Efficient Algorithms and Parallel Implementations for Power Series Multiplication

Hamid Fathi

Computer Science Department
Western University

August, 2024

Western

Outline

Table of Contents

Introduction

- Power series are polynomial-like objects with potentially infinite terms.
- Power series have many applications in approximating functions and solving differential equations.
- Given two formal power series f and g, we are interested in expanding the product fg.
- Other operations such as division can be reduced to multiplication [1].
- Power series multiplication methods: fixed precision, varying precision

Example: Improving Product Precision

$$f = g = nx^{n-1}, n \in \mathbb{N} \tag{1}$$

$$f^{(3)}(x) = 1 + 2x + 3x^2 + 4x^3 + \ldots \tag{2}$$

$$g^{(3)}(x) = 1 + 2x + 3x^2 + 4x^3 + \ldots \tag{3}$$

$$fg^{(3)} = 1 + 4x + 10x^2 + 20x^3 + \ldots \tag{4}$$

Improving the product precision:

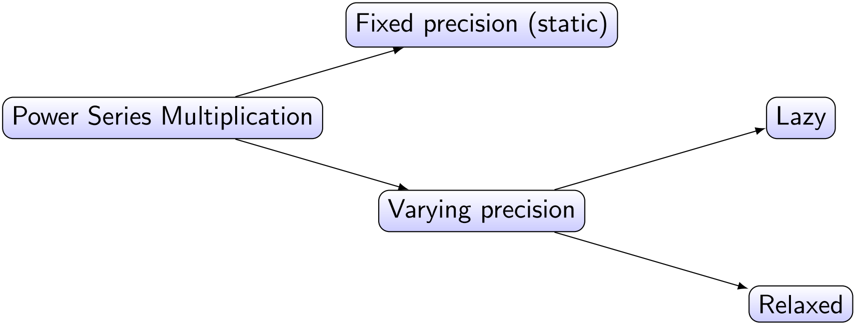
$$f^{(7)}(x) = 1 + 2x + 3x^2 + 4x^3 + 5x^4 + 6x^5 + 7x^6 + 8x^7 + \ldots \tag{5}$$

$$g^{(7)}(x) = 1 + 2x + 3x^2 + 4x^3 + 5x^4 + 6x^5 + 7x^6 + 8x^7 + \ldots \tag{6}$$

$$fg^{(7)} = 1 + 4x + 10x^2 + 20x^3 + 35x^4 + 56x^5 + 84x^6 + 120x^7 + \ldots \tag{7}$$

Previous computations can be reused.

Power Series Multiplication Methods



Power Series Multiplication Methods in the Literature

- Zealous or static algorithms expand f and g up to order n, multiply them and truncate the result at order n.
  - Fast multiplication algorithms e.g. DnC and FFT can be used.
  - They incur the overhead of recomputing known terms to enhance the product precision.
  - In many applications, the product precision cannot be determined a priori.
- Lazy algorithms compute the coefficients of fg one by one upon request [2].
  - No more computation is done than what is needed.
  - Computation of more coefficients of fg can be resumed without having to recompute previous coefficients.
  - Uses naive polynomial multiplication with $\mathcal{O}(n^2)$.
- Relaxed algorithms share the properties of lazy and zealous algorithms [3].
  - They employ algorithms with sub-quadratic time complexity.
  - They save computations by reusing already known product terms when higher product precision is requested.

Contributions

- We report on a parallel implementation of the relaxed multiplication algorithm for univariate power series by Joris van der Hoeven.
- We investigate the practicality of various schemes for multiplying multivariate power series, including the method proposed by Eric Schost at ISSAC 2005.
- We design, implement, and parallelize a new scheme for multivariate power series multiplication based on decomposition and using FFT and compare its performance with direct multiplication in serial and parallel modes.

Table of Contents

Relaxed Karatsuba Method

Assume $n = 2^k, k \in \mathbb{N}$.

Karatsuba Method For Polynomials (TPS)

$$f^{(n-1)} = \underbrace{f_0 + f_1 z + \dots + f_{\lceil n/2 \rceil - 1} z^{\lceil n/2 \rceil - 1}}_{f_*} + \underbrace{f_{\lceil n/2 \rceil} z^{\lceil n/2 \rceil} + \dots + f_{n-1} z^{n-1}}_{f^* z^{\lceil n/2 \rceil}}$$

$$g^{(n-1)} = \underbrace{g_0 + g_1 z + \dots + g_{\lceil n/2 \rceil - 1} z^{\lceil n/2 \rceil - 1}}_{g_*} + \underbrace{g_{\lceil n/2 \rceil} z^{\lceil n/2 \rceil} + \dots + g_{n-1} z^{n-1}}_{g^* z^{\lceil n/2 \rceil}}$$

$$f = f_* + f^* z^{\lceil n/2 \rceil}$$
$$g = g_* + g^* z^{\lceil n/2 \rceil}$$

$$fg = \underbrace{f_* g_*}_{low} + (\underbrace{(f_* + f^*)(g_* + g^*)}_{mid} - f_* g_* - \underbrace{f^* g^*}_{high}) z^{\lceil n/2 \rceil} + f^* g^* z^{2\lceil n/2 \rceil}$$

Relaxed Karatsuba Method

Relaxed Power Series Multiplication Using Karatsuba Method

$$f^{(2n-1)} = \underbrace{f_0 + f_1 z + \ldots + f_{n-1} z^{n-1}}_{f_*} + \underbrace{f_n z^n + \ldots + f_{2n-1} z^{2n-1}}_{f^* z^n}$$

$$g^{(2n-1)} = \underbrace{g_0 + g_1 z + \ldots + g_{n-1} z^{n-1}}_{g_*} + \underbrace{g_n z^n + \ldots + g_{2n-1} z^{2n-1}}_{g^* z^n}$$

$$f = f_* + f^* z^n$$
$$g = g_* + g^* z^n$$

$$fg' = \underbrace{f_* g_*}_{low} + (\underbrace{(f_* + f^*)(g_* + g^*)}_{mid} - f_* g_* - \underbrace{f^* g^*}_{high}) z^n + f^* g^* z^{2n}$$

Relaxed Karatsuba Method

The Karatsuba algorithm [4] can be a relaxed method of univariate power series multiplication.

- Fast, with a time complexity of $\mathcal{O}(n^{log_2 3})$.
- Product precision can be increased without recomputing previous terms. In fact, fg is already the result of one of the three multiplications, i.e. $f_* g_*$, needed to compute $fg^{'}$.

Serial Karatsuba Algorithm

Algorithm 1: DnCMultiply(f, g)

Input : Polynomials $f = f_0 + \ldots + f_{n-1}z^{n-1}$, $g = g_0 + \ldots + g_{n-1}z^{n-1}$.
Output: The product fg.
Function `DNCMultiply`(f, g, n)**:**

```
    if n ≤ Threshold then
        return Σ_{i=0}^{2n-2} ( Σ_{j=max(0,i+1-n)}^{min(n-1,i)} f_j g_{i-j} ) z^i;
    end
    else
        low ← DNCMultiply(f_*, g_*, n/2);
        mid ← DNCMultiply((f_* + f^*), (g_* + g^*), n/2);
        high ← DNCMultiply(f^*, g^*, n/2);
        return low + mid × z^{⌈n/2⌉} + high × z^{2⌈n/2⌉};
    end
return
```

if $n \leq \mathit{Threshold}$ **then**

  return $\sum_{i=0}^{2n-2} \left(\sum_{j=\max(0,i+1-n)}^{\min(n-1,i)} f_j g_{i-j} \right) z^i$;

end

else

  $low \leftarrow \texttt{DNCMultiply}(f_*, g_*, n/2)$;

  $mid \leftarrow \texttt{DNCMultiply}((f_* + f^*), (g_* + g^*), n/2)$;

  $high \leftarrow \texttt{DNCMultiply}(f^*, g^*, n/2)$;

  return $low + mid \times z^{\lceil n/2 \rceil} + high \times z^{2\lceil n/2 \rceil}$;

end

return

$T_1(n) = 3T_1(n/2) + \mathcal{O}(n) \Rightarrow$ Time complexity is $\mathcal{O}(n^{\log_2 3})$

Parallel Karatsuba Algorithm

Algorithm 2: DnCMultiply(f, g)

Input : Polynomials $f = f_0 + \ldots + f_{n-1}z^{n-1}$, $g = g_0 + \ldots + g_{n-1}z^{n-1}$.
Output: The product fg.
Function `DNCMultiply`(f, g, n)**:**
  if $n \leq$ *Threshold* **then**
    return $\sum_{i=0}^{2n-2}\left(\sum_{j=\max(0,i+1-n)}^{\min(n-1,i)} f_j g_{i-j}\right) z^i$;
  end
  else
    $low \leftarrow$ `DNCMultiply`$(f_*, g_*, n/2)$;
    $mid \leftarrow$ `DNCMultiply`$((f_* + f^*), (g_* + g^*), n/2)$;
    $high \leftarrow$ `DNCMultiply`$(f^*, g^*, n/2)$;
    return $low + mid \times z^{\lceil n/2 \rceil} + high \times z^{2\lceil n/2 \rceil}$;
  end
return

$T_\infty(n) = T_\infty(n/2) + \mathcal{O}(n) \Rightarrow$ Time complexity is $\mathcal{O}(n)$

Results

Implementation in OOP (Inheritance and Composition)

Power series (individual, f, g, and the product power series, $h = fg$) class structure is based on [3].

```
class TPS {
  int arraySize;
  mpq_t *coefficientArray;
};
class SeriesRep {
  TPS *phi;
  void (*PSGenerator)(int index, mpq_t *coefficient);
  virtual void next();
};
class Series {
  SeriesRep sR;
  virtual void getCoefficient(int index, mpq_t coefficient);
};
class LazyProdSeriesRep: public SeriesRep;
class RelaxedProdSeriesRep: public SeriesRep;
class LazyProdSeries: public Series{LazyProdSeriesRep lPSR;};
class RelaxedProdSeries: public Series{RelaxedProdSeriesRep rPSR;};
```

- Classes `LazyProdSeriesRep` and `RelaxedProdSeriesRep` inherit from `SeriesRep` class.
- The `next()` method of the `SeriesRep` class is overridden in `LazyProdSeriesRep` and `RelaxedProdSeriesRep`.
- Classes `LazyProdSeries` and `RelaxedProdSeries` inherit from the `Series` class.
- Class `Series` "has a" `SeriesRep`.
- Class `LazyProdSeries` "has a" `LazyProdSeriesRep`.
- Class `RelaxedProdSeries` "has a" `RelaxedProdSeriesRep`.

Power Series Multiplication Algorithm	Description	Polynomial Multiplication Used
Fixed precision Naive	Computes $(f^{(d)}g^{(d)} \mod x^d)$ for a given d without reusing previous results.	Plain (Quadratic)
Lazy (varying precision)	Computes the terms of degrees $d/2, \ldots, d-1$ of fg and stores them, assuming results up to $d/2-1$.	Plain
Fixed precision Karatsuba	Computes $(f^{(d)}g^{(d)} \mod x^d)$ for a given d without reusing previous results.	Karatsuba
Relaxed (varying precision)	Computes the terms of degree $d-1$ of fg, reusing previous results, and stores those terms.	Karatsuba

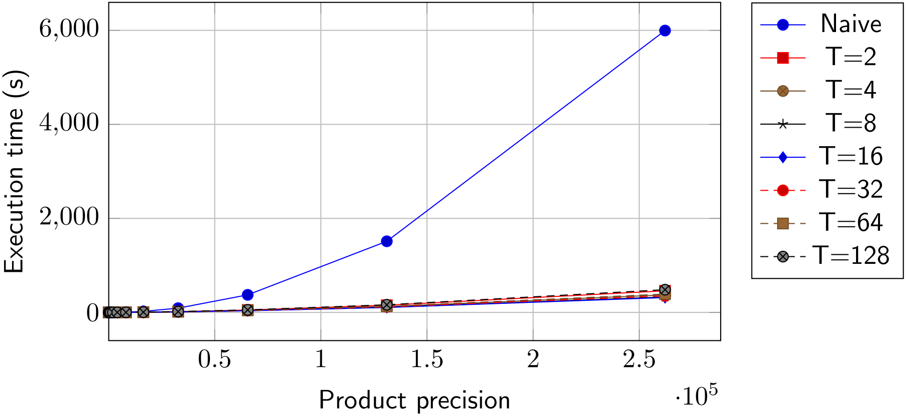
Table: Descriptions of the implemented power series multiplication algorithms.

Results: Static Karatsuba vs. Static Naive (Quadratic)

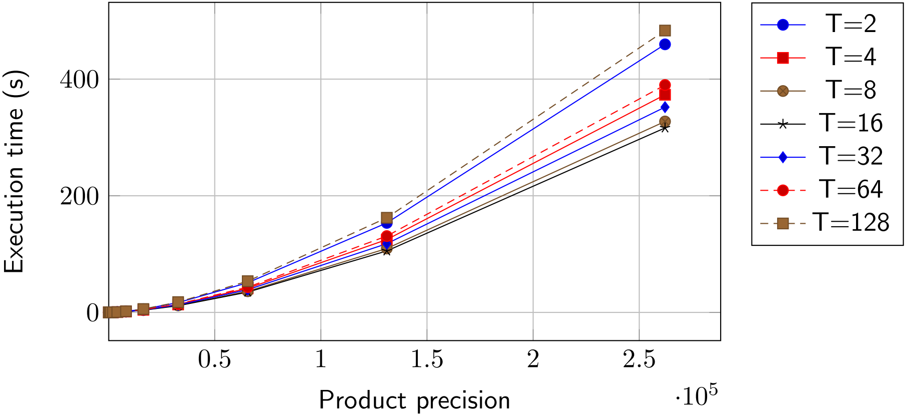
Product Precision	Static Naive	Static Karatsuba T = 2	T = 4	T = 8	T = 16	T = 32	T = 64	T = 128
2	5.47e-5	2.10e-5	2.50e-5	4.50e-5	9.00e-5	0.0003	0.0009	0.0036
4	8.81e-5	1.50e-5	6.00e-6	2.60e-5	6.40e-5	0.0003	0.0009	0.0035
8	0.0001	3.90e-5	2.70e-5	2.00e-5	6.00e-5	0.0003	0.0009	0.0033
16	0.0002	0.0001	8.50e-5	7.60e-5	5.90e-5	0.0003	0.0009	0.0032
32	0.0005	0.0004	0.0003	0.0002	0.0002	0.0002	0.0009	0.0031
64	0.0014	0.0012	0.0009	0.0008	0.0007	0.0008	0.0009	0.0030
128	0.0026	0.0035	0.0027	0.0024	0.0022	0.0025	0.0029	0.0035
256	0.0064	0.0102	0.0080	0.0072	0.0069	0.0082	0.0080	0.0083
512	0.0221	0.0254	0.0216	0.0190	0.0181	0.0203	0.0214	0.0240
1024	0.0848	0.0677	0.0547	0.0488	0.0468	0.0516	0.0566	0.0703
2^{11}	0.3400	0.2040	0.1641	0.1426	0.1366	0.1525	0.1697	0.2100
2^{12}	1.3647	0.6150	0.4961	0.4328	0.4126	0.4628	0.5142	0.6383
2^{13}	5.5781	1.8566	1.5011	1.3127	1.2566	1.4128	1.5612	1.9354
2^{14}	22.2973	5.6013	4.5192	3.9757	3.8064	4.2768	4.7292	5.8319
2^{15}	90.3683	16.8652	13.6735	12.0067	11.5153	12.9148	14.2982	17.6861
2^{16}	**373.7040**	**50.8692**	**41.3140**	**36.2653**	**34.8404**	**39.0014**	**43.2828**	**53.5939**
2^{17}	**1512.7900**	**153.4520**	**124.8990**	**109.5690**	**105.4290**	**117.8210**	**130.8930**	**162.2700**
2^{18}	**5995.5200**	**459.6320**	**373.1990**	**327.3920**	**316.3050**	**351.9760**	**389.8980**	**483.2490**

Execution times in seconds.

Results: Static Karatsuba vs. Static Naive (Quadratic)



Results: Karatsuba Method with Different DnC Thresholds

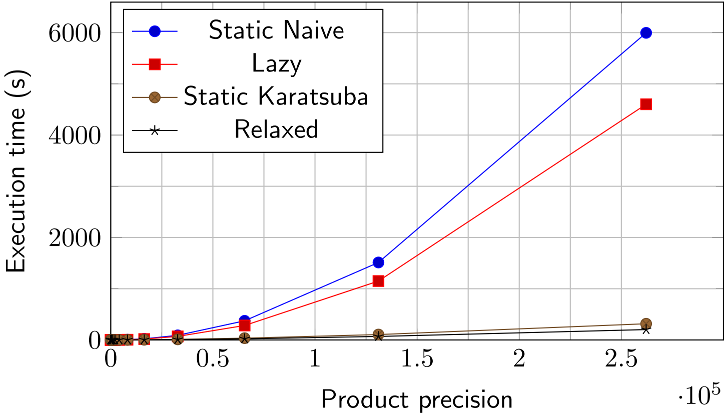


Results: Static (Naive and Karatsuba) vs. Dynamic (Lazy and Relaxed) Methods

Product Precision	Static Naive Multiplication	Lazy Multiplication	Static Karatsuba Multiplication	Relaxed Multiplication
2	0.0001	0.0001	0.0001	0.0001
4	0.0001	0.0001	0.0001	0.0001
8	0.0001	0.0001	0.0001	0.0001
16	0.0002	0.0001	0.0001	0.0001
32	0.0005	0.0002	0.0002	0.0002
64	0.0014	0.0003	0.0007	0.0006
128	0.0026	0.0011	0.0022	0.0017
256	0.0064	0.0042	0.0069	0.0047
512	0.0221	0.0167	0.0181	0.0131
1024	0.08480	0.0667	0.0468	0.0319
2^{11}	0.3400	0.2614	0.1366	0.0910
2^{12}	1.3647	1.0560	0.4126	0.2736
2^{13}	5.5781	4.2607	1.2566	0.8177
2^{14}	22.2973	17.2254	3.8064	2.4597
2^{15}	90.3683	69.6232	11.5153	7.4391
$\mathbf{2^{16}}$	**373.704**	**282.392**	**34.8404**	**22.4544**
$\mathbf{2^{17}}$	**1512.79**	**1149.76**	**105.429**	**68.3184**
$\mathbf{2^{18}}$	**5995.52**	**4601.09**	**316.305**	**202.588**

Execution times in seconds.

Results: Static (Naive and Karatsuba) vs. Dynamic (Lazy and Relaxed) Methods



Results: Static Karatsuba vs. Relaxed Methods

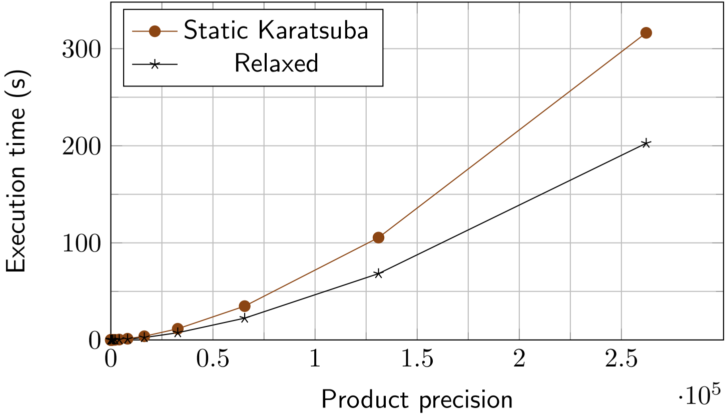
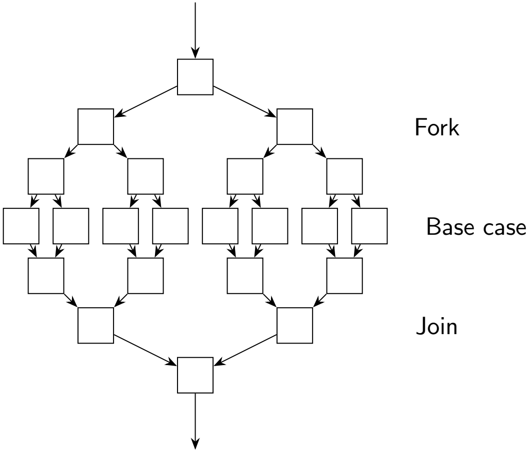


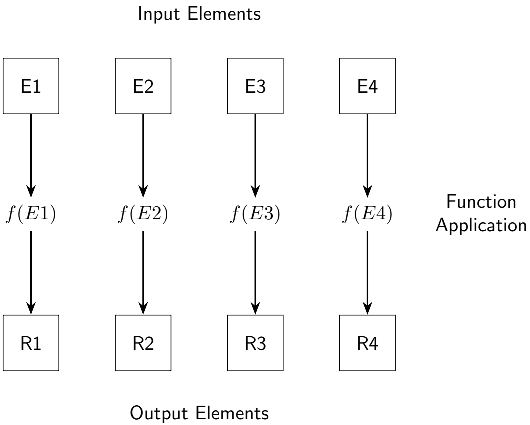
Table of Contents

Parallel Programming Patterns: Fork-Join



The recursive divide-and-conquer Karatsuba method is parallelized based on the fork-join model [5, 6].

Parallel Programming Patterns: Map



The partition method is parallelized based on the map model [5, 7].

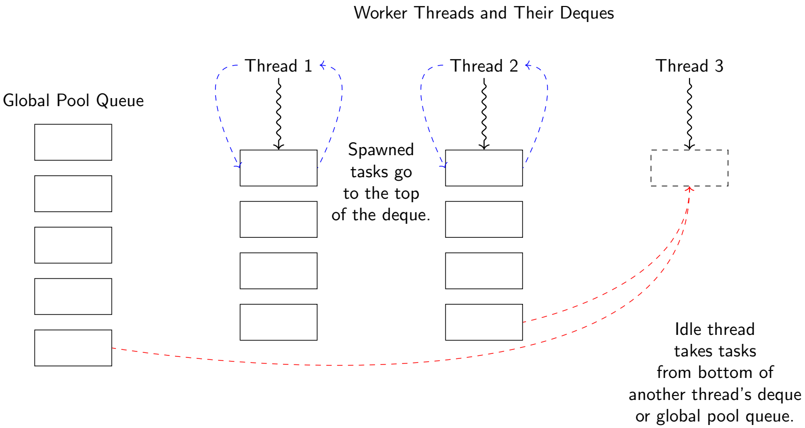
Multithreading

- In designing fast and high-performance algorithms, it is essential to consider the capabilities and constraints of modern hardware architectures.
- Increase in the CPU frequency was not feasible after around 2005 due to the exponential increase in power consumption and heat generation.
- The computer architecture industry shifted focus to multicore processors to improve performance.
- Concurrency vs. Parallelism
- Parallelism via multiprocessing: Higher overhead; OS protects shared data.
- Parallelism via multithreading: Lower overhead; programmer must synchronize.

Thread Pool Development

- To avoid oversubscription.
- To avoid the overhead of creating and destroying threads.
- Cilk [8] brings external dependencies and compatibility and portability issues.

Thread Pool with Work Stealing Scheduler

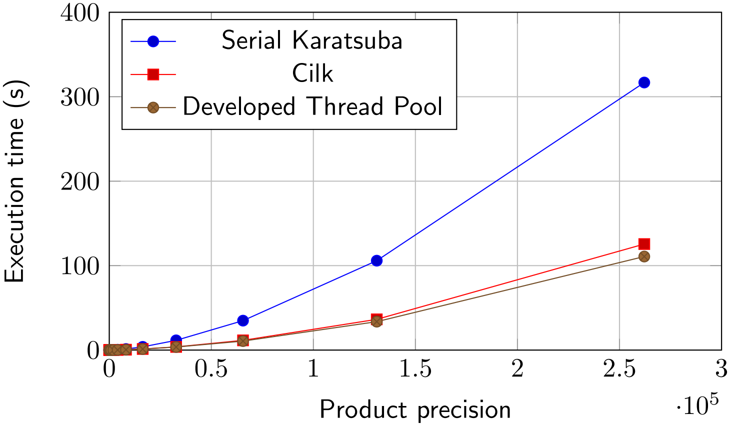


Results: Parallel Static Karatsuba via Thread Pool and Cilk

Product Precision	Serial Karatsuba	Parallel Karatsuba with Cilk	Parallel Karatsuba with Developed Thread Pool	Developed Thread Pool Speedup
2	7.24e-5	0.0001	0.0001	1.8166
4	8.08e-5	7.44e-5	0.0001	1.6717
8	9.28e-5	7.15e-5	0.0001	1.4927
16	9.13e-5	7.66e-5	0.0001	1.5029
32	0.0002	0.0011	0.0003	1.5298
64	0.0005	0.0009	0.0007	1.3520
128	0.0016	0.0016	0.0014	1.1552
256	0.0048	0.0049	0.0027	1.7779
512	0.0179	0.0087	0.0067	2.6720
1024	0.0468	0.0225	0.0182	2.5698
2^{11}	0.1374	0.0600	0.0513	2.6802
2^{12}	0.4159	0.1589	0.1537	2.7066
2^{13}	1.2632	0.4436	0.4223	2.9920
2^{14}	3.8125	1.2809	1.2045	3.1640
2^{15}	11.5335	3.7492	3.6231	3.1821
$\mathbf{2^{16}}$	**34.8514**	**11.5012**	**10.6497**	**3.2736**
$\mathbf{2^{17}}$	**105.8640**	**36.4011**	**33.6434**	**3.1465**
$\mathbf{2^{18}}$	**316.8210**	**125.5320**	**110.7680**	**2.8603**

The parallel execution time is about one-third of the serial execution time. This aligns with the Karatsuba algorithm.

Results: Parallel Static Karatsuba via Thread Pool and Cilk



Results: Parallel Relaxed Karatsuba via Thread Pool and Cilk

Product Precision	Serial Relaxed	Parallel Relaxed with Cilk	Parallel Relaxed with Developed Thread Pool	Developed Thread Pool Speedup
2	0.0001	0.0007	0.0008	0.125
4	0.0001	0.0001	0.0001	1.000
8	0.0001	0.0001	0.0001	1.000
16	0.0001	0.0001	0.0001	1.000
32	0.0002	0.0003	0.0006	0.333
64	0.0006	0.0007	0.0011	0.545
128	0.0017	0.0011	0.0013	1.308
256	0.0047	0.0027	0.0044	1.068
512	0.0131	0.0069	0.0054	2.426
1024	0.0319	0.0191	0.0155	2.058
2^{11}	0.0910	0.0493	0.0374	2.433
2^{12}	0.2736	0.1381	0.0985	2.777
2^{13}	0.8177	0.3981	0.2937	2.784
2^{14}	2.4597	1.1558	0.8202	2.999
2^{15}	7.4391	3.4227	2.3974	3.102
$\mathbf{2^{16}}$	**22.4544**	**10.4685**	**7.1766**	**3.129**
$\mathbf{2^{17}}$	**68.3184**	**33.5267**	**21.9230**	**3.116**
$\mathbf{2^{18}}$	**202.588**	**116.4900**	**74.9096**	**2.705**

Execution times in seconds.

Results: Parallel Relaxed Karatsuba via Thread Pool and Cilk

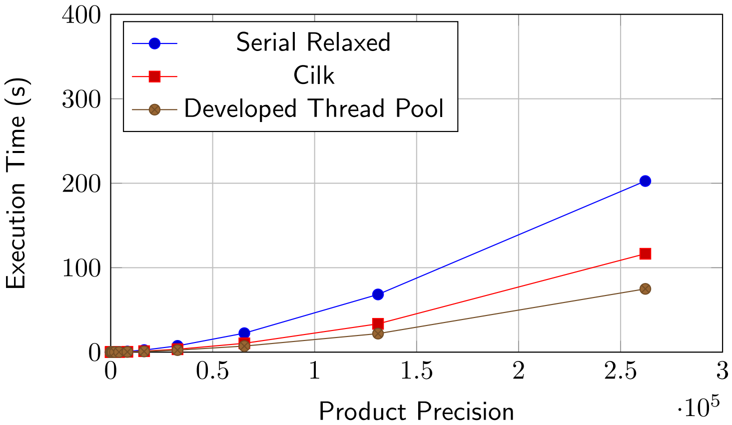


Table of Contents

Multivariate Power Series: Definitions

- Let $\mathbb{K}$ be a field of characteristic zero. We denote by $\mathbb{K}[[X_1, \ldots, X_n]]$ the ring of formal power series with coefficients in $\mathbb{K}$ and with ordered variables $X_1 < \cdots < X_n$.
- For $f \in \mathbb{K}[[X_1, \ldots, X_n]]$, we write

$$f = \sum_{e \in \mathbb{N}^n} a_e X^e,$$

  where $a_e \in \mathbb{K}$, $X^e = X_1^{e_1} \cdots X_n^{e_n}$, $e = (e_1, \ldots, e_n) \in \mathbb{N}^n$, and $|e| = e_1 + \cdots + e_n$.
- Let k be a non-negative integer. The *homogeneous part* and *polynomial part* of f in degree k are denoted $f_{(k)}$ and $f^{(k)}$, and are defined by

$$f_{(k)} = \sum\nolimits_{|e|=k} a_e X^e \quad \text{and} \quad f^{(k)} = \sum\nolimits_{i \leq k} f_{(i)}.$$

Computation Reuse in Multivariate Power Series Multiplication

In the multiplication of multivariate power series, doubling the product precision allows limited reuse of previous computations, particularly as the number of variables increases.

Fraction of Terms to Reuse

Let $T(n,k)$ be the maximum number of terms in a power series with n variables and a total degree of k.

$$T(n,k) = \sum_{0 \le i \le k} \binom{n+i-1}{i} = \binom{n+k}{k} \tag{8}$$

$$R = \frac{T(n,k)}{T(n,2k)} = \frac{\binom{n+k}{k}}{\binom{n+2k}{2k}} \tag{9}$$

Examples

For $n = 2,\ k = 2^7,\ R \approx 0.25.$
For $n = 3,\ k = 2^7,\ R \approx 0.13.$
For $n = 4,\ k = 2^7,\ R \approx 0.06.$

An Evaluation-Interpolation Scheme

- Standard results from the theory of Gröbner bases [9, 10] and the deflation techniques in [11] are used.
- Let $G = \{g_1, \ldots, g_m\}$ be the reduced minimal Gröbner basis of a monomial ideal $\mathcal{I}$ of $\mathbb{K}[X_1, \ldots, X_n]$, w.r.t. some admissible monomial order τ, which refines the partial order comparing total degrees of monomials.
- In practice, the ideal $\mathcal{I}$ would often be $\mathcal{M}^k$ or $\langle X_1^k, X_2^k, \ldots, X_n^k \rangle$ for some positive integer k. Let $A, B \in \mathbb{K}[X_1, \ldots, X_n]$. Our goal is to compute

$$C := A \cdot B \mod \mathcal{I} \tag{10}$$

An Evaluation-Interpolation Scheme: Introductory Example

- We first consider an example with $n = 1$ and $\mathcal{I} = \langle X_1^2 \rangle$. Thus:

$$A \;=\; a_1\, X_1 \;+\; a_0 \;\text{ and }\; B \;=\; b_1\, X_1 \;+\; b_0, \tag{11}$$

  for some $a_1, a_0, b_1, b_0 \in \mathbb{K}$. We have $T = \{1, X_1\}$.
- To apply an evaluation-interpolation scheme, if $\mathcal{I}$ were $\langle X_1(X_1 - 1) \rangle$, then the Chinese Remaindering Theorem can be used.
- To reduce to this case, we replace $\mathcal{I}$ with $\mathcal{I}_Z := \langle X_1(X_1 - Z) \rangle$, where Z is a new variable, which is meant to be specialized to zero in order to retrieve the ideal $\mathcal{I}$.
- We evaluate A and B at $X_1 = 0$ and $X_1 = Z$. We build the corresponding evaluation matrix, by evaluating each of the basis elements of $T = \{1, X_1\}$ at each of the points of $\{0, Z\}$, yielding to:

$$M_{\mathrm{eval}} \;=\; \begin{pmatrix} 1 & 0 \\ 1 & Z \end{pmatrix} \tag{12}$$

  Next, we compute the coordinates of A and B in the basis given by T:

$$A = \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} \;\text{ and }\; B = \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} \tag{13}$$

Evaluation-Interpolation Scheme: Introductory Example

The values of A and B at the points of $P_Z = \{0, Z\}$ are given by:

$$A_{\mathsf{eval}} = \begin{bmatrix} a_0 \\ Za_1 + a_0 \end{bmatrix} \text{ and } B_{\mathsf{eval}} = \begin{bmatrix} b_0 \\ Zb_1 + b_0 \end{bmatrix} \tag{14}$$

$$(AB)_{\mathsf{eval}} = \begin{bmatrix} a_0 b_0 \\ (Za_1 + a_0)(Zb_1 + b_0) \end{bmatrix} \tag{15}$$

$$M_{\mathrm{interp}} = \begin{bmatrix} 1 & 0 \\ -\frac{1}{\_Z} & \frac{1}{\_Z} \end{bmatrix} \tag{16}$$

Computing the matrix-vector product $M_{\mathrm{interp}}(AB)_{\mathrm{eval}}$ produces:

$$AB := a_0\, b_0 + \left(-\frac{a_0\, b_0}{\_Z} + \frac{(\_Za_1 + a_0)(\_Zb_1 + b_0)}{\_Z} \right) x1 . \tag{17}$$

Finally, we evaluate this latter at $Z = 0$ which yields:

$$a_0\, b_1\, x1 + a_1\, b_0\, x1 + a_0\, b_0 , \tag{18}$$

that is, $A \cdot B \mod \mathcal{I}$.

Results: Evaluation-Interpolation Scheme

$n=2$		$n=3$	
d_i	Time (s)	d_i	Time (s)
2	0.0001	2	0.0003
4	0.0010	3	0.0017
6	0.0077	4	0.0091
8	0.0213	5	0.0380
10	0.0523	6	0.1121
12	0.1168		
14	0.2084		
16	0.3622		

Table: Execution time (s) of the evaluation-interpolation scheme for power series of different number of variables and partial degrees.

A Decomposition Scheme (Partition)

Computing $(fg)^{(2k)}$ from $f^{[k]}g^{[k]}$

We observe that there exist polynomials $Q_1^f, \ldots, Q_n^f, Q_1^g, \ldots, Q_n^g$ such that we have:

$$f^{(2k)} = f^{[k]} + Q_1^f X_1^k + \cdots + Q_n^f X_n^k \text{ and } g^{(2k)} = g^{[k]} + Q_1^g X_1^k + \cdots + Q_n^g X_n^k, \quad (19)$$

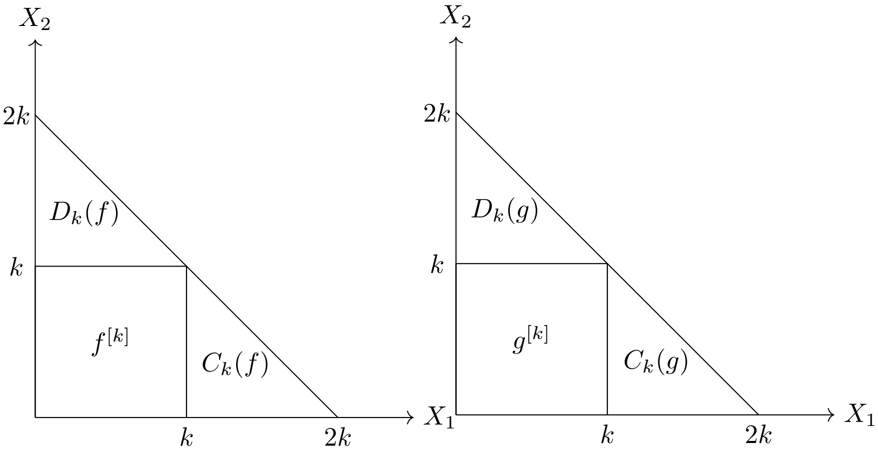
so that none of the $Q_1^f, \ldots, Q_n^f, Q_1^g, \ldots, Q_n^g$ has a constant term, and each of them has a total degree of at most k.

Proposition

We have:

$$(fg)^{(2k)} \equiv f^{[k]}g^{[k]} + (Q_1^f g + Q_1^g f)X_1^k + \cdots + (Q_n^f g + Q_n^g f)X_n^k. \mod \mathcal{M}^{2k+1}. \quad (20)$$

Example: A Decomposition Scheme (Partition)



$$f(X_1, X_2) = \left[f^{[k]} + C_k(f) X_1^k + D_k(f) X_2^k \right] \quad (21)$$

$$g(X_1, X_2) = \left[g^{[k]} + C_k(g) X_1^k + D_k(g) X_2^k \right] \quad (22)$$

Example: A Decomposition Scheme (Partition)

Example: $f(X_1, X_2), g(X_1, X_2)$

We have:

$$
\begin{array}{rcl}
(fg)^{(2k)} & = & \left[f^{[k]} + C_k(f)X_1^k + D_k(f)X_2^k\right] \cdot \left[g^{[k]} + C_k(g)X_1^k + D_k(g)X_2^k\right] \mod \mathcal{M}^{2k+1} \\
& = & f^{[k]}\left(g^{[k]} + C_k(g)X_1^k + D_k(g)X_2^k\right) + g^{[k]}\left(C_k(f)X_1^k + D_k(f)X_2^k\right)
\end{array}
\tag{23}
$$

Therefore, we perform 5 multiplications instead of 9 (eliminating 4 multiplications).

- In the case of three variables, we perform 7 multiplications instead of 16 (eliminating 9 multiplications).
- In the case of four variables, we perform 9 multiplications instead of 25 (eliminating 16 multiplications).
- In the case of five variables, we perform 11 multiplications instead of 36 (eliminating 25 multiplications).

Results: Decomposition (Partition) Scheme, 2 Variables

Total Degree (k)	Direct Method	Partition Method	Parallel Direct Method	Parallel Partition Method	Partition Method Speedup
2	0.0001	0.0002	0.0023	0.0002	0.9016
4	0.0003	0.0005	0.0025	0.0005	0.9765
8	0.0011	0.0015	0.0047	0.0005	2.8048
16	0.0042	0.0052	0.0068	0.0015	3.4791
32	0.0128	0.0179	0.0139	0.0050	3.5881
64	0.0494	0.0579	0.0298	0.0142	4.0757
128	0.2116	0.2355	0.0815	0.0656	3.5903
256	0.8285	1.0217	0.3018	0.2308	4.4278
512	3.3755	3.9346	0.9162	0.9589	4.1033
1024	14.4911	16.8674	3.6976	4.0522	4.1629
2^{11}	61.8196	73.0748	15.6075	16.4796	4.4356
2^{12}	273.0004	309.6342	67.2966	71.7569	4.3162

Table: Performance comparison for 2 variables in seconds. The Parallel Partition Method is 5 serial multiplications executed in parallel. The Parallel Direct Method is one parallel multiplication.

Results: Decomposition (Partition) Scheme, 2 Variables

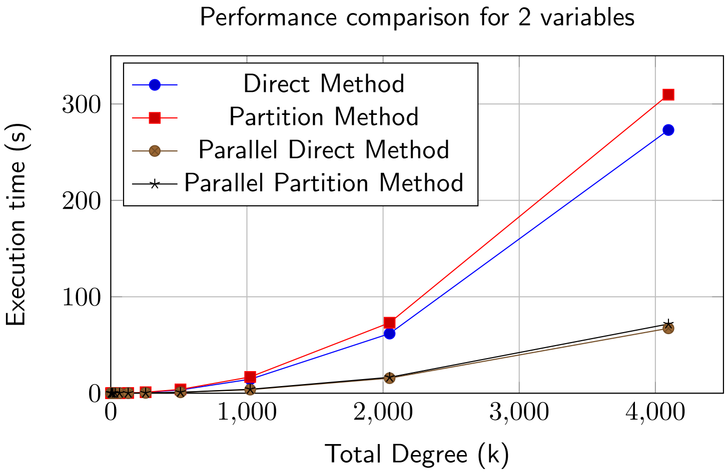


Figure: Execution time comparison for 2 variables using Direct, Partition, Parallel Direct, and Parallel Partition methods.

Results: Decomposition (Partition) Scheme, 3 Variables

Total Degree (k)	Direct Method	Partition Method	Parallel Direct Method	Parallel Partition Method	Partition Method Speedup
2	0.0008	0.0009	0.0040	0.0006	1.4761
4	0.0054	0.0050	0.0075	0.0018	2.8185
8	0.0366	0.0328	0.0271	0.0062	5.3004
16	0.2902	0.2292	0.0905	0.0417	5.4897
32	2.4415	1.9020	0.5981	0.3285	5.7904
64	18.5132	17.0732	3.6049	3.0133	5.6647
128	194.3451	129.6254	35.0982	21.7583	5.9590
160	355.3166	259.5845	61.8392	41.6352	6.2358

Table: Performance comparison for 3 variables in seconds. The Parallel Partition Method is 7 serial multiplications executed in parallel. The Parallel Direct Method is one parallel multiplication.

Results: Decomposition (Partition) Scheme, 3 Variables

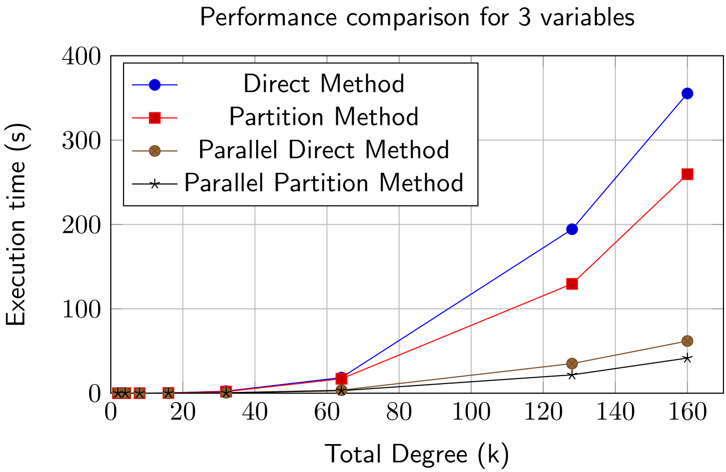


Figure: Execution time comparison for 3 variables using Direct, Partition, Parallel Direct, and Parallel Partition methods.

Results: Decomposition (Partition) Scheme, 4 Variables

Total Degree (k)	Direct Method	Partition Method	Parallel Direct Method	Parallel Partition Method	Partition Method Speedup
2	0.0052	0.0036	0.0073	0.0015	2.4711
4	0.0541	0.0360	0.0284	0.0080	4.5106
8	0.9321	0.4569	0.2197	0.0961	4.7554
16	14.6326	7.3545	2.7641	1.5972	4.6050
32	265.8875	132.6068	37.8152	17.7673	7.4638
40	421.1498	204.8680	70.5869	27.0816	7.5664

Table: Performance comparison for 4 variables in seconds. The Parallel Partition Method is 9 serial multiplications executed in parallel. The Parallel Direct Method is one parallel multiplication.

Results: Decomposition (Partition) Scheme, 4 Variables

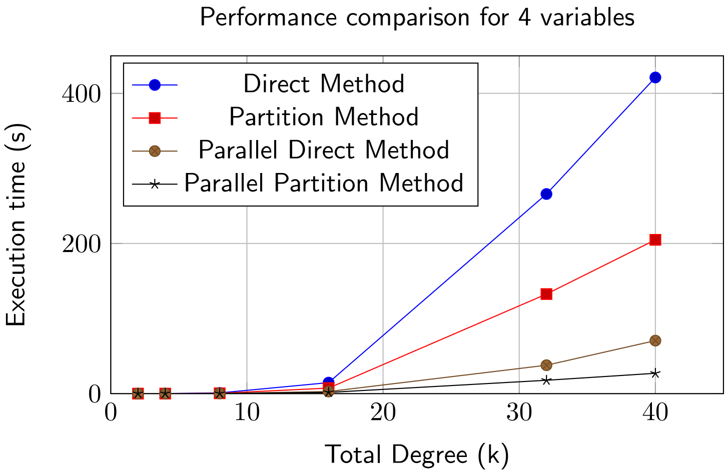


Figure: Execution time comparison for 4 variables using Direct, Partition, Parallel Direct, and Parallel Partition methods.

Results: Decomposition (Partition) Scheme, 5 Variables

Total Degree (k)	Direct Method	Partition Method	Parallel Direct Method	Parallel Partition Method	Partition Method Speedup
2	0.0465	0.0192	0.0325	0.0047	4.1256
4	1.1803	0.4159	0.2401	0.0871	4.7768
6	9.4207	2.4694	1.6607	0.3709	6.6585
8	45.8923	11.5437	6.9359	1.3823	8.3512
10	148.4728	43.2585	20.4864	5.2195	8.2857
12	336.4207	104.9664	43.8913	12.0742	8.6923

Table: Performance comparison for 5 variables in seconds. The Parallel Partition Method is 11 serial multiplications executed in parallel. The Parallel Direct Method is one parallel multiplication.

Results: Decomposition (Partition) Scheme, 5 Variables

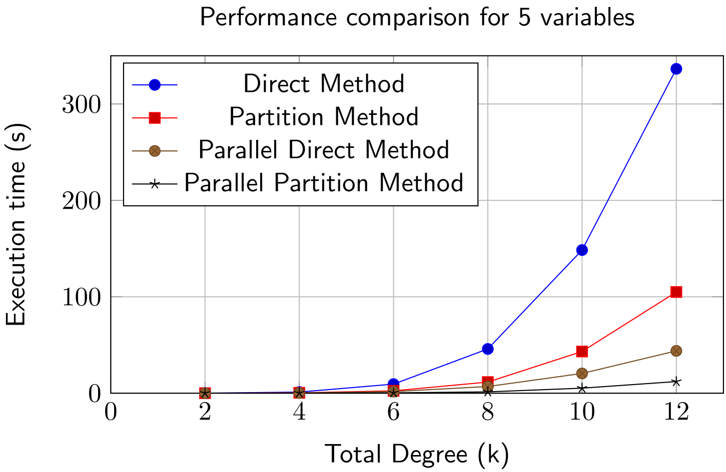


Figure: Execution time comparison for 5 variables using Direct, Partition, Parallel Direct, and Parallel Partition methods.

Table of Contents

References I

Allan K. Steel.
Reduce everything to multiplication.
Technical report, The University of Sydney, 2015.

Joris van der Hoeven.
Lazy multiplication of formal power series.
In W. W. Küchlin, editor, *Proceedings of the International Symposium on Symbolic and Algebraic Computation (ISSAC '97)*, pages 17–20, Maui, Hawaii, July 1997.

Joris van der Hoeven.
Relax, but don't be too lazy.
J. Symb. Comput., 34(6):479–542, 2002.

A. A. Karatsuba and Y. Ofman.
Multiplication of multidigit numbers on automata.
Soviet Physics Doklady, 7:595–596, 1963.
URL: `http://cr.yp.to/bib/entries.html#1963/karatsuba`.

References II

Michael D. McCool, Arch D. Robison, and James Reinders.
Structured parallel programming patterns for efficient computation, 2012.

Anthony Williams.
C++ Concurrency in Action.
Manning Publications, Shelter Island, NY, second edition, 2019.

Alexander Brandt.
The Design and Implementation of a High-Performance Polynomial System Solver.
Phd dissertation, The University of Western Ontario, 2022.
Electronic Thesis and Dissertation Repository. 8733.

Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou.
Cilk: an efficient multithreaded runtime system.
SIGPLAN Not., 30(8):207–216, aug 1995.

References III

David A. Cox, John Little, and Donal O'Shea.
Ideals, varieties, and algorithms - an introduction to computational algebraic geometry and commutative algebra (2. ed.).
Undergraduate texts in mathematics. Springer, 1997.

David A Cox, John Little, and Donal O'shea.
Using algebraic geometry, volume 185.
Springer Science & Business Media, 2005.

Éric Schost.
Multivariate power series multiplication.
In *Symbolic and Algebraic Computation, International Symposium ISSAC 2005, Beijing, China, July 24-27, 2005, Proceedings*, pages 293–300. ACM, 2005.

Thank you!

E-mail: sfathi4@uwo.ca