IBM Advanced Studies

NSERC CRSNG

Algebraic tools supporting comprehensive optimization of parametric GPU kernels

Mohammadali Asadi, Alexander Brandt, Robert HC Moir and Marc Moreno Maza

ORCCA — Ontario Research Centre for Computer Algebra

Western Science

Introduction

The **Basic Polynomial Algebra Subprograms (BPAS) library** is a free, open-source, and multi-threaded library providing support for polynomial algebra on modern computer architectures, in particular hardware accelerators. Typical operations are polynomial arithmetic, multi-point evaluation and interpolation, and real root isolation. BPAS is written in C and C++ with CilkPlus extensions targeting multi-core processors.

▷ Objectives ◁

Part of the objective of the IBM-NSERC CRD CAS Project 880 is to provide efficient implementations of sophisticated algebraic algorithms for nonlinear polynomial system solving, required to determine optimal parameter values for parametric GPU kernels. **BPAS** is being extended to achieve this by porting the capabilities of the `RegularChains` library of the computer algebra system (CAS) MAPLE, while simultaneously improving efficiency and taking advantage of parallelism.

▷ Importance of Sparse Polynomial Arithmetic ◁

The `solve` command in MAPLE uses the `RegularChains` library to solve polynomial systems by computing triangular decompositions, which are essentially simplified algebraic descriptions of the solution set. These algorithms depends heavily on low-level operations, such as polynomial arithmetic, as well as other mid-level algebraic computations. Optimizing system solving thus requires optimizing these fundamental algorithms.

Sparse polynomial arithmetic is foundational to triangular decomposition. **Sparse** means that zero coefficients are not stored in the data structure, providing memory-efficient and natural representations. The amount of space required to store polynomials in a dense format (where all coefficients are stored) increases exponentially in the number of variables, and thus prohibitively large for nonlinear system solving. We have aggressively optimized the arithmetic operations for polynomials over $\mathbb{Z}$ and $\mathbb{Q}$.

Sparse Polynomial Arithmetic

▷ Notation and Representation ◁

Sparse polynomials in a polynomial ring $\mathbb{D}[x_1,\ldots,x_m]$, for an integral domain $\mathbb{D}$ and variable ordering $x_1 < x_2 < \cdots < x_m$, have two main representations: **distributed** and **recursive**. Both are needed in **BPAS**, depending on the operation.

In a distributed format, polynomials a are represented as a sum of terms

$$a = \sum_{i=1}^{n_a} A_i = \sum_{i=1}^{n_a} a_i X^{\alpha_i},$$

where n_a is the number of (non-zero) terms, $0 \neq a_i \in \mathbb{D}$, α_i is an m-tuple of exponents for the variables $X = (x_1,\ldots,x_m)$. A term of a is represented by $A_i = a_i X^{\alpha_i}$, so that each coefficient $a_i \in \mathbb{D}$ and each monomial $X^{\alpha_i} \in \mathbb{D}[x_1,\ldots,x_m]$ with $\alpha_i > \alpha_{i+1}$.

In a recursive format, a polynomial is regarded as an element of $\mathbb{D}[x_1,\ldots,x_{m-1}][x_m]$, coefficients are elements of $\mathbb{D}[x_1,\ldots,x_{m-1}]$ and monomials are elements of $\mathbb{D}[x_m]$. Viewed recursively, a is essentially a univariate polynomial in $R[x_m]$ with coefficients in $R = \mathbb{D}[x_1,\ldots,x_{m-1}]$. Operations that are essentially univariate, such as pseudo-division (fraction-free division), require such a recursive view.

▷ Key Aspects of Sparse Arithmetic ◁

Two main issues require special care to perform sparse polynomial operations efficiently:

(1) not computing multiple terms with the same monomial (since they will be combined into a single term in the result); and

(2) computing the terms in order, so that expensive sorting is not required.

We follow and extend the algorithms of [2] for arithmetic that generate terms in order. The key idea for this method is that the polynomials $a = \sum_{i=1}^{n_a} a_i X^{\alpha_i}$ and $b = \sum_{j=1}^{n_b} b_j X^{\beta_j}$ are each initially sorted, so if we multiply a term $A_i = a_i X^{\alpha_i}$ by b, then the monomials $X^{\alpha_i+\beta_j}$ in $A_i \cdot b$ satisfy $X^{\alpha_i+\beta_j} > X^{\alpha_i+\beta_{j+1}}$.

For **multiplication**, we produce terms in order by maintaining an ordered list of the maximal element $(a_i \cdot b_j)X^{\alpha_i+\beta_j}$ of each sub-product $A_i \cdot b$ that has yet to be used in the product. Maintaining this ordered list is implemented efficiently using a heap.

Polynomial **division** of a by b requires computing q and r such that $a = qb + r$. Division is essentially multiplication with a continuously updating operand, q. Division generates new terms of q from the product of the previously produced quotient terms with b. With slight tweaks to multiplication this is easy to achieve.

Pseudo-division involves computing i, q and r such that $h^i a = qb + r$, where $h = \mathrm{lc}(b) \in R$, and $a, b, q, r \in R[x_m]$. Accordingly, pseudo-division uses a slightly-modified version of the division algorithm to deal with h^i.

We formally prove the correctness of these algorithms and provide pseudocode in [1].

Implementation

Our implementation focuses on effective memory usage and management. Memory traversal is optimized through data locality and memory-efficient data structures.

▷ Polynomial Data Structures ◁

- Distributed polynomials are stored in an **alternating array**, alternating between coefficients and monomials. Corresponding coefficients and monomials have optimal data locality. *Coefficients* are GMP multi-precision integers or rational numbers.
- *Monomials*, assuming a consistent variable ordering, are only a list (array) of exponents, an **exponent vector**; the symbols themselves are not needed during computation.
- Exponent vectors are encoded using **exponent packing**. Integers of small absolute value have many leading zero bits when encoded into machine integers. Leading zero bits are removed by packing many into a single 64-bit word, improving memory usage.
  - Lower-ordered variables are given more bits in the packing to facilitate intermediate expression swell experienced during polynomial system solving.
  - Having *exactly one* machine word reduces monomial comparisons and multiplications to a single machine instruction (cf. iterating through a list of integers).

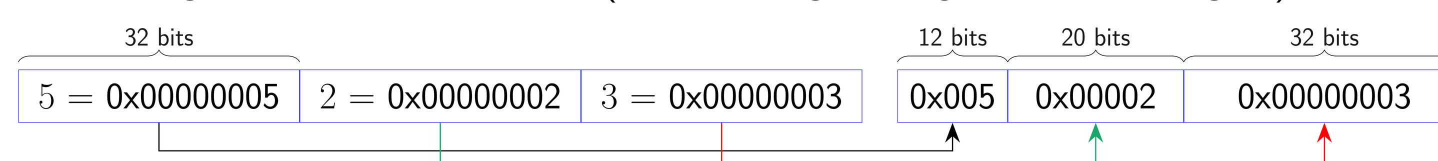


Figure 2: The exponent vector for $x^5y^2z^3$ packed into a single machine word.

- We use an in-place conversion from distributed to recursive representation. The recursive view adds an auxiliary alternating array which alternates between (1) main variable degree, (2) number of terms in coefficient, (3) pointer to coefficient in distributed array.

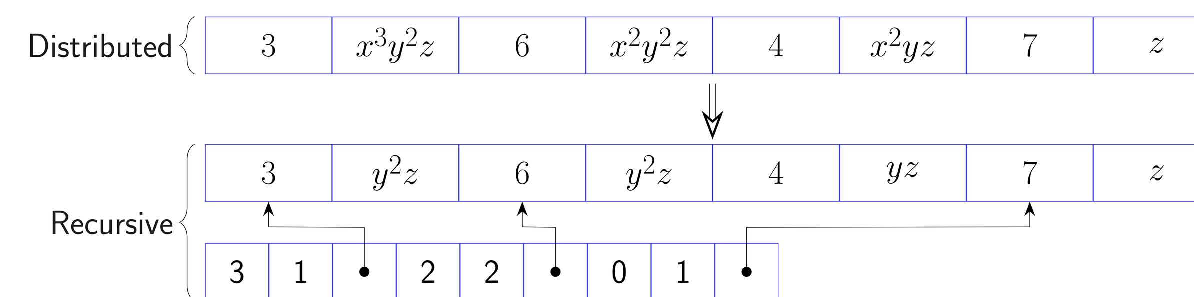


Figure 3: A distributed polynomial converted to recursive representation, showing the auxiliary array.

▷ Heap-Based Arithmetic Optimization ◁

Our algorithms rely on producing terms of the result (product, quotient, remainder) *in order*. We use a **binary heap** as an effective data structure for retrieving maximum elements from a continuously updating data set, essentially performing heap sort.

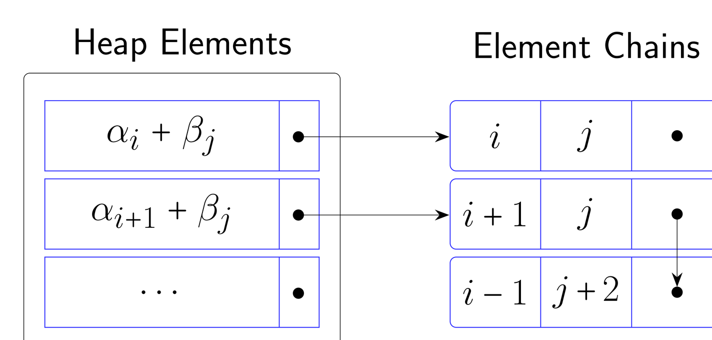


Figure 4: A heap of product terms, showing element chaining and index-based storing of coefficients. In this case, terms $A_{i+1} \cdot B_j$ and $A_{i-1} \cdot B_{j+2}$ have equal monomials and are chained together.

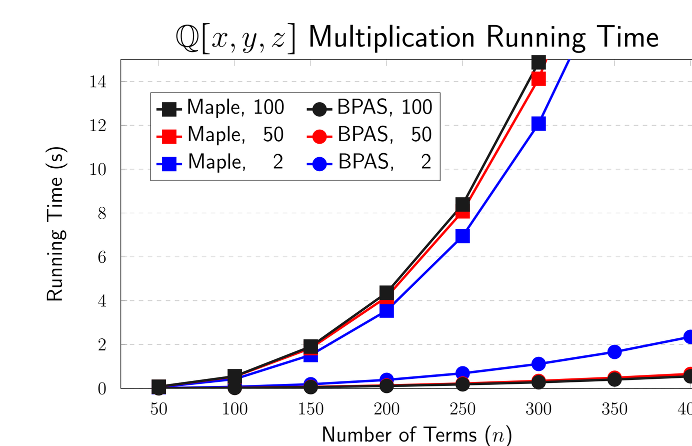
- **Memory usage** of the heap is minimized by storing only the monomial of product terms (as coefficients do not play a role in comparison), and the indices of coefficients which multiply together to form the product monomial's corresponding coefficient.
- **Comparisons** are minimized using heap **chaining**. Monomials found to be equal are chained together in a linked list so that only one such monomial is stored in the heap, reducing heap size, memory usage, and required comparisons for heapsort.

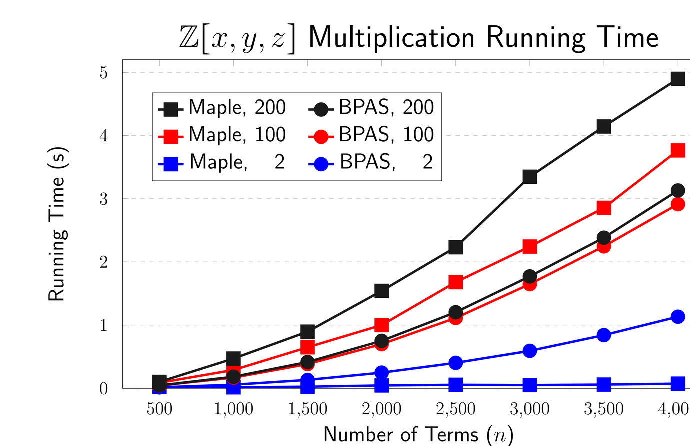
Experimentation

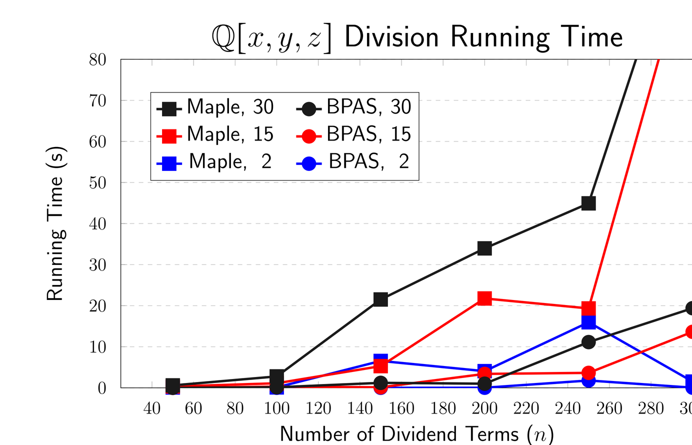
MAPLE has become the leader in high-performance (integer) polynomial arithmetic thanks to the work of Monagan and Pearce [3], out-performing many computer algebra systems like SINGULAR, TRIP, and PARI.

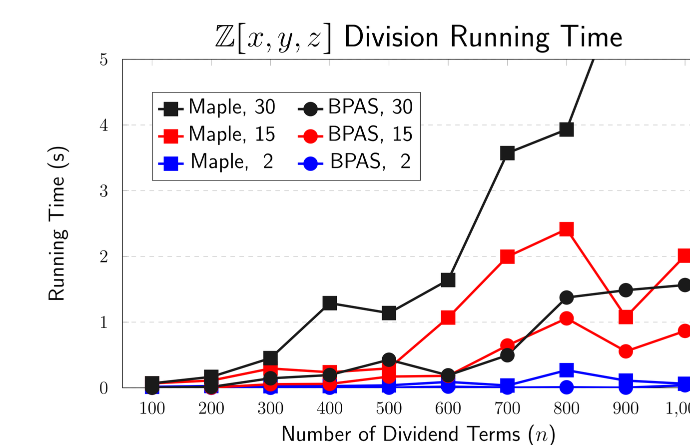
We compare **BPAS** to MAPLE for arithmetic over $\mathbb{Z}$ and $\mathbb{Q}$ with varying *sparsity*.

- All examples have operand polynomials with a maximum coefficient size of 128 bits.
- Division examples have divisors with half the number of terms as dividends.
- A sparsity of 2 is a fully dense polynomial.









Toward Polynomial System Solving

Implementing efficient polynomial system solving in **BPAS** requires first developing the required underlying algorithms and data structures for triangular sets and regular chains. The first steps toward this began with developing the `TriangularSet` class in **BPAS**.

▷ TriangularSet ◁

`TriangularSet` is the fundamental data structure underlying nonlinear polynomial system solving, where solutions are expressed as special kinds of triangular sets. The **BPAS** implementation has a number of advantages over its MAPLE analogue.

- Polynomial rings are handled simply and automatically by the polynomials themselves.
- `TriangularSet` provides support for both fixed triangular sets (for algebraic computations) and variable triangular sets (needed in differential algebra computations).
- The class also includes algorithms for pseudo-division and normal form of a polynomial by a triangular set, both core algorithms for nonlinear system solving.

▷ RegularChain ◁

We have also started development of the `RegularChain` class, required for nonlinear system solving. This class will support solutions of zero-dimensional systems (solutions are points) and positive-dimensional systems (solutions are parameterized spaces).

References

[1] Mohammadali Asadi, Alexander Brandt, Robert HC Moir, and Marc Moreno Maza. Sparse polynomial arithmetic with the BPAS library, 2018.

[2] Stephen C Johnson. Sparse polynomial arithmetic. *ACM SIGSAM Bulletin*, 8(3):63–71, 1974.

[3] Michael Monagan and Roman Pearce. Parallel sparse polynomial division using heaps. In *Proceedings of PASCO 2010*, pages 105–111. ACM, 2010.