Parallelization of Triangular Decompositions: Design and Implementation with the BPAS library

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

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

Sage/Oscar Days, July 19, 2021

Outline

1. Introduction
2. Preliminaries
3. Triangularize: task pool parallelization
4. Intersect: Asynchronous Generators
5. Removing Redundancies: Divide-and-Conquer
6. Experimentation

Decomposing a non-linear system

Many ways to “solve” a polynomial system

$$\left\{\begin{array}{r} x^2+y+z=1 \\ x+y^2+z=1 \\ x+y+z^2=1 \end{array}\right. \xRightarrow{\text{Gröbner basis}} \left\{\begin{array}{r} x+y+z^2=1 \\ (y+z-1)(y-z)=0 \\ z^2\left(z^2+2y-1\right)=0 \\ z^2\left(z^2+2z-1\right)(z-1)^2=0 \end{array}\right.$$

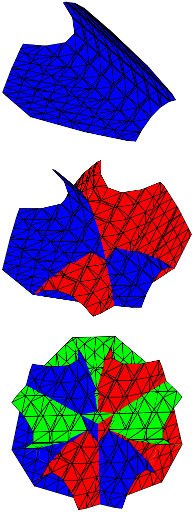
$$\Big\Downarrow \ \text{Triangular Decomposition}$$

$$\left\{\begin{array}{r} x-z=0 \\ y-z=0 \\ z^2+2z-1=0 \end{array}\right., \quad \left\{\begin{array}{r} x=0 \\ y=0 \\ z-1=0 \end{array}\right., \quad \left\{\begin{array}{r} x=0 \\ y-1=0 \\ z=0 \end{array}\right., \quad \left\{\begin{array}{r} x-1=0 \\ y=0 \\ z=0 \end{array}\right.$$

Both solutions are equivalent.

→ by using triangular decomposition, **multiple components** are found, suggesting possible **component-level parallelism**

Incremental decomposition of a non-linear system



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

$$\varnothing$$

$$F[1] \quad \downarrow$$

$$\left\{ x^2 + y + z = 1 \right\}$$

$$F[2] \quad \downarrow$$

$$\left\{ \begin{array}{r} x + y^2 + z = 1 \\ y^4 + (2z-2)y^2 + y + (z^2 - z) = 0 \end{array} \right\}$$

$$F[3] \quad \swarrow \qquad \swarrow \qquad \searrow \qquad \searrow$$

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

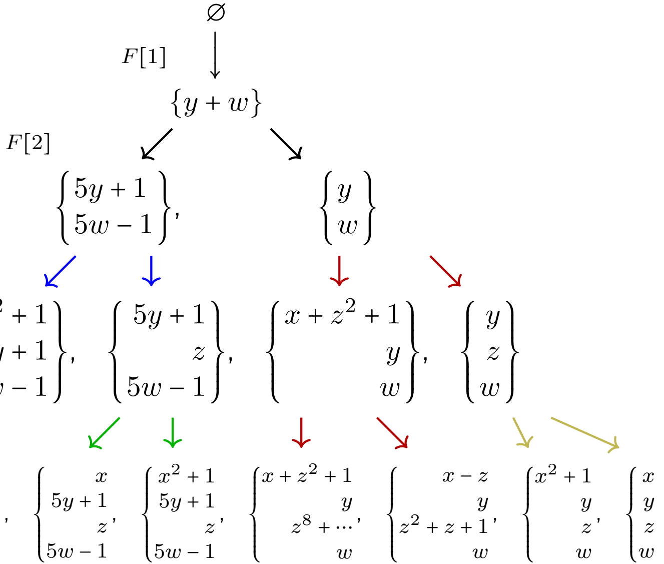
Our Goal: take advantage of different components to gain better performance in high-level decomposition algorithms via **parallelism**

Motivations and challenges

- Many challenges exist in parallelizing triangular decompositions:
  - ↳ 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
- Potential **parallelism is problem-dependent** and not algorithmic; it exhibits **irregular parallelism**
- Where a splitting is found in an **intermediate step**, subsequent steps can operate concurrently on each independent component
  - ↳ Finding splittings in the geometry is as difficult as solving the system
- An implementation must exploit all possible parallelism, without adding too much overhead, in particular in the cases where there is no parallelism.

A more interesting example (1/2)

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



$\varnothing$

$F[1]$

$\{y + w\}$

$F[2]$

$$\left\{\begin{matrix} 5y+1 \\ 5w-1 \end{matrix}\right\}, \quad \left\{\begin{matrix} y \\ w \end{matrix}\right\}$$

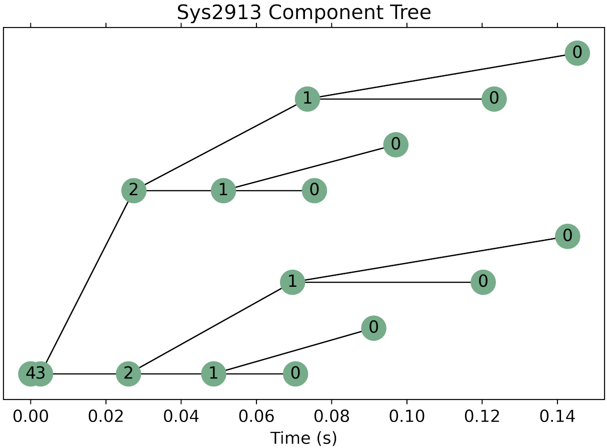
$F[3]$

$$\left\{\begin{matrix} x+z^2+1 \\ 5y+1 \\ 5w-1 \end{matrix}\right\}, \left\{\begin{matrix} 5y+1 \\ z \\ 5w-1 \end{matrix}\right\}, \left\{\begin{matrix} x+z^2+1 \\ y \\ w \end{matrix}\right\}, \left\{\begin{matrix} y \\ z \\ w \end{matrix}\right\}$$

$F[4]$

$$\left\{\begin{matrix} x+z^2+1 \\ 5y+1 \\ z^8+\cdots \\ 5w-1 \end{matrix}\right., \left\{\begin{matrix} x-z \\ 5y+1 \\ z^2+z+1 \\ 5w-1 \end{matrix}\right., \left\{\begin{matrix} x \\ 5y+1 \\ z \\ 5w-1 \end{matrix}\right., \left\{\begin{matrix} x^2+1 \\ 5y+1 \\ z \\ 5w-1 \end{matrix}\right., \left\{\begin{matrix} x+z^2+1 \\ y \\ z^8+\cdots \\ w \end{matrix}\right., \left\{\begin{matrix} x-z \\ y \\ z^2+z+1 \\ w \end{matrix}\right., \left\{\begin{matrix} x^2+1 \\ y \\ z \\ w \end{matrix}\right., \left\{\begin{matrix} x \\ y \\ z \\ w \end{matrix}\right.$$

A more interesting example (2/2)



→ more parallelism exposed as more components found

→ yet, work unbalanced between branches

→ mechanism needed for dynamic parallelism: “workpile” or “task pool”

Previous Works

- Parallelization of high-level algebraic and geometric algorithms was more common roughly 30 years ago
  - ↳ Such as in Gröbner bases [1, 3, 4] and CAD [11]
- Recent work on parallelism has been on *low-level* routines with *regular parallelism*:
  - ↳ Polynomial arithmetic [5, 8]
  - ↳ Modular methods for GCDs and factorization [6, 9]
- Recently, high-level algorithms, often with *irregular parallelism* have neither seen much attention nor received thorough parallelization
  - ↳ The normalization algorithm of [2] finds components serially, then processes each component with a simple parallel map
  - ↳ Early work on parallel triangular decomposition was limited by symmetric multi-processing and inter-process communication [10]

Main Results

Basic Polynomial Algebra Subprograms http://www.bpaslib.org/

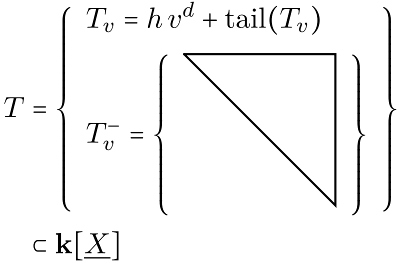
- An implementation of triangular decomposition fully in C/C++
- Parallelization effectively exploits as much parallelism as possible throughout the triangular decomposition algorithm
- Implementation framework for parallelization based on task pools, generating functions, pipelines, fork-join
- An extensive evaluation of our implementation against over 3000 real-world polynomial systems

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 $\mathrm{sat}(T_v^-)$



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

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

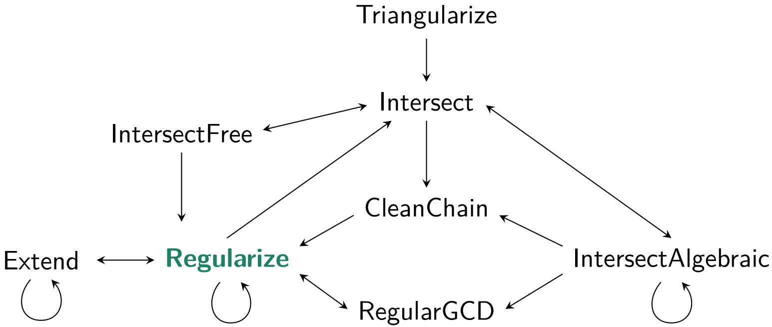
$$f = (y+1)x^2 - x$$

$$\begin{tikzcd}
& T_1 = \begin{cases} y+1=0 \\ z-1=0 \end{cases} \arrow[r, "f = x"] & T_1 = \begin{cases} x = 0 \\ y+1=0 \\ z-1=0 \end{cases} \\
T = \begin{cases} y^2-1=0 \\ z-1=0 \end{cases} \arrow[ur, blue, "y+1=0"] \arrow[dr, red, "y+1\neq 0"'] & & \\
& T_2 = \begin{cases} y-1=0 \\ z-1=0 \end{cases} \arrow[r, "f = 2x^2 - x"] & T_2 = \begin{cases} 2x^2 - x = 0 \\ y-1=0 \\ z-1=0 \end{cases}
\end{tikzcd}$$

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

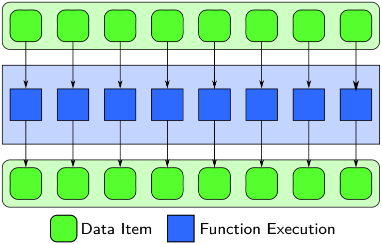
Parallel map and workpile

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

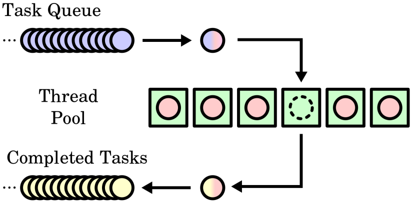
↳ execute a function on each item in a collection concurrently

↳ with multiple Maps, tasks must execute in *lockstep*

Map Pattern [7]



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 $V(F)$
$\mathcal{T} := \{\varnothing\}$
for $p \in F$ do
    $\mathcal{T}' := \{\}$
    for $T \in \mathcal{T}$ Map                        ▷ map Intersect over the current components
        $\mathcal{T}' := \mathcal{T}' \cup$ Intersect$(p, T)$
    $\mathcal{T} := \mathcal{T}'$
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 $V(F)$
Tasks ← $\{ (F, \varnothing) \}$;  $\mathcal{T} \leftarrow \{\}$
while $|Tasks| > 0$ do
    $(P, T)$ ← pop a task from Tasks
    Choose a polynomial $p \in P$; $P' \leftarrow P \setminus \{p\}$
    for $T'$ in Intersect($p, T$) do
        if $|P'| = 0$ then $\mathcal{T} \leftarrow \mathcal{T} \cup \{T'\}$
        else Tasks ← Tasks $\cup \{(P', T')\}$
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

1. Introduction
2. Preliminaries
3. Triangularize: task pool parallelization
4. Intersect: Asynchronous Generators
5. Removing Redundancies: Divide-and-Conquer
6. Experimentation

Generators and Pipelines

Generators

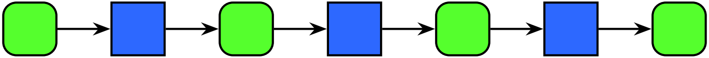
- → A generator function (i.e. iterator) yields data items one at a time, allowing the function's control flow to resume on its next execution.

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.
for $(g_i, T_i) \in$ RegularGCD$(p, T_v, v, T_v^-)$ do
    if $\dim(T_i) \neq \dim(T_v^-)$ then
        for $T_{i,j} \in$ Intersect$(p, T_i)$ do
            yield $T_{i,j}$
    else
        if $g_i \notin \mathbf{k}$ and $\deg(g_i, v) > 0$ then
            yield $T_i \cup \{g_i\}$
        for $T_{i,j} \in$ Intersect$(\mathrm{lc}(g_i, v), T_i)$ do
            for $T' \in$ Intersect$(p, T_{i,j})$ do
                yield $T'$
```

→ **yield** “produces” a single data item, and then continues computation

→ each **for** loop consumes a data one at a time from the generator

Generators are both producers and consumers

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

```
for $(g_i, T_i) \in$ RegularGCD$(p, T_v, T_v^-)$ do
    if dim$(T_i) \neq$ dim$(T_v^-)$ then
        for $T_{i,j} \in$ Intersect$(p, T_i)$ do
            yield $T_{i,j}$
    else
        if $g_i \notin \mathbf{k}$ and $\deg(g_i, v) > 0$ then
            yield $T_i \cup \{g_i\}$
        for $T_{i,j} \in$ Intersect$(\mathrm{lc}(g_i, v), T_i)$ do
            for $T' \in$ Intersect$(p, T_{i,j})$ do
                yield $T'$
```

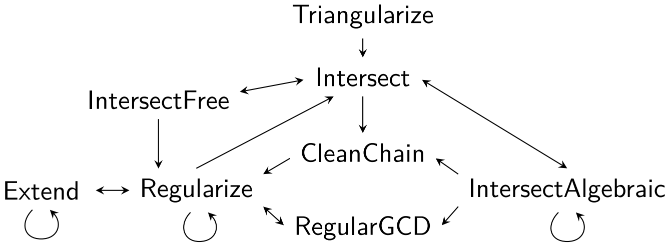
Algorithm 4 **Regularize**(p, T)

```
for $(g_i, T_i) \in$ RegularGCD$(p, T_v, T_v^-)$ do
                ▷ assume $\dim(T_i) = \dim(T_v^-)$
    if $0 < \deg(g_i, v) < \deg(T_v, v)$ then
        yield $T_i \cup g_i$
        yield $T_i \cup \mathrm{pquo}(T_v, g_i)$
        for $T_{i,j} \in$ Intersect$(\mathrm{lc}(g_i, v), T_i)$ do
            for $T' \in$ Regularize$(p, T_{i,j})$ do
                yield $T'$
    else
        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

1. Introduction

2. Preliminaries

3. Triangularize: task pool parallelization

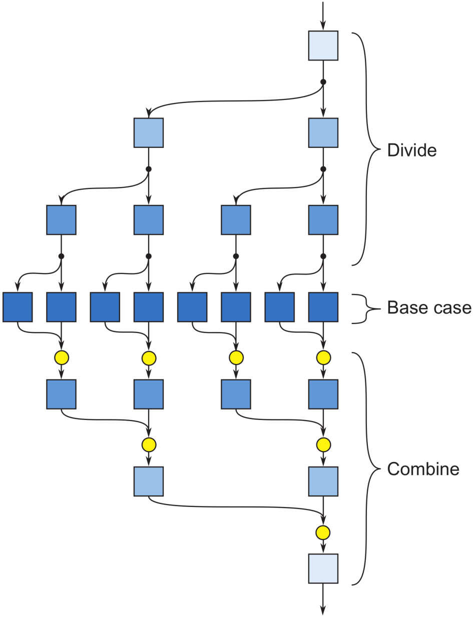
4. Intersect: Asynchronous Generators

5. Removing Redundancies: Divide-and-Conquer

6. Experimentation

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 irredudant 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
  $\mathcal{T}_1' := \varnothing$;   $\mathcal{T}_2' := \varnothing$
  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

1. Introduction
2. Preliminaries
3. Triangularize: task pool parallelization
4. Intersect: Asynchronous Generators
5. Removing Redundancies: Divide-and-Conquer
6. Experimentation

Experimentation Setup

Thanks to Maplesoft, we have a collection of over 3000 real-world systems from: actual user data, the literature, bug reports.

Of these $>$3000 systems, 828 require greater than 0.1s to solve

→ Non-trivial systems to warrant the overheads of parallelism

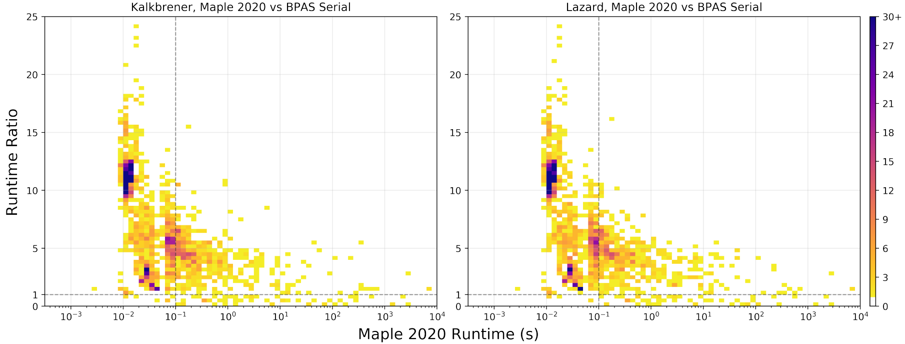
203 of these 828 systems (25%) **do not split** at all

→ No speed-up expected; *some slow-down* is expected in these cases

→ however, we include them to ensure that *slow-down is minimal*

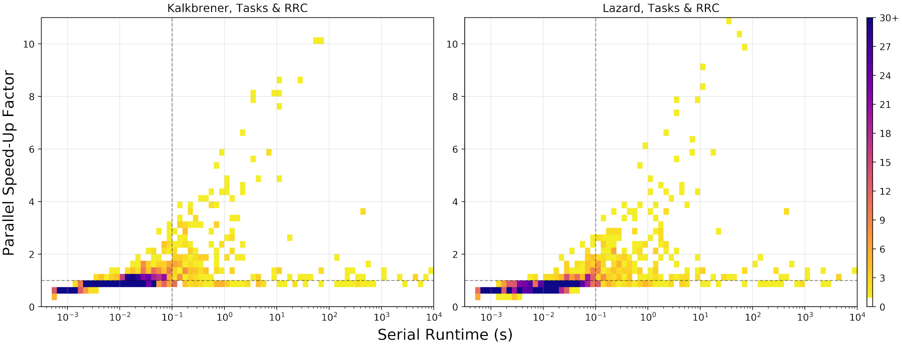
These experiments are run on a node with 2x6-core Intel Xeon X560 processors (24 physical threads with hyperthreading)

BPAS serial vs Maple



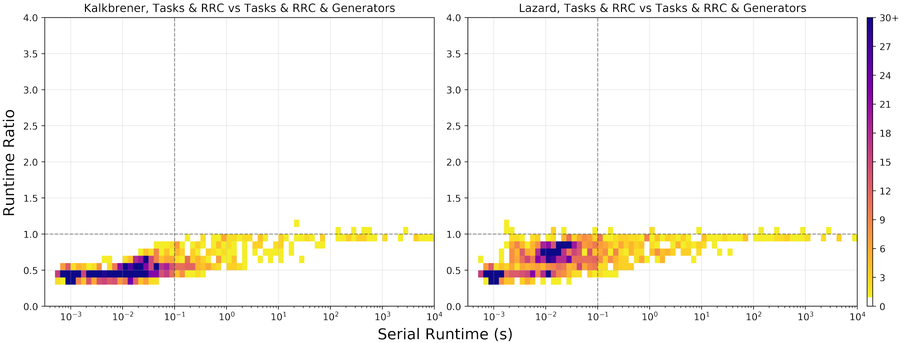
Comparing the runtime performance of triangular decomposition in the `RegularChains` library of MAPLE 2020 against the serialized implementation in BPAS.

Speedup obtained from tasks and fork-join



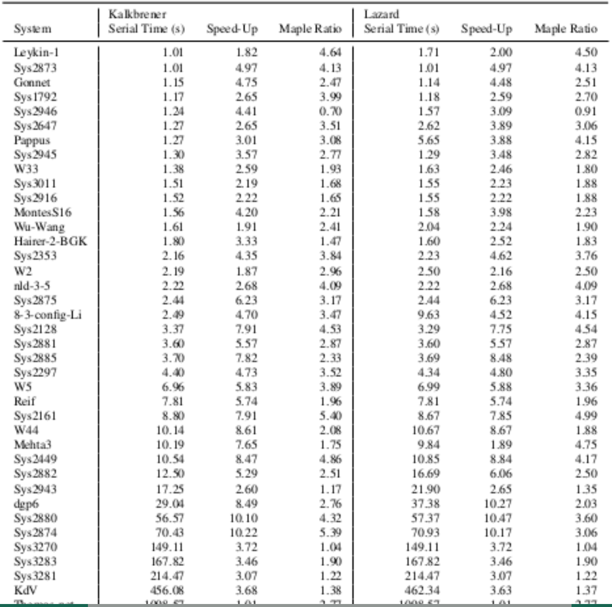
The parallel-speedup obtained from using parallel triangularize tasks and parallel removal of redundant components (RRC) together for solving in Kalkbrener and Lazard modes.

Adding generators



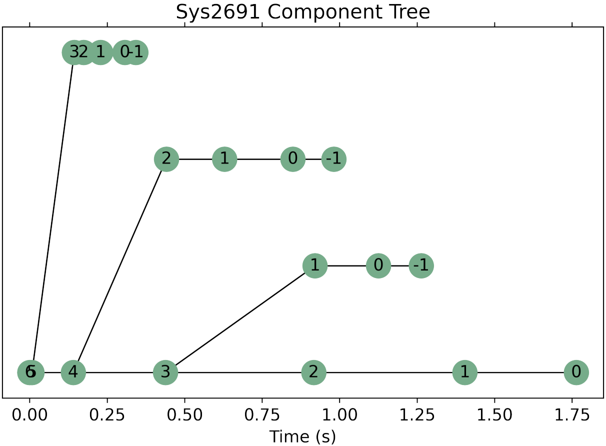
Using parallel triangularize tasks and parallel removal of redundant components (RRC) as the base case, compare the addition of asynchronous generators to overall performance for solving in Kalkbrener and Lazard modes.

Timings for a few well-known systems



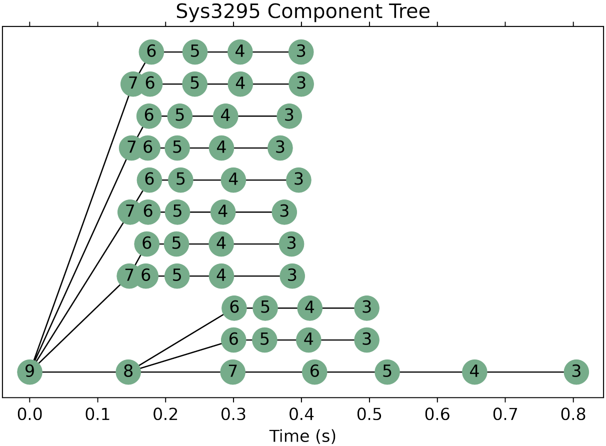
System	Kalkbrener Serial Time (s)	Speed-Up	Maple Ratio	Lazard Serial Time (s)	Speed-Up	Maple Ratio
Leykin-1	1.01	1.82	4.64	1.71	2.00	4.50
Sys2873	1.01	4.97	4.13	1.01	4.97	4.13
Gonnet	1.15	4.75	2.47	1.14	4.48	2.51
Sys1792	1.17	2.65	3.99	1.18	2.59	2.70
Sys2946	1.24	4.41	0.70	1.57	3.09	0.91
Sys2647	1.27	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

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)

Conclusion & Future Work

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.

Further parallelism can be found through:

- → solving over a prime field, which produces more splittings;

Thank You!

Basic Polynomial Algebra Subprograms

http://www.bpaslib.org/

References

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

[2] 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.

[3] B. Buchberger. “The parallelization of critical-pair/completion procedures on the L-Machine”. In: *Proc. of the Jap. Symp. on functional programming*. 1987, pp. 54–61.

[4] J. C. Faugere. “Parallelization of Gröbner Basis”. In: *Parallel Symbolic Computation PASCO 1994 Proceedings*. Vol. 5. World Scientific. 1994, p. 124.

[5] M. Gastineau and J. Laskar. “Parallel sparse multivariate polynomial division”. In: *Proceedings of PASCO 2015*. 2015, pp. 25–33.

[6] J. Hu and M. B. Monagan. “A Fast Parallel Sparse Polynomial GCD Algorithm”. In: *ISSAC 2016, Waterloo, ON, Canada, July 19-22, 2016*. 2016, pp. 271–278.

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

[8] M. B. Monagan and R. Pearce. “Parallel sparse polynomial multiplication using heaps”. In: *ISSAC*. 2009, pp. 263–270.

[9] 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.

[10] M. Moreno Maza and Y. Xie. “Component-level parallelization of triangular decompositions”. In: *PASCO 2007 Proceedings*. ACM. 2007, pp. 69–77.

[11] 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.