On The Parallelization of Triangular Decompositions

Mohammadali Asadi, **Alexander Brandt**,
Robert H.C. Moir, Marc Moreno Maza

Ontario Research Center for Computer Algebra
Department of Computer Science
University of Western Ontario, Canada

Wednesday April 10, 2019

A First Triangular Decomposition

A **triangular decomposition** is an of analogue Gaussian elimination for polynomial systems; a method for solving systems of polynomial equations.

$$\begin{cases} 2x + y + z = 1 \\ x + 2y + z = 1 \\ x + y + 2z = 1 \end{cases} \xRightarrow{Gaussian\ Elim.} \left[\begin{array}{ccc|c} 2 & 1 & 1 & 1 \\ 0 & \frac{3}{2} & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & \frac{4}{3} & \frac{1}{3} \end{array}\right]$$

$$\xRightarrow{Tri.\ Decomp.} \begin{cases} x - \frac{1}{4} = 0 \\ y - \frac{1}{4} = 0 \\ z - \frac{1}{4} = 0 \end{cases}$$

The decomposition has a *triangular shape* by its pair-wise different main variables $\implies$ **Triangular Set**.

↳ Main var: the largest variable in a polynomial, given an order (e.g. $x > y > z$)

A Decomposition of a Non-Linear System

$$\begin{cases} x^2 + y + z = 1 \\ x + y^2 + z = 1 \\ x + y + z^2 = 1 \end{cases} \xRightarrow{Gr\ddot{o}bner\ \ Basis} \begin{cases} x + y + z^2 = 1 \\ (y + z - 1)(y - z) = 0 \\ z^2 \left(z^2 + 2y - 1\right) = 0 \\ z^2 \left(z^2 + 2z - 1\right)(z - 1)^2 = 0 \end{cases}$$

$$\begin{cases} x - z = 0 \\ y - z = 0 \\ z^2 + 2z - 1 = 0 \end{cases}, \begin{cases} x = 0 \\ y = 0 \\ z - 1 = 0 \end{cases}, \begin{cases} x = 0 \\ y - 1 = 0 \\ z = 0 \end{cases}, \begin{cases} x - 1 = 0 \\ y = 0 \\ z = 0 \end{cases}$$

The Gröbner bases solution is equivalent (via a union) of the solutions of its decomposition into four triangular sets.

↳ This system decomposes into multiple **components**.

Outline

Introductory Definitions

- A polynomial belongs to a **polynomial ring** composed of a *ground ring* and an (ordered) set of variables.
  - ↳ e.g. $\mathbb{K}[x_1 < x_2 < \ldots < x_v]$
  - ↳ $\mathbb{K}$ is the ground ring, say $\mathbb{Q}$, $x_1, \ldots, x_v$ are variables.
  - ↳ X can be short-hand for $x_1, \ldots, x_v$.

- A set of polynomials $F \subset \mathbb{K}[x_1 < x_2 < \ldots < x_v]$ can form **system of equations** by setting $f = 0$ for each $f \in F$.

- The **algebraic variety** of F is the geometric sense of the set of solutions of F.
  - ↳ $V(F) = \{(a_1, \ldots, a_v) \in \mathbb{K}^n \mid f(a_1, \ldots, a_v) = 0 \ \forall f \in F\}$
  - ↳ This assumes $\mathbb{K}$ is algebraically closed.
  - ↳ e.g. $x^2 + 1 = 0$ has no solution in $\mathbb{R}$ but does in $\mathbb{C}$.

Some Notations

- A polynomial p has:
  - ↳ a main variable, mvar(p), and
  - ↳ an initial, init(p), which is the leading coefficient of p with respect to the main variable.
  - ↳ a tail, tail(p), the terms left after setting its initial to 0.
  - ↳ e.g. for $x > y > z$, $p = yx^2 + x^2 + z$
    mvar(p) = x, init(p) $= y + 1$, tail(p) $= z$.

- A triangular set T is an ordered set of polynomials $T = \{t_1, t_2, \ldots, t_v\}$ where t_i are ordered by their (pairwise distinct) main variables.
  - ↳ T_x "picks out" the polynomial in T with main variable x.
  - ↳ mvar(T) is the set of main variables of the polynomials in T.

Triangular Sets and Regular Chains

$$F_1 = \begin{cases} yx - 1 = 0 \\ y = 0 \\ z - 1 = 0 \end{cases}$$

- This set is inconsistent; there are no solutions.
- Back-substituting $y = 0$ yields $-1 = 0$.

$$F_2 = \begin{cases} (y+1)x^2 - x = 0 \\ y^2 - 1 = 0 \\ z - 1 = 0 \end{cases}$$

- For $y = -1$, x has 1 solution.
- For $y = 1$, x has 2 solutions.

Both are triangular sets, neither are **regular chains**. Regular Chains are special triangular sets without these "issues".

↳ In a regular chain the leading coefficients (initials) of each polynomial must not be zero (e.g. F_1) nor a *zero-divisor* (e.g. F_2).

↳ e.g. the initial of $yx - 1$ is y, the initial of $(y+1)x^2 - x$ is $y + 1$.

Regular Chains

A triangular set $T = \{T_1 < T_2 < \ldots < T_n\}$, ordered by main variable, is a **regular chain** if the initial of every polynomial in the set is **regular** with respect to the polynomials less than ("below") it.

↳ More mathematically, a TS is a RC if the initial of polynomial T_i is regular modulo the saturated ideal of $T_1, \ldots, T_{i-1}$ for $i = 2, \ldots, n$.

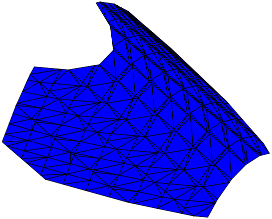
A polynomial is **regular** modulo $\langle F \rangle$ if it is neither 0 nor a zero-divisor modulo $\langle F \rangle$.

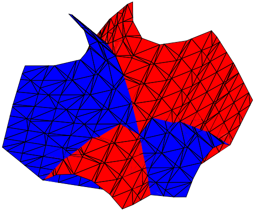
A polynomial p is a **zero-divisor** modulo $\langle F \rangle$ if there exists q such that $pq \in \langle F \rangle$ but $p, q \notin \langle F \rangle$. e.g.:

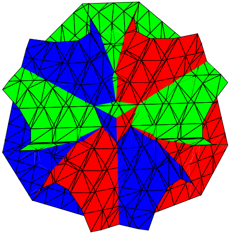
$$F = \begin{cases} y^2 - 1 \\ z - 1 \end{cases}, \qquad p = y + 1,\ q = y - 1 \implies pq = y^2 - 1 = 0 \mod \langle F \rangle$$

Outline

Incremental Solving





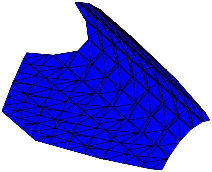


$$F^{(1)} = \left\{ \; x^2 + y + z = 1 \right.$$

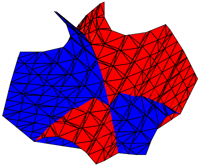
$$F^{(2)} = \begin{cases} x^2 + y + z = 1 \\ x + y^2 + z = 1 \end{cases}$$

$$F^{(3)} = \begin{cases} x^2 + y + z = 1 \\ x + y^2 + z = 1 \\ x + y + z^2 = 1 \end{cases}$$

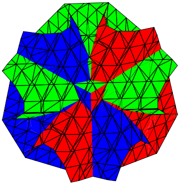
Incremental Solving in Triangular Decomposition



$$T^{(1)} = \text{Intersect}(\varnothing, x^2 + y + z = 1)$$
$$T^{(1)} = \left\{ \; x^2 + y + z = 1 \right.$$



$$T^{(2)} = \text{Intersect}(T^{(1)},\; x + y^2 + z = 1)$$
$$T^{(2)} = \left\{ \begin{array}{r} x + y^2 + z = 1 \\ y^4 + (2z-2)y^2 + y + (z^2 - z) = 0 \end{array} \right.$$



$$T^{(3)} = \text{Intersect}(T^{(2)},\; x + y + z^2 = 1)$$
$$T_1^{(3)} = \left\{ \begin{array}{r} x + y = 1 \\ y^2 - y = 0 \\ z = 0 \end{array} \right. \qquad T_2^{(3)} = \left\{ \begin{array}{r} 2x + z^2 = 1 \\ 2y + z^2 = 1 \\ z^3 + z^2 - 3z = -1 \end{array} \right.$$

Triangularization by Intersection

Triangularize(F)

Input: $F = \{f_1, f_2, \ldots, f_n\}$, Output: a triangular decomposition of F.

$\mathcal{T} \leftarrow \{\varnothing\}$
for $i = 1 \mathinner{..} n$ **do**
    $\mathcal{T}' \leftarrow \varnothing$
    for $T \in \mathcal{T}$ **do**
        $\mathcal{T}' \leftarrow \mathcal{T}' \cup \text{Intersect}(f_i, T)$
    end for
    $\mathcal{T} \leftarrow \mathcal{T}'$
end for
return $\mathcal{T}$

An Intersect Which Splits

To Intersect(f, T), we must first *regularize* f.

- Since $x \notin \text{mvar}(T)$ we regularize $\text{init}(f) = y + 1$.
- Via GCD computation we discover $y + 1$ is a zero-divisor modulo T.
- Since $T_y = g(T_y/g)$, T splits into:
  - ↳ T_1 (replacing T_y with $g = \gcd(\text{init}(f), T_y) = y + 1$, and
  - ↳ T_2 (replacing T_y by its quotient with g).
  - ↳ We can then work with T_1 and T_2 independently due to the Chinese remaindering theorem: $\mathbb{K}[X]/\langle T\rangle \simeq \mathbb{K}[X]/\langle T_1\rangle \bigoplus \mathbb{K}[X]/\langle T_2\rangle$

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

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

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

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

A Simple Intersect

IntersectFree(f, T)
Input: f a polynomial, T a regular chain, mvar(f) $\notin$ mvar(T)
Output: $T_1, \ldots, T_e$ such that $\bigcup_{i=1}^{e} T_i$ "encode the solutions" of f together with T.

for $p, C \in$ Regularize(init(f), T) **do**
    if $p = 0$ **then** yield Intersect(tail(f), C)
    else
        yield $C \cup f$
        for $D \in$ Intersect(init(f), C) **do**
            yield $\mathcal{T}$ $\cup$ Intersect(tail(f), E)

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

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

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

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

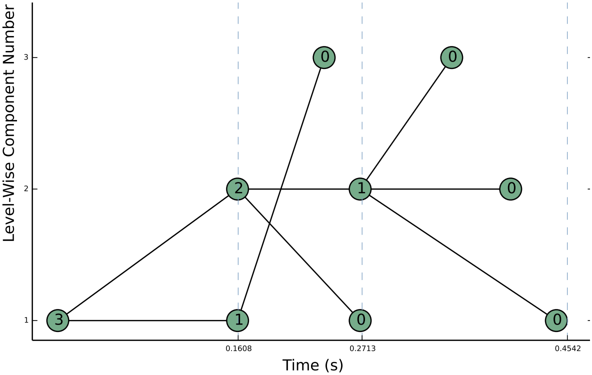
Opportunities For Parallelization

Regular chains may split due to:

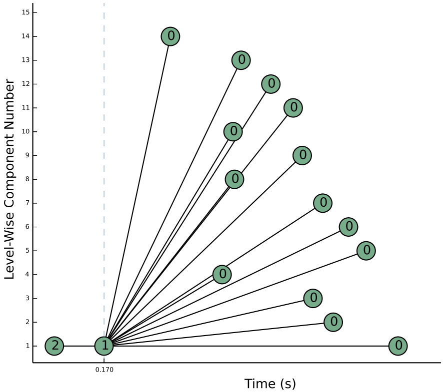
- Case discussions from zero-divisors or factorizations.

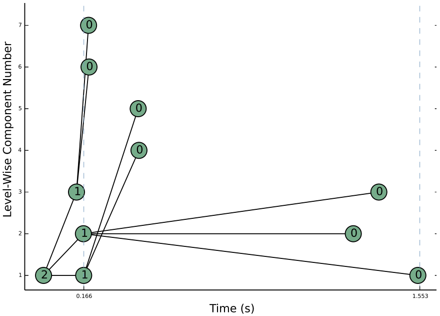
$$T = \begin{cases} y^2 - 3y + 2 = 0 \\ z - 1 = 0 \end{cases} \implies \begin{cases} y - 2 = 0 \\ z - 1 = 0 \end{cases} \cup \begin{cases} y - 1 = 0 \\ z - 1 = 0 \end{cases}$$

- **The geometry of the problem.** Triangularize intersects in parallel.



Parallelization Challenges (1/2)

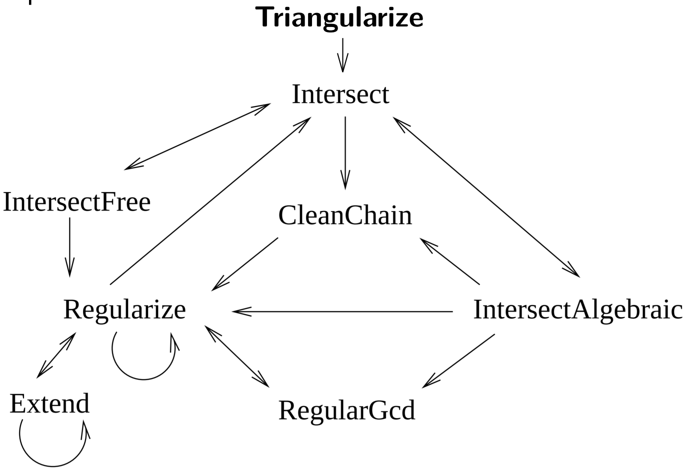




- Computations may not split until the very end.
- Splitting may be very unbalanced.

Parallelization Challenges (2/2)

The sub-algorithms for triangular decomposition are highly recursive, and mutually dependent.



Parallelism Adaptive to Geometry

The parallelization must be dynamic and adaptive to the geometry of the problem currently being solved.

- No simple "divide-and-conquer".

Coarse-grained parallelism in the top-level **Triangularize** algorithm as it calls **Intersect**.

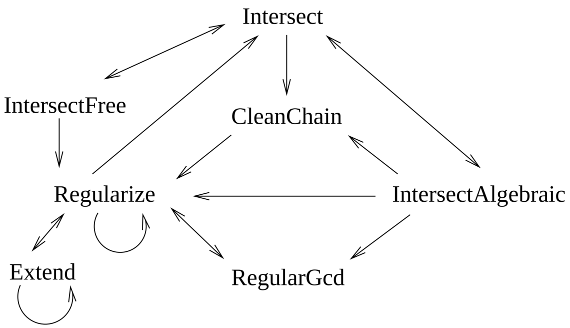
↳ One thread for one branch of the solution space.

↳ But what about load balancing? One branch has all the work?

Triangularize(F)

```
𝒯 ← {∅}
for i = 1..n do
    𝒯′ ← ∅
    for T ∈ 𝒯 do
        𝒯′ ← 𝒯′ ∪ Intersect(f_i, T)
    end for
    𝒯 ← 𝒯′
end for
return 𝒯
```

Parallelism Adaptive to Load-Balancing



- Regularize is the core function which actually causes splits.
  - ↳ But, *all roads lead to regularize*.
  - ↳ Hence, all methods return a list of components.
- Preferably, each method would return a *stream* of components, one by one, as they are computed.

Outline

Basic Polynomial Algebra Subprograms

Basic Polynomial Algebra Subprograms

Our triangular decomposition algorithm is implemented within the open-source BPAS library.

- Low-level routines written in C for efficiency.
- A clean user interface is provided via a C++ wrapper.
- BPAS provides low-level parallelism in polynomial arithmetic, but here we only discuss parallelism within triangular decompositions.

Coarse-Grained Parallelism

- Coarse-grained parallelism is achieved by spawning threads as required.
- At one "level" we have one polynomial to intersect with k components, yielding $k-1$ threads and k concurrent runs.
- InterParallel takes a thread-safe container $\mathcal{T}'$ to accumulate results.
- Our implementation uses `pthreads`.

TriangularizeParallel(F)
Input: $F = \{f_1, f_2, \ldots, f_n\}$, Output: a triangular decomposition of F.

$\mathcal{T} \leftarrow \{\varnothing\}$
for $i = 1 \,..\, n$ **do**
    $\mathcal{T}' \leftarrow \varnothing$; $k \leftarrow |\mathcal{T}|$
    for $i = 1 \,..\, k-1$ **do**
        spawn InterParallel$(f_i, \mathcal{T}[i], \mathcal{T}')$
    InterParallel$(f_n, \mathcal{T}[k], \mathcal{T}')$
    join()
return $\mathcal{T}$

Fine-Grained Parallelism

Each low-level routine should *stream* components between themselves rather than accumulating them all into a single list before returning.

↳ This is only beneficial if different routines can execute simultaneously.

We implement each algorithm as a **coroutine** (i.e. generator).

- Use commonly in cooperative tasks, iterators, pipes.
- A generalization of the classic **producer-consumer** paradigm.
- Every method is both a producer and a consumer, and at different levels of recursion.
- Each coroutine (potentially) runs asynchronously, returning components one at a time.
- **The caller can continue processing one component while the callee accumulates more components.**

The AsyncGenerator class

```
template <class Object>
class AsyncGenerator {

    /* Create a generator from a function call. */
    AsyncGenerator(Function f, Args.. args);

    /* Add a new object to the generated. */
    virtual void generateObject(Object obj) = 0;

    /**
     * Finalize the AsyncGenerator by declaring it has
     * finished generating all possible objects.
     */
    virtual void setComplete() = 0;

    /**
     * Obtain the next Object which was generated by reference.
     * returns false iff no more objects available
     */
    virtual bool getNextObject(Object obj) = 0;

};
```

Inner-Workings of `AsyncGenerator`

General design:

- The consumer creates the AsyncGenerator by passing it a function and its arguments.
- The AsyncGenerator inserts itself into the function call arguments.
- The AsyncGenerator (possibly) spawns a thread to call function.
- The producer produces output via the AsyncGenerator parameter.

```
void Regularize(Poly p, RegChain T, AsyncGen<Poly,RegChain> res) {
    //...
    res.generatreObject(next);
}

void IntersectFree(Poly p, RegChain T, AsyncGen<RegChain> res) {
    AsyncGen<Poly,RegChain> regularRes(Regularize, p.initial(), T);
    Poly,RegChain next;
    while (regularRes.getNextObject(next)) {
        //...
    }
}
```

Optimizing The Generators

Due to the highly recursive nature of these algorithms, we look to mitigate the cost of continuously creating and joining threads for each method call.

- In large examples, *tens of thousands* of inner functions calls occur.
- A thread-pool is a classic solution.
- We create a thread pool of **FunctionExectors**.
- When pool is dry, execute serially.

Each FunctionExecutor implements an **event loop**, waiting for function objects to be passed to it and executed on that thread.

- Functions are not first-class objects C/C++.
- Function pointers are not generic enough.
- With C++11 finagling we can make it work.
- `std::bind` unifies functions arguments.
- `std::function` wraps functions pointers.

Outline

Experimentation Setup

Thanks to **Maple**, an industrial partner and computer algebra system, we have a collection of over 3000 real-world systems.

- Systems come from actual user data, literature, bug reports.

These experiments are run on a node with 2x6-core Intel Xeon X560 processors at 2.67 GHz, 32KB L1 data ache, 256KB L2 cache, 48 GB of RAM.

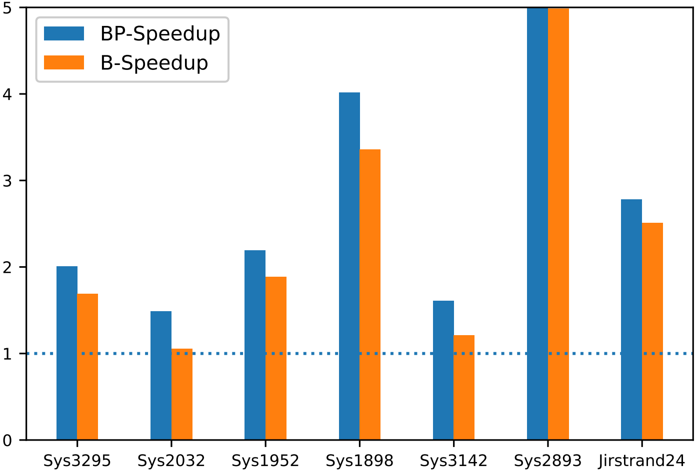
Important to note, these problems *do not* necessarily scale with the number of processors.

- Potential speed-up is fully dictated by the **geometry of the problem**.

A First Comparison

Coarse- and fine-grained parallelism vs coarse-grained parallelism alone.

- "P" stands for with `AsyncGenerator` thread pool.



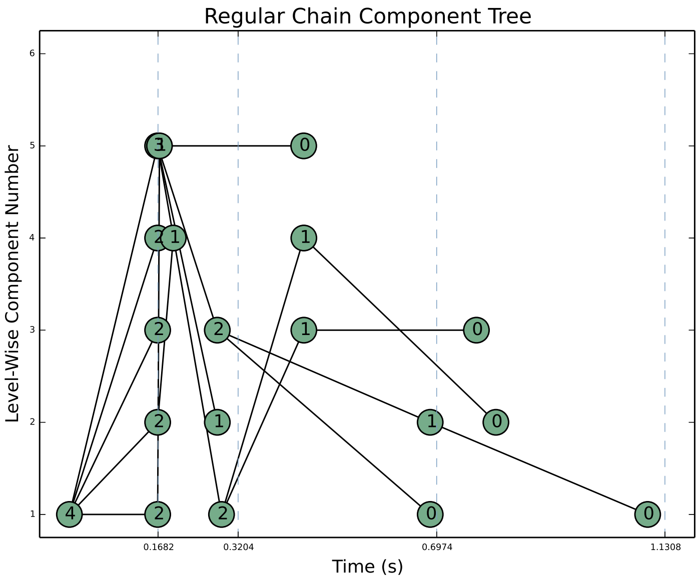
A Real-World System

Sys2893, “Fee-1”.

$$
\begin{cases}
-2qp - 2p^2 - 2q + 8p - 2 \\[1em]
-3qcp + 2qpd + 4p^2d + 3cp + qd - 7pd \\[1em]
q^2c^2 - 2q^2cd - 2qcpd + q^2d^2 + 2qpd^2 + p^2d^2 - 2qc^2 + 4qcp + 2qcd + 2cpd \\
-4qd^2 - 4pd^2 + c^2 + 2qp + 10p^2 - 4cd + 4d^2 - 2q - 8p + 2 \\[1em]
3q^2c^2 + 12qcpd - 3q^2d^2 + 6qpd^2 - 3p^2d^2 - 6qc^2 + 12qcd + 12cpd \\
-4q^2 + 3c^2 + 5p^2 - 12cd + 12d^2 - 6p + 5
\end{cases}
$$

It’s solution has 3 components, polynomials with 1000-bit coefficients and 100 terms.

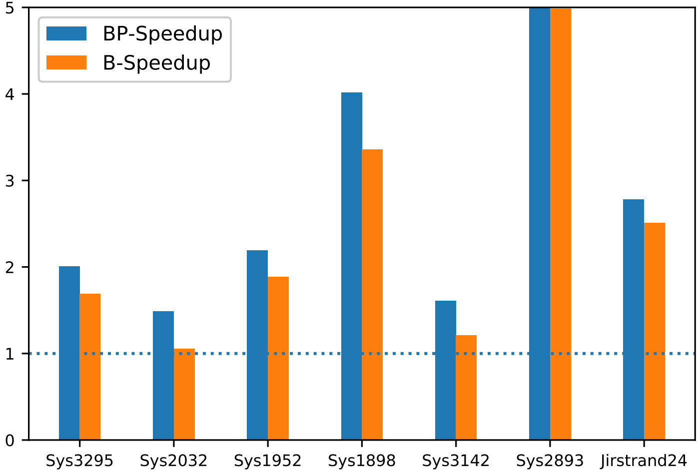
Inspecting the Geometry: Sys2893



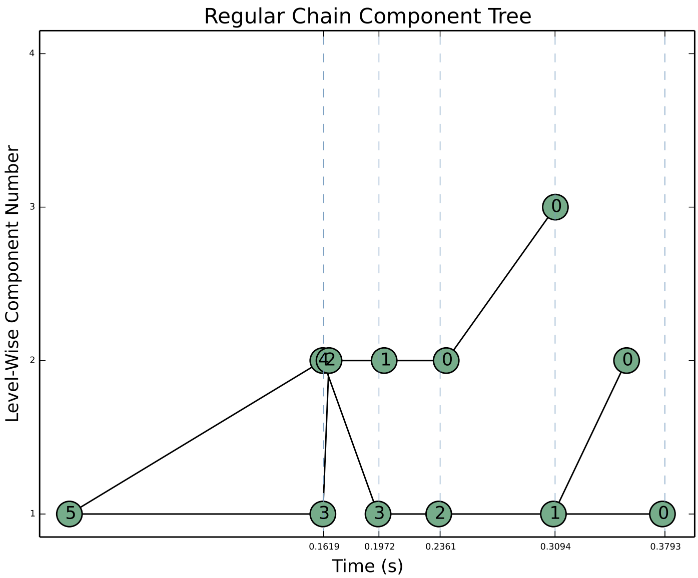
A First Comparison (Again)

Coarse- and fine-grained parallelism vs coarse-grained parallelism alone.

- "P" stands for with `AsyncGenerator` thread pool.



Inspecting the Geometry: Sys3142



Future Work

- Explore further opportunities in parallelizing low-level operations like GCD computations, factorizations, and polynomial arithmetic.

- Examine patterns in the geometry of solutions of different systems to categorize systems and assign various decomposition flavours.

- Continue the tuning of parallelism and performance, particularly in data transfer between threads and coroutines.
  - ↳ Polynomials are not small objects.

Thank you!

Questions?