Design and Implementation of Multi-Threaded Algorithms in Polynomial Algebra

Marc Moreno Maza

Ontario Research Center for Computer Algebra
Departments of Computer Science and Mathematics
University of Western Ontario, Canada

ISSAC 2021, Saint Petersburg, Russia, July 20

Acknowledgements

- Many thanks to the ISSAC 2021 organizers, in particular Ekaterina Shemyakova for her kind invitation and, Marc Mezzarobba for his help with the proceedings.
- I would have loved to visit Saint Petersburg and our Russian colleagues.
- This tutorial is based on research projects in which many of my former and current graduate students have played an essential role. By alphabetic order: Ali Asadi, Alexander Brandt Changbo Chen, Xiaohui Chen, Svyatoslav Covanov, Sardar Haque, Xin Li, Farnam Mansouri, Davood Mohajerani, Robert Moir, Wei Pan, Delaram Talaashrafi, Linxiao Wang, Ning Xie, Yuzhen Xie, Haoze Yuan.
- This tutorial is based on collaborations with MIT/CSAIL, Intel and IBM Canada, with funding support from IBM and NSERC of Canada.
- Special thanks go to Alexander Brandt who helped me put these notes together and who is leading the development of the Basic Polynomial Algebra Subprograms (BPAS) [1]. ⏩ skip slide

My journey in parallel computing (1/2)

- First experiences with the `pthread` library (with Xin Li, 2006) for computing normal forms and distributed computing (with Yuzhen Xie, 2006) for computing triangular decompositions.
- Visiting scholar at MIT/CSAIL (2008-2009) in the team of Charles Leiserson,
  - ↳ I discover `Cilk` and cache complexity,
  - ↳ Yuzhen Xie starts BPAS with multi-dimensional FFTs/TFTs
- First steps with NVIDIA CUDA, with Wei Pan in 2008-2011 then with Sardar Haque in 2009-2012.
- Return to Canada and collaboration with IBM, more interested in `OpenMP` than `Cilk`:
  - ↳ automatic translation between `OpenMP` and `Cilk`, see the Metafork (Xiaohui Chen & Ning Xie)
  - ↳ participation to the C++ committee, designing C++ 11.

My journey in parallel computing (2/2)

- In 2013, IBM Canada (and us 🙂) switches to automatic parallelization from C to CUDA:
  - ↳ this culminates with KLARAPTOR, a tool for dynamically finding optimal kernel launch parameters targeting CUDA programs (Alex Brandt, Davood Mohajerani, Linxiao Wang)
  - ↳ work on polyhedral geometry (Delaram Talaashrafi, Rui-Juan Jung Linxiao Wang) to support automatic parallelization.
- With the arrival of Alexander Brandt, the BPAS library took off (see next slide):
  - ↳ Key contributions from Svyatoslav Covanov (FFT) and Ali Asadi (subresultants)
  - ↳ Ali Asadi, Alex Brandt and Robert Moir ported the algorithms of Maple's `RegularChains` library to BPAS.
  - ↳ Experimentation with pipelining (Alex Brandt, Delaram Talaashrafi)

The computer algebra community and parallel computing

- Parallelization of high-level algebraic and geometric algorithms was more common roughly 30 years ago
  - ↳ Such as in Gröbner bases [5, 11, 18] and CAD [39]
  - ↳ see the proceedings of CAP [15, 40] and PASCO [26, 27]

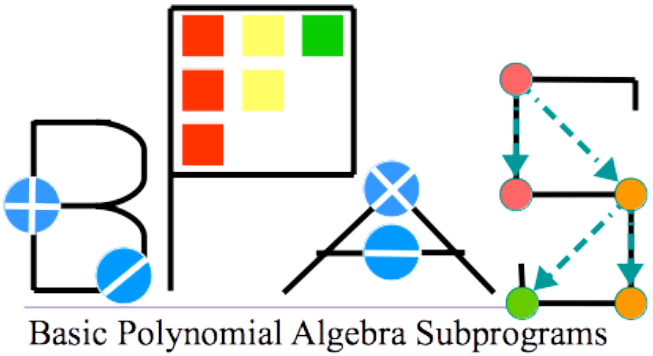
- In the past 15 years, work on parallelism has been on *low-level* routines with *regular parallelism*:
  - ↳ Polynomial arithmetic [22, 33]
  - ↳ Modular methods for GCDs and factorization [29, 35]
  - ↳ see the proceedings of PASCO [16, 19, 36, 37].

- Recently, high-level algorithms are receiving attention again:
  - ↳ The normalization algorithm of [6] finds components serially, then processes each component with a parallel map
  - ↳ Today, parallel decomposition of polynomial systems is made (much) easier by multi/many-core processors, new concurrency platforms and libraries (e.g. C++ 11) and, of course, better algebraic algorithms 🙂.

Tentative Plan

- We will not define multithreading, but I will show some pictures 🙂.
- Part 1: Memory access patterns (45 min)
- Part 2: Parallel programming patterns (30 mins)
- Part 3: BPAS multi-threading (15 mins)
- Part 4: Extracting patterns (10 mins)
- The conclusions will be after Parts 1 and 2.
- I prefer to take questions between parts

The BPAS library



http://www.bpaslib.org/

A high-performance polynomial algebra library

- Core of library written in C, wrapped in C++ interface for usability and object-oriented programming

Optimized algorithms and data structures, data locality, and parallelism

- Sparse multivariate polynomials [3], dense univariate and bivariate [38]
- Triangular decomposition of polynomial systems [2, 4]

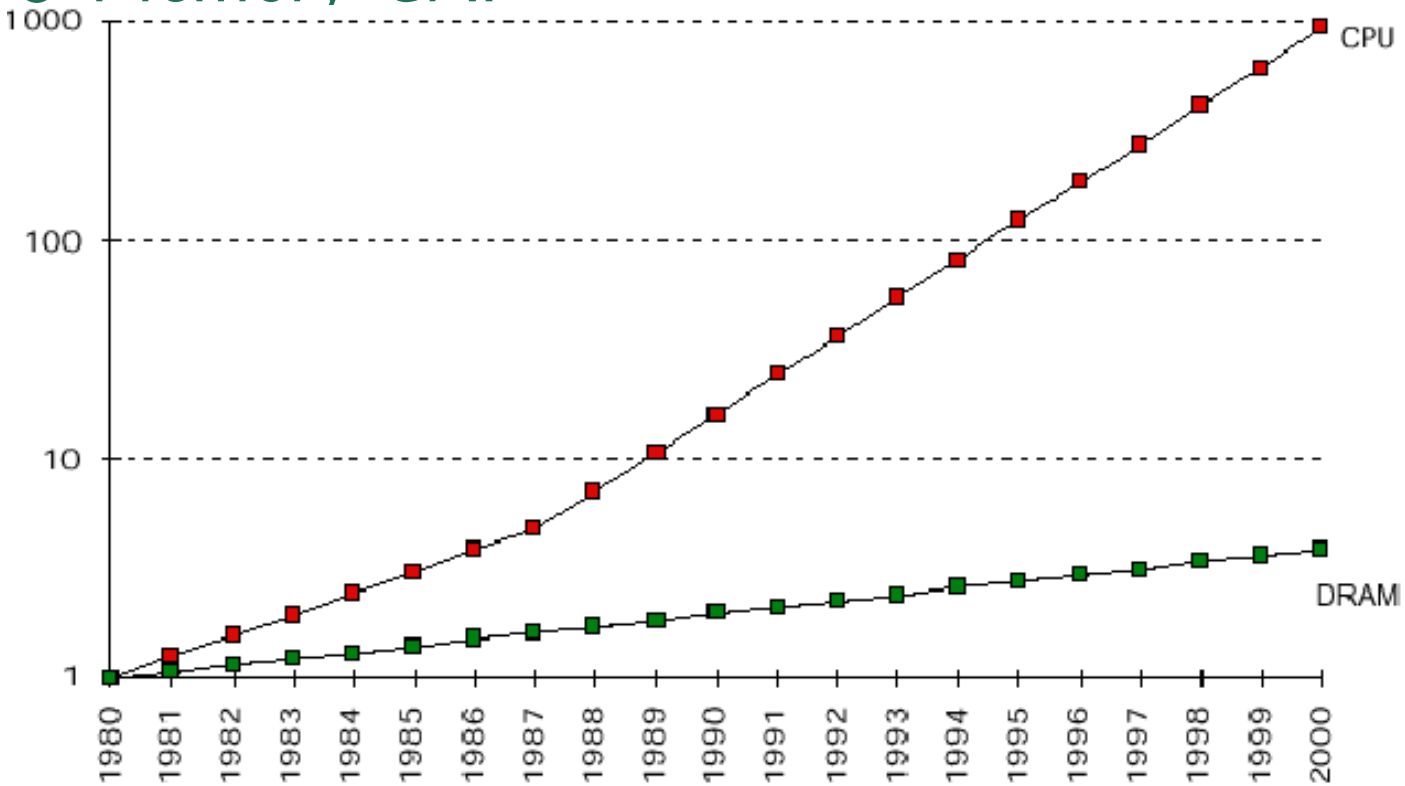
User-friendly, object-oriented interface based on template meta-programming [9]

- A natural encoding of the algebraic hierarchy
- "Dynamic" creation of algebraic types through composition
- Compile-time type safety between algebraic types

Generic support for parallel programming and parallel patterns (this talk)

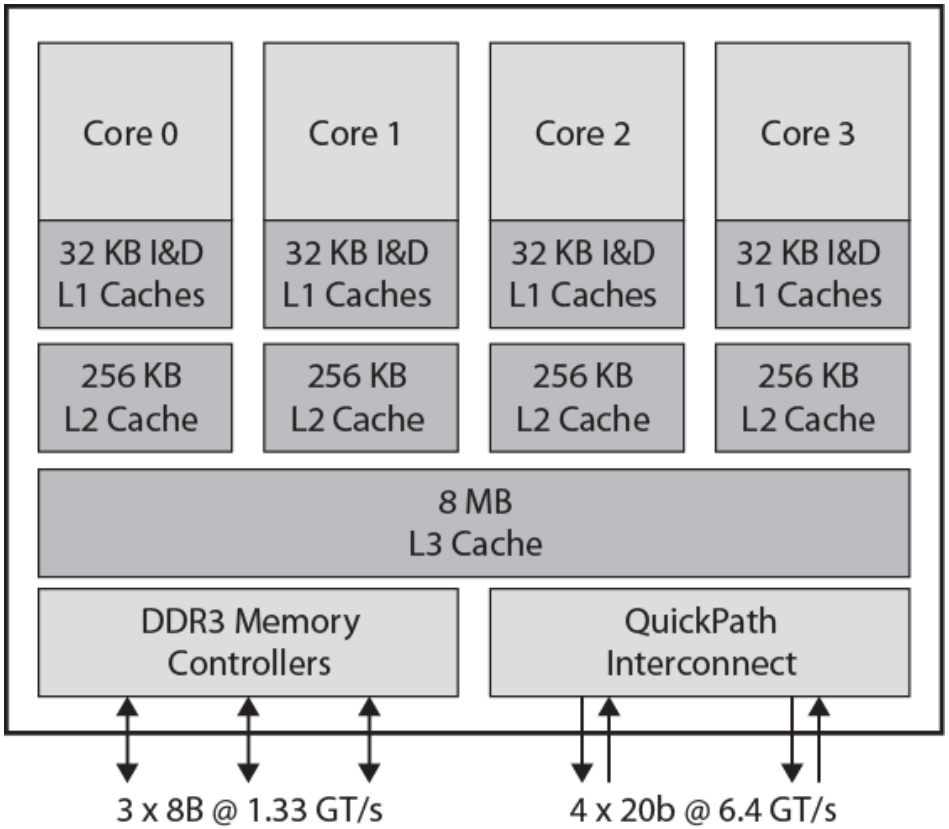
Outline

The CPU-Memory GAP



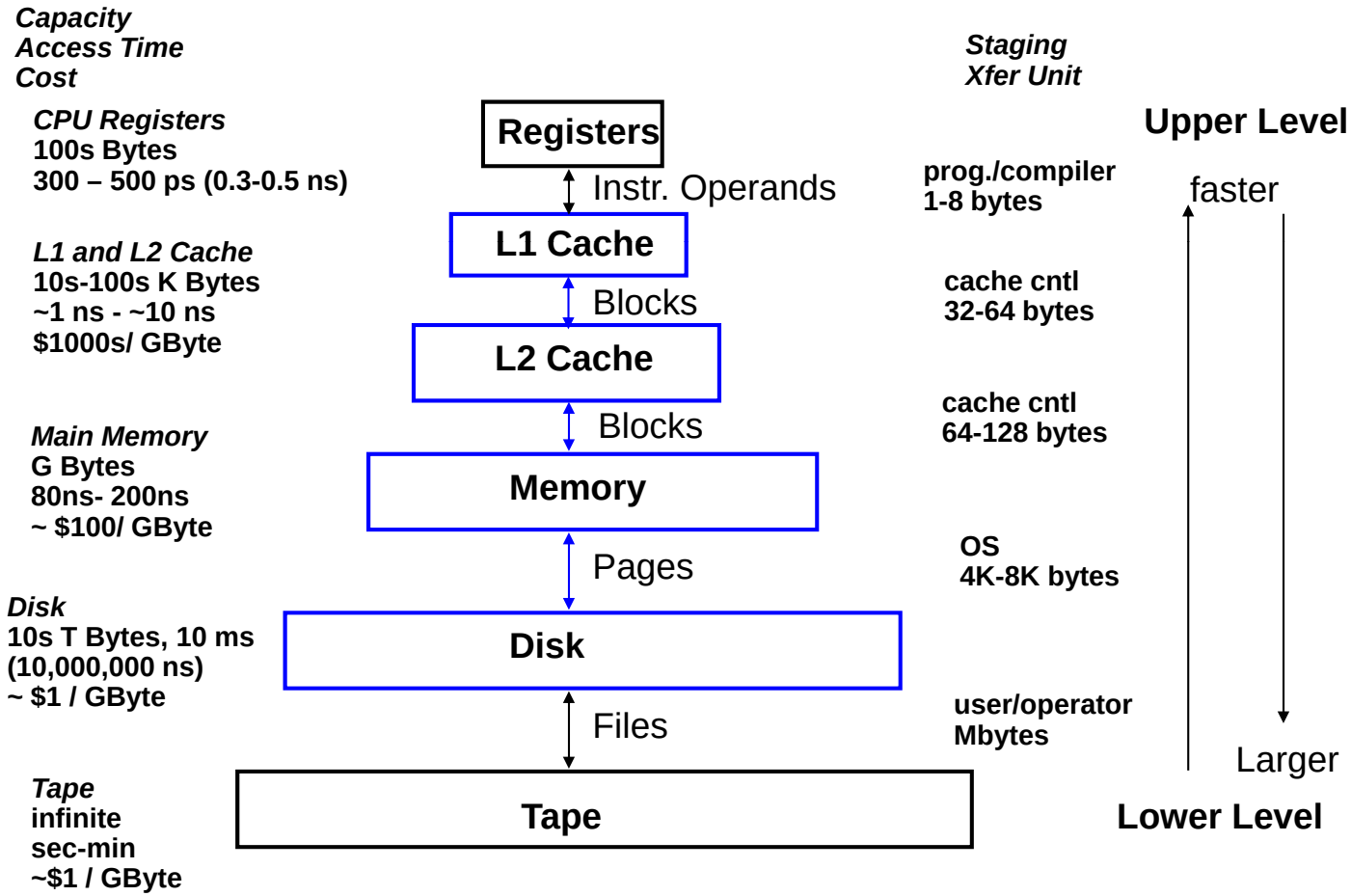
- In the 1980's, a memory access and a CPU operation were both as slow as the other
- CPU frequency increased between 1985 and 2005 has reduced CPU op times much more than DRAM technology improvement could reduce memory access times
- Even after the introduction of multicore processors, the gap is still huge.

Multicore processor



- In the 1st Gen. Intel Core i7, each core had an L1 data cache and an L1 instruction cache, together with a unified L2 cache
- The cores share an L3 cache
- Note the sizes of the successive caches

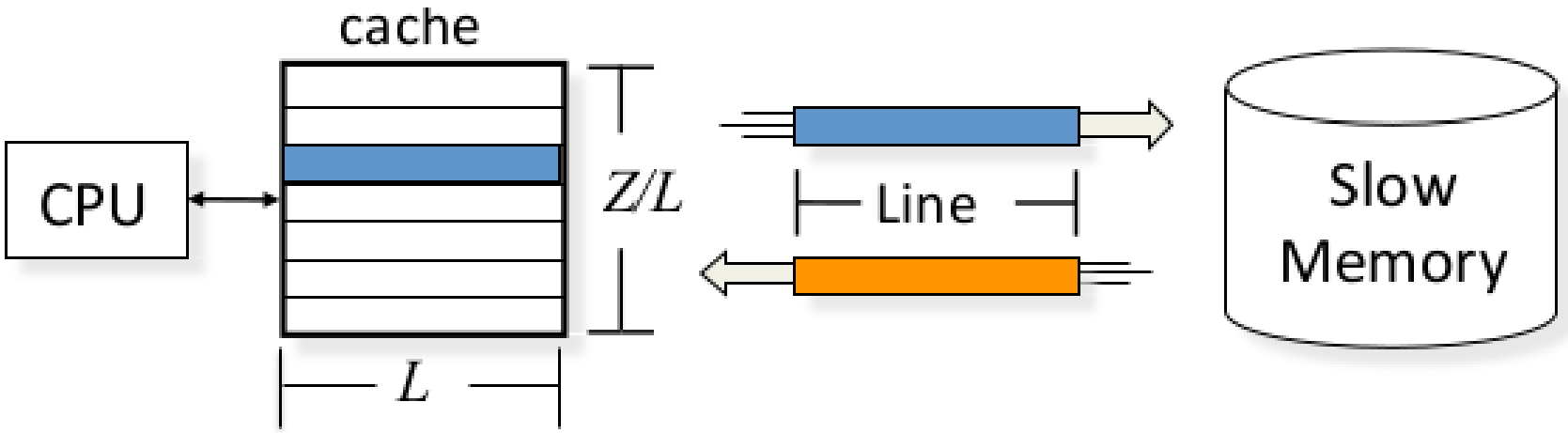
Hierarchical memory



- Data moves in blocks (cache-lines, pages) between levels
- On the right, note the block sizes
- On the left, note the access times, sizes and prices.

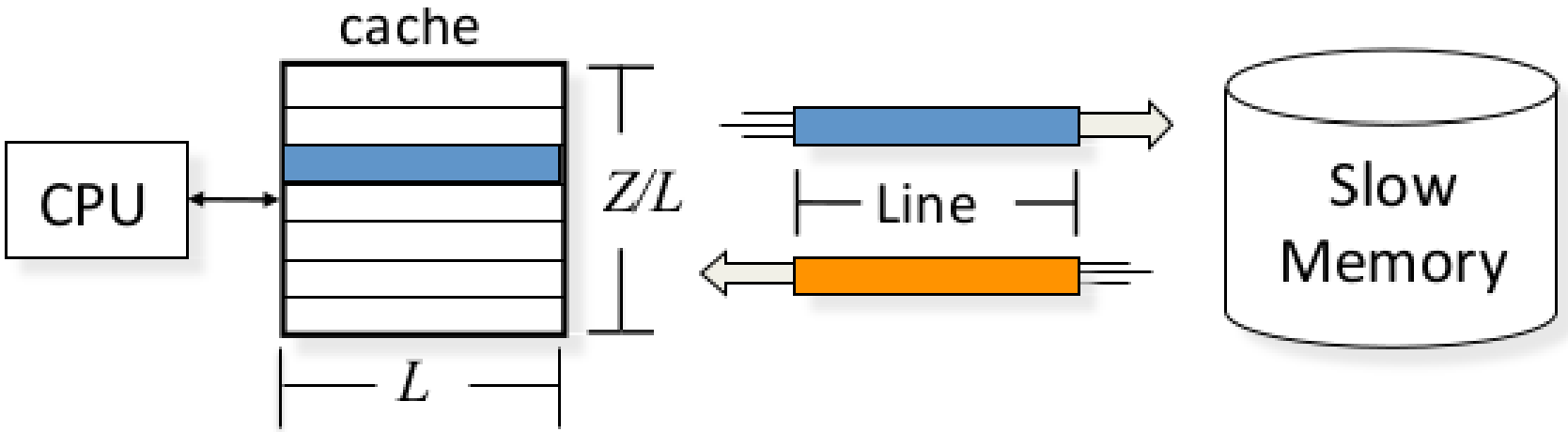
Outline

The ideal cache model (1/5)



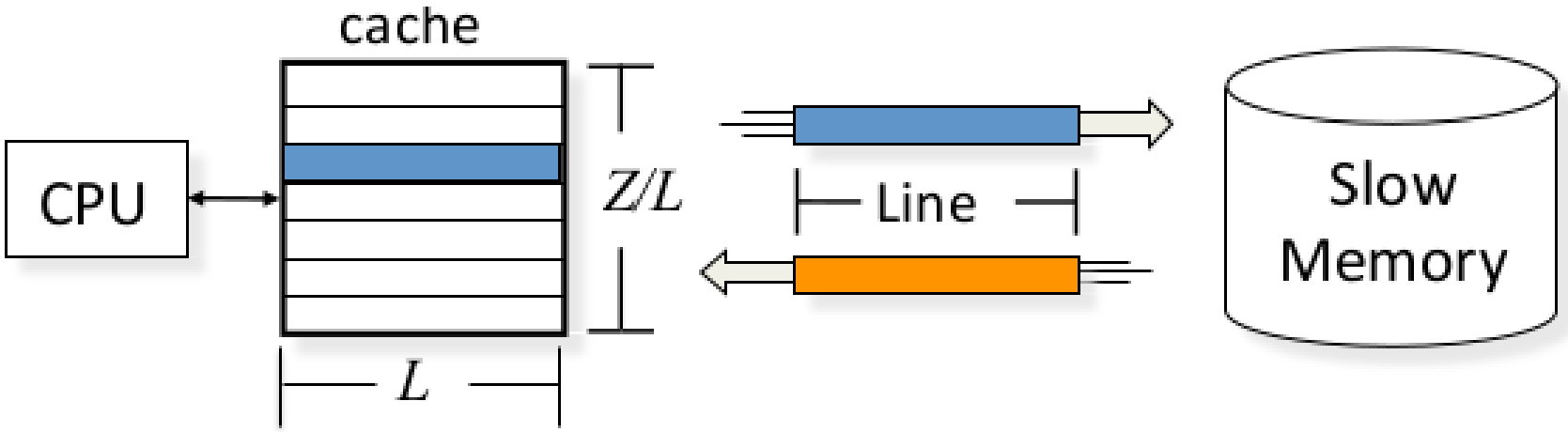
- Computer with a **two-level memory hierarchy**:
  - ↳ an ideal (data) cache of Z words partitioned into Z/L *cache lines*, where L is the number of words per cache line.
  - ↳ an arbitrarily large main memory.
- Data moved between cache and main memory are always cache lines.
- The cache is **tall**, that is, Z is much larger than L, say $Z \in \Omega(L^2)$.

The ideal cache model (2/5)



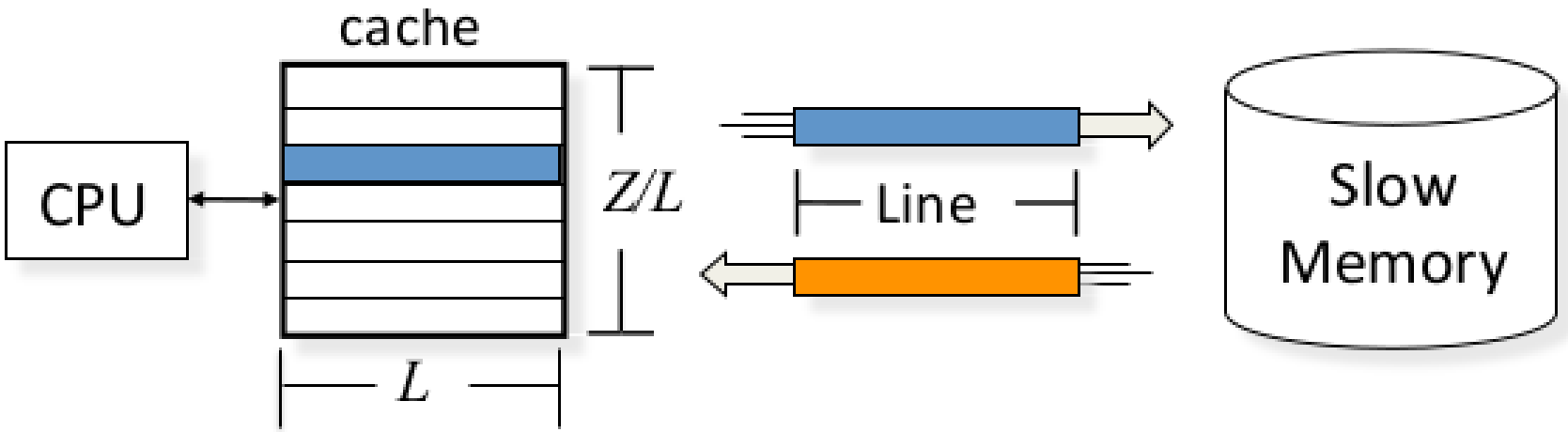
- The processor can only reference words that reside in the cache.
- If the referenced word belongs to a line already in cache, a **cache hit** occurs, and the word is delivered to the processor.
- Otherwise, a **cache miss** occurs, and the line is fetched and installed into the cache.

The ideal cache model (3/5)



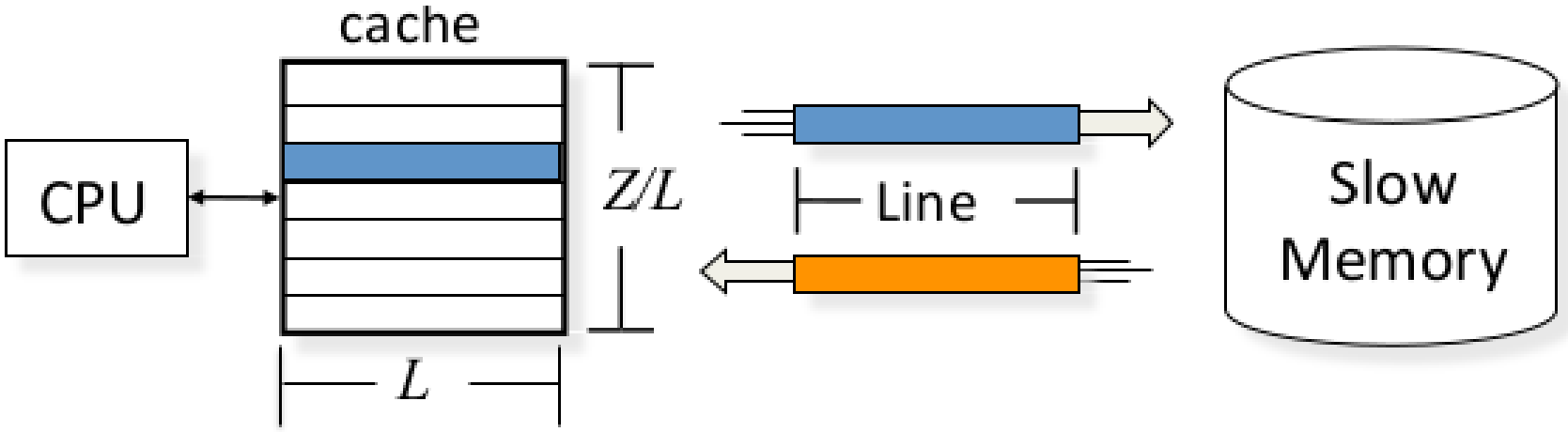
- The ideal cache is **fully associative**: cache lines can be stored anywhere in the cache.
- The ideal cache uses the **optimal off-line strategy of replacing** the cache line whose next access is furthest in the future, and thus it exploits temporal locality perfectly.

The ideal cache model (4/5)



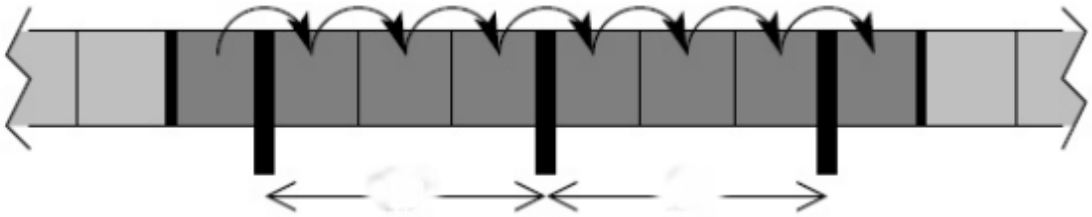
- For an algorithm with an input of size n, the ideal-cache model uses two complexity measures:
  - the **work complexity** $W(n)$, which is its conventional running time in a RAM model.
  - the **cache complexity** $Q(n; Z, L)$, the number of cache misses it incurs (as a function of the size Z and line length L of the ideal cache).
  - When Z and L are clear from context, we simply write $Q(n)$ instead of $Q(n; Z, L)$.

The ideal cache model (5/5)



- An algorithm is said to be **cache aware** if its behavior (and thus performances) can be tuned (and thus depend on) on the particular cache size and line length of the targeted machine.
- Otherwise the algorithm is **cache oblivious**.
- Cache oblivious naturally performs well on hierarchical memories.

Scanning



- Scanning n words stored in a contiguous segment of memory with cache-line size L costs at most $\lceil n/L \rceil + 1$ cache misses.
- If this vector of n words is aligned in memory, then this estimate is simply $\lceil n/L \rceil$.

Proof.

- Let (q, r) be the quotient and remainder in the integer division of n by L.
- Let u (resp. w) be the total number of words stored in cache-lines fully (not fully) used by those n consecutive words. Thus, we have $n = u + w$. Three cases arise.
  1. if $w = 0$ then $(q, r) = (\lfloor n/L \rfloor, 0)$ and the scanning costs exactly q; thus the conclusion is clear since $\lceil n/L \rceil = \lfloor n/L \rfloor$ in this case.
  2. if $0 < w < L$ then $(q, r) = (\lfloor n/L \rfloor, w)$ and the scanning cost is at most $q + 2$; the conclusion is clear since $\lceil n/L \rceil = \lfloor n/L \rfloor + 1$ in this case.
  3. if $L \leq w < 2L$ then $(q, r) = (\lfloor n/L \rfloor, w - L)$ and the scanning cost is at most $q + 1$; the conclusion is clear again.

Adding vectors

- Consider $m \geq 2$ vectors $V_1, \ldots, V_m$ of size $n \geq 1$ aligned in memory.
- Consider $m-1$ scalars $\alpha_1, \ldots, \alpha_{m-1}$, stored in a contiguous segment of memory in $m-1$ words.
- Assume that the ideal cache has at least $\lceil m/L \rceil + 4$ cache-lines.
- Then, computing the linear combination $\alpha_1 V_1 + \cdots + \alpha_{m-1} V_{m-1}$ and writing it to V_m can be done in no more cache misses than those required for scanning $V_1, \ldots, V_m, \alpha_1, \ldots, \alpha_{m-1}$,
- thus, within $m\lceil n/L \rceil + \lceil m/L \rceil + 1$ cache misses.

Proof.

- We first load $\alpha_1, \ldots, \alpha_{m-1}$ into the cache, thus using at most $\lceil m/L \rceil + 1$ cache-lines.
- In the pseudo-code below, vector indexing starts at 0.
  1. For b with $0 \leq b \leq \lfloor n/L \rfloor$, for each j with $1 \leq j < m$, for each i with $0 \leq i < L$ do:
     1. $k := b * L + i$,
     2. if $k < n$ then $V_m[k] := V_m[k] + \alpha_j V_j[k]$
- Use the optimal replacement policy and the fact that vectors are aligned in memory

Counting sort: the algorithm

```
allocate an array Count[0..k]; initialize each array cell to zero
for each input item x:
    Count[key(x)] = Count[key(x)] + 1
total = 0
for i = 0, 1, ... k:
    c = Count[i]
    Count[i] = total
    total = total + c
allocate an output array Output[0..n-1]
for each input item x:
    store x in Output[Count[key(x)]]
    Count[key(x)] = Count[key(x)] + 1
return Output
```

- *Counting sort* takes as input a collection of n items, each of which known by a key in the range $0 \cdots k$.
- The algorithm computes a *histogram* of the number of times each key occurs.
- Then performs a *prefix sum* to compute positions in the output.

Counting sort: poor spatial locality

```
allocate an array Count[0..k]; initialize each array cell to zero
for each input item x:
    Count[key(x)] = Count[key(x)] + 1
total = 0
for i = 0, 1, ... k:
    c = Count[i]
    Count[i] = total
    total = total + c
allocate an output array Output[0..n-1]
for each input item x:
    store x in Output[Count[key(x)]]
    Count[key(x)] = Count[key(x)] + 1
return Output
```

- For n large enough: $Q(n; Z, L) = 3n + 3n/L + 2k/L$ cache misses (worst case).
- The possibly random distribution of the input values creates possibly many non-cold misses, see counting_sort.pdf for an animation.

Counting sort: improved by a *blocking strategy*

```
alloacate an array bucketsize[0..m-1]; initialize each array cell to zero
for each input item x:
    bucketsize[floor(key(x) m/(k+1))] := bucketsize[floor(key(x) m/(k+1))] + 1
total = 0
for i = 0, 1, ... m-1:
    c = bucketsize[i]
    bucketsize[i] = total
    total = total + c
alloacate an array bucketedinput[0..n-1];
for each input item x:
    q := floor(key(x) m/(k+1))
    bucketedinput[bucketsize[q] ] := key(x)
    bucketsize[q] := bucketsize[q] + 1
return bucketedinput
```

- Split the input value range into m buckets (given by well-chosen pivot values) so that counting sort can be applied in succession to several smaller input arrays, with smaller value ranges, incurring cold misses only, see counting_sort_bucket.pdf for an animation.
- This yields $Q(n; Z, L) = 9n/L + 3m/L + m + 2k/L$ (assuming $m < Z/(1 + L)$) improving on $3n + 3n/L + 2k/L$.

Counting sort: experimentation

- Experimentation on an *Intel(R) Core(TM) i7 CPU @ 2.93GHz*. It has L2 cache of 8MB.
- CPU times in seconds for both classical and cache-friendly counting sort algorithm.
- The keys are random machine integers in the range $[0, n]$.

n	classical counting sort	cache-friendly counting sort (bucketing + sorting)
100000000	13.74	4.66 (= 3.04 + 1.62)
200000000	30.20	9.93 (= 6.16 + 3.77)
300000000	50.19	16.02 (= 9.32 + 6.70)
400000000	71.55	22.13 (= 12.50 +9.63)
500000000	94.32	28.37 (= 15.71 + 12.66)
600000000	116.74	34.61 (= 18.95 + 15.66)

Cache-friendly counting sort: extension to sample sort

1. Split the input array into $\sqrt{n}$ contiguous subarrays of size $\sqrt{n}$ and sort those subarrays recursively.
2. Choose $m := \sqrt{n} - 1$ "good" pivot values $p_1 \leq p_2 \leq \cdots \leq p_m$.
3. Distribute subarrays into buckets $B_1, \ldots, B_{m+1}$ according to pivots. Bucket B_i has size $n_i \simeq \sqrt{n}$, expectedly.
4. Recursively sort the buckets
5. Copy-concatenate the buckets back to the input array.

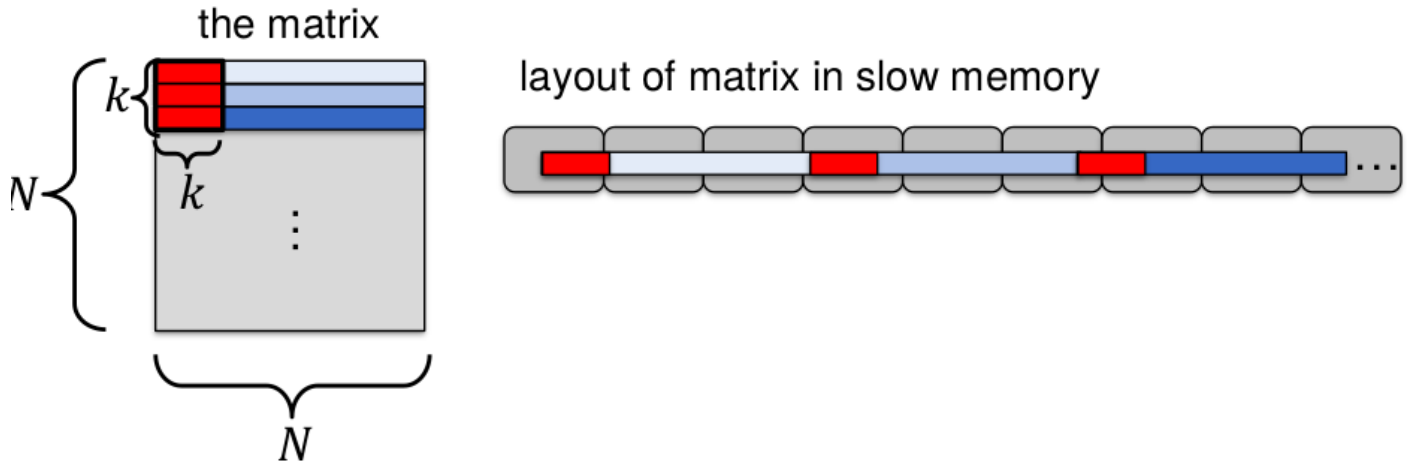
Cache complexity analysis of Sample sort

- Step 1 costs $\sqrt{n}Q(\sqrt{n})$, Step 4 (expectedly) costs $\sqrt{n}Q(\sqrt{n})$ also and Steps 2, 3, 5 cost $\Theta(n/L)$. Thus, we have:

$$Q(n) = \begin{cases} n/L & \text{if} \quad n < Z \quad \text{(base case)} \\ 2\sqrt{n}Q(\sqrt{n}) + \Theta(n/L) & \text{if} \quad n \geq Z \quad \text{(recurrence)} \end{cases}$$

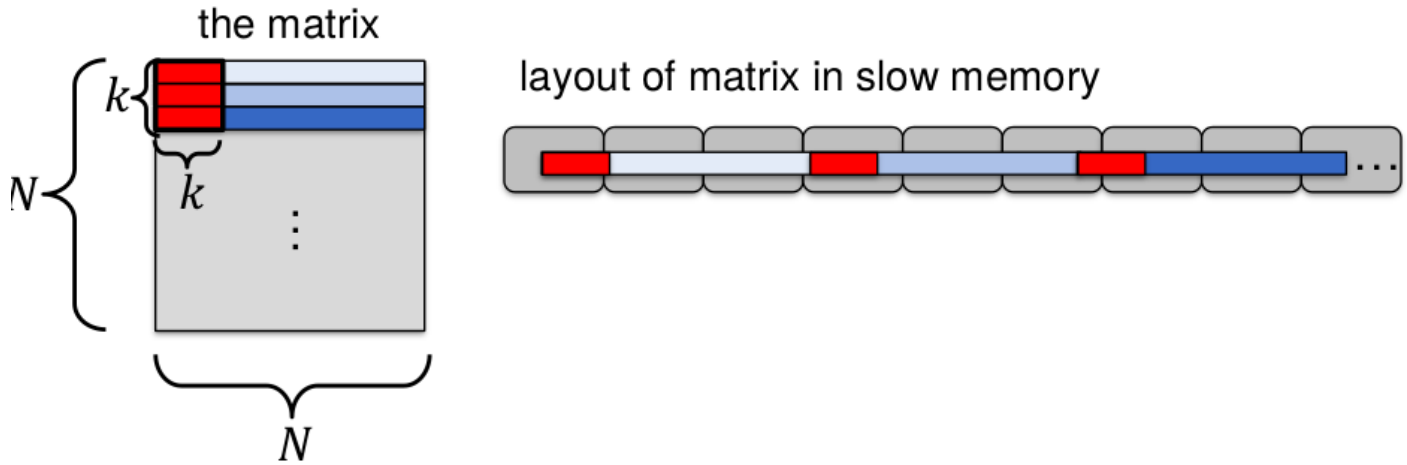
- This yields $Q(n) \in \Theta(\frac{n}{L}\log_Z(n))$.

Transposition of a matrix



- Assume that multi-dimensional arrays (and in particular dense rectangular matrices) are stored in memory using a row-major layout.
- Assume that each array coefficient is stored on a single word.
- Therefore, reading a $k \times k$ block may incur $k(\lceil k/L \rceil + 1)$ caches misses.
- In this exercise sheet, determine the cache complexity of the proposed algorithms for transposing a square matrix of order n. Assume n large (say $n > Z$) and n is a power of 2.

Transposition of a matrix



- Assume that multi-dimensional arrays (and in particular dense rectangular matrices) are stored in memory using a row-major layout.
- Assume that each array coefficient is stored on a single word.
- Therefore, reading a $k \times k$ block may incur $k(\lceil k/L \rceil + 1)$ caches misses.
- In this exercise sheet, determine the cache complexity of the proposed algorithms for transposing a square matrix of order n. Assume n large (say $n > Z$) and n is a power of 2.
- Algo 1: $\Theta(n^2)$. Algo 2: $\Theta(\log_2(\frac{n}{Z})\,\frac{n^2}{L})$. Algo 3: $\Theta(n^2/L)$. Proofs and precise estimates below. ⏩ skip slide

Matrix transposition: various algorithms

- **Matrix transposition problem:** Given an $m \times n$ matrix A stored in a row-major layout, compute and store A^T into an $n \times m$ matrix B also stored in a row-major layout.
- We shall describe a recursive cache-oblivious algorithm which uses $\Theta(mn)$ work and incurs $\Theta(1 + mn/L)$ cache misses, which is optimal.
- The straightforward algorithm employing doubly nested loops incurs $\Theta(mn)$ cache misses on one of the matrices when $m \gg Z/L$ and $n \gg Z/L$.
- We shall start with an apparently good algorithm and use complexity analysis to show that it is even worse than the straightforward algorithm.

Matrix transposition: a first divide-and-conquer (1/4)

- For simplicity, assume that our input matrix A is square of order n and that n is a power of 2, say $n = 2^k$.
- We divide A into four square quadrants of order $n/2$ and we have

$$A = \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \Rightarrow {}^tA = \begin{pmatrix} {}^tA_{1,1} & {}^tA_{2,1} \\ {}^tA_{1,2} & {}^tA_{2,2} \end{pmatrix}.$$

- This observation yields an "in-place" algorithm:
  1. If $n = 1$ then return A.
  2. If $n > 1$ then
     1. recursively compute ${}^tA_{1,1}, {}^tA_{2,1}, {}^tA_{1,2}, {}^tA_{2,2}$ in place as
     $$\begin{pmatrix} {}^tA_{1,1} & {}^tA_{1,2} \\ {}^tA_{2,1} & {}^tA_{2,2} \end{pmatrix}$$
     2. exchange ${}^tA_{1,2}$ and ${}^tA_{2,1}$.
- What is the number $M(n)$ of memory accesses to A, performed by this algorithm on an input matrix A of order n?

Matrix transposition: a first divide-and-conquer (2/4)

- $M(n)$ satisfies the following recurrence relation

$$M(n) = \begin{cases} 0 & \text{if} \quad n = 1 \\ 4M(n/2) + 2(n/2)^2 & \text{if} \quad n > 1. \end{cases}$$

- Unfolding the tree of recursive calls or using the *Master's Theorem*, one obtains:

$$M(n) = 2(n/2)^2 \ \log_2(n).$$

- This is worse than the straightforward algorithm (which employs doubly nested loops). Indeed, for this latter, we have $M(n) = n^2 - n$. Explain why!
- Despite of this negative result, we shall analyze the cache complexity of this first divide-and-conquer algorithm. Indeed, it provides us with an easy training exercise
- We shall study later a second and efficiency-optimal divide-and-conquer algorithm, whose cache complexity analysis is more involved.

Matrix transposition: a first divide-and-conquer (3/4)

- We shall determine $Q(n)$ the number of cache misses incurred by our first divide-and-conquer algorithm on a (Z, L)-ideal cache machine.
- For n small enough, the entire input matrix or the entire block (input of some recursive call) fits in cache and incurs only the cost of a scanning. Because of possible misalignment, that is, $n(\lceil n/L \rceil + 1)$.
- **Important:** For simplicity, some authors write n/L instead of $\lceil n/L \rceil$. This can be dangerous.
- **However:** these simplifications are fine for asymptotic estimates, keeping in mind that n/L is a rational number satisfying

$$n/L - 1 \leq \lfloor n/L \rfloor \leq n/L \leq \lceil n/L \rceil \leq n/L + 1.$$

  Thus, for a fixed L, the functions $\lfloor n/L \rfloor$, n/L and $\lceil n/L \rceil$ are asymptotically of the same order of magnitude.
- We need to translate "for n small enough" into a formula. We claim that there exists a real constant $\alpha > 0$ s.t. for all n and Z we have

$$n^2 < \alpha Z \quad \Rightarrow \quad Q(n) \leq n^2/L + n.$$

Matrix transposition: a first divide-and-conquer (4/4)

- $Q(n)$ satisfies the following recurrence relation

$$Q(n) = \begin{cases} n^2/L + n & \text{if} \quad n^2 < \alpha Z \quad \text{(base case)} \\ 4Q(n/2) + \frac{n^2}{2L} + n & \text{if} \quad n^2 \geq \alpha Z \quad \text{(recurrence)} \end{cases}$$

- Indeed, **exchanging 2 blocks** amount to $2((n/2)^2/L + n/2)$ accesses.
- Unfolding the recurrence relation k times (more details in class) yields

$$Q(n) = 4^k\, Q(\frac{n}{2^k}) + k\,\frac{n^2}{2L} + (2^k - 1)n.$$

- The minimum k for reaching the base case satisfies $\frac{n^2}{4^k} = \alpha Z$, that is, $4^k = \frac{n^2}{\alpha Z}$, that is, $k = \log_4(\frac{n^2}{\alpha Z})$. This implies $2^k = \frac{n}{\sqrt{\alpha Z}}$ and thus

$$\begin{aligned} Q(n) &\leq \frac{n^2}{\alpha Z}\,(\alpha Z/L + \sqrt{\alpha Z}) + \log_4(\tfrac{n^2}{\alpha Z})\,\tfrac{n^2}{2L} + \tfrac{n}{\sqrt{\alpha Z}}\,n \\ &\leq n^2/L + 2\tfrac{n^2}{\sqrt{\alpha Z}} + \log_4(\tfrac{n^2}{\alpha Z})\,\tfrac{n^2}{2L}. \end{aligned}$$

A matrix transposition cache-oblivious algorithm (1/2)

- If $n \geq m$, the Rec-Transpose algorithm partitions

$$A = (A_1 \ A_2) \ , \quad B = \begin{pmatrix} B_1 \\ B_2 \end{pmatrix}$$

and recursively executes Rec-Transpose(A_1, B_1) and Rec-Transpose(A_2, B_2).

- If $m > n$, the Rec-Transpose algorithm partitions

$$A = \begin{pmatrix} A_1 \\ A_2 \end{pmatrix} \ , \quad B = (B_1 \ B_2)$$

and recursively executes Rec-Transpose(A_1, B_1) and Rec-Transpose(A_2, B_2).

A matrix transposition cache-oblivious algorithm (2/2)

- Recall that the matrices are stored in row-major layout.
- Let α be a constant sufficiently small such that the following two conditions hold:
  - (i) two sub-matrices of size $m \times n$ and $n \times m$, where $\max\{m, n\} \leq \alpha L$, fit in cache
  - (ii) even if each row starts at a different cache line.
- We distinguish three cases for the input matrix A:
  - ↳ Case I: $\max\{m, n\} \leq \alpha L$.
  - ↳ Case II: $m \leq \alpha L < n$ or $n \leq \alpha L < m$.
  - ↳ Case III: $m, n > \alpha L$.

Case I: $\max\{m, n\} \leq \alpha L$.

- Both matrices fit in $O(1) + 2mn/L$ lines.
- From the choice of α, the number of lines required for the entire computation is at most Z/L.
- Thus, no cache lines need to be evicted during the computation. Hence, it feels like we are simply scanning A and B.
- Therefore $Q(m, n) \in O(1 + mn/L)$.

Case II: $m \leq \alpha L < n$ or $n \leq \alpha L < m$.

- Consider $n \leq \alpha L < m$. The Rec-Transpose algorithm divides the greater dimension m by 2 and recurses.
- At some point in the recursion, we have $\alpha L/2 \leq m \leq \alpha L$ and the whole computation fits in cache. At this point:
  - ↳ the input array resides in contiguous locations, requiring at most $\Theta(1 + nm/L)$ cache misses
  - ↳ the output array consists of nm elements in n rows, where in the **worst case** every row starts at a different cache line, leading to at most $\Theta(n + nm/L)$ cache misses.
- Since $m/L \in [\alpha/2, \alpha]$, the **total** cache complexity for this base case is $\Theta(1 + n)$, yielding the recurrence (where the resulting $Q(m, n)$ is a **worst case estimate**)

$$Q(m,n) = \begin{cases} \Theta(1+n) & \text{if } m \in [\alpha L/2, \alpha L] \ , \\ 2Q(m/2,n) + O(1) & \text{otherwise ;} \end{cases}$$

whose solution satisfies $Q(m, n) = \Theta(1 + mn/L)$.

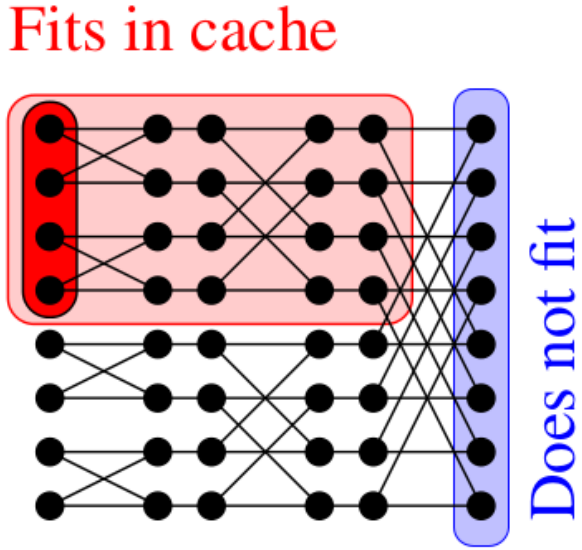
Case III: $m, n > \alpha L$.

- As in Case II, at some point in the recursion both n and m fall into the range $[\alpha L/2, \alpha L]$.
- The whole problem fits into cache and can be solved with at most $\Theta(m + n + mn/L)$ cache misses.
- The **worst case cache miss estimate** satisfies the recurrence

$$Q(m,n) = \begin{cases} \Theta(m+n+mn/L) & \text{if } m,n \in [\alpha L/2, \alpha L] \ , \\ 2Q(m/2,n) + O(1) & \text{if } m \geq n \ , \\ 2Q(m,n/2) + O(1) & \text{otherwise;} \end{cases}$$

  whose solution is $Q(m,n) = \Theta(1 + mn/L)$.
- **Therefore, the Rec-Transpose algorithm has optimal cache complexity.**
- Indeed, for an $m \times n$ matrix, the algorithm must write to mn distinct elements, which occupy at least $\lceil mn/L \rceil$ cache lines.

1-D FFTs: classical cache friendly algorithm



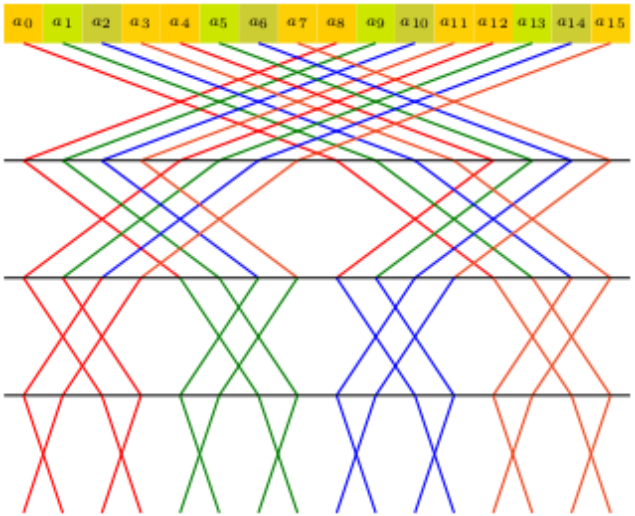
FFT$([a_0, a_1, \cdots, a_{n-1}], \omega)$

```
if n ≤ HTHRESHOLD then
    ArrayBitReversal(a_0, a_1, ..., a_{n-1})
    return FFT_iterative_in_cache([a_0, a_1, ..., a_{n-1}], ω)
end if
Shuffle(a_0, a_1, ..., a_{n-1})
[a_0, a_1, ..., a_{n/2-1}] = FFT([a_0, a_1, ..., a_{n/2-1}], ω^2)
[a_{n/2}, a_{n/2+1}, ..., a_{n-1}] = FFT([a_{n/2}, a_{n/2+1}, ..., a_{n-1}], ω^2)
return [a_0 + a_{n/2}, a_1 + ω · a_{n/2+1}, ..., a_{n/2-1} − ω^{n/2-1} · 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$((\alpha_0\alpha_1...\alpha_{N-1}), \omega, N = J \cdot K, \Omega = \omega^{N/K})$
  for $0 \leq k < K-1$ **do** ▷ Inner transforms
    for $0 \leq k' < J-1$ **do**
      $\gamma[k][k'] = \alpha_{k'K+k}$
    end for
    $c[k] = FFT(\gamma[k], \omega^K, J, \Omega)$
  end for
  for $0 \leq j < J-1$ **do** ▷ Outer transforms
    for $0 \leq k < K-1$ **do**
      $\delta[j][k] = c[k][j] * \omega^{jk}$ ▷ Computation of coefficients
    end for
    $d[j] = FFT(\delta[j], \omega^J, K, \Omega)$
    for $0 \leq j' < K-1$ **do**
      $\beta_{j'J+j} = d[j][j']$
    end for
  end for
  return $b = (\beta_0, ..., \beta_{N-1})$
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 $\boxed{O(n/L(\log_2(n)/\log_2(Z)))}$ which is optimal.

1-D FFTs in BPAS

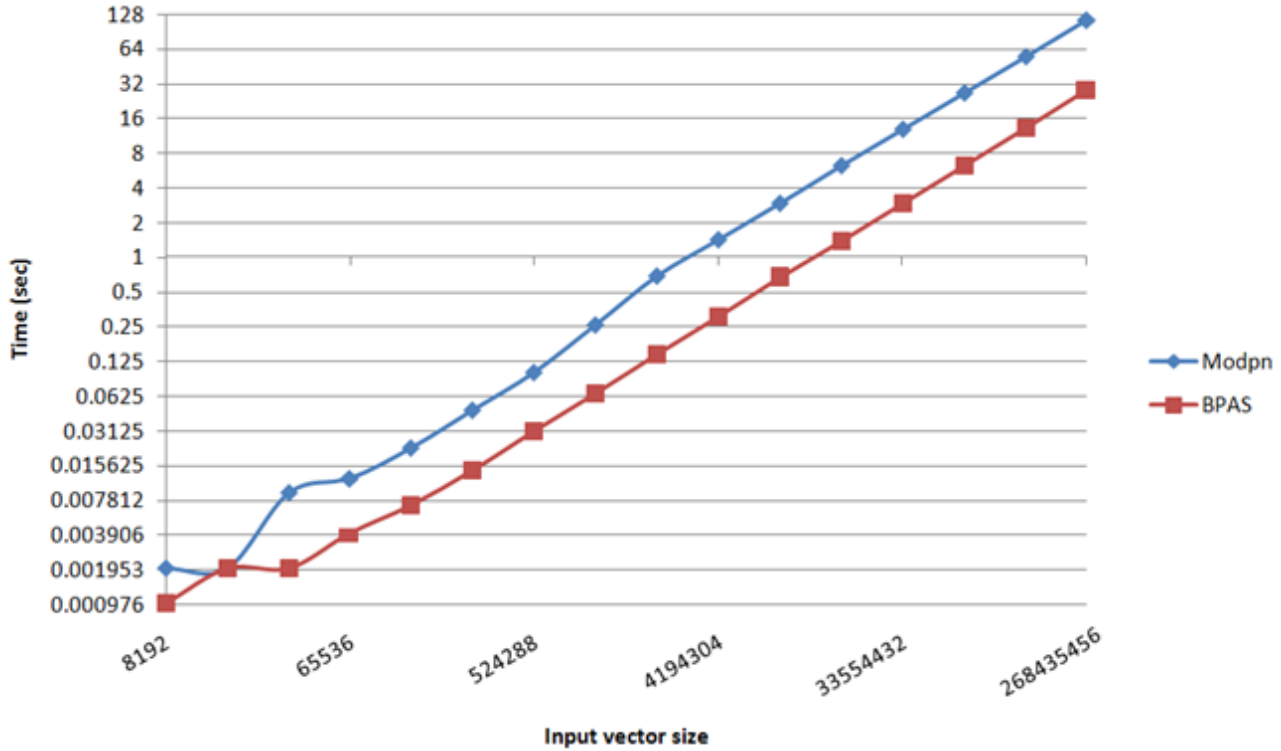


Figure: 1-D modular FFTs: Modpn (**serial**) vs BPAS (**serial**).

- In addition to the above optimal blocking strategy, instruction level parallelism (ILP) is carefully considered: vectorized instructions are explicitly used and instruction pipeline usage is highly optimized.
- BPAS 1-D FFT code automatically generated by configurable Python scripts.

Notations

- For two positive integers a, b, we write a/b instead of $\lfloor a/b \rfloor$.
- Let $\mathbf{k}$ be a finite field so that each element of $\mathbf{k}$ can be stored in a machine word.
- We assume that each polynomial P of $\mathbf{k}[x]$ is stored in a vector V_P of $d+1$ words, aligned in memory, where d is the degree of P, and so that the coefficient of x^i in P is stored in the $(d-i)$-th slot of V_P, for $0 \leq i \leq d$.
- Let $A = \sum_{i=0}^{i=m-1} a_i x^i$ and $B = \sum_{i=0}^{i=n-1} b_i x^i$ be in $\mathbf{k}[x]$ with $m \geq n$.

Plain polynomial multiplication

- Recall $A = \sum_{i=0}^{i=m-1} a_i x^i$ and $B = \sum_{i=0}^{i=n-1} b_i x^i$ in $\mathbf{k}[x]$ with $m \geq n$.
- Counting cache misses, the plain multiplication incurs
$$O((m/L+1)n)$$
- This estimate can be substantially improved by performing the plain multiplication in a divide-and-conquer manner, following the scheme of the matrix multiplication algorithm of [20].
- This recursive algorithm is presented in [14]; it runs within
$$O(mn/(ZL))$$
- It leads to clear gains on Graphics Processing Units (GPUs) due to the fine grained control of hardware resources.
- However, with a CPU implementation, for relatively small n and m, any plain multiplication algorithm is outperformed by an FFT-based polynomial multiplication.

Plain polynomial division

- Let $Q = \mathrm{quo}(A, B)$ and $R = \mathrm{rem}(A, B)$
- The schoolbook plain Euclidean division, using a two-loop nest, computes Q and R, within

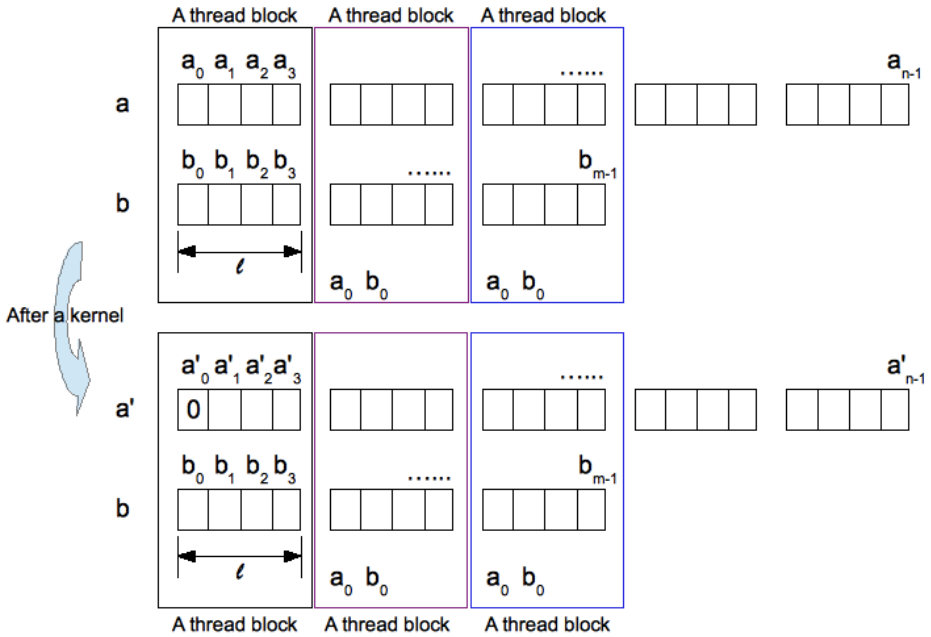
$$O((m - n + 1)(n/L + 3))$$

- By means of a *blocking strategy*, this estimate can be improved to

$$O(((2Z + 9L)(m - n + 1)(n/(Z^2 L) + 1))$$

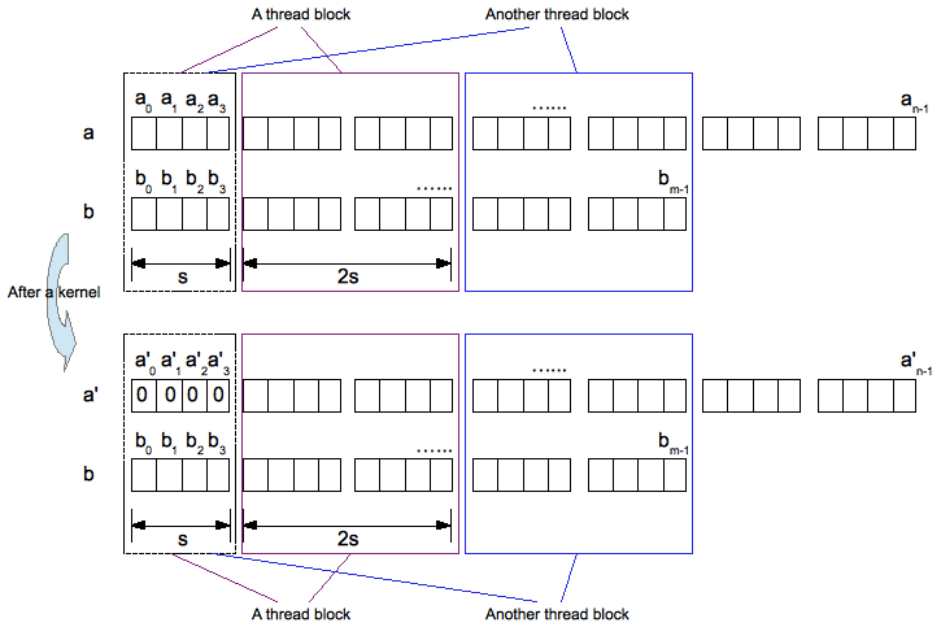
  See [24, 25].
- This strategy is inspired by the Half-Gcd algorithm, see Lemma 11.1 in Chapter 11 of [23] and the next slide.

Plain polynomial division on the GPU: naive approach



- A naive approach consists in doing one division step (that is, a combination of two vectors) by one kernel.
- That is, each thread (in thread-block) reads two coefficients (one from A and one form B), combines them and writes the result back to A
- This causes many memory transactions (equivalent to cache misses) between the global memory of the GPU and the local memories of the streaming multi-processors,

Plain polynomial division on the GPU: optimized approach



- Each thread-block loads the s leading coefficients of both A and B, and a segment of $2s$ consecutive coefficients from both A and B.
- So that each thread-block can independently work on (possibly) s division steps.
- The authors of [24, 25] show how to determine s in a way minimizing cache misses, which leads to the above complexity estimate.

Outline

Optimizing cache complexity

- For all results discussed above, the key towards cache-oblivious or cache optimal algorithms is a *blocking strategy*.
- This blocking strategy may take different forms: from the *buckets* of counting sort to *matrix blocks* in dense linear algebra.
- While blocking strategies naturally lead to recursive algorithms, the implementation of the latter are often made in the form of *for-loop nests*, which is more suitable for compiler optimization.

In the context of multi/many-core processors

- When multiple threads are cooperating, cores executing those threads share a common physical address space, causing a *cache coherence problem*.
- Two well-known consequences of this problem are *true sharing* and *false sharing*:
  - ↳ In the former, two cores are accessing the same memory address, with at least one of them for writing.
  - ↳ In the latter, two cores are accessing the same cache-line (but not the same memory address), with at least one of them for writing.
- Other parallel overheads should be watched like *memory contention*, *scheduling and synchronization costs*, which are very hard to take into account in complexity analysis [24, 28, 30].
- Nevertheless, on multicore processors, a good practical indication about what to expect in $W(n)/Q(n; Z, L)$, in addition to the more standard ration $T_1(n)/T_\infty(n)$.

Data reshaping

- Other performance degradation can come from *for-loop overheads*.
- If a loop has a few iterations, then overheads due to *branch misprediction* can have an impact, since a misprediction delay can be between 10 and 35 clock cycles [17].
- Trying to avoid those issues with for-loop nests has several advantages, including reducing overheads due to loop counter manipulation.

- In the context of dense multivariate polynomials over finite fields, this idea was studied in [31, 38] for *multi-threaded multi-dimensional FFTs (and TFTs)* and their application to polynomial multiplication.
- The authors systematically reduce multivariate polynomials to **balanced bivariate polynomials**. *Balanced* here means that partial degrees are equal or as close as possible.
- A theoretical study, supported by extensive experimentation, shows that this approach *minimizes cache misses* and *maximizes parallelism.*

Outline

Overview

- As we saw, a blocking strategy extracts opportunities for improving data and spatial locality
- Similarly, a *parallel programming pattern* is a meta-algorithm or algorithmic structure used to organize code for efficient parallel computation.
- Well-known parallel patterns are *fork-join*, *parallel map*, *stencil*, and *pipeline*.
- See the book [32] of McCool, Reinders and Robison.
- We shall illustrate those patterns with two types of computations:
  1. polynomial system solving via incremental triangular decomposition (Maple's Triangularize command) [2, 4, 12],
  2. power series arithmetic via lazy evaluation [7, 8, 10].

Preliminary observations

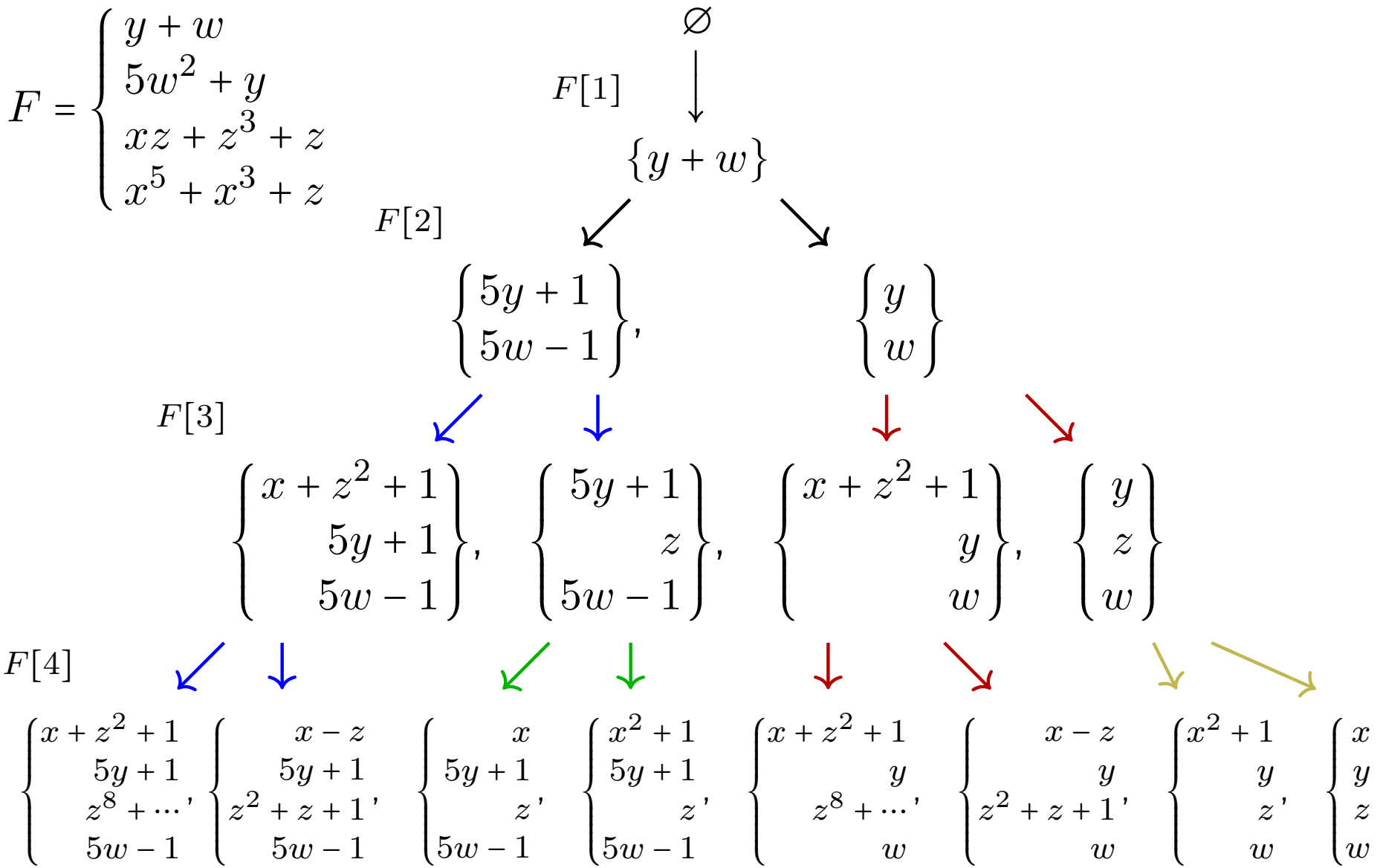
- Different parallel patterns may apply to the same algorithm:
  - ↳ the construction of Pascals' Triangle can be done in a divide-and-conquer way (dividing the triangle into 2 triangles and a square), or
  - ↳ as a stencil pattern (in conjunction with a blocking strategy) to the formula, see [13]
- Keep in mind the targeted hardware:
  - ↳ the parallelization of vector addition on a CPU multicore processor is limited (and yields low speedup factors) because of various overheads (memory contention, etc.) while
  - ↳ it can implemented in a SIMD fashion on GPU with good speedup factors
- Keep in mind thread scheduling and synchronization costs
  - ↳ Doing $A\vec{v}$, for a triangular dense square matrix A, by computing concurrently all products of a row of A by $\vec{v}$ would keep cores idle for (at least) half of the execution time, while
  - ↳ the fork-join pattern (applied to a divide-and-conquer approach) would make better use of resources.

Regular vs irregular tasks

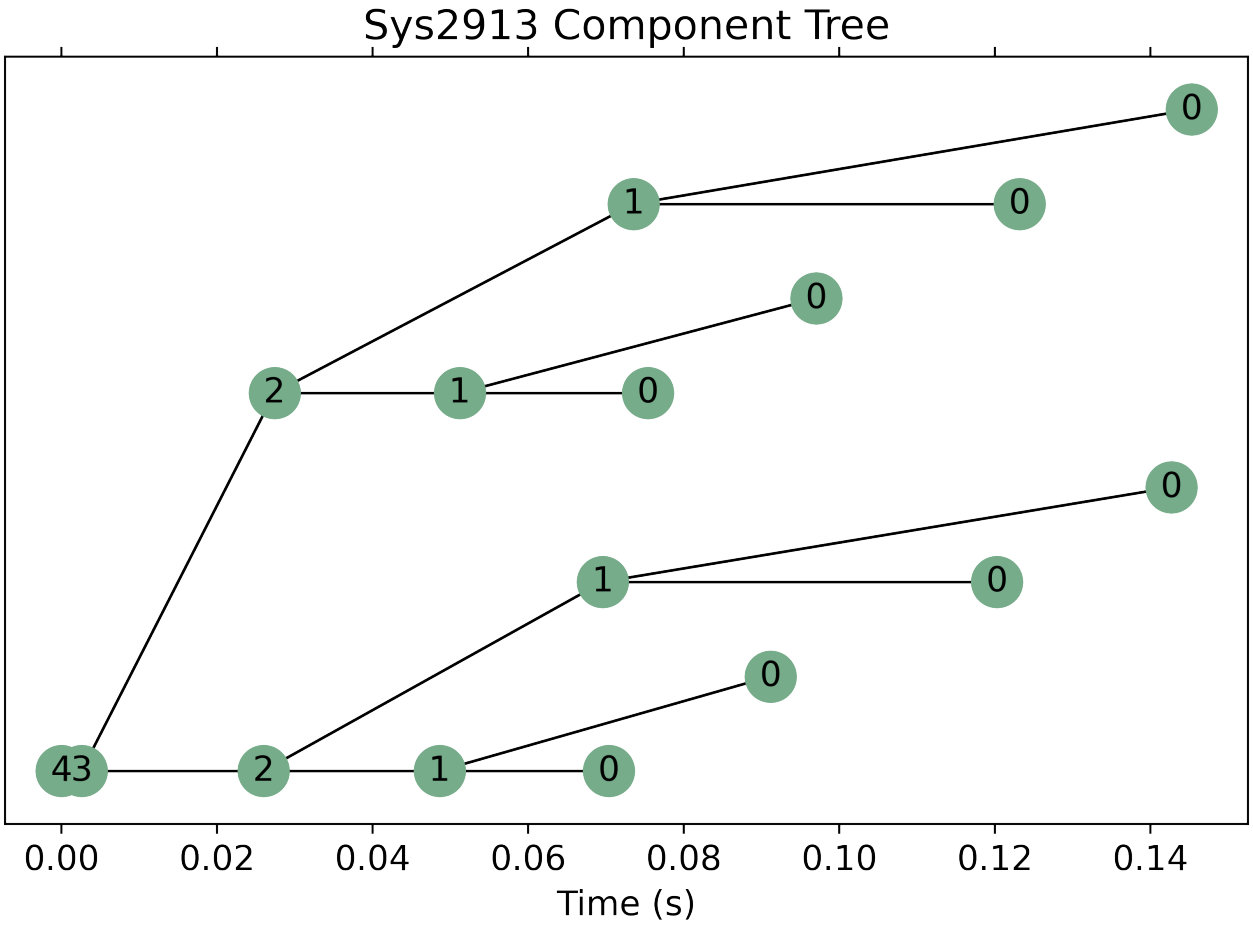
- In our previous three examples, the work to be executed in parallel can be decomposed evenly and easily with predictable dependencies and predictable computing resource needs; one then uses the term *regular parallelism*.
- The term *irregular parallelism* refers to the opposite case, when the decomposition of work into tasks creates unbalanced work, dissimilar tasks, unpredictable dependencies or unpredictable computing resource needs.
- Decomposing polynomial systems (unless they have properties like *strongly unmixed* or *regular sequence*) fits in that second category:
  - ↳ Some systems never split
  - ↳ Some split only at the final step, leaving very little concurrency
  - ↳ Some split into one "main" component and several degenerative cases

Incremental solving: a toy example

$$F = \begin{cases} y + w \\ 5w^2 + y \\ xz + z^3 + z \\ x^5 + x^3 + z \end{cases}$$



Incremental solving: tracing a non-toy example



- more parallelism exposed as more components found
- yet, work unbalanced between branches
- mechanism needed for dynamic parallelism: “workpile” or “task pool”

Outline

Regular chains, notations

Let $\mathbf{k}$ be a perfect field, and $\mathbf{k}[\underline{X}]$ have ordered vars. $\underline{X} = X_1 < \cdots < X_n$

A triangular set T is a regular chain if either T is empty, or T_v^- is a regular chain and h is regular modulo $\mathrm{sat}(T_v^-)$

$$T = \left\{ \begin{array}{l} T_v = h\, v^d + \mathrm{tail}(T_v) \\ \\ T_v^- = \left\{ \begin{array}{l} \\ \\ \end{array} \right. \end{array} \right\} \subset \mathbf{k}[\underline{X}]$$

Example:

$$T = \left\{ \begin{array}{r} (2y + ba)x - by + a^2 \\ 2y^2 - by - a^2 \\ a + b \end{array} \right\} \subset \mathbb{Q}[b < a < y < x]$$

Saturated ideal of a regular chain:

- $\mathrm{sat}(T) = (\mathrm{sat}(T_v^-) + T_v) : h^\infty$
- $\mathrm{sat}(\varnothing) = \langle 0 \rangle$

Quasi-component of a regular chain:

- $W(T) := V(T) \setminus V(h_T)$, $h_T := \prod_{p \in T} h_p$
- $\overline{W(T)} = V(\mathrm{sat}(T))$

Triangular decomposition algorithms

A **triangular decomposition** of an input system $F \subseteq \mathbf{k}[\underline{X}]$ is a set of regular chains $T_1, \ldots, T_e$ such that:

(a) $V(F) = \bigcup_{i=1}^{e} \overline{W(T_i)}$, in the sense of Kalkbrener, or

(b) $V(F) = \bigcup_{i=1}^{e} W(T_i)$, in the sense of Wu and Lazard

Triangular decomposition by incremental **intersection** has key subroutines:

Intersect. Given $p \in \mathbf{k}[\underline{X}]$, $T \subset \mathbf{k}[\underline{X}]$, compute $T_1, \ldots, T_e$ such that:
$V(p) \cap W(T) \subseteq \bigcup_{i=1}^{e} W(T_i) \subseteq V(p) \cap \overline{W(T)}$

Regularize: Given $p \in \mathbf{k}[\underline{X}]$, $T \subset \mathbf{k}[\underline{X}]$, compute $T_1, \ldots, T_e$ such that:
(i). $W(T) \subseteq \bigcup_{i=1}^{e} W(T_i) \subseteq \overline{W(T)}$, and
(ii). $p \in \mathrm{sat}(T_i)$ or p is regular modulo $\mathrm{sat}(T_i)$, for $i = 1, \ldots, e$

RegularGCD: Given $p \in \mathbf{k}[\underline{X}]$ with main variable v, $T = \{T_v\} \cup T_v^-$, find pairs (g_i, T_i) such that:
(i). $W(T_v^-) \subseteq \bigcup_{i=1}^{e} W(T_i) \subseteq \overline{W(T_v^-)}$, and
(ii). g_i is a *regular gcd* of p, T_v w.r.t. T_i

Finding splittings: GCDs and Regularize

Let $p \in \mathbf{k}[\underline{X}] \setminus \mathbf{k}$ with main variable v. Let $T = T_v^- \cup T_v$. All are square free.

A **regular GCD** g of p and T_v w.r.t. $\mathrm{sat}(T_v^-)$ has:

1. h_g is regular modulo $\mathrm{sat}(T_v^-)$
2. $g \in \langle p, T_v \rangle$ (every solution of p and T_v solves g as well)
3. if $\deg(g, v) > 0$, then g pseudo-divides p and T_v.

Let $q = \mathrm{pquo}(T_v, g)$. In Regularize, g says where p vanishes or is regular:

$$W(T) \subseteq W(T_v^- \cup g) \cup W(T_v^- \cup q) \cup (V(h_g) \cap W(T)) \subseteq \overline{W(T)}$$

In Intersect, splittings are found via recursive calls:

$$V(p) \cap W(T) \subseteq$$
$$W(T_v^- \cup g) \cup (V(p) \cap (V(h_g) \cap W(T)))$$
$$\subseteq V(p) \cap \overline{W(T)}$$

The foundation of splitting: regularity testing

To intersect a polynomial with an existing regular chain, it must have a regular initial, regularizing finds splittings via a **case discussion**

- either the initial is regular, or it is not regular

$$f = (y+1)x^2 - x$$

$$T = \begin{cases} y^2 - 1 = 0 \\ z - 1 = 0 \end{cases}$$

$y+1=0$:

$$T_1 = \begin{cases} y+1=0 \\ z-1=0 \end{cases} \xrightarrow{f = x} T_1 = \begin{cases} x = 0 \\ y+1=0 \\ z-1=0 \end{cases}$$

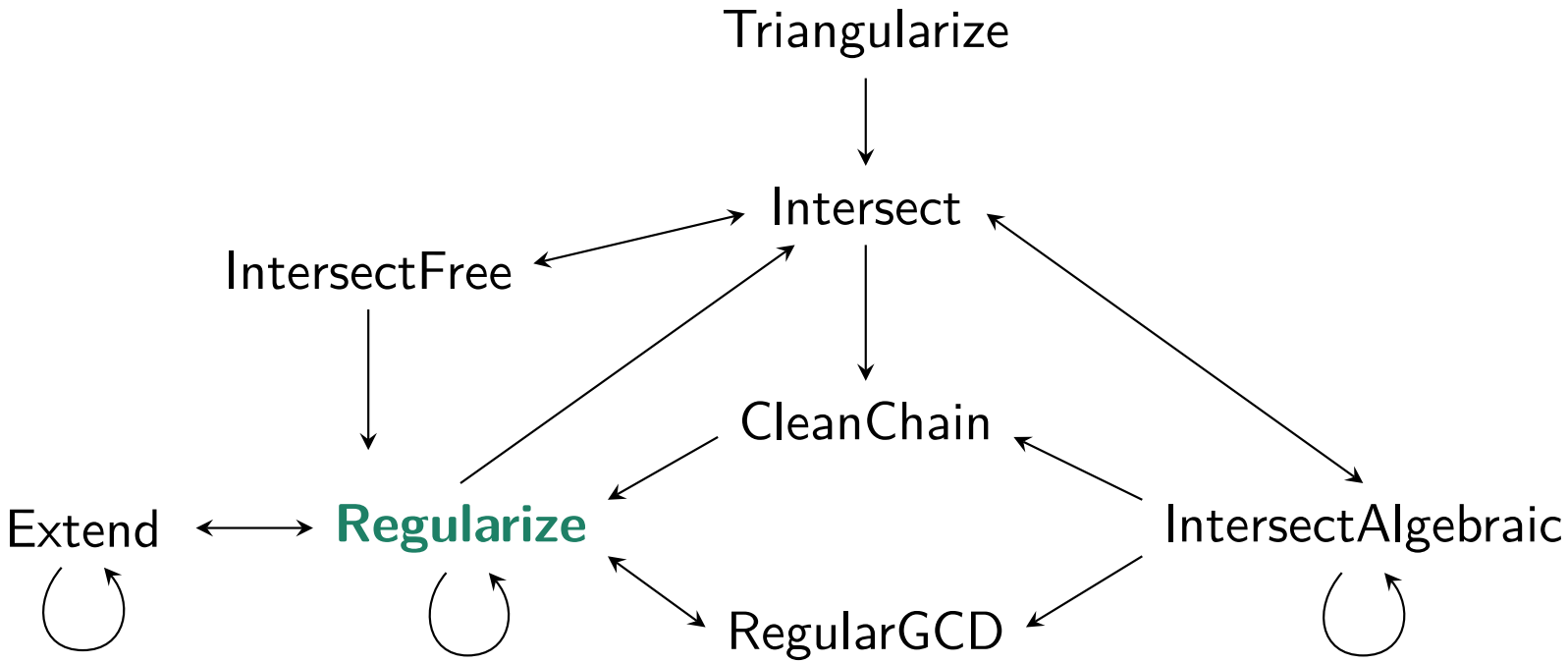
$y+1 \neq 0$:

$$T_2 = \begin{cases} y-1=0 \\ z-1=0 \end{cases} \xrightarrow{f = 2x^2 - x} T_2 = \begin{cases} 2x^2 - x = 0 \\ y-1=0 \\ z-1=0 \end{cases}$$

All roads lead to Regularize

The Triangularize algorithm iteratively calls intersect, then a network of mutually recursive functions do the heavy-lifting.

- In all cases, polynomials are forced to be regular and splittings are (possibly) found via **Regularize**



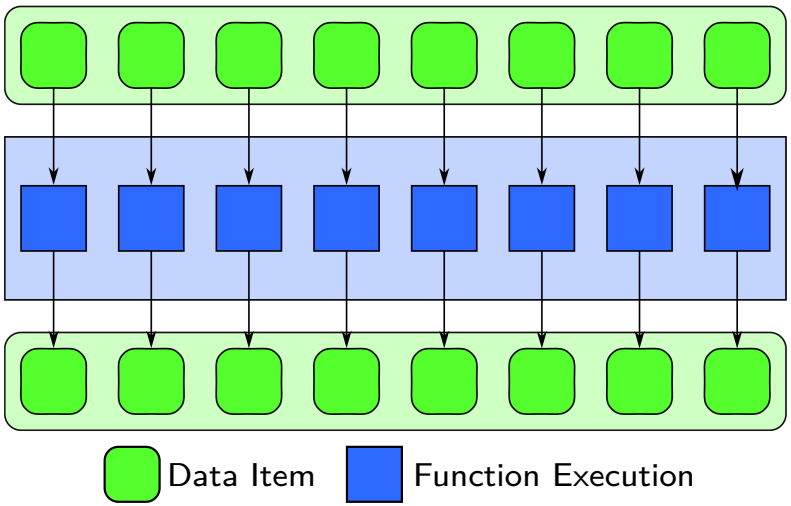
Outline

Parallel map and workpile

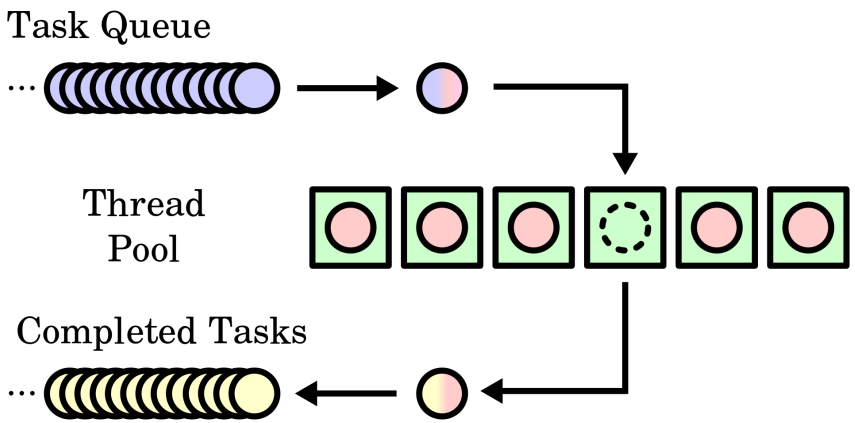
Map is the possibly the most well-known parallel programming pattern

- ↳ execute a function on each item in a collection concurrently
- ↳ with multiple Maps, tasks must execute in *lockstep*

Map Pattern [32]



Thread Pool (Wikipedia)



Workpile generalizes Map to a *queue of a tasks*, allowing tasks to add more tasks, thus enabling *load-balancing* as tasks start asynchronously

- ↳ one possible implementation of workpile is a **thread pool**

Triangularize: incremental triangular decomposition

Algorithm 1 Triangularize(F)

Input: a finite set $F \subseteq \mathbf{k}[\underline{X}]$
Output: regular chains $T_1, \ldots, T_e \subseteq \mathbf{k}[\underline{X}]$ encoding the solutions of $V(F)$

1: $\mathcal{T} := \{\emptyset\}$
2: **for** $p \in F$ **do**
3: $\quad \mathcal{T}' := \{\}$
4: $\quad$ for $T \in \mathcal{T}$ **Map** $\quad \triangleright$ map Intersect over the current components
5: $\quad\quad \mathcal{T}' := \mathcal{T}' \cup$ **Intersect**(p, T)
6: $\quad \mathcal{T} := \mathcal{T}'$
7: **return** RemoveRedundantComponents($\mathcal{T}$)

- **Coarse-grained parallelism**: each Intersect represents substantial work
- At each "level" there are $|\mathcal{T}|$ components with which to intersect, yielding $|\mathcal{T}|$ concurrent calls to intersect
- Performs a *breadth-first search*, with intersects occurring in lockstep

Triangularize: a task-based approach

Algorithm 2 TriangularizeByTasks(F)

Input: a finite set $F \subseteq \mathbf{k}[\underline{X}]$
Output: regular chains $T_1, \ldots, T_e \subseteq \mathbf{k}[\underline{X}]$ encoding the solutions of $V(F)$

1: *Tasks* $\leftarrow \{ (F, \varnothing) \}$; $\mathcal{T} \leftarrow \{\}$
2: **while** $|\mathit{Tasks}| > 0$ **do**
3: $(P, T) \leftarrow$ pop a task from *Tasks*
4: Choose a polynomial $p \in P$; $P' \leftarrow P \setminus \{p\}$
5: **for** T' in **Intersect**(p, T) **do**
6: **if** $|P'| = 0$ **then** $\mathcal{T} \leftarrow \mathcal{T} \cup \{T'\}$
7: **else** *Tasks* $\leftarrow$ *Tasks* $\cup \{(P', T')\}$
8: **return** RemoveRedundantComponents($\mathcal{T}$)

- *Tasks* is really a task scheduler augmented with a thread pool
- Tasks create more tasks, workers pop Tasks until none remain.
- Adaptive to load-balancing, no inter-task synchronization

Outline

Generators and pipelines

Generators

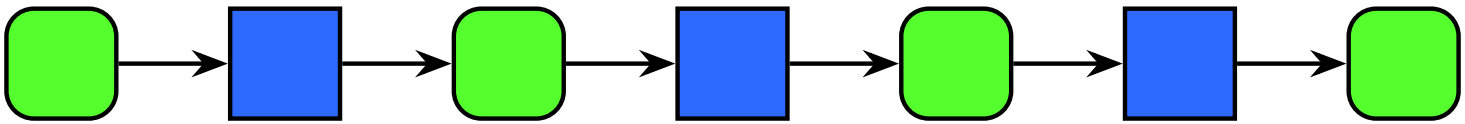
- A generator function (i.e. iterator) yields data items one at a time, allowing the function's control flow to resume on its next execution.

Asynchronous Generators, Producer-Consumer

- *async generators* can concurrently produce items while the generator's caller is consuming items, creating a producer-consumer pair

Pipeline

- By connecting many producer-consumer pairs we create a *pipeline*
- Pipelines need not be linear, they can be *directed acyclic graphs*



Intersect as a generator

Algorithm 3 **Intersect**(p, T)

Input: $p \in \mathbf{k}[\underline{X}] \setminus \mathbf{k}$, $v := \mathrm{mvar}(p)$, a regular chain T s.t. $T = T_v^- \cup T_v$
Output: regular chains $T_1, \ldots, T_e$ satisfying specs.

```
1:  for (g_i, T_i) ∈ RegularGCD(p, T_v, v, T_v^-) do
2:      if dim(T_i) ≠ dim(T_v^-) then
3:          for T_{i,j} ∈ Intersect(p, T_i) do
4:              yield T_{i,j}
5:      else
6:          if g_i ∉ k and deg(g_i, v) > 0 then
7:              yield T_i ∪ {g_i}
8:          for T_{i,j} ∈ Intersect(lc(g_i, v), T_i) do
9:              for T' ∈ Intersect(p, T_{i,j}) do
10:                 yield T'
```

- **yield** "produces" a single data item, and then continues computation
- each **for** loop consumes a data item one at a time from the generator

Generators are both producers and consumers

Algorithm 3 **Intersect**(p, T)

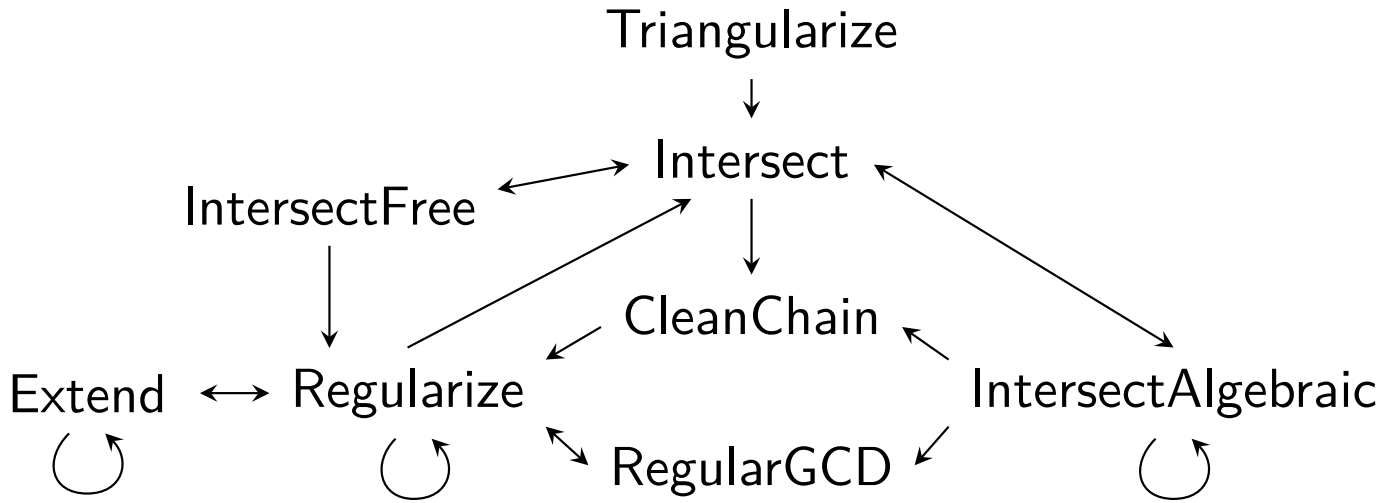
```
1:  for (g_i, T_i) ∈ RegularGCD(p, T_v, T_v^-) do
2:      if dim(T_i) ≠ dim(T_v^-) then
3:          for T_{i,j} ∈ Intersect(p, T_i) do
4:              yield T_{i,j}
5:      else
6:          if g_i ∉ k and deg(g_i, v) > 0 then
7:              yield T_i ∪ {g_i}
8:          for T_{i,j} ∈ Intersect(lc(g_i, v), T_i) do
9:              for T' ∈ Intersect(p, T_{i,j}) do
10:                 yield T'
```

Algorithm 4 **Regularize**(p, T)

```
1:  for (g_i, T_i) ∈ RegularGCD(p, T_v, T_v^-) do
2:                  ▷ assume dim(T_i) = dim(T_v^-)
3:      if 0 < deg(g_i, v) < deg(T_v, v) then
4:          yield T_i ∪ g_i
5:          yield T_i ∪ pquo(T_v, g_i)
6:          for T_{i,j} ∈ Intersect(lc(g_i, v), T_i) do
7:              for T' ∈ Regularize(p, T_{i,j}) do
8:                  yield T'
9:      else
10:         yield T_i
```

- Establishing mutually recursive functions as generators allows data to **stream** between subroutines; subroutines are effectively *non-blocking*
- function call stack of generators creates a *dynamic parallel pipeline*.

The subroutine pipeline

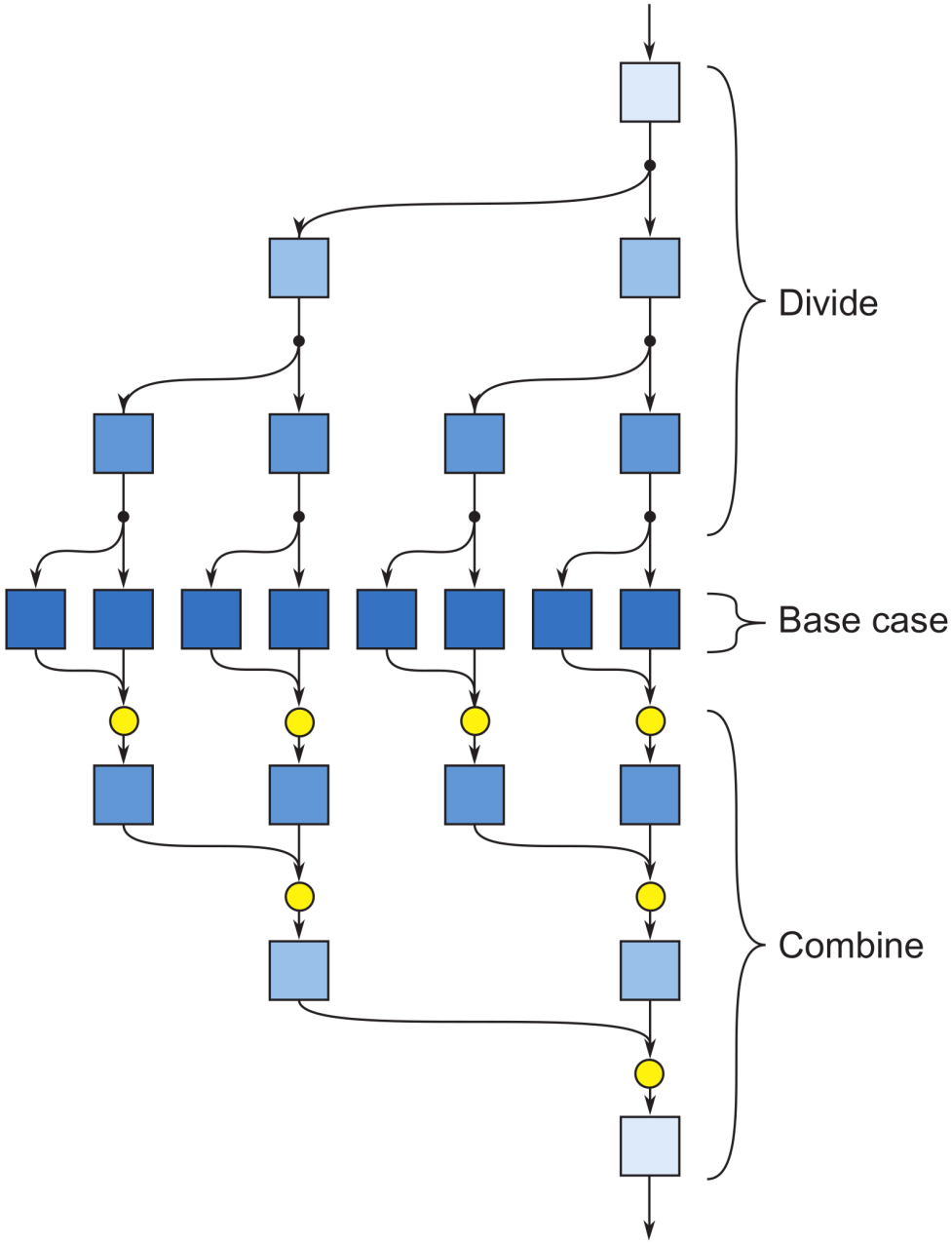


- All subroutines, as generators, allow the pipeline to evolve dynamically with the call stack.
- The call stack forms a **tree** if several generators are invoked by one consumer
- This pipeline creates **fine-grained parallelism** since work diminishes with each recursive call
- A thread pool is used and shared among all generators; generators run synchronously if the pool is empty

Outline

Divide-and-conquer and fork-join

- Divide a problem into sub-problems, solving each recursively
- Combine sub-solutions to produce a full solution
- **Fork**: execute multiple recursive calls in parallel (divide)
- **Join**: merge parallel execution back into serial execution (combine)



Removal of redundant components

After a system is solved, and many components found, we can remove components from the solution set that are contained within others

- Follow a merge-sort approach; **spawn**/fork and **sync**/join

---

Algorithm 5 RemoveRedundantComponents($\mathcal{T}$)

Input: a finite set $\mathcal{T} = \{T_1, \ldots, T_e\}$ of regular chains
Output: an irredundant set $\mathcal{T}'$ with the same algebraic set as $\mathcal{T}$

if $e = 1$ **then return** $\mathcal{T}$
$\ell \leftarrow \lceil e/2 \rceil$; $\mathcal{T}_{\leq \ell} \leftarrow \{T_1, \ldots, T_\ell\}$; $\mathcal{T}_{> \ell} \leftarrow \{T_{\ell+1}, \ldots, T_e\}$
$\mathcal{T}_1 :=$ **spawn** RemoveRedundantComponents($\mathcal{T}_{\leq \ell}$)
$\mathcal{T}_2 :=$ RemoveRedundantComponents($\mathcal{T}_{> \ell}$)
sync
$\mathcal{T}_1' := \emptyset$; $\mathcal{T}_2' := \emptyset$
for $T_1 \in \mathcal{T}_1$ **do**
  if $\forall T_2$ **in** $\mathcal{T}_2$ IsNotIncluded (T_1, T_2) **then** $\mathcal{T}_1' := \mathcal{T}_1' \cup \{T_1\}$
for $T_2 \in \mathcal{T}_2$ **do**
  if $\forall T_1$ **in** $\mathcal{T}_1'$ IsNotIncluded (T_2, T_1) **then** $\mathcal{T}_2' := \mathcal{T}_2' \cup \{T_2\}$
return $\mathcal{T}_1' \cup \mathcal{T}_2'$

Outline

Experimentation setup

Thanks to Maplesoft, we have a collection of over 3000 real-world systems from: actual user data, the literature, bug reports. In this experimentation, we solve 2815 of these systems in under 2 hours.

Of these 2815 systems, 300 require greater than 0.1s to solve

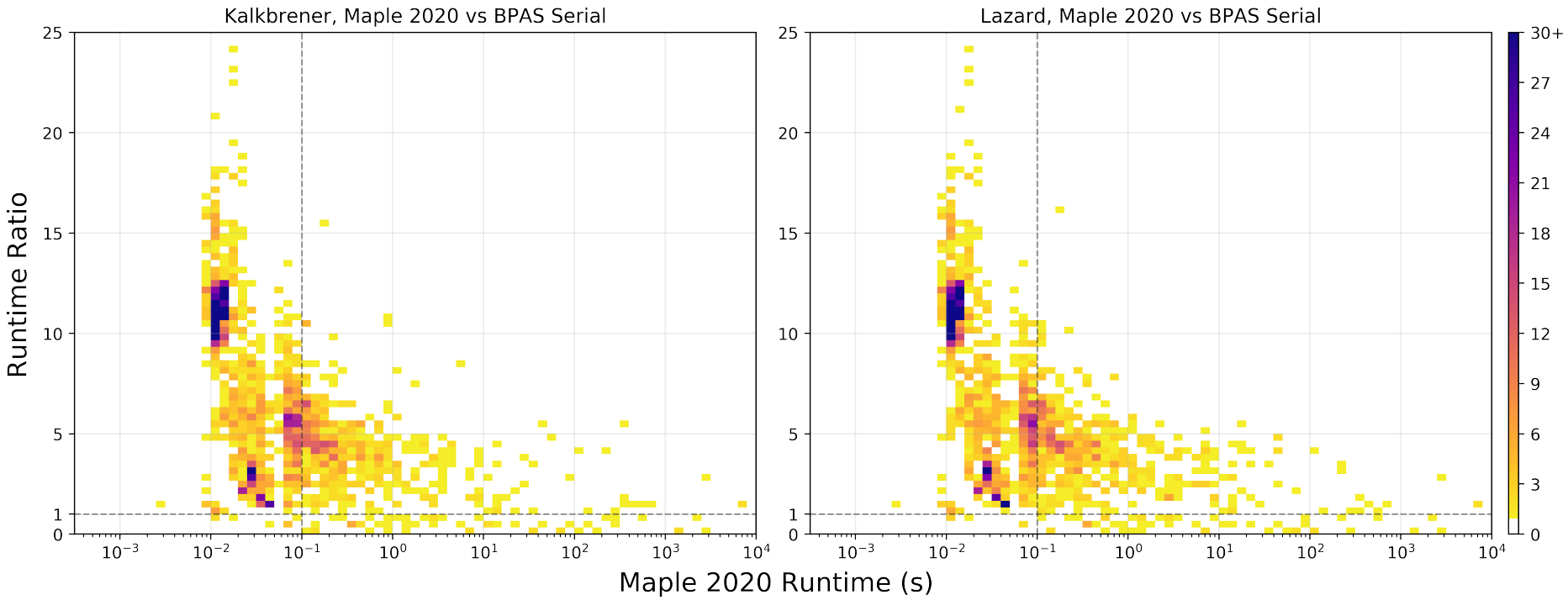
- Non-trivial systems to warrant the overheads of parallelism

1739 of these 2815 systems **do not split** at all

- No speed-up expected; *some slow-down* is expected in these cases
- however, we include them to ensure that *slow-down is minimal*

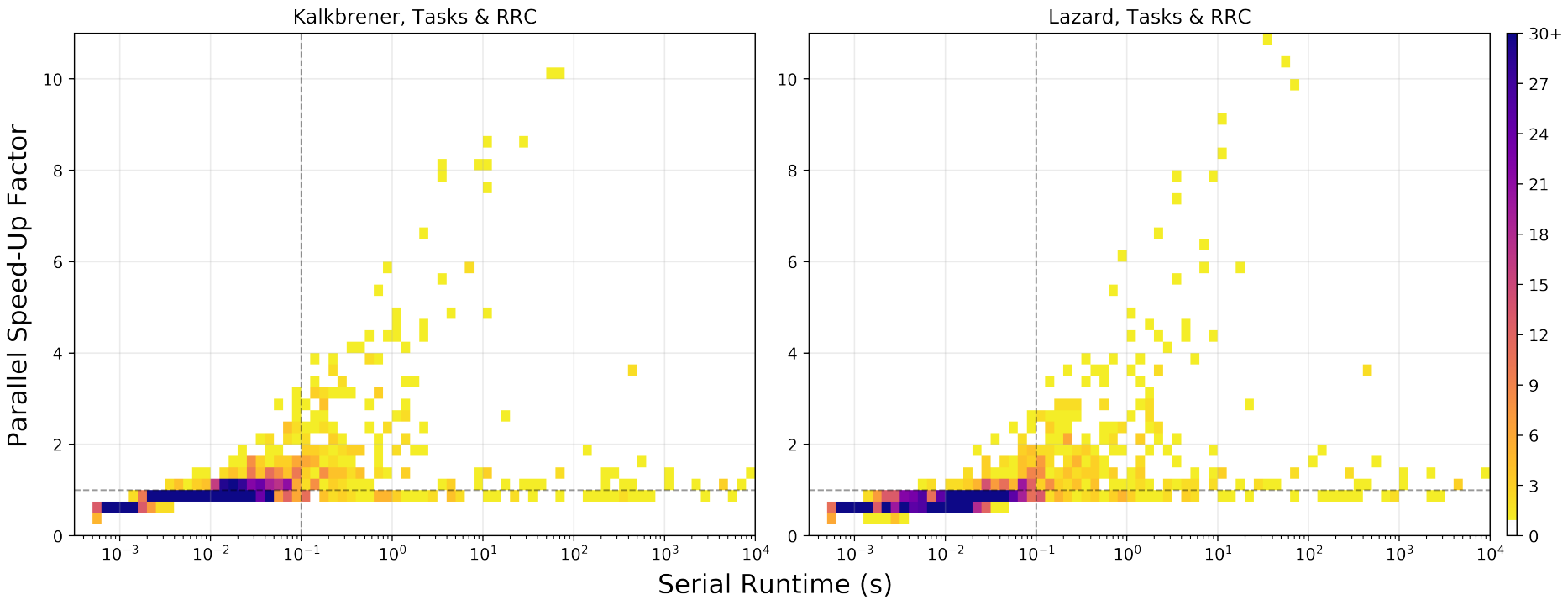
These experiments are run on a node with 2x6-core Intel Xeon X560 processors (24 physical threads with hyperthreading)

BPAS serial vs Maple



Comparing the runtime performance of triangular decomposition in the `RegularChains` library of MAPLE 2020 against the serialized implementation in BPAS.

Speedup obtained from tasks and fork-join

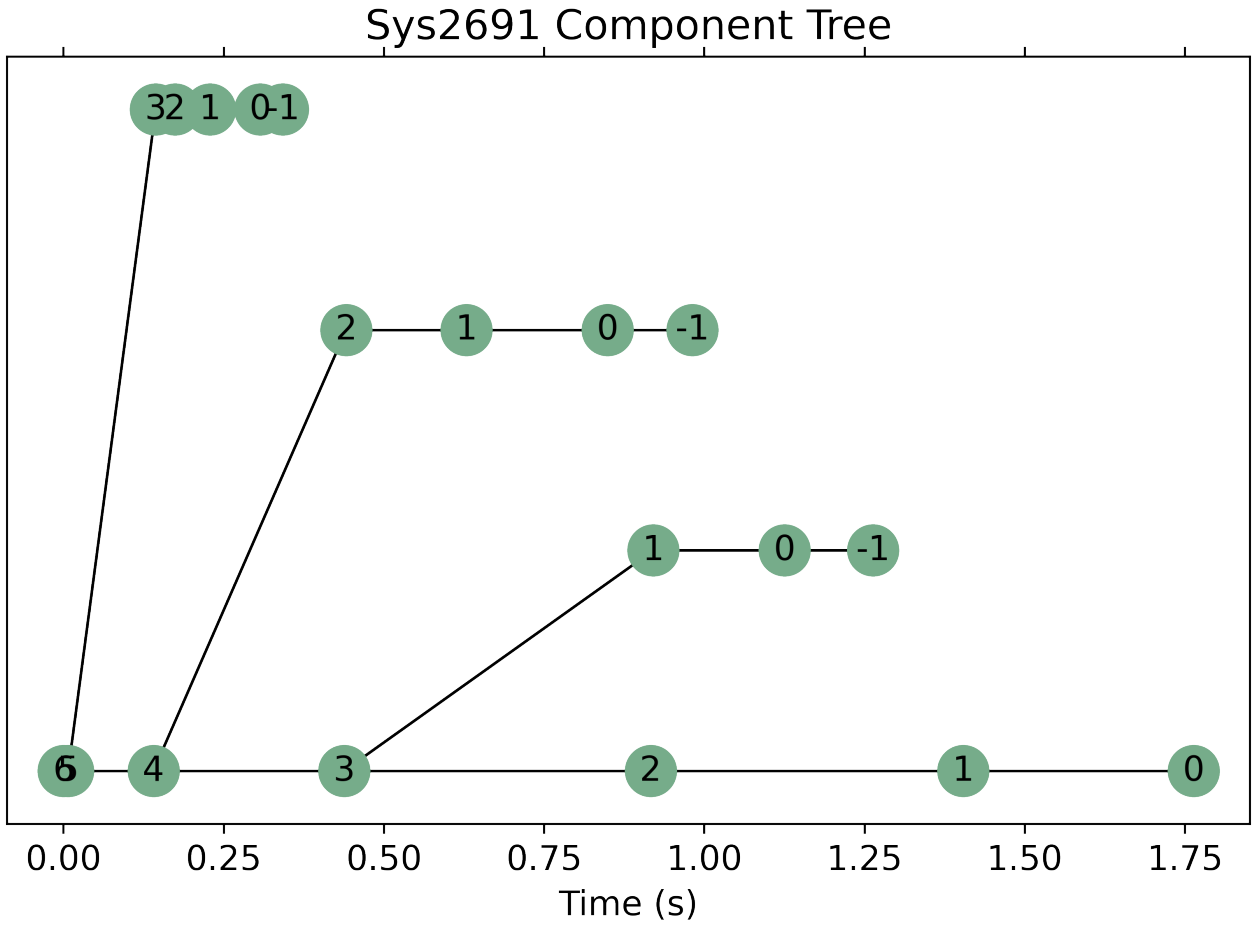


The parallel-speedup obtained from using parallel triangularize tasks and parallel removal of redundant components (RRC) together for solving in Kalkbrener and Lazard modes.

Timings for a few well-known systems

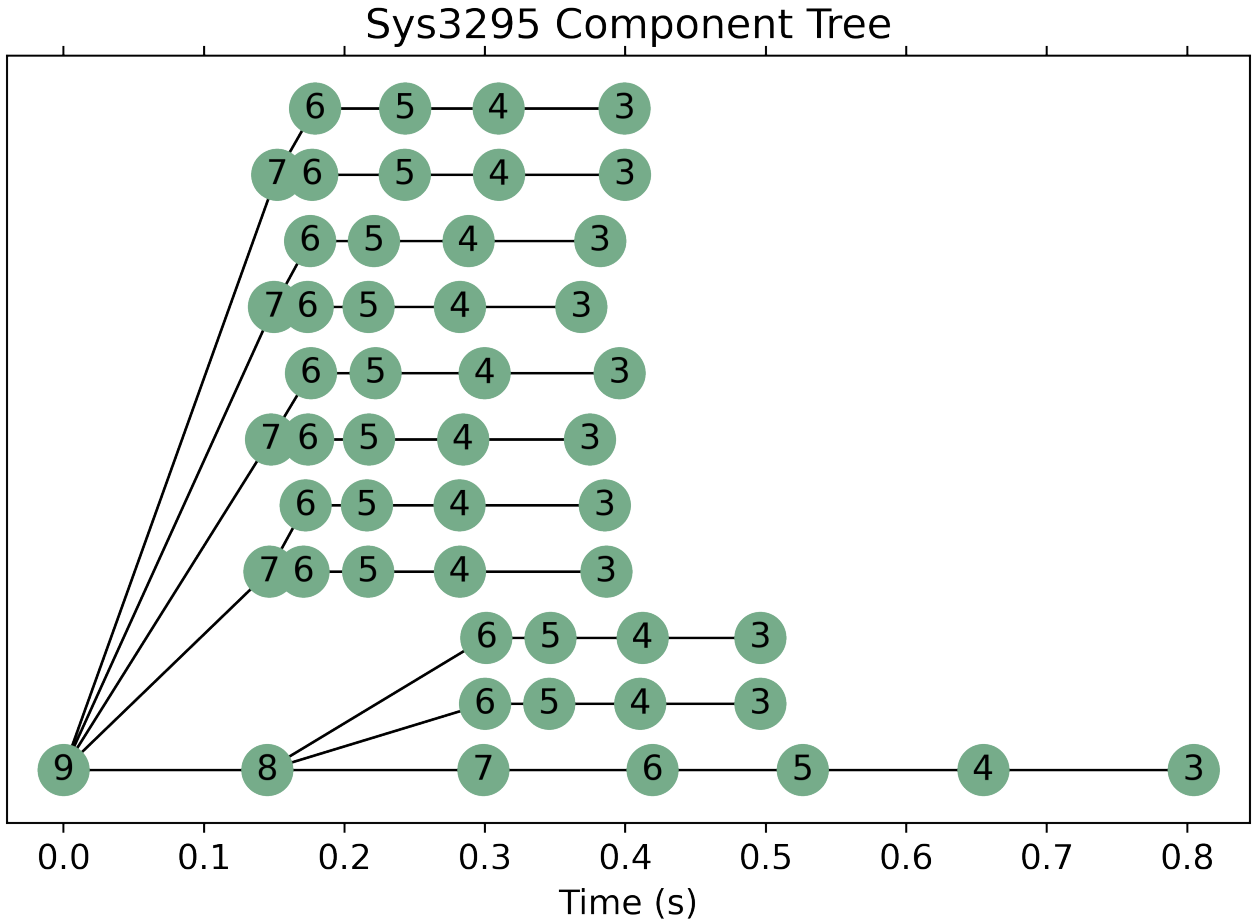
System	Kalkbrener Serial Time (s)	Speed-Up	Maple Ratio	Lazard Serial Time (s)	Speed-Up	Maple Ratio
Leykin-1	1.01	1.82	4.64	1.71	2.00	4.50
Sys2873	1.01	4.97	4.13	1.01	4.97	4.13
Gonnet	1.15	4.75	2.47	1.14	4.48	2.51
Sys1792	1.17	2.65	3.99	1.18	2.59	2.70
Sys2946	1.24	4.41	0.70	1.57	3.09	0.91
Sys2647	1.27	2.65	3.51	2.62	3.89	3.06
Pappus	1.27	3.01	3.08	5.65	3.88	4.15
Sys2945	1.30	3.57	2.77	1.29	3.48	2.82
W33	1.38	2.59	1.93	1.63	2.46	1.80
Sys3011	1.51	2.19	1.68	1.55	2.23	1.88
Sys2916	1.52	2.22	1.65	1.55	2.22	1.88
MontesS16	1.56	4.20	2.21	1.58	3.98	2.23
Wu-Wang	1.61	1.91	2.41	2.04	2.24	1.90
Hairer-2-BGK	1.80	3.33	1.47	1.60	2.52	1.83
Sys2353	2.16	4.35	3.84	2.23	4.62	3.76
W2	2.19	1.87	2.96	2.50	2.16	2.50
nld-3-5	2.22	2.68	4.09	2.22	2.68	4.09
Sys2875	2.44	6.23	3.17	2.44	6.23	3.17
8-3-config-Li	2.49	4.70	3.47	9.63	4.52	4.15
Sys2128	3.37	7.91	4.53	3.29	7.75	4.54
Sys2881	3.60	5.57	2.87	3.60	5.57	2.87
Sys2885	3.70	7.82	2.33	3.69	8.48	2.39
Sys2297	4.40	4.73	3.52	4.34	4.80	3.35
W5	6.96	5.83	3.89	6.99	5.88	3.36
Reif	7.81	5.74	1.96	7.81	5.74	1.96
Sys2161	8.80	7.91	5.40	8.67	7.85	4.99
W44	10.14	8.61	2.08	10.67	8.67	1.88
Mehta3	10.19	7.65	1.75	9.84	1.89	4.75
Sys2449	10.54	8.47	4.86	10.85	8.84	4.17
Sys2882	12.50	5.29	2.51	16.69	6.06	2.50
Sys2943	17.25	2.60	1.17	21.90	2.65	1.35
dgp6	29.04	8.49	2.76	37.38	10.27	2.03
Sys2880	56.57	10.10	4.32	57.37	10.47	3.60
Sys2874	70.43	10.22	5.39	70.93	10.17	3.06
Sys3270	149.11	3.72	1.04	149.11	3.72	1.04
Sys3283	167.82	3.46	1.90	167.82	3.46	1.90
Sys3281	214.47	3.07	1.22	214.47	3.07	1.22
KdV	456.08	3.68	1.38	462.34	3.63	1.37
Themos-net	1098.57	1.01	2.77	1098.57	1.01	2.77
tryme	3100.90	1.18	0.75	3100.90	1.18	0.75
childDraw-2	4499.91	1.25	0.32	4499.91	1.25	0.32
Sys1651	4792.44	1.16	1.39	4792.44	1.16	1.39
Sys2984	4793.55	1.16	1.39	4793.55	1.16	1.39

Inspecting the geometry: Sys2691



- Bottom "main" branch is majority of the work.
- Little overlap with the quickly-solved degenerative branches
- 2.13× speedup achieved; 88% efficient compared to work/span ratio

Inspecting the geometry: Sys3295



- Up to 11 active branches at once, but overlap is only for 0.1s
- 4.94× speedup; 75% efficient
- Could consider other parallelism in "main" branch once all other tasks have finished and released resources (poly arithmetic, subresultants)

Incremental decomposition: conclusion

We have tackled irregular parallelism in a high-level algebraic algorithm

- our solution dynamically finds and exploits opportunities for concurrency
- uses dynamic parallel task management, async. generators, and DnC
- Dnc is also used to construct subresultant chains via evaluation/interpolation techniques
- While async. generators do not help much (because the corresponding tasks became too fine-grained as we were optimizing polynomial arithmetic) they did help in the past (ISSAC 2021).
- All our parallel patterns (task management, async. generators, and DnC) are part of the BPAS library and do not rely on any other concurrency plafform;
- The benefit is that all those parallel patterns rely on the same scheduler.

Incremental decomposition: future work

Further parallelism can be found through:

- solving over a prime field, which produces more splittings;
- then lifting to solutions over the rational numbers, which can be done by evaluation/interpolation techniques.

Our parallel techniques could be employed in further high-level algorithms.

- e.g. factorization: pipelining between square-free, distinct-degree, and equal-degree factorization

Outline

Notations

$\mathbb{A} = \mathbf{k}[[X_1, \ldots, X_n]]$ is the ring of multivariate power series over an algebraically closed field. Its maximal ideal is $\mathcal{M} = \langle X_1, \ldots, X_n \rangle$.

- $f = \sum_{e \in \mathbb{N}^n} a_e X^e \in \mathbf{k}[[X_1, \ldots, X_n]]$
- $X^e = X_1^{e_1} \cdots X_n^{e_n}, \quad |e| = e_1 + \cdots + e_n$
- $f_{(k)} = \sum_{|e|=k} a_e X^e$ is the **homogeneous part** of f of degree k
- $f_{(k)} \in \mathcal{M}^k \setminus \mathcal{M}^{k+1}$
- The units of $\mathbb{A}$ are $\{u \,|\, u \notin \mathcal{M}\}$

$\mathbb{A}[Y]$ is the ring of Univariate Polynomials over Power Series (UPoPS)

- $f = \sum_{i=0}^{d} a_i Y^i, \; a_i \in \mathbb{A}, \; a_d \neq 0$ is a UPoPS of degree d
- Denote degree of f in Y by $\deg f = d$

Lazy evaluation for power series arithmetic

Power series implemented using a *lazy evaluation* scheme allow for terms to be computed on demand, increasing **precision** as needed.

Our lazy power series require:

1. an **update function** to compute homogeneous parts of a given degree
2. capturing parameters required for the update function
3. storing previously computed homogeneous parts

Where update parameters are power series, they are called **ancestors**.

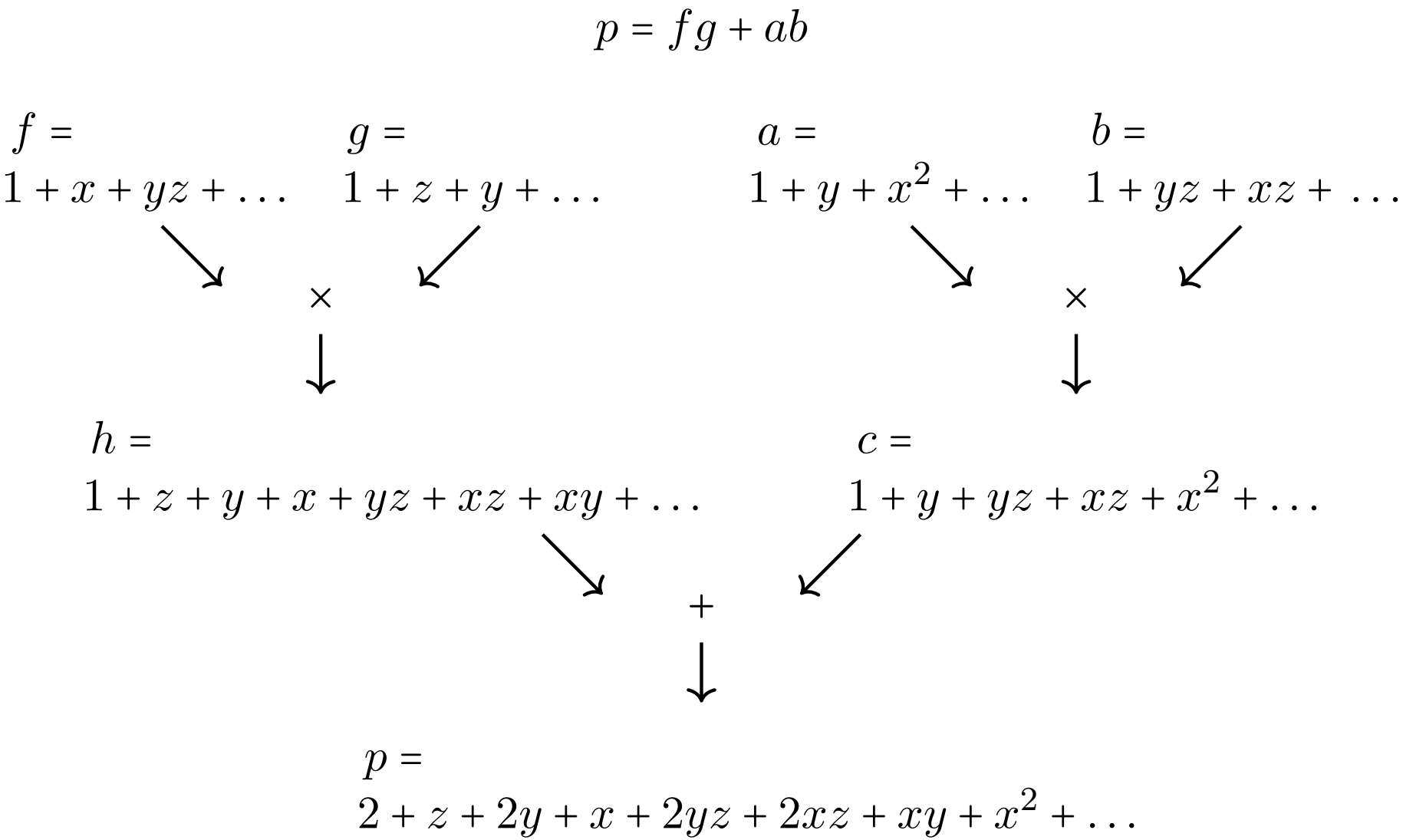
Addition, $f = g + h$

- $f_{(k)} = g_{(k)} + h_{(k)}$

Multiplication $f = gh$

- $f_{(k)} = \sum_{i=0}^{k} g_{(i)} h_{(k-i)}$

Ancestry example



Weierstrass preparation theorem in $\mathbf{k}[[X_1, \ldots, X_n]][Y]$

Theorem (Weierstrass preparation)

Let $f \in \mathbf{k}[[X_1, \ldots, X_n]][Y]$ *and assume* $f \not\equiv 0 \mod \mathcal{M}[Y]$. *Write* $f = \sum_{i=0}^{d+m} a_i Y^i \in \mathbf{k}[[X_1, \ldots, X_n]][Y]$, *where* $d \geq 0$ *is the smallest integer such that* $a_d \notin \mathcal{M}$ *and* $m \in \mathbb{Z}^+$.

Then, there exists a unique pair p, α *satisfying the following:*

1. $f = p\,\alpha$,
2. α *is an invertible element of* $\mathbf{k}[[X_1, \ldots, X_n]][[Y]]$,
3. p *is a monic polynomial of degree* d,
4. *writing* $p = Y^d + b_{d-1}Y^{d-1} + \cdots b_1 Y + b_0$, *we have* $b_{d-1}, \ldots, b_0 \in \mathcal{M}$.

Theorem (Algebraic complexity for Weierstrass)

For $f = p\,\alpha \in \mathbf{k}[[X_1]][Y], \deg p = d, \deg \alpha = m$, *computing* p *and* α *to precision* k *requires* $d(m+1)k^2 + dmk$ *operations in* $\mathbf{k}$.

Weierstrass preparation by lazy evaluation

Let $f = \sum_{\ell}^{d+m} a_{\ell=0} Y^{\ell}$, $p = Y^d + \sum_{j=0}^{d-1} b_j Y^j$, $\alpha = \sum_{i=0}^{m} c_i Y^i$ be UPoPS.

↳ a_ℓ, b_j, c_i are power series ↳ $b_j \in \mathcal{M}$ for $j = 0, \ldots, d-1$

$$
f = \alpha p \implies
\begin{aligned}
a_0 &= b_0 c_0 \\
a_1 &= b_0 c_1 + b_1 c_0 \\
&\vdots \\
a_{d-1} &= b_0 c_{d-1} + b_1 c_{d-2} + \cdots + b_{d-2} c_1 + b_{d-1} c_0 \\
a_d &= b_0 c_d + b_1 c_{d-1} + \cdots + b_{d-1} c_1 + c_0 \\
&\vdots \\
a_{d+m-1} &= b_{d-1} c_m + c_{m-1} \\
a_{d+m} &= c_m
\end{aligned}
$$

We update p and α by solving these equations modulo $\mathcal{M}^k$, $k = 1, 2, \ldots$

modulo $\mathcal{M}$ we have:

(1) $b_j \equiv 0 \mod \mathcal{M}$, $j = 0, \ldots, d-1$ (2) $c_i \equiv a_i \mod \mathcal{M}$ for $i = 0, \ldots, m$.

⏩ skip slide

Lazy Weierstrass Phase 1: update p

Let $f = \sum_{\ell}^{d+m} a_\ell Y^\ell$, $p = Y^d + \sum_{j}^{d-1} b_j Y^j$, $\alpha = \sum_{i}^{m} c_i Y^i$ be UPoPS.

↳ a_ℓ, b_j, c_i are power series

↳ $b_j \in \mathcal{M}$ for $j = 0, \ldots, d-1$

$$\begin{aligned} a_0 &= b_0 c_0 \\ a_1 - b_0 c_1 &= b_1 c_0 \\ a_2 - b_0 c_2 - b_1 c_1 &= b_2 c_0 \\ &\vdots \\ a_{d-1} - b_0 c_{d-1} - b_1 c_{d-2} + \cdots - b_{d-2} c_1 &= b_{d-1} c_0 \end{aligned}$$

- let $F_j = a_j - \sum_{i=0}^{j-1} b_i c_{j-i}$
- with power series division we can solve $F_j = b_j c_0$ from $j = 0$ to $d-1$
  - ↳ $b_{j\,(k)} = {}^{1}\!/_{c_{0\,(0)}} \left(F_{j\,(k)} - \sum_{i=1}^{k-1} b_{j\,(i)} c_{0\,(k-i)} \right)$
- each F_j lazily updated through lazy power series arithmetic

Lazy Weierstrass Phase 2: update α

Let $f = \sum_{\ell}^{d+m} a_\ell Y^\ell$, $p = Y^d + \sum_{j}^{d-1} b_j Y^j$, $\alpha = \sum_{i}^{m} c_i Y^i$ be UPoPS.

↳ a_ℓ, b_j, c_i are power series

↳ $b_j \in \mathcal{M}$ for $j = 0, \ldots, d-1$

$$\begin{aligned} c_m &= a_{d+m} \\ c_{m-1} &= a_{d+m-1} - b_{d-1}c_m \\ c_{m-2} &= a_{d+m-2} - b_{d-2}c_m - b_{d-1}c_{m-1} \\ &\vdots \\ c_0 &= a_d - b_0 c_d - b_1 c_{d-1} - \cdots - b_{d-1}c_1 \end{aligned}$$

- $c_{m-i\,(k)} = a_{d+m-i\,(k)} - \sum_{j=1}^{i} \left(b_{d-j}c_{m-i+j}\right)_{(k)}$, for $i \leq d$
- each c_{m-i} lazily updated through lazy power series arithmetic
- $\left(b_{d-j}c_{m-i+j}\right)_{(k)}$ only requires up to $c_{m-i+j\,(k-1)}$ since $b_{d-j\,(0)} = 0$
- each c_{m-i} can thus be updated simultaneously

Parallelization opportunities in Weierstrass

parallel map-reduce or **parallel for** loops

In phase 1: $b_{j_{(k)}} = {}^{1}\!/_{c_{0(0)}} \left(F_{j_{(k)}} - \sum_{i=1}^{k-1} b_{j_{(i)}} c_{0(k-i)} \right)$, with $F_j = a_j - \sum_{i=0}^{j-1} b_i c_{j-i}$

- compute summation using map-reduce:

$$\sum_{i=1}^{k} b_{j_{(i)}} c_{0(k-i)}$$

- notice in multivariate case, e.g., $b_{j_{(1)}} c_{0(k-1)}$ is less work than $b_{j_{(\frac{k}{2})}} c_{0(k-\frac{k}{2})}$

In phase 2: $c_{m-i(k)} = a_{d+m-i(k)} - \sum_{j=1}^{i} \left(b_{d-j} c_{m-i+j} \right)_{(k)}$

- compute each $c_{m-i(k)}$, $0 \leq i \leq m$ simultaneously, and/or
- compute product homogeneous parts using a map-reduce:

$$\left(b_{d-j} c_{m-i+j} \right)_{(k)} = \sum_{\ell=1}^{k} b_{d-j_{(\ell)}} c_{m-i+j_{(k-\ell)}}$$

- load-balance issue again in the multivariate case

Update to degree: Map-Reduce

Algorithm 6 UpdateToDegParallel$(k,\ f,\ t)$

Input: $k \in \mathbb{Z}^+$, $f \in \mathbf{k}[[X_1, \ldots, X_n]]$ known to at least precision $k-1$. If f has ancestors, it is the result of a binary operation. $t \in \mathbb{Z}^+$ for the number of threads to use.

Output: f is updated to precision k, in place.

1: $g,\ h \leftarrow$ FirstAncestor(f), SecondAncestor(f)
2: UpdateToDegParallel$(k,\ g,\ t)$;
3: UpdateToDegParallel$(k,\ h,\ t)$;

4: **if** f is a product **then** ▷ compute $f_{(k)}$ by map-reduce
5: $\quad \mathcal{V} \leftarrow [0, \ldots, 0]$ ▷ 0-indexed list of size t
6: $\quad$ **parallel_for** $j \leftarrow 0$ to $t-1$
7: $\quad\quad$ **for** $i \leftarrow {}^{jk}/_{t}$ to ${}^{(j+1)k}/_{t} - 1$ **while** $i \leq k$ **do**
8: $\quad\quad\quad \mathcal{V}[j] \leftarrow \mathcal{V}[j] + g_{(i)} h_{(k-i)}$
9: $\quad f_{(k)} \leftarrow \sum_{j=0}^{t-1} \mathcal{V}[j]$ ▷ reduce

10: **else if** f is a p from a Weierstrass preparation **then**
11: $\quad$ WeierstrassPhase1Parallel$(k,g,f,h,$WeierstrassData$(f),t)$
12: **else if** f is an α from a Weierstrass preparation **then**
13: $\quad$ WeierstrassPhase2Parallel$(k,\ g,\ h,\ f,\ t)$
14: **else**
15: $\quad$ UpdateToDeg$(k,\ f)$ ▷ fallback to serial algorithm

Hensel's Lemma

Theorem (Hensel's Lemma)

Let $f = Y^d + \sum_{i=0}^{d-1} a_i Y^i$ *be a monic polynomial in* $\mathbf{k}[[X_1, \ldots, X_n]][Y]$.
Let $\bar{f} = f(0, \ldots, 0, Y) = (Y - c_1)^{d_1}(Y - c_2)^{d_2} \cdots (Y - c_r)^{d_r}$ *for* $c_1, \ldots, c_r \in \mathbf{k}$ *and positive integers* $d_1, \ldots, d_r$. *Then, there exists* $f_1, \ldots, f_r \in \mathbf{k}[[X_1, \ldots, X_n]][Y]$, *all monic in Y, such that:*

1. $f = f_1 \cdots f_r$,
2. $\deg f_i, Y = d_i$ *for* $1 \leq i \leq r$, *and*
3. $\bar{f}_i = (Y - c_i)^{d_i}$ *for* $1 \leq i \leq r$.

Proof:

Let $g = f(X_1, \ldots, X_n, Y + c_r) = Y^d + \sum_{i=0}^{d-1} b_i Y^i$, sending c_r to the origin. By construction, $b_0, \ldots, b_{d_r - 1} \in \mathcal{M}$ and Weierstrass preparation can be applied to produce $g = p\,\alpha$ with $\deg p = d_r, \deg \alpha = d - d_r$.

Reversing the shift, $f_r = p(Y - c_r)$.
Induction on $\hat{f} = \alpha(Y - c_r)$ completes the proof. □

Hensel Factorization

Algorithm 7 HenselFactorization(f)

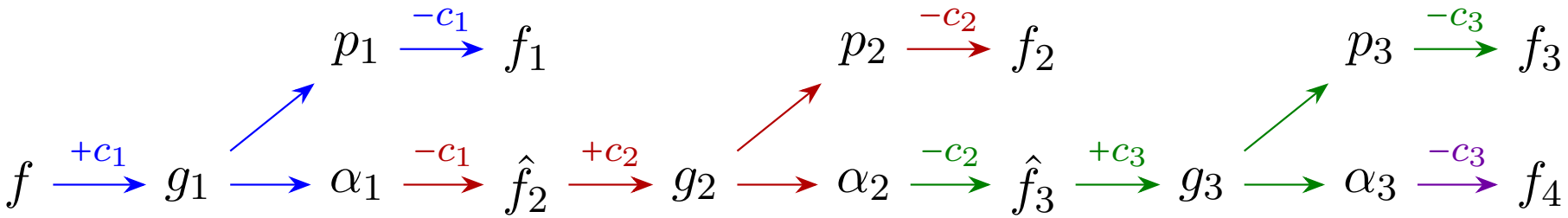
Input: $f = Y^d + \sum_{i=0}^{d-1} a_i Y^i, a_i \in \mathbf{k}[[X_1, \ldots, X_n]]$.
Output: $f_1, \ldots, f_r$ satisfying Theorem 3.

1: $\bar{f} = f(0, \ldots, 0, Y)$
2: $(c_1, \ldots, c_r), (d_1, \ldots, d_r) \leftarrow$ roots and their multiplicities of $\bar{f}$
3: $c_1, \ldots, c_r \leftarrow \text{SORT}([c_1, \ldots, c_r])$ by increasing multiplicity
4: $\hat{f}_1 \leftarrow f$
5: **for** $i \leftarrow 1$ to $r-1$ **do**
6: $\quad g_i \leftarrow \hat{f}_i(Y + c_i)$
7: $\quad p_i, \alpha_i \leftarrow$ WeierstrassPreparation(g)
8: $\quad f_i \leftarrow p_i(Y - c_i)$
9: $\quad \hat{f}_{i+1} \leftarrow \alpha_i(Y - c_i)$
10: $f_r \leftarrow \hat{f}_r$
11: **return** $f_1, \ldots, f_r$

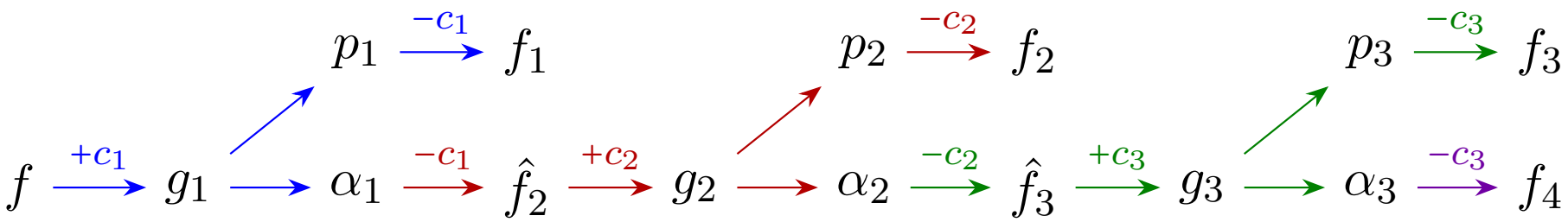
Parallel Opportunities in Hensel

- The output of one Weierstrass becomes input to another
- $f_{i+i(k)}$ relies on $f_{i(k)}$
- Can compute $f_{i(k+1)}$ and $f_{i+i(k)}$ concurrently in a **pipeline**

	Stage 1 (f_1)	Stage 2 (f_2)	Stage 3 (f_3)	Stage 4 (f_4)
Time 1	$f_{1(1)}$			
Time 2	$f_{1(2)}$	$f_{2(1)}$		
Time 3	$f_{1(3)}$	$f_{2(2)}$	$f_{3(1)}$	
Time 4	$f_{1(4)}$	$f_{2(3)}$	$f_{3(2)}$	$f_{4(1)}$
Time 5	$f_{1(5)}$	$f_{2(4)}$	$f_{3(3)}$	$f_{4(2)}$
Time 6	$f_{1(6)}$	$f_{2(5)}$	$f_{3(4)}$	$f_{4(3)}$



Parallel Challenges and Composition



- Degrees and computational work diminish with each stage
  - ↳ $\deg g_1 = d,\ \deg g_2 = d - d_1,\ \deg g_3 = d - d_1 - d_2,\ \ldots$
- To load-balance each stage, give a decreasing number of threads to each stage to be used within Weierstrass preparation.
- From Theorem 1: updating f_i requires $\mathcal{O}(d_i \hat{d}_{i+1} k^2)$ operations
- Assign t_i threads to stage i so that $d_i \hat{d}_{i+1}/t_i$ is equal for each stage.
- Better still, update a *group of successive factors* per stage.
  - ↳ To each stage s assign factors $f_{s_1}, \ldots, f_{s_2}$ and t_s threads so that $\sum_{i=s_1}^{s_2} d_i \hat{d}_{i+1}/t_s$ is roughly equal for each stage.

⏩ skip slide

Hensel Pipeline Algorithm

Algorithm 8 HenselPipeStage(k, f_i, t, gen)

Input: $k \in \mathbb{Z}^+$, f_i a UPoPS, $t \in \mathbb{Z}^+$ the number of threads to use within this stage. gen a generator for the previous pipeline stage.

Output: a sequence of integers j signalling f_i is known to precision j; the sequence ends at k.

1: $p \leftarrow \text{Precision}(f_i)$ ▷ current precision of f_i
2: **do**
▷ A blocking function call until gen yields
3: $k' \leftarrow \text{gen}()$
4: **for** $j \leftarrow p$ to k' **do**
5: UpdateToDegParallel(j, f_i, t)
6: **yield** j
7: $p \leftarrow k'$
8: **while** $k' < k$

Algorithm 9 HenselFactorizationPipeline(k, $\mathcal{F}$, $\mathcal{T}$)

Input: $k \in \mathbb{Z}^+$, $\mathcal{F} = \{f_1, \ldots, f_r\}$ the output of HenselFactorization. $\mathcal{T} \in \mathbb{Z}^r$ a 0-indexed list of the number of threads to use in each stage, $\mathcal{T}[r-1] > 0$.

Output: $f_1, \ldots, f_r$ updated in-place to precision k.

▷ an anonymous function asynchronous generator
1: gen $\leftarrow$ ( ) $\rightarrow$ {**yield** k}
2: **for** $i \leftarrow 0$ to $r-1$ **do**
3: **if** $\mathcal{T}[i] > 0$ **then**
▷ Capture function as a function object, passing the previous gen as input
4: gen $\leftarrow$ AsyncGenerator(HenselPipeStage, k, f_{i+1}, $\mathcal{T}[i]$, gen)
▷ ensure last stage completes before returning
5: **do**
6: $k' \leftarrow \text{gen}()$
7: **while** $k' < k$

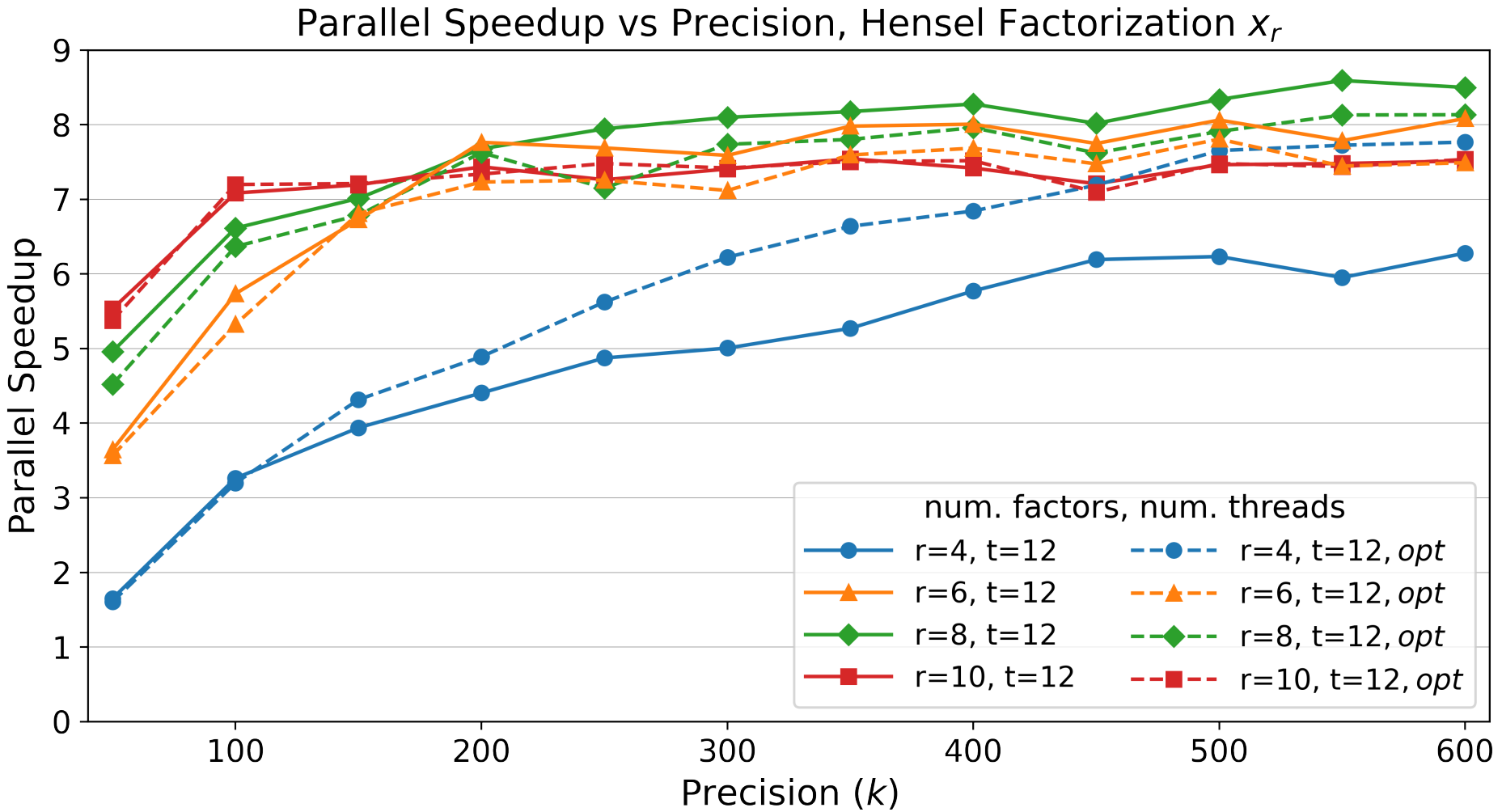
Hensel Pipeline Example

$$f = (Y-1)(Y-2)(Y-3)(Y-4) + X_1(Y^3+Y)$$
$$\bar{f}_1 = Y-1, \quad \bar{f}_2 = Y-2, \quad \bar{f}_3 = Y-3, \quad \bar{f}_4 = Y-4$$

factor	serial time (s)	shift time (s)	$d_i \hat{d}_{i+1}$	Complexity-est. threads	parallel time (s)	wait time (s)	Time-est. threads	parallel time (s)	wait time (s)
f_1	18.1989	0.0012	$1 \cdot 3$	6	4.5380	0.0000	7	3.5941	0.0000
f_2	6.6681	0.0666	$1 \cdot 2$	4	4.5566	0.8530	3	3.6105	0.6163
f_3	3.4335	0.0274	$1 \cdot 1$	1	4.5748	1.0855	0	-	-
f_4	0.0009	0.0009	$1 \cdot 0$	1	4.5750	4.5707	2	3.6257	1.4170

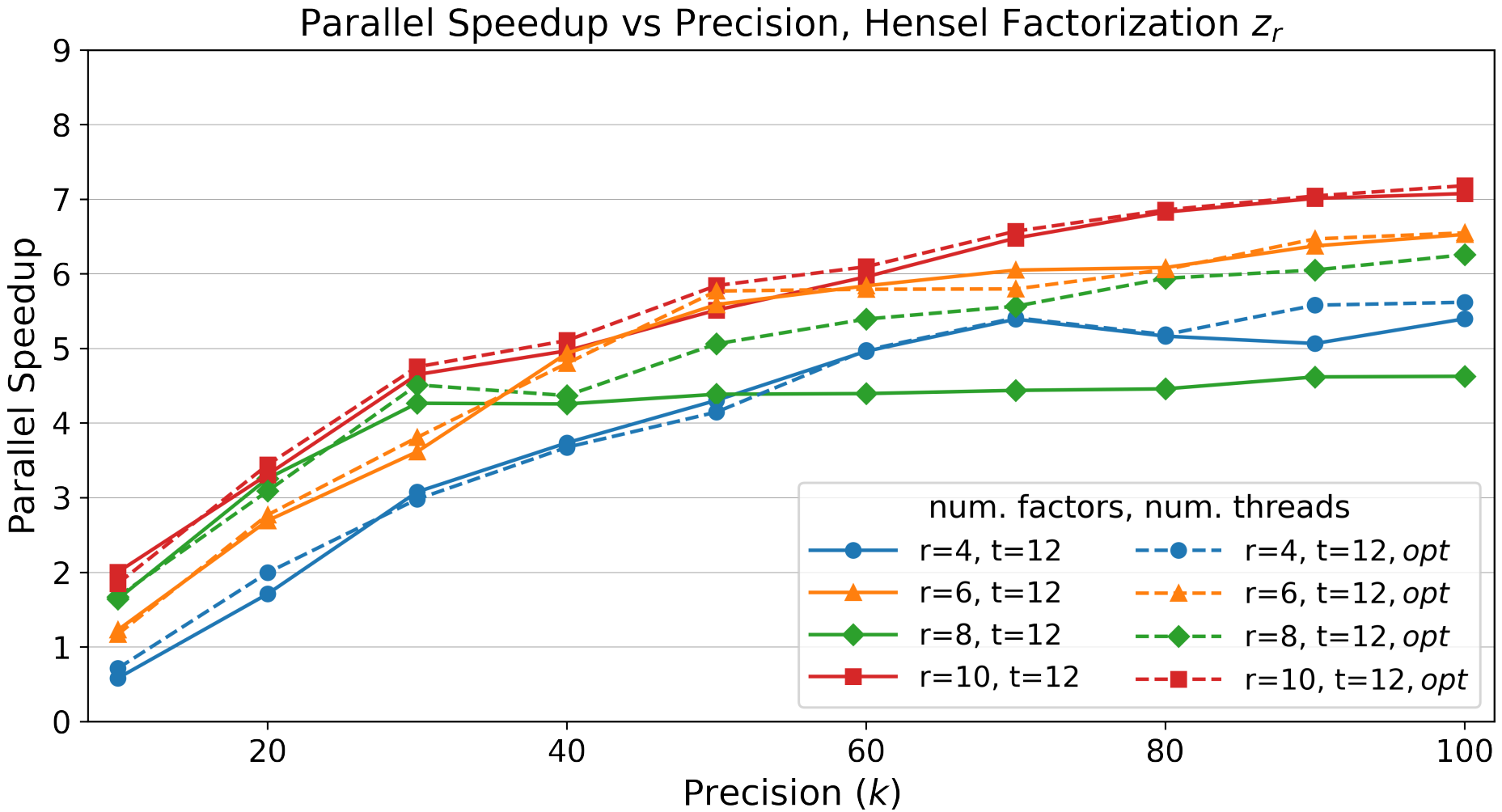
- f_4 requires at least one thread so that it (the last factor) gets updated
- work estimates based on complexity results okay, but does not account for, e.g., coefficient size or data locality.
- can also use serial time to suggest thread assignments

Parallel Speed-up Hensel Factorization 1



$$x_r = \prod_{i=1}^{r} (Y - i) + X_1 (Y^3 + Y)$$

Parallel Speed-up Hensel Factorization 2



$$z_r = \prod_{i=1}^{r} (Y + X_1 + X_2 - i) + X_1 X_2 (Y^3 + Y)$$

Outline

Compiler-Level automatic parallelization

- CILK and OPENMP provide automatic parallelization though compiler extensions
- Very easy but flexibility more challenging

```
void mergeSort(int* A, int i,
    int j) {
  //... base case, k
  cilk_spawn mergeSort(A, i, k);
  mergeSort(A, k, j);
  cilk_sync
  merge(A, i, k, j);
}
```

```
void mergeSort(int* A, int i, int
    j) {
  //... base case, k
  #pragma omp parallel
      num_threads(2)
  {
    #pragma omp sections {
      #pragma omp section {
        mergeSort(A, i, k);
      }
      #pragma omp section {
        mergeSort(A, k, j);
      }
    }
  }
  merge(A, i, k, j);
}
```

Fork-Join parallelism with BPAS

- Object-oriented
- Standard C++, no compiler extensions
- Extends the *Thread Support Library* of C++11

```
1  void mergeSort(int* A, int i, int j) {
2      //... base case, k
3      threadID id;
4      ExecutorThreadPool& pool =
5          ExecutorThreadPool::getThreadPool();
6
7      pool.obtainThread(id);
8      pool.executeTask(id, std::bind(mergeSort, A, i, k));
9      mergeSort(A, k, j);
10
11     pool.returnThread(id);
12
13     merge(A, i, k, j);
14 }
```

Outline

Threading primitives

C++11 introduced the *Thread Support Library*

- `std::thread`
  - ↳ C++ class encapsulating a thread (often a `pthread`) and its low-level spawn and join
- `std::mutex`
  - ↳ shared object between threads to indicate *mutual exclusion* to a **critical region**.
  - ↳ `mutex` is *locked* or *owned* by at most one thread at a time.
- `std::lock_guard`, `std::unique_lock`
  - ↳ temporary object wrapping a `mutex` whose object lifetime automatically locks and unlocks the `mutex`.
  - ↳ the constructor **blocks** and only returns once the shared `mutex` is successfully owned by the calling thread.
- `std::condition_variable`
  - ↳ blocks the current thread and temporarily releases a lock
  - ↳ receives notification from another thread to awaken the blocked thread

std::function

Functors, function objects, callable objects

- First-class objects which are callable using normal function syntax
- Are often constructed by passing function names, function pointers
- `std::bind` binds arguments to a function or function object, returning a function object which requires fewer arguments

```
1  void printInteger(int a) {
2      std::cout << a << std::endl;
3  }
4
5  //Function object from function name
6  std::function<void(int)> f_printInt(printInteger);
7  f_printInt(12);
8
9  //Function object binding arguments to function name
10 std::function<void()> f_print42( std::bind(printInteger,42) );
11 f_print42();
```

Parallel overheads

Creating and managing multiple threads of execution can be expensive

- Every thread spawn requires non-insignificant amount of time
- If more threads are active than the hardware supports, **over-subscription** occurs and repeated **context switching** slows down the program
- Thread synchronization, locking `mutexs`, accessing critical regions require special care

Thread pools mitigate the first two, by supplying a fixed number of long-running threads.

Parallel programming patterns are algorithmic designs for efficient thread scheduling and minimizing locking

Outline

Long-Running threads

Threads typically terminate once their assigned function/code block finishes

In order to implement and benefit from a thread pool, we require a mechanism which allows threads to:

1. Remain active until explicitly told to exit (or the entire program exits)
2. Receive new code blocks to execute on demand

`FunctionExecutorThread`s are such long-running threads which receive functions or code blocks and executes them asynchronously.

FunctionExecutorThread usage

```
1  int A[N];
2  int* ret = new int();
3  FunctionExecutorThread t;
4
5  t.sendRequest( [=]() void -> {
6      int s = 0;
7      for (int i = 0; i < N; ++i) {
8          s += A[i];
9      }
10     *ret = s;
11 });
12
13 doSomethingElse();
14
15 //make sure result is available before continuing
16 t.waitForThread();
17
18 std::cout << "sum: " << *ret << std::endl;
```

Object streams

The key to the implementation of `FunctionExectorThread` is the `AyncObjectStream` class. It provides:

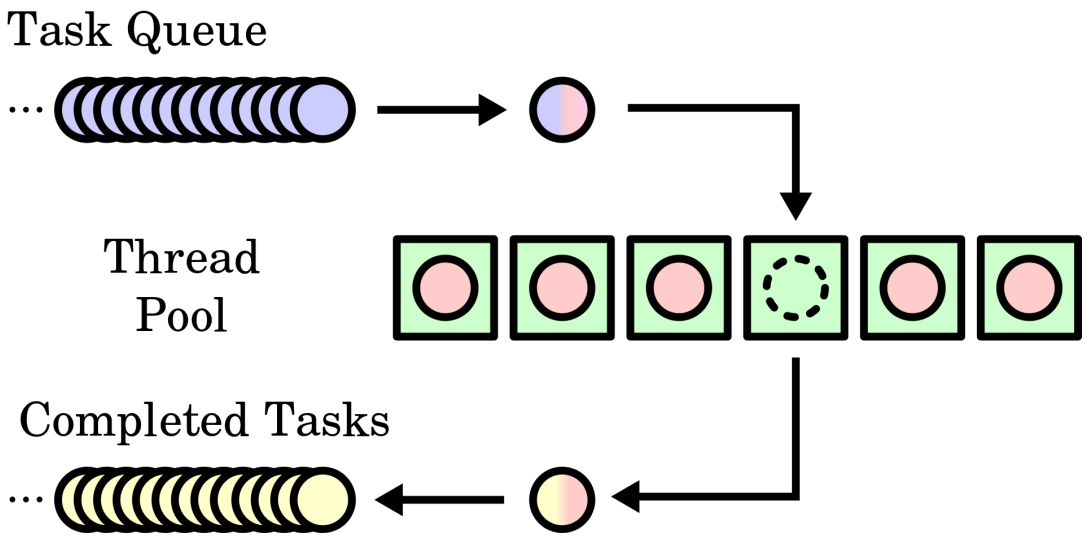
1. a queue for tasks (or any object) and
2. a blocking mechanism to keep the `FunctionExecutorThread` alive and idle when waiting for tasks

- Actually a class template for any kind of object being passed between two threads
- Implements a queue satisfying the **producer-consumer problem**
- A **std::queue** combined with a mutex and condition variable

Thread pools

A **thread pool** manages a collection of long-running threads and a queue of tasks

- spawn all threads once at the beginning of program
- idle threads receive and execute tasks as required
- if all threads busy, tasks are added to queue



ExecutorThreadPool

- A thread pool built using **FunctionExecutorThreads**
- An internal queue of tasks and queue of threads
- When threads are busy, they are temporarily removed from the pool
- When all threads busy, tasks are added to task queue

```
1  class ExecutorThreadPool {
2
3  private:
4      std::deque<FunctionExecutorThread*> threadPool;
5      std::deque<std::function<void()>> taskPool;
6      std::mutex m_mutex;
7      std::condition_variable m_cv; //used in waitForThreads
8
9      void tryPullTask();
10     void putBackThread(FunctionExecutorThread* t);
11
12 public:
13     void addTask(std::function<void()> f);
14     void waitForThreads();
15 }
```

ExecutorThreadPool: flexible usage

- In support of certain **parallel patterns**, clients can (temporarily) obtain ownership of threads from the pool, rather than using addTask
- Abstract away actual threads through **thread IDs**
- Once thread obtained, repeat Steps 2–3 as often as necessary

```
1  class ExecutorThreadPool {
2    //Storage for threads removed from pool by obtainThread
3    std::vector<FunctionExecutorThread*> occupiedThreads;
4
5    //Step 1: obtain a thread's ID, removing it from the pool
6    void obtainThread(threadID& id);
7
8    //Step 2: execute a task on a particular thread
9    void executeTask(threadID id, std::function<void()>& f);
10
11   //Step 3 (optional): wait for thread to become idle
12   void waitForThread(threadID id);
13
14   //Step 4: return thread to pool (waits before returning)
15   void returnThread(threadID id);
16 }
```

Outline

Fork-Join with ExecutorThreadPool

```
1  void mergeSort(int* A, int i, int j) {
2      if (j <= i) { return; }
3      int k = i + (j-1)/2;
4      mergeSort(A, i, k);
5      mergeSort(A, k, j);
6      merge(A, i, k, j);
7  }
```

```
1  void mergeSort(int* A, int i, int j) {
2      if (j <= i) { return; }
3      int k = i + (j-1)/2;
4      threadID id;
5      ExecutorThreadPool& pool = getThreadPool();
6
7      pool.obtainThread(id);
8      pool.executeTask(id, std::bind(mergeSort, A, i, k));
9      mergeSort(A, k, j);
10
11     pool.returnThread(id);
12     merge(A, i, k, j);
13 }
```

Workpile with ExecutorThreadPool

```
1  void processInt(std::queue<int> B, int a) {
2      a -= 10;
3      if (a > 0) {
4          getThreadPool().addTask(std::bind(processInt, B, a));
5      } else {
6          B.push(a);
7      }
8  }
9
10 void WorkpileExample(std::queue<int> B, std::queue<int> A) {
11     ExecutorThreadPool& pool = getThreadPool();
12     while (!A.empty()) {
13         pool.addTask( std::bind(processInt, B, A.front()) );
14         A.pop();
15     }
16     pool.waitForAllThreads();
17 }
```

AsyncGenerator and AsyncObjectStream

We want an *object-oriented* approach to create and use generators.

`AsyncObjectStream` already solves the producer-consumer problem.

- It provides a queue which blocks and notifies the consumer as data is produced, implemented using a condition variable
- As a class template, can be used within `AsyncGenerator` to yield any type of object

```
1  template <class Object>
2  class AsyncObjectStream {
3      void addResult(Object&& res); //Producer
4
5      void resultsFinished(); //Producer
6
7      bool getNextObject(Object& res); //Consumer
8
9      void streamEmpty(); //Consumer
10 };
```

AsyncGenerator

`AsyncGenerator` is itself a class template, templated by `Object`, the type of object to generate.

- The `AsyncGenerator` acts as interface between producer and consumer
- The consumer constructs the `AsyncGenerator`, passing the constructor the producer's function and arguments
- The producer's signature should be:

```
1  void producerFunction(..., AsyncGenerator<Object>&);
```

- The `AsyncGenerator` being constructed inserts itself into the producer's list of arguments so that it has reference to the generator object

AsyncGenerator example

```
1  void FibonacciGen(int n, AsyncGenerator<int>& gen) {
2      int Fn_1 = 0;
3      int Fn = 1;
4      for (int i = 0; i < n; ++i) {
5          gen.generateObject(Fn_1); //yield Fn_1 and continue
6          Fn = Fn + Fn_1;
7          Fn_1 = Fn - Fn_1;
8      }
9      gen.setComplete();
10 }
11
12 void Fib() {
13     int n;
14     std::cin >> n;
15     AsyncGenerator<int> gen(FibonacciGen, n);
16
17     int fib;
18     //get one integer at a time until generator is finished
19     while (gen.getNextObject(fib)) {
20         std::cerr << fib << std::endl;
21     }
22 }
```

Cooperative parallelism

With several simultaneous clients of `ExecutorThreadPool` (workpile, fork-join, generators), some tasks should be given priority.

- Some tasks are more **coarse-grained**, offer more potential speed-up
- Some tasks may expose more parallelism and should be executed first

Often, parallelism coming from Fork-Join or Map is preferred over Producer-Consumer.

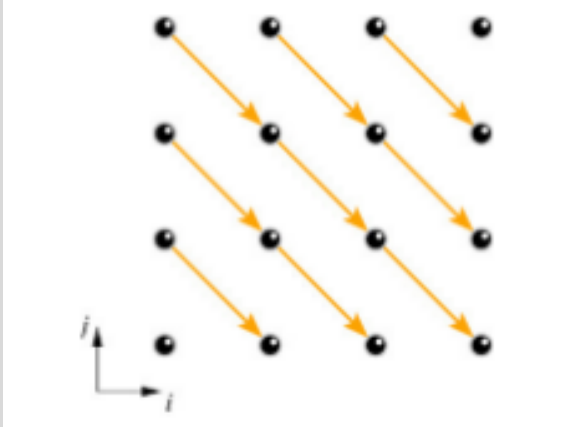
- **Goal**: allow Fork-Join and Map to access thread pool threads over Producer-Consumer while still keeping the latter possible when there are idle threads
- **Solution**: **priority tasks**
- `addTask()` vs `addPriorityTask()`
- If all threads busy, `addPriorityTask()` temporarily spawns new thread to start execution immediately

Outline

Automatic parallelization: plain multiplication

Serial dense univariate polynomial multiplication

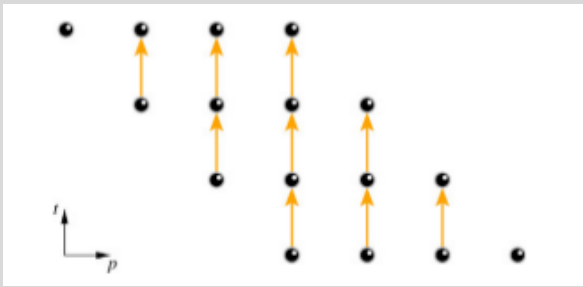
```
for(i=0; i<=n; i++){
  c[i] = 0; c[i+n] = 0;
  for(j=0; j<=n; j++)
    c[i+j] += a[i] * b[j];
}
```



Dependence analysis suggests to set $t(i,j) = n - j$ and $p(i,j) = i + j$.

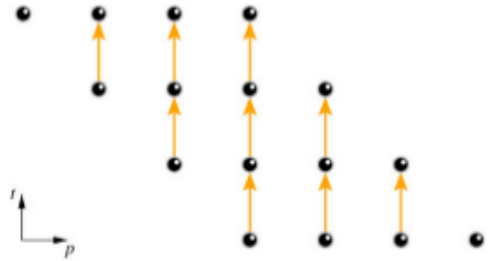
Asynchronous parallel dense univariate polynomial multiplication

```
parallel_for (p=0; p<=2*n; p++){
  c [ p ] =0;
  for (t=max(0,n-p); t<= min(n,2*n-p);t++)
    C [ p ]  = C [ p ]
     + A [ t+p-n ]  * B [ n-t ] ;
}
```



Generating parametric code & use of tiling techniques

```
parallel_for (p=0; p<=2*n; p++){
  c [ p ] =0;
  for (t=max(0,n-p); t<= min(n,2*n-p);t++)
    C [ p ]  = C [ p ]
      + A [ t+p-n ]  * B [ n-t ] ;
}
```



Improving the parallelization

- The above generated code is not practical for multicore implementation: the number of processors is in $\Theta(n)$. (Not to mention poor locality!) and the work is unevenly distributed among the workers.
- We group the virtual processors (or threads) into 1D blocks, each of size B. Each thread is known by its block number b and a local coordinate u in its block.
- Blocks represent good units of work which have good locality property.
- This yields the following constraints: $0 \leq u < B,\ p = bB + u$.

Generating parametric code: using tiles

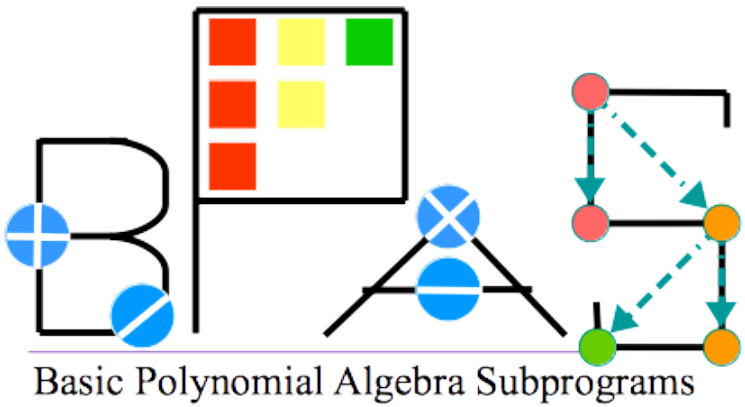
We apply `RegularChains:-QuantifierElimination` on the left system (in order to get rid off i, j) leading to the relations on the right:

$$\left\{\begin{array}{c} o < n \\ 0 \leq i \leq n \\ 0 \leq j \leq n \\ t = n - j \\ p = i + j \\ 0 \leq b \\ o \leq u < B \\ p = bB + u, \end{array}\right. \qquad \left\{\begin{array}{c} B > 0 \\ n > 0 \\ 0 \leq b \leq 2n/B \\ 0 \leq u < B \\ 0 \leq u \leq 2n - Bb \\ p = bB + u, \end{array}\right. \tag{1}$$

From where we derive the following program:

```
for (p=0; p<=2*n; p++) c [ p ] = 0;
parallel_for (b=0; b<= 2 n / B; b++) {
    parallel_for (u=0; u<=min(B-1, 2*n - B * b); u++) {
        p = b * B + u;
        for (t=max(0,n-p); t<=min(n,2*n-p) ;t++)
            c [ p ] = c [ p ] + a [ t+p-n ] * b [ n-t ];
    }
}
```

Thank You!



http://www.bpaslib.org/

References

[1] M. Asadi, A. Brandt, C. Chen, S. Covanov, F. Mansouri, D. Mohajerani, R. H. C. Moir, M. Moreno Maza, D. Talaashrafi, L. Wang, N. Xie, and Y. Xie. *Basic Polynomial Algebra Subprograms (BPAS)*. www.bpaslib.org. 2021.

[2] M. Asadi, A. Brandt, R. H. C. Moir, M. M. Maza, and Y. Xie. "On the parallelization of triangular decompositions". In: *International Symposium on Symbolic and Algebraic Computation (ISSAC '20), Kalamata, Greece, July 20-23, 2020*. ACM, 2020, pp. 22–29.

[3] M. Asadi, A. Brandt, R. H. C. Moir, and M. Moreno Maza. "Algorithms and Data Structures for Sparse Polynomial Arithmetic". In: *Mathematics* 7.5 (2019), p. 441.

[4] M. Asadi, A. Brandt, R. H. C. Moir, M. Moreno Maza, and Y. Xie. "Parallelization of Triangular Decompositions: Techniques and Implementation". In: *J. Symb. Comput.* (2021). *(to appear).*

[5] G. Attardi and C. Traverso. "Strategy-Accurate Parallel Buchberger Algorithms". In: *J. Symbolic Computation* 22 (1996), pp. 1–15.

[6] J. Böhm, W. Decker, S. Laplagne, G. Pfister, A. Steenpaß, and S. Steidel. "Parallel algorithms for normalization". In: *J. Symb. Comput.* 51 (2013), pp. 99–114.

[7] A. Brandt, M. Kazemi, and M. Moreno Maza. "Power Series Arithmetic with the BPAS Library". In: *Computer Algebra in Scientific Computing (CASC '20)*. Vol. 12291. LNCS. Springer, 2020, pp. 108–128.

[8] A. Brandt and M. M. Maza. "On the Complexity and Parallel Implementation of Hensel's Lemma and Weierstrass Preparation". In: *CoRR* abs/2105.10798 (2021). arXiv: 2105.10798. URL: https://arxiv.org/abs/2105.10798.

[9] A. Brandt, R. H. C. Moir, and M. Moreno Maza. "Employing C++ Templates in the Design of a Computer Algebra Library". In: *Mathematical Software - ICMS 2020, Braunschweig, Germany, July 13-16, 2020*. Vol. 12097. LNCS. Springer, 2020, pp. 342–352.

[10] A. Brandt and M. Moreno Maza. "On the Complexity and Parallel Implementation of Hensel's Lemma and Weierstrass Preparation". In: *Computer Algebra in Scientific Computing (CASC '21)*. (*To appear*). 2021.

[11] B. Buchberger. "The parallelization of critical-pair/completion procedures on the L-Machine". In: *Proceedings of the Japanese Symposium on functional programming*. 1987, pp. 54–61.

[12] C. Chen and M. Moreno Maza. "Algorithms for computing triangular decomposition of polynomial systems". In: *J. Symb. Comput.* 47.6 (2012), pp. 610–642.

[13] C. Chen, M. Moreno Maza, and Y. Xie. "Cache Complexity and Multicore Implementation for Univariate Real Root Isolation". In: *J. of Physics: Conference Series* 341 (2011).

[14] M. F. I. Chowdhury, M. M. Maza, W. Pan, and É. Schost. "Complexity and Performance Results for non FFT-based Univariate Polynomial Multiplication.". In: *Proceedings of Advances in mathematical and computational methods: addressing modern of science, technology, and society, AIP conference proceedings*. volume 1368. 2011, pp. 259–262.

[15] J. Della Dora and J. Fitch, eds. *Computer Algebra and Parallelism, First International Workshop, Grenoble, France, September 1989*. Academic Press, 1988. ISBN: 978-0122090424.

[16] J. Dumas, E. L. Kaltofen, and C. Pernet, eds. *Proceedings of the 2015 International Workshop on Parallel Symbolic Computation, PASCO 2015, Bath, United Kingdom, July 10-12, 2015*. ACM, 2015. ISBN: 978-1-4503-3599-7.

[17] S. Eyerman, J. E. Smith, and L. Eeckhout. "Characterizing the branch misprediction penalty". In: *2006 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2006, March 19-21, 2006, Austin, Texas, USA, Proceedings*. IEEE Computer Society, 2006, pp. 48–58.

[18] J. C. Faugere. "Proceedings of the 1st International Workshop on Parallel Symbolic Computation, PASCO 1994, Linz, Austria". In: vol. 5. World Scientific. 1994, p. 124.

[19] J. Faugère, M. B. Monagan, and H. Loidl, eds. *Proceedings of the 2017 International Workshop on Parallel Symbolic Computation, PASCO 2017, Kaiserslautern, Germany, July 23-24, 2017*. ACM, 2017. ISBN: 978-1-4503-5288-8.

[20] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. "Cache-Oblivious Algorithms". In: *ACM Transactions on Algorithms* 8.1 (2012).

[21] M. Gastineau and J. Laskar. "Highly Scalable Multiplication for Distributed Sparse Multivariate Polynomials on Many-Core Systems". In: *Computer Algebra in Scientific Computing (CASC '13)*. Vol. 8136. LNCS. Springer, 2013, pp. 100–115.

[22] M. Gastineau and J. Laskar. "Parallel sparse multivariate polynomial division". In: *Proceedings of the 2015 International Workshop on Parallel Symbolic Computation, PASCO 2015*. ACM, 2015, pp. 25–33.

[23] J. von zur Gathen and J. Gerhard. *Modern Computer Algebra (3. ed.)* Cambridge University Press, 2013. ISBN: 978-1-107-03903-2.

[24] S. A. Haque, M. M. Maza, and N. Xie. "A Many-Core Machine Model for Designing Algorithms with Minimum Parallelism Overheads". In: *Parallel Computing: On the Road to Exascale, Proceedings of the International Conference on Parallel Computing, ParCo 2015, 1-4 September 2015, Edinburgh, Scotland, UK*. Vol. 27. Advances in Parallel Computing. IOS Press, 2015, pp. 35–44.

[25] S. A. Haque, M. M. Maza, and N. Xie. "A Many-core Machine Model for Designing Algorithms with Minimum Parallelism Overheads". In: *CoRR* abs/1402.0264 (2014). arXiv: 1402.0264. URL: http://arxiv.org/abs/1402.0264.

[26] H. Hong, ed. *Proceedings of the 1st International Workshop on Parallel Symbolic Computation, PASCO 1994, Linz, Austria*. World scientific, 1994. ISBN: 978-9810220402.

[27] H. Hong, E. Kaltofen, and M. A. Hitz, eds. *Proceedings of the 2nd International Workshop on Parallel Symbolic Computation, PASCO 1997, July 20-22, 1997, Kihei, Hawaii, USA*. ACM, 1997. ISBN: 0-89791-951-3.

[28] S. Hong and H. Kim. "An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness". In: *Proc. of ISCA*. 2009, pp. 152–163.

[29] J. Hu and M. B. Monagan. "A Fast Parallel Sparse Polynomial GCD Algorithm". In: *International Symposium on Symbolic and Algebraic Computation (ISSAC '16), Waterloo, ON, Canada, July 19-22, 2016*. 2016, pp. 271–278.

[30] L. Ma, K. Agrawal, and R. D. Chamberlain. "A memory access model for highly-threaded many-core architectures". In: *Future Generation Comp. Syst.* 30 (2014), pp. 202–215.

[31] M. M. Maza and Y. Xie. "FFT-Based Dense Polynomial Arithmetic on Multi-cores". In: *High Performance Computing Systems and Applications, 23rd International Symposium, HPCS 2009*. Vol. 5976. LNCS. Springer, 2009, pp. 378–399.

[32] M. McCool, J. Reinders, and A. Robison. *Structured parallel programming: patterns for efficient computation*. Elsevier, 2012.

[33] M. B. Monagan and R. Pearce. "Parallel sparse polynomial multiplication using heaps". In: *International Symposium on Symbolic and Algebraic Computation, (ISSAC '09), Seoul, Republic of Korea, July 29-31, 2009*. ACM, 2009.

[34] M. B. Monagan and R. Pearce. "Sparse polynomial division using a heap". In: *J. Symb. Comput.* 46.7 (2011), pp. 807–822.

[35] M. B. Monagan and B. Tuncer. "Sparse Multivariate Hensel Lifting: A High-Performance Design and Implementation". In: *ICMS 2018 - 6th International Conference, South Bend, IN, USA, July 24-27, 2018, Proceedings*. 2018, pp. 359–368.

[36] M. Moreno Maza and J. Roch, eds. *Proceedings of the 4th International Workshop on Parallel Symbolic Computation, PASCO 2010, July 21-23, 2010, Grenoble, France*. ACM, 2010. ISBN: 978-1-4503-0067-4.

[37] M. Moreno Maza and S. M. Watt, eds. *Proceedings of the 3rd International Workshop on Parallel Symbolic Computation, PASCO 2007, 27-28 July 2007, London, Ontario, Canada*. ACM, 2007. ISBN: 978-1-59593-741-4.

[38] M. Moreno Maza and Y. Xie. "Balanced Dense Polynomial Multiplication on Multi-Cores". In: *Int. J. Found. Comput. Sci.* 22.5 (2011), pp. 1035–1055.

[39] B. D. Saunders, H. R. Lee, and S. K. Abdali. "A parallel implementation of the cylindrical algebraic decomposition algorithm". In: *ISSAC*. Vol. 89. 1989, pp. 298–307.

[40] R. Zippel, ed. *Computer Algebra and Parallelism, Second International Workshop, Ithaca, USA, May 9-11, 1990*. Vol. 584. LNCS. Springer, 1992. ISBN: 3-540-55328-2.