that BPAS supports (prime fields, ring of integers, field of rational numbers). Level 3 gathers advanced arithmetic operations taking as input a zero-dimensional regular chain, e.g. normal form of a polynomial, multivariate real root isolation.

H. Hong and C. Yap (Eds.): ICMS 2014, LNCS 8592, pp. 669–676, 2014.

Level 1 functions are highly optimized in terms of data locality and parallelism. In particular, the underlying algorithms are nearly optimal in terms of cache complexity [5]. This is the case, for instance, for our modular multidimensional FFTs/TFTs [14], modular dense polynomial arithmetic [15] and Taylor shift [3] algorithms.

At Level 2, the user can choose between algorithms that either minimizes work (at the possible expense of decreasing parallelism) or maximizes parallelism (at the possible expense of increasing work). For instance, five different integer polynomial multiplication algorithms are available, namely: Schönhage-Strassen, 8-way Toom-Cook, 4-way Toom-Cook, divide-and-conquer plain multiplication and the two-convolution method [2].

- The first one has optimal work (i.e. algebraic complexity) 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, see [12] for details.
- The fifth algorithm relies on modular 2D FFTs which are computed by means of the row-column scheme; this algorithm delivers high scalability and can fully utilize the hardware on fat multicore nodes.

Another example of Level 2 functionality is parallel Taylor shift computation for which four different algorithms are available: the two plain algorithms presented in [3], Algorithm (E) of [7] and an optimized version of Algorithm (F) of [7].

- The first two are highly effective when both the input data size and the number of available cores are small.
- The third algorithm creates parallelism by means of a divide-and-conquer procedure and relies on polynomial multiplication; this approach is effective when 8-way Toom-Cook multiplication is selected.
- The fourth algorithm reduces a Taylor shift computation to a single polynomial multiplication; this latter approach outperforms the other three, as soon as the two-convolution multiplication dominates its counterparts, that is, when either input data size and the number of available cores become large.

This variety of parallel solutions leads, at Level 3, to adaptive algorithms which select appropriate Level 2 functions depending on available resources (number of cores, input data size). An example is parallel real root isolation. Many procedures for this purpose are based on a *subdivision scheme*. However, on many examples, this scheme exposes only a limited amount of opportunities for concurrent execution, see [3]. It is, therefore, essential to extract as much as parallelism from the underlying routines, such as Taylor shift computations.

2 User Interface

Inspired by computer algebra systems like `AXIOM` [9] and `Magma` [1], the `BPAS` library makes use of type constructors so as to provide genericity. For instance `SparseUnivariatePolynomial (SUP)` can be instantiated over any `BPAS` ring. On the other hand, for efficiency consideration, certain polynomial type constructors, like `DistributedDenseMultivariateModularPolynomia (DDMMP)`, are only available over finite fields in order to ensure that the data encoding a `DDMMP` polynomial consists only of consecutive memory cells.

For the same efficiency consideration, the most frequently used polynomial rings, like `DenseUnivariateIntegerPolynomial (DUZP)` and `DenseUnivariate-RationalNumberPolynomial (DUQP)` are primitive types. Consequently, `DUZP` and `SUP<Integer>` implement the same functionalities; however the implementation of the former is further optimized.

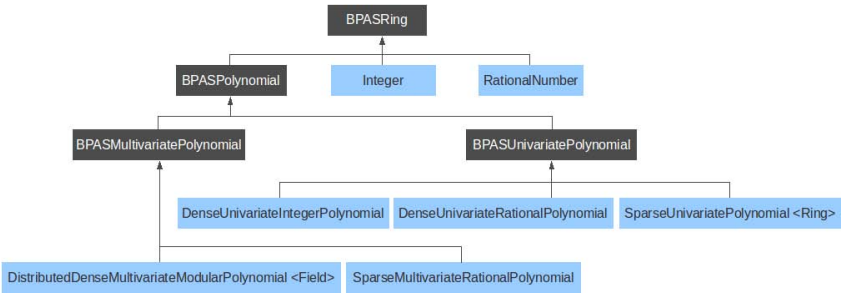


Fig. 1. A snapshot of `BPAS` algebraic data structures

Figure 1 shows a subset of `BPAS`'s tree of algebraic data structures. Dark and blue boxes correspond respectively to abstract and concrete classes. `BPAS` counts many other classes for instance `Intervals` and `RegularChains`.

Figure 2 first shows how two dense univariate polynomials are read from a file and how their product is computed. Then, on the same code fragment, a (zero-dimensional) regular chain is read from a file and its real roots are isolated.

3 Implementation Techniques

Modular FFTs are at the core of asymptotically fast algorithms for dense polynomial arithmetic operations. A substantial body of code of the `BPAS` library is, therefore, devoted to the computation of one-dimensional and multi-dimensional FFTs over finite fields. In the current release, the characteristic of those fields is of machine word size while larger characteristics are work in progress.

The techniques used for the multi-dimensional FFTs are described in [14,15] while those for one-dimensional FFTs are inspired by the design of the FFTW [4].

```
#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;
}
```

Fig. 2. A snapshot of BPAS code

BPAS one-dimensional FFTs code is optimized in terms of cache complexity and register usage. To achieve this, 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. This threshold can be specified by the user through an environment variable HTHRESHOLD or determined automatically when installing the library. 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. 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.

Table 1. One-dimensional modular FFTs: Modpn vs BPAS

Size	Modpn	BPAS	Speedup
16777216	6.232	1.391	4.48
33554432	12.987	2.957	4.392
67108864	26.783	6.266	4.274
134217728	55.329	13.235	4.181
268435456	113.8	27.901	4.079

Table 1 compares running times (in sec. on Intel Xeon 5600) of one-dimensional modular FFTs computed by the Modpn library [11] and BPAS, both using serial C code in this case. The first column of Table 1 gives the size of the input vector; coefficients are in a prime field whose characteristic is a 57-bit prime.

Modular FFTs support the implementation of several algorithms performing dense polynomial arithmetic. As an example, we consider parallel multiplication of dense polynomials with integer coefficients by means of the *two-convolution method* [2] and which is illustrated on Figure 3. Given two univariate polynomials $a(y)$, $b(y)$ with integer coefficients, their product $c(y)$ is computed as follows.

(S1) Convert $a(y)$, $b(y)$ to bivariate integer polynomials $A(x,y)$, $B(x,y)$ 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 M is essentially the maximum bit size of a coefficient in a and b.

(S2) Consider $C^+(x,y) \equiv A(x,y)\ B(x,y) \bmod \langle x^K + 1\rangle$ and $C^-(x,y) \equiv A(x,y)\ B(x,y) \bmod \langle x^K - 1\rangle$. Compute $C^+(x,y)$ and $C^-(x,y)$ modulo machine-word primes so as to use modular 2D FFTs.

(S3) Consider $C(x,y) = \frac{C^+(x,y)}{2}(x^K - 1) + \frac{C^-(x,y)}{2}(x^K + 1)$ and evaluate $C(x,y)$ at $x = \beta$, which finally gives $c(y) = a(y)\,b(y)$.

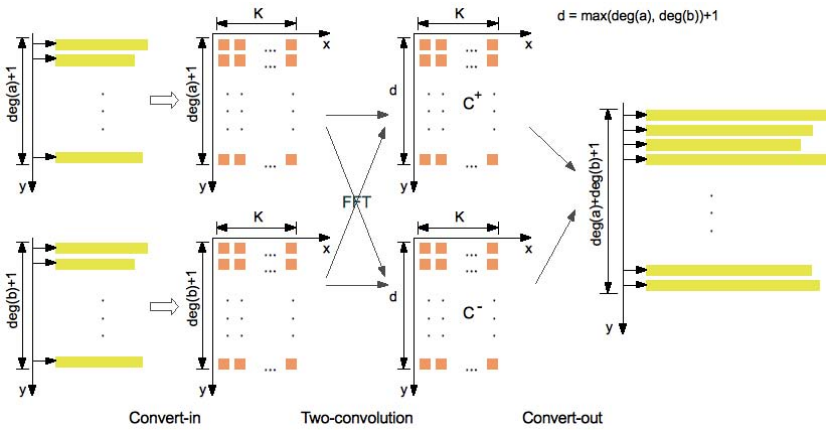


Fig. 3. Multiplication scheme for dense univariate integer polynomials

The conversions from the univariate polynomials $a(y)$, $b(y)$ to the bivariate polynomials $A(x,y)$, $B(x,y)$ in Step (S1) as well as the conversions from the bivariate polynomials $C^+(x,y)$ and $C^-(x,y)$ in Step (S3) require only additions and shift operations on machine words. Moreover, the polynomials $C^+(x,y)$ and $C^-(x,y)$ are reconstructed from their modular images (in practice two modular images are sufficient) within Step (S3). Consequently, the data produced by 2D FFT computations is converted in a *single pass* into the final result $c(y)$. Similarly the bivariate polynomials $A(x,y)$, $B(x,y)$ are obtained from $a(y)$, $b(y)$ (here again by means of additions and shift operations on machine words) in a single pass. Since `BPAS`' 2D FFT computations are optimal in terms of cache complexity [15], the whole multiplication procedure is optimal for that same complexity measure. Last, but not least, `BPAS`' 2D FFTs are computed by the row-column scheme which provides lots of parallelism with limited overheads on multicore architectures. As a result, our multiplication code, based on this two-convolution method scales well on multicores as illustrated hereafter.

4 Experimental Evaluation

As mentioned above, one of the main purposes of the BPAS library is to take advantage of hardware accelerators and support the implementation of polynomial system solvers. With this goal, polynomial multiplication plays a central role. Moreover, both sparse and dense representations are important. Indeed, input polynomial systems are often sparse while many algebraic transactions, like substitution, tend to densify data. Parallel sparse polynomial arithmetic has been studied by Gastineau and Laskar in [6] and by Monagan and Pearce in [13].

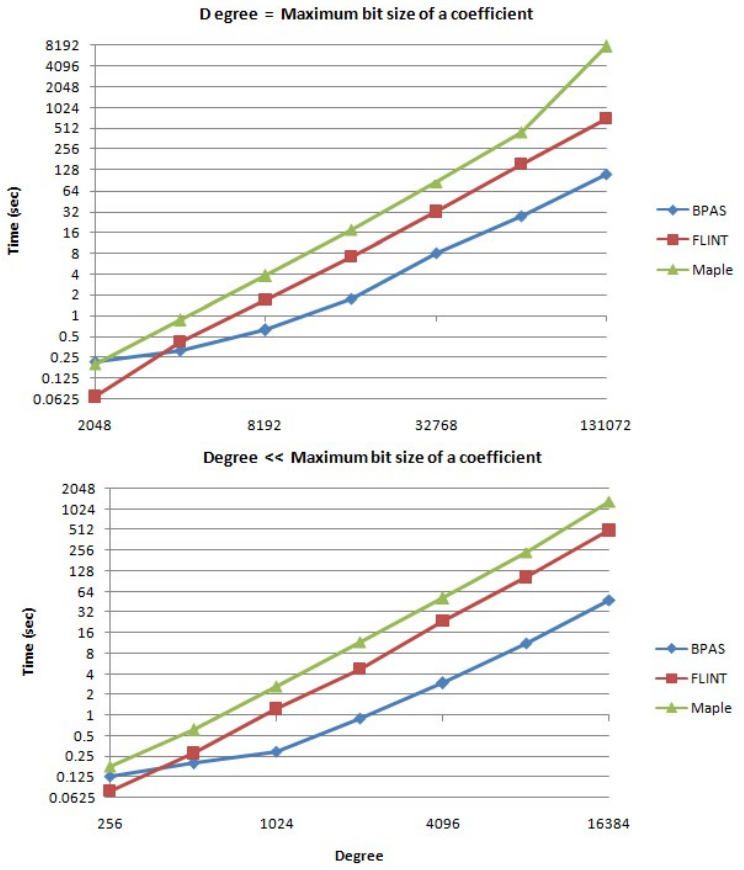


Fig. 4. Dense integer polynomial multiplication: BPAS vs FLINT vs Maple

Up to our knowledge, BPAS is the first publicly available library for parallel dense integer polynomial arithmetic. For this reason, we compare BPAS' parallel dense polynomial multiplication against state-of-the-art counterpart implementation in FLINT 2.4.3 and Maple 18. On Figure 4, the input of each test case is a pair of polynomials of degree d where each coefficient has bit size N. Two plots

are provided: one for which d = N holds and one for which d is much smaller than N.

The `BPAS` library is implemented with the multi-threaded language `CilkPlus` [10] and we compiled our code with the `CilkPlus` branch of GCC[1]. 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 2 shows that the work overhead (measured by `Cilkview`, the performance analysis tool of `CilkPlus`) of the `BPAS` method w.r.t. to a method based on Schönhage & Strassen algorithm (KS) is only around 2 (see Column 3), whereas `BPAS` provides large amount of parallelism (see Column 2).

Table 2. `Cilkview` analysis of `BPAS` and KS (* shows the number of instructions)

Size	Work(KS)*	Work(`BPAS`)*	Span(`BPAS`)*	$\frac{\text{Work(BPAS)}}{\text{Span(BPAS)}}$	$\frac{\text{Work(BPAS)}}{\text{Work(KS)}}$
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

5 Application

Turning to parallel univariate real root isolation, we have integrated our parallel integer polynomial multiplication into the algorithm proposed in [3]. To this end, we perform the Taylor Shift operation, that is, the map $f(x) \longmapsto f(x+1)$, by means of Algorithm (E) in [7], which reduces calculations to integer polynomial multiplication in large degrees and to using algorithm of [3] in small degrees. In Table 3, we call `BPAS` this adaptive algorithm combining FFT-based arithmetic (via Algorithm (E)) and plain arithmetic (via [3]).

Table 3. Univariate real root isolation running time for four examples

	Size	`BPAS`	CMY [3]	*realroot*	#Roots
Cnd	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

We run these two parallel real root algorithms, `BPAS` and CMY [3], which are both implemented in `CilkPlus`, against Maple 18 serial *realroot* command, which implements a state-of-the-art algorithm. Table 3 shows the running times

[1] `http://gcc.gnu.org/svn/gcc/branches/cilkplus/`

(in sec.) of four well-known test problems, including `Cnd`, `Chebycheff`, `Laguerre` and `Wilkinson`. Moreover, for each test problem, the degree of the input polynomial varies in a range. The results reported in Table 3 show that integrating parallel integer polynomial multiplication into our real root isolation code has substantially improved the performance of the latter.

Acknowledgments. This work was supported by the NSFC (11301524) and the CSTC (cstc2013jjys0002).

References

1. Bosma, W., Cannon, J., Playoust, C.: The Magma algebra system. I. The user language. J. Symbolic Comput. 24(3-4), 235–265 (1997)
2. Chen, C., Mansouri, F., Moreno Maza, M., Xie, N., Xie, Y.: Parallel Multiplication of Dense Polynomials with Integer Coefficient. Technical report, The University of Western Ontario (2013)
3. Chen, C., Moreno Maza, M., Xie, Y.: Cache complexity and multicore implementation for univariate real root isolation. J. of Physics: Conf. Series 341 (2011)
4. Frigo, M., Johnson, S.G.: The design and implementation of FFTW3 93(2), 216–231 (2005)
5. Frigo, M., Leiserson, C.E., Prokop, H., Ramachandran, S.: Cache-oblivious algorithms. ACM Transactions on Algorithms 8(1), 4 (2012)
6. Gastineau, M., Laskar, J.: Highly scalable multiplication for distributed sparse multivariate polynomials on many-core systems. In: Gerdt, V.P., Koepf, W., Mayr, E.W., Vorozhtsov, E.V. (eds.) CASC 2013. LNCS, vol. 8136, pp. 100–115. Springer, Heidelberg (2013)
7. von zur Gathen, J., Gerhard, J.: Fast algorithms for taylor shifts and certain difference equations. In: ISSAC, pp. 40–47 (1997)
8. Hart, W., Johansson, F., Pancratz, S.: FLINT: Fast Library for Number Theory. V. 2.4.3, `http://flintlib.org`
9. Jenks, R.D., Sutor, R.S.: AXIOM, The Scientific Computation System. Springer (1992)
10. Leiserson, C.E.: The Cilk++ concurrency platform. The Journal of Supercomputing 51(3), 244–257 (2010)
11. Li, X., Moreno Maza, M., Rasheed, R., Schost, É.: The modpn library: Bringing fast polynomial arithmetic into maple. J. Symb. Comput. 46(7), 841–858 (2011)
12. Mansouri, F.: On the parallelization of integer polynomial multiplication. Master's thesis, The University of Western Ontario, London, ON, Canada (2014), `http://www.csd.uwo.ca/~moreno/Publications/farnam-thesis.pdf`
13. Monagan, M.B., Pearce, R.: Parallel sparse polynomial multiplication using heaps. In: ISSAC, pp. 263–270. ACM (2009)
14. Moreno Maza, M., Xie, Y.: FFT-based dense polynomial arithmetic on multi-cores. In: Mewhort, D.J.K., Cann, N.M., Slater, G.W., Naughton, T.J. (eds.) HPCS 2009. LNCS, vol. 5976, pp. 378–399. Springer, Heidelberg (2010)
15. Moreno Maza, M., Xie, Y.: Balanced dense polynomial multiplication on multicores. Int. J. Found. Comput. Sci. 22(5), 1035–1055 (2011)
16. Schönhage, A., Strassen, V.: Schnelle multiplikation großer zahlen. Computing 7(3-4), 281–292 (1971)