Sparse Polynomial Arithmetic

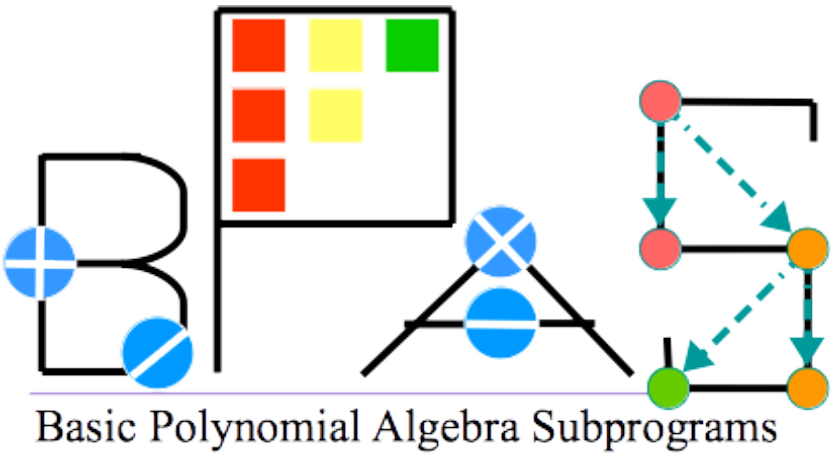
Alex Brandt

ORCCA, Department of Computer Science
University of Western Ontario

UWORCS 2018

April 12, 2018

Introduction



- High performance algorithms for sparse multivariate polynomial arithmetic over the rational numbers and integers.
  - → $\mathbb{Q}$ is practical for use in polynomial system solving.
  - → $\mathbb{Z}$ to compare against the leading high-performance implementation.[1],[2],[3]

- With careful implementations, we outperform the same algorithms ($\mathbb{Z}[X]$ arithmetic) implemented in *Maple*.

- Implementation: data structures, algorithmic tricks, and data locality.

Plan

Introduction: Polynomial Definitions

A quick refresher of polynomials and related definitions...

- **Variable** - a symbol representing some number.
  x, y, z, $\ldots$
- **Monomial** - a product of variables, each to some exponent.
  x^5yz^3
- **Coefficient** - a numerical multiplicative factor of a monomial.
  13, $\frac{7}{9}$, 2.1463
- **Term** - a coefficient-monomial product.
  $13x^5yz^3$
- **Polynomial** - a summation of terms.
  $13x^3 + 7x + 9$ (univariate)

  $13x^5yz^3 + \frac{7}{9}x^3y^2 + 11$ (multivariate)

Introduction: Sparse Polynomials

We are concerned with **Sparse Polynomials**. This is the natural representation of polynomials and allows for computational savings. Sparse here has a dual meaning:

- A polynomial *is sparse* if it has few non-zero coefficients.
  $x^9 + 1$ vs. $3x^4 + 7x^3 + 4x^2 + 1$
- A polynomial *is represented sparsely* if only its non-zero coefficients are stored.
  $1 * x^9 + 1 * x^0$ vs. $1 * x^9 + 0 * x^8 + 0 * x^7 + \ldots + 1 * x^0$

Introduction: Notations

Throughout this presentation we use the following notation:

$$a = \sum_{i=1}^{n} a_i X^{\alpha_i} \qquad b = \sum_{i=1}^{m} b_i X^{\beta_i} \qquad c = \sum_{i=1}^{k} c_i X^{\gamma_i}$$

- a, b, c are polynomials with n, m, k terms, respectively.
- a_i, b_i, c_i are non-zero coefficients.
- α_i, β_i, γ_i are **Exponent Vectors**, representing the exponents of a multivariate monomial over a product of some variables, X.
- $lm(a) = X^{\alpha_1}$ is the *leading monomial* of a.
- $lt(a) = a_1 X^{\alpha_1}$ is the *leading term* of a.

- $\mathbb{Q}[X]$: multivariate polynomials with rational number coefficients.
- $\mathbb{Z}[X]$: multivariate polynomials with integer coefficients.

A Note on Polynomial Ordering

To obtain a **Canonical Representation** of a polynomial it should have its terms sorted in some way. Certain operations thus become more efficient:

- Equality testing, leading coefficient, degree, etc.

Sorting uses some *total ordering* of monomials.

- Increasing or decreasing order is often arbitrary, depending on the application and data structure.

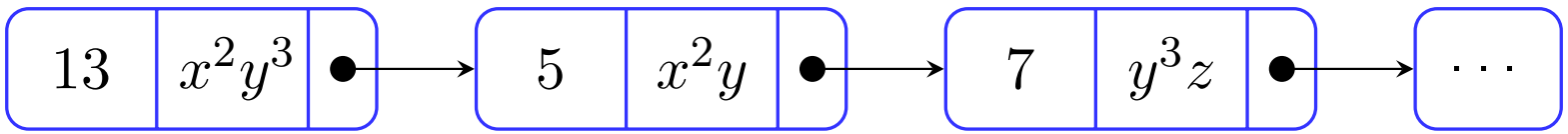
We use decreasing lexicographical ordering for monomials. For a variable ordering $x > y$ we get the following monomial ordering:

$$x^n y^n > x^{n-1} y^n > \cdots > xy > x > y^n > y^{n-1} > \cdots > y$$

Polynomial Representations

Sparse Polynomial Representation - Linked List

A simple scheme to encode a polynomial is a **Linked List**. Each node represents a single term of the polynomial and these nodes are strung together to form a list.



Advantages:

- Easy to insert or remove terms anywhere within the list

Disadvantages:

- Indexing, counting the number of terms, are $O(n)$ operations.
- Poor data locality, latent memory accesses impact performance.

Sparse Polynomial Representation - Alternating Arrays

In contrast, an **Alternating Array** provides a compact representation with good data locality.

An alternating array *alternates* between a coefficient data element and a monomial data element.

- Each term is encoded by a pair of array elements.

13	x^2y^3	5	x^2y	7	y^3z	$\cdots$

Some operations,like random insertion, are difficult, but memory efficiency is more important for our arithmetic algorithms.

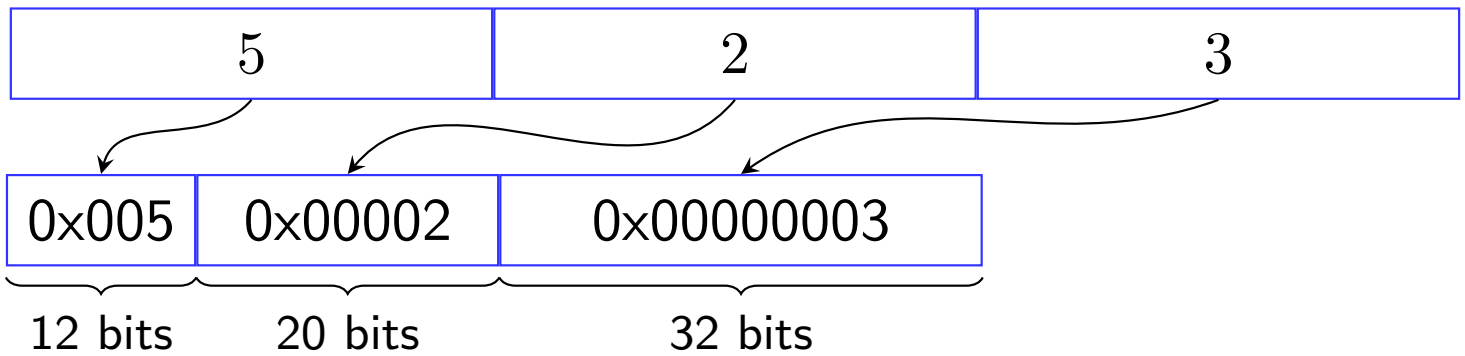
Maple implements a similar scheme[3] but is limited to integer coefficients.

Sparse Polynomial Representation - Exponent Packing

The simplest way to represent a monomial is a list of integers: one integer for each variable.

- Inefficient use of memory as exponents rarely approach the maximum value of a 32-bit int (4294967295).

Exponent packing[4],[5] uses bit-wise operations to store many exponents in a single machine word, improving memory usage and cache complexity. Consider the monomial $x^5y^2z^3$.



Monomial comparison and multiplication now reduce to a single instruction on one machine word.

Addition

Sparse Addition (1/2)

The simplest method for adding two polynomials together is to iteratively add one term from one to the other while maintaining the canonical, sorted order.

$$a + b = \left(a + b_1 X^{\beta_1}\right) + b_2 X^{\beta_2} + b_3 X^{\beta_3} + \ldots$$
$$a + b = \left(a + b_1 X^{\beta_1} + b_2 X^{\beta_2}\right) + b_3 X^{\beta_3} + \ldots$$
$$a + b = \left(a + b_1 X^{\beta_1} + b_2 X^{\beta_2} + b_3 X^{\beta_3}\right) + \ldots$$

- This process is analogous to insertion sort.
- Using alternating arrays, it is prohibitively poor performance to insert at random positions.
- We can use the fact that both a and b are initially sorted...

Sparse Addition (2/2)

Solution: Model addition as one merge step of *merge sort*.

Just as in merge sort, a and b are each pre-sorted partitions to be merged into a single partition.

- Walk left-to-right through a and b, taking the larger of the two current terms.
- Append it to the end of the sum.
- If terms are equal then we combine like terms.

$$\left.\begin{aligned} a &= 13x^2y^3 + 5x^2y \;\; + 7y^3z + \ldots \\ b &= 6x^2y \;\; + 12xz^2 + \;\; 4y^2 \;\; + \ldots \end{aligned}\right\} \quad c = 13x^2y^3 + 11x^2y + \ldots$$

The key idea here is that the terms in the sum are *generated in sorted order*.

Sparse Addition: Benchmarks

Addition is a very simple algorithm.

- Very little arithmetic work.
- A *memory bound* problem; must access memory efficiently.

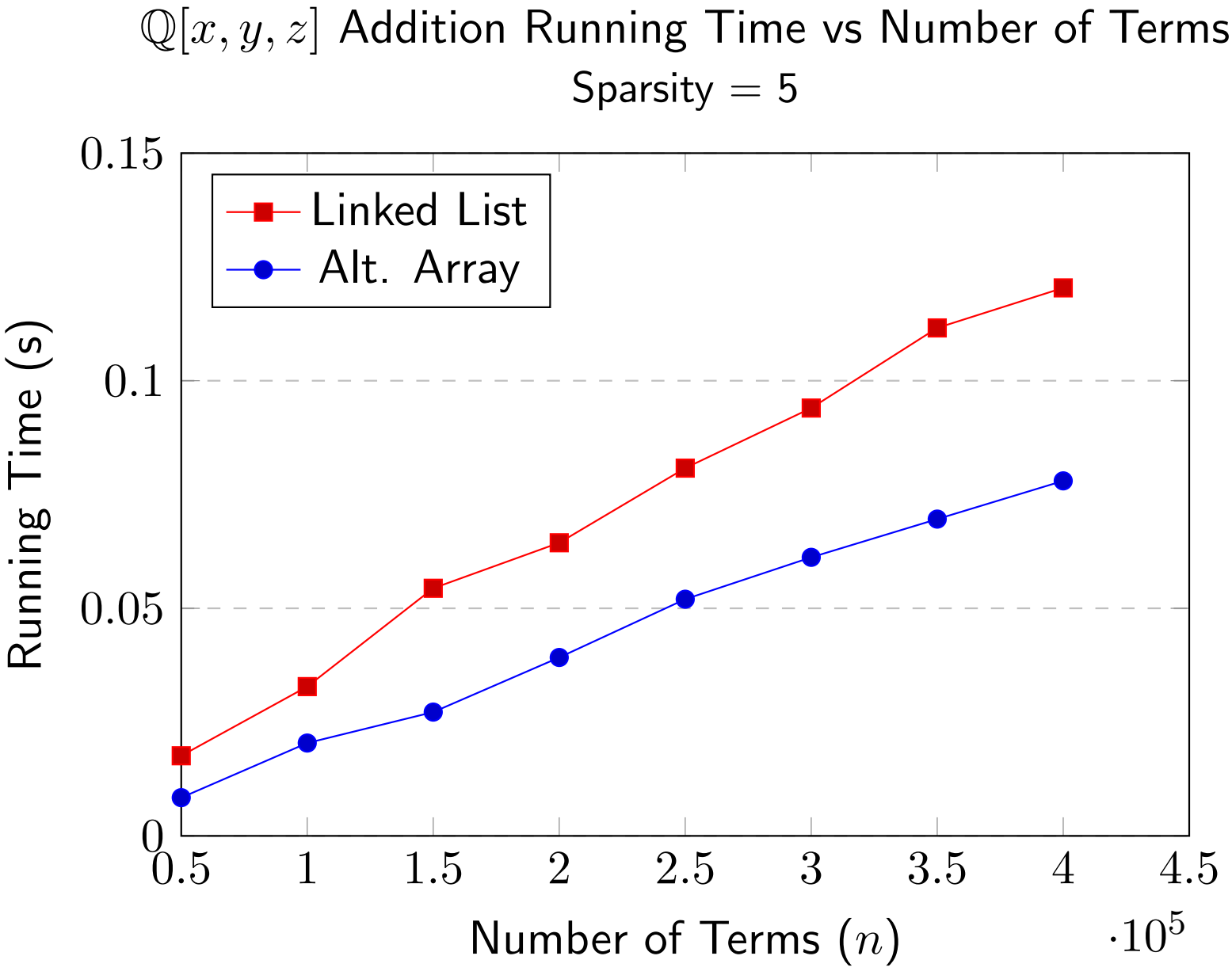
We compare linked lists, without exponent packing, against alternating arrays with exponent packing.

Note: *sparsity* is quantified as the maximum difference between exponents in successive monomials.

- This difference is done by viewing an exponent vector of v variables as digits of a number in radix r:

$$r = \sqrt[v]{\text{sparsity} \times \text{number of terms}}$$

Sparse Addition: Benchmarks



Multiplcation

Sparse Multiplication (1/2)

The task of multiplying two polynomials is actually three in one.

1. Generate the terms of the multiplication,
2. Combine like-terms,
3. Sort the resulting terms in decreasing order.

The simplest multiplication involves distributing each term of the multiplier (a) over the multiplicand (b), combining like-terms.

$$c = a \cdot b = (a_1 x^{\alpha_1} * b) + (a_2 x^{\alpha_2} * b) + \ldots$$

- Produces all $n \cdot m$ terms regardless of combining like-terms.
- Terms must be sorted after combining.

Like addition, we should try to *generate terms in sorted order*.

Sparse Multiplication: (2/2)

Solution: An n-way merge of "streams" where each stream represents a single term of a distributed over b.[6]

- Since b is sorted, and we multiply by a single term of a, the resulting "stream" is also sorted.

$$a \cdot b = \begin{cases} (a_1 * b_1) X^{\alpha_1+\beta_1} + (a_1 * b_2) X^{\alpha_1+\beta_2} + (a_1 * b_3) X^{\alpha_1+\beta_3} + \dots \\ (a_2 * b_1) X^{\alpha_2+\beta_1} + (a_2 * b_2) X^{\alpha_2+\beta_2} + (a_2 * b_3) X^{\alpha_2+\beta_3} + \dots \\ \vdots \\ (a_n * b_1) X^{\alpha_n+\beta_1} + (a_n * b_2) X^{\alpha_n+\beta_2} + (a_n * b_3) X^{\alpha_n+\beta_3} + \dots \end{cases}$$

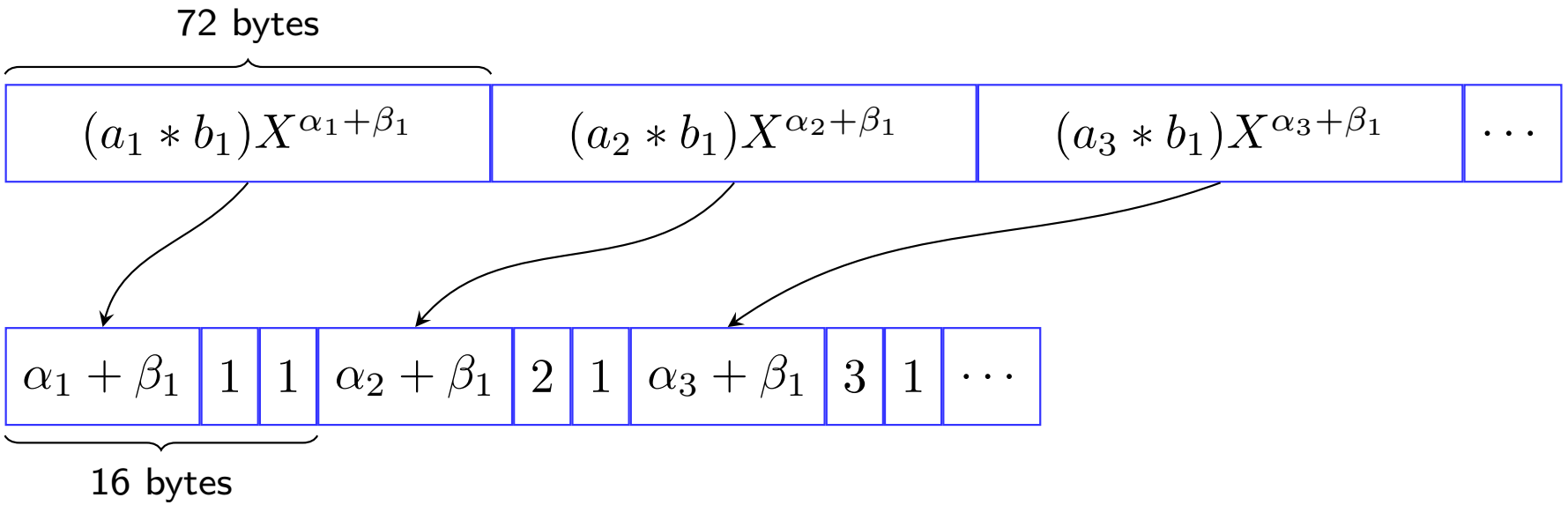
At each step we must choose the maximum among the head of all streams and then move to the next term in that stream.

- Choosing the maximum from a repeatedly updated collection is efficiently implemented using a **Heap**.

Heap Tricks: Memory Efficient Heap Elements

A heap relies on many comparisons of its elements and essentially random memory access.

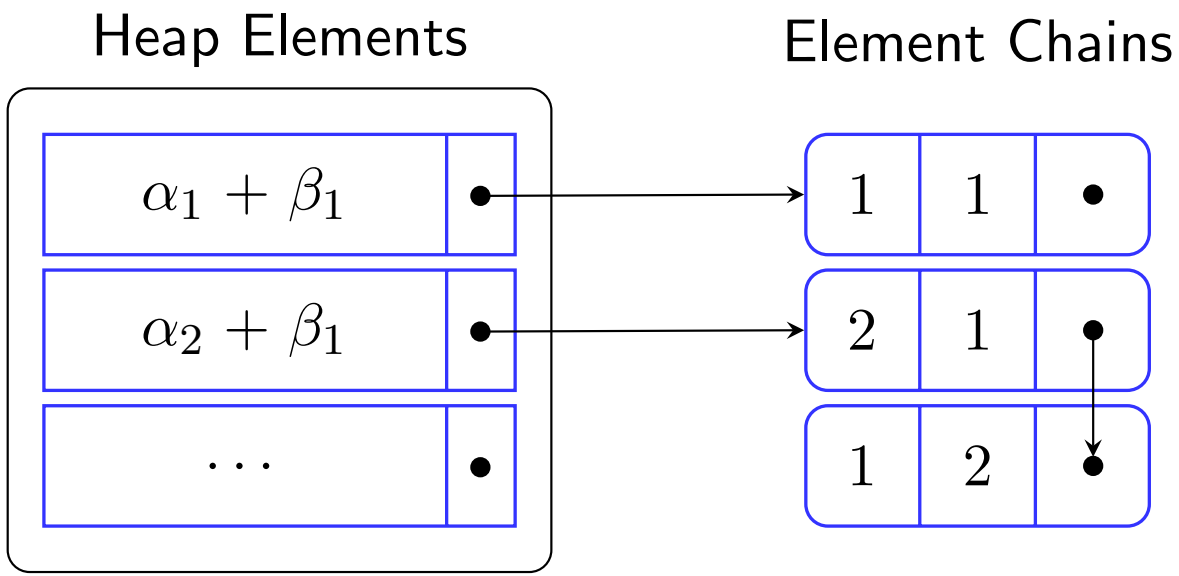
- Comparisons are already efficient due to exponent packing.
- It is necessary to only store the product exponent vector, $\alpha_1 + \beta_1$, and the *index* of the corresponding coefficients.
- Reduction of element size allows the entire heap to fit in cache for efficient random random access.



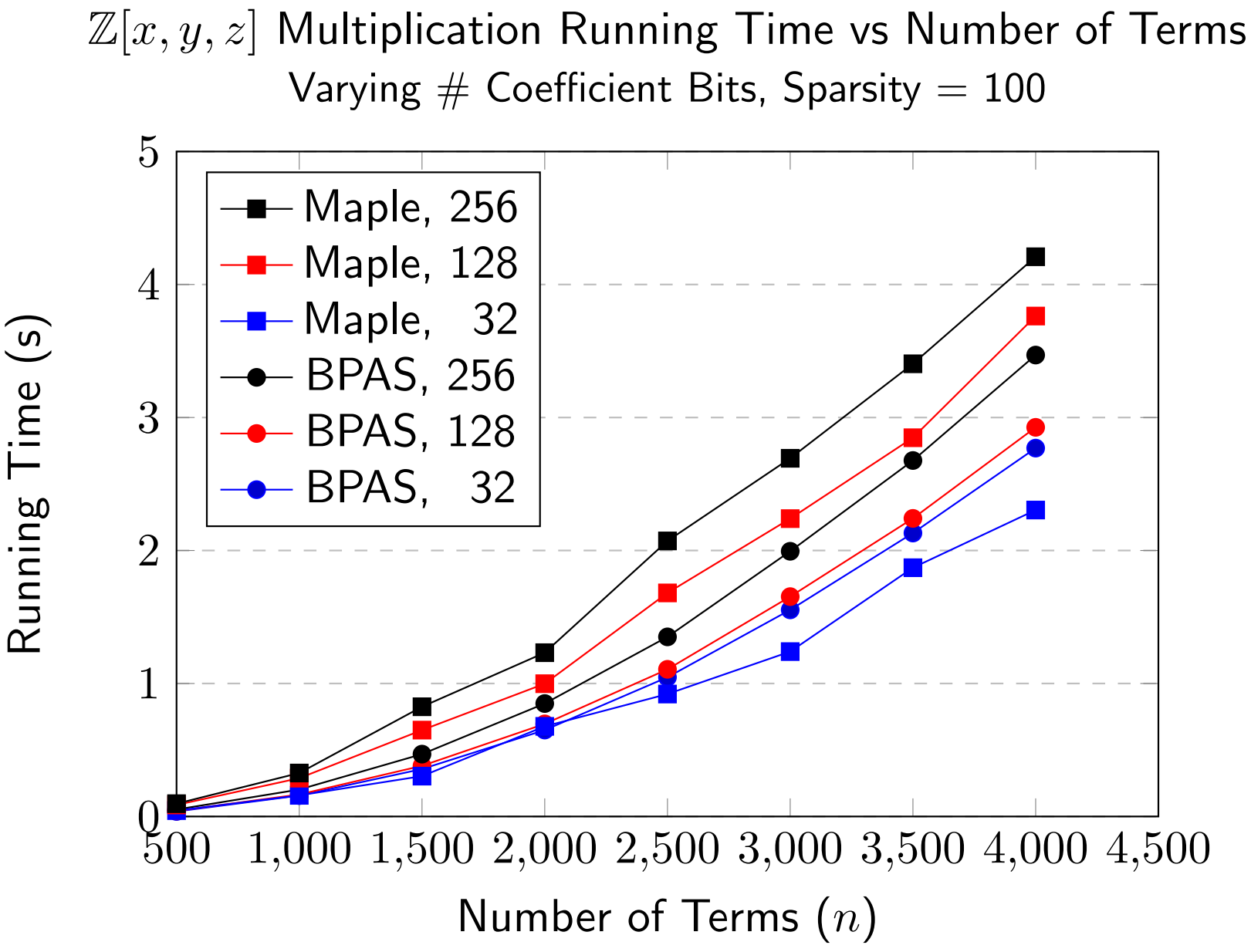
Heap Tricks: Element Chains

In our application, heap elements (monomials) are frequently equal. We can reduce the working size of the heap by *chaining*[5] equal elements.

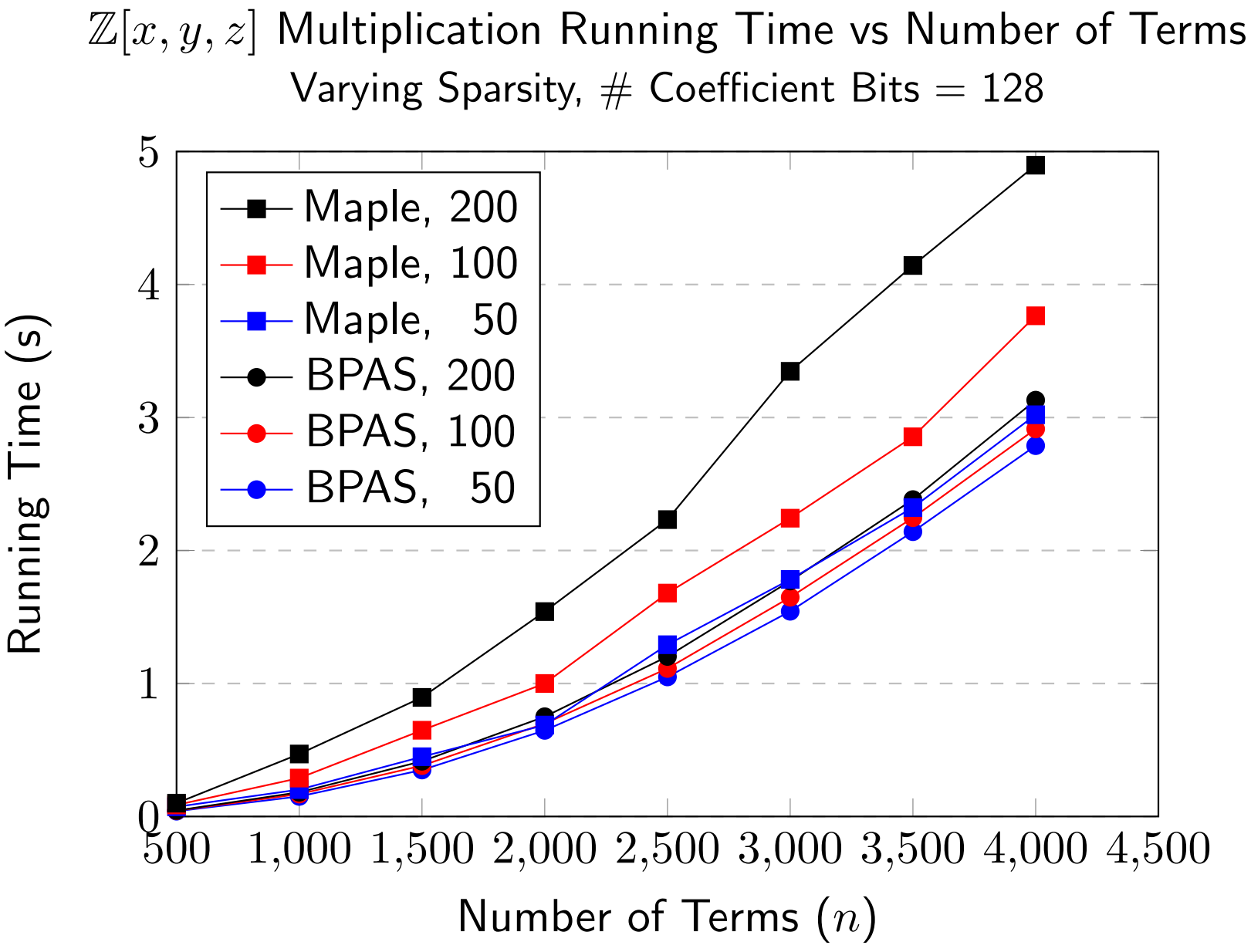
- Heap elements found to have the same exponent vector chain together into a linked list instead of inserting a new element.
- Our heap becomes a heap of linked lists.
- Allows extracting many elements for the cost of one extraction.



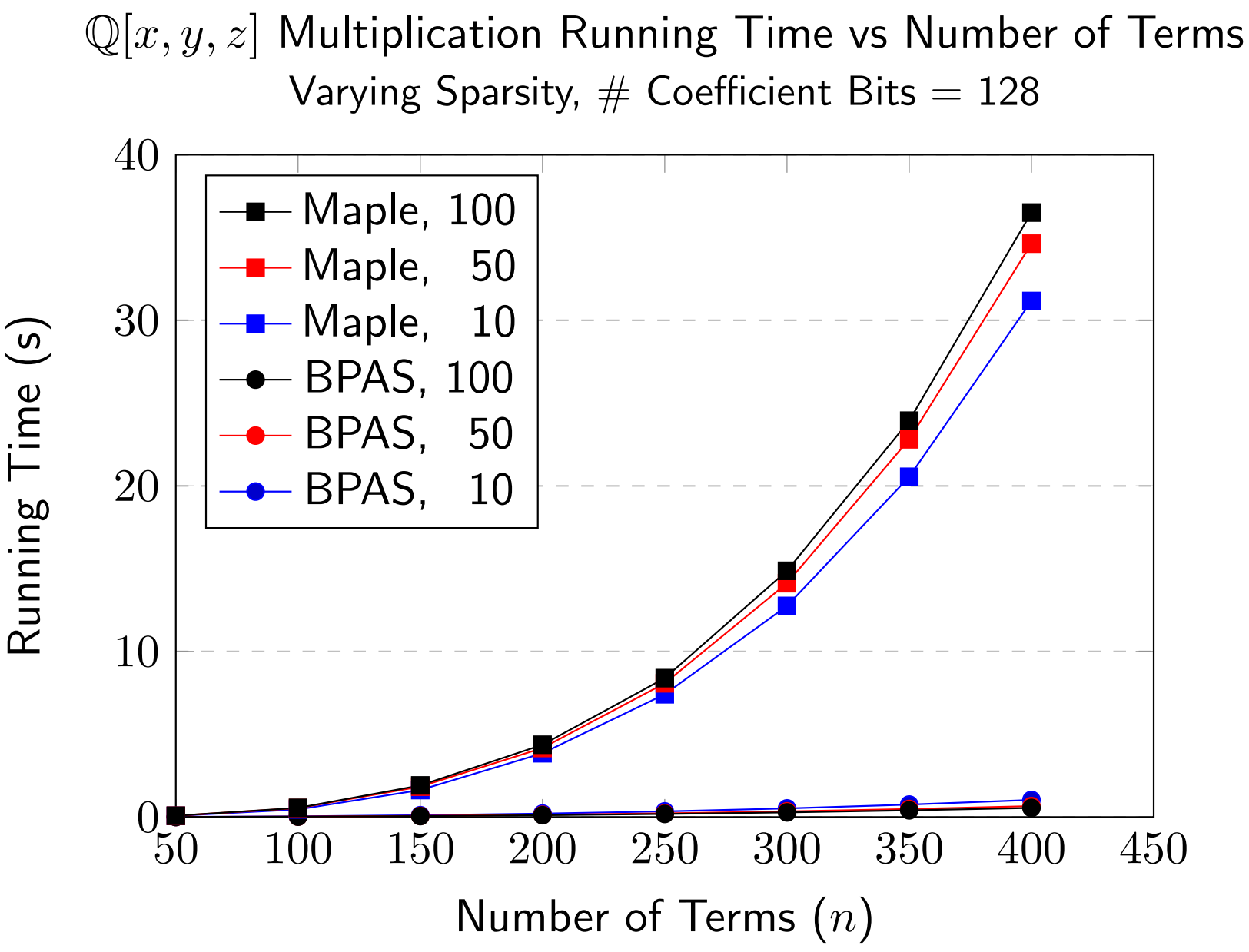
Benchmarks: $\mathbb{Z}[X]$ with Varying Coefficient Size



Benchmarks: $\mathbb{Z}[X]$ with Varying Sparsity



Benchmarks: $\mathbb{Q}[X]$ with Varying Sparsity



Division

Sparse Division: (1/2)

The goal of division is to, given c and b, find a and r such that $c = a \cdot b + r$ with $lm(b) > lm(r)$.

For simplicity, a_i, b_i, and c_i in the following discussion are *terms*.

Let's take the following example of basic long division:

$$b_1 + b_2 \overline{)\, c_1 + c_2 + c_3}$$

Sparse Division: (1/2)

The goal of division is to, given c and b, find a and r such that $c = a \cdot b + r$ with $lm(b) > lm(r)$.

For simplicity, a_i, b_i, and c_i in the following discussion are *terms*.

Let's take the following example of basic long division:

$$b_1 + b_2 \overline{)\, c_1 + c_2 + c_3}^{\,a_1}$$

Sparse Division: (1/2)

The goal of division is to, given c and b, find a and r such that $c = a \cdot b + r$ with $lm(b) > lm(r)$.

For simplicity, a_i, b_i, and c_i in the following discussion are *terms*.

Let's take the following example of basic long division:

$$\begin{array}{r} a_1 \\ b_1 + b_2 \overline{)\, c_1 + c_2 + c_3} \\ -a_1(b_1 + b_2) \end{array}$$

Sparse Division: (1/2)

The goal of division is to, given c and b, find a and r such that $c = a \cdot b + r$ with $lm(b) > lm(r)$.

For simplicity, a_i, b_i, and c_i in the following discussion are *terms*.

Let's take the following example of basic long division:

$$\begin{array}{r} a_1 \\ b_1 + b_2 \overline{)\, c_1 + c_2 + c_3} = c^{(1)} \\ \underline{-a_1(b_1 + b_2)} \\ c_1^{(2)} + c_2^{(2)} = c^{(2)} \end{array}$$

Sparse Division: (1/2)

The goal of division is to, given c and b, find a and r such that $c = a \cdot b + r$ with $lm(b) > lm(r)$.

For simplicity, a_i, b_i, and c_i in the following discussion are *terms*.

Let's take the following example of basic long division:

$$
\begin{array}{r}
 & a_1 + a_2 \\
\cline{2-2}
b_1 + b_2 \,\big) & c_1 + c_2 + c_3 \quad = c^{(1)} \\
 & -a_1(b_1 + b_2) \\
\cline{2-2}
 & c_1^{(2)} + c_2^{(2)} \quad = c^{(2)} \\
 & -a_2(b_1 + b_2) \\
\cline{2-2}
 & c_1^{(3)}
\end{array}
$$

Sparse Division: (1/2)

The goal of division is to, given c and b, find a and r such that $c = a \cdot b + r$ with $lm(b) > lm(r)$.

For simplicity, a_i, b_i, and c_i in the following discussion are *terms*.

Let's take the following example of basic long division:

$$
\begin{array}{rl}
 & a_1 + a_2 \\
b_1 + b_2 \,\overline{)}\, & c_1 + c_2 + c_3 = c^{(1)} \\
 & \underline{-a_1(b_1 + b_2)} \\
 & \quad c_1^{(2)} + c_2^{(2)} = c^{(2)} \\
 & \underline{-a_2(b_1 + b_2)} \\
 & \quad\quad\quad c_1^{(3)} = r
\end{array}
$$

Sparse Division: (1/2)

The goal of division is to, given c and b, find a and r such that $c = a \cdot b + r$ with $lm(b) > lm(r)$.

For simplicity, a_i, b_i, and c_i in the following discussion are *terms*.

Let's take the following example of basic long division:

$$
\begin{array}{rl}
 & a_1 + a_2 \\
b_1 + b_2 \,\overline{)}\, & c_1 + c_2 + c_3 = c^{(1)} \\
 & \underline{-a_1(b_1 + b_2)} \\
 & c_1^{(2)} + c_2^{(2)} = c^{(2)} \\
 & \underline{-a_2(b_1 + b_2)} \\
 & c_1^{(3)} = r
\end{array}
$$

- $c^{(1)} = c$
- $a_i = c_1^{(i)} \ / \ b_1$
- $c^{(i+1)} = c^{(i)} - a_i b$
- $r = c^{(i)}$ when b_1 cannot divide $c_1^{(i)}$

Sparse Division: (1/2)

The goal of division is to, given c and b, find a and r such that $c = a \cdot b + r$ with $lm(b) > lm(r)$.

For simplicity, a_i, b_i, and c_i in the following discussion are *terms*.

Let's take the following example of basic long division:

$$
\begin{array}{rl}
 & a_1 + a_2 \\
b_1 + b_2 \,\overline{)}\, & c_1 + c_2 + c_3 = c^{(1)} \\
 & \underline{-a_1(b_1 + b_2)} \\
 & c_1^{(2)} + c_2^{(2)} = c^{(2)} \\
 & \underline{-a_2(b_1 + b_2)} \\
 & c_1^{(3)} = r
\end{array}
$$

- $c^{(1)} = c$
- $a_i = c_1^{(i)} \ / \ b_1$
- $c^{(i+1)} = c^{(i)} - a_i b$
- $r = c^{(i)}$ when b_1 cannot divide $c_1^{(i)}$

At each step we calculate a full m (size of b) product terms from $a_i \cdot b$ and then perform a subtraction with those m terms.

Sparse Division: (2/2)

With some rearrangement of the previous recurrence relation we obtain a simple equation for a_i:

$$a_i = lt\left(c - \sum_{j=1}^{i-1} a_j b\right) / \; lt(b)$$

This formula has many notable benefits:

- Terms of a are still produced in-order,
- Avoids repeated subtraction and updating of the dividend,
- Reuse multiplication algorithm to obtain $\sum_{j=1}^{i-1} a_j b$ terms in-order.

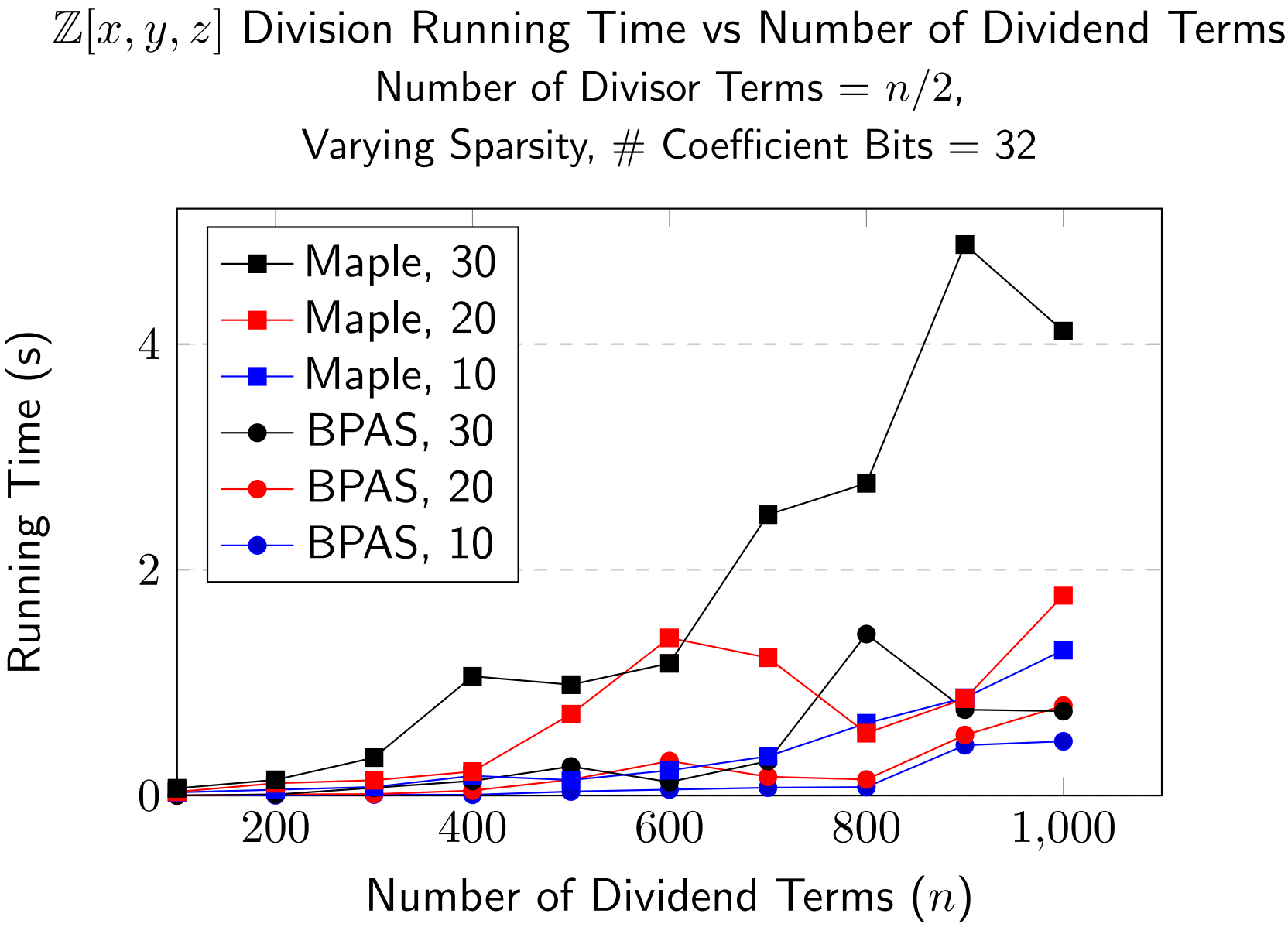
Sparse Division: Algorithm

Sparse Division

1: $a \leftarrow 0$
2: **while** `true` **do**
3: $\quad \tilde{c} \leftarrow \mathsf{lt}(c - ab)$
4: $\quad$ **if** $\mathsf{lt}(b) \nmid \tilde{c}$ **then**
5: $\quad\quad$ **break**
6: $\quad a \leftarrow a + (\tilde{c} \,/\, \mathsf{lt}(b))$
7: **end while**
8: $r \leftarrow c - ab$
9: **return** (a, r)

- For $\mathsf{lt}(c - ab)$ it is sufficient to keep track of which terms have *already been used* instead of doing subtraction.
- Produce only enough terms of $a \cdot b$ to get the next leading term, $\tilde{c}$, since a updates continually.
- $a + (\tilde{c}/\mathsf{lt}(b))$ is only an append as terms are in-order.

Benchmarks: $\mathbb{Z}[X]$ with Varying Sparsity



Future Work

Future work is focused on improving performance and efficiency.

- Pseudo division (WIP).
- Improved integer arithmetic.
  - → Dynamic switching between machine-word and multi-precision integers.[3]
- Experiment with new exponent packing schemes.
- Investigate different heap implementations.
  - → Binary-tree heap, Fibonacci heap, Brodal queue
- Parallelization of algorithms.

References

[1] M. Monagan and R. Pearce. "Sparse polynomial multiplication and division in Maple 14". In: *ACM Communications in Computer Algebra* 44.3/4 (2011), pp. 205–209.

[2] M. Monagan and R. Pearce. "Sparse polynomial division using a heap". In: *Journal of Symbolic Computation* 46.7 (2011), pp. 807–822.

[3] M. Monagan and R. Pearce. "The design of Maple's sum-of-products and POLY data structures for representing mathematical objects". In: *ACM Communications in Computer Algebra* 48.3/4 (2015), pp. 166–186.

[4] A. D. Hall Jr. "The ALTRAN system for rational function manipulation-a survey". In: *Proceedings of the second ACM symposium on Symbolic and algebraic manipulation*. ACM. 1971, pp. 153–157.

[5] M. Monagan and R. Pearce. "Polynomial division using dynamic arrays, heaps, and packed exponent vectors". In: *International Workshop on Computer Algebra in Scientific Computing*. Springer. 2007, pp. 295–315.

[6] S. C. Johnson. "Sparse polynomial arithmetic". In: *ACM SIGSAM Bulletin* 8.3 (1974), pp. 63–71.

Thank you!

Questions?