On The Parallelization of Triangular Decompositions

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

ISSAC 2020, September 7, 2020

Outline

Decomposing a Non-Linear System

Many ways to “solve” a system

$$\begin{cases} x^2 + y + z = 1 \\ x + y^2 + z = 1 \\ x + y + z^2 = 1 \end{cases} \xrightarrow{\text{Gröbner Basis}} \begin{cases} x + y + z^2 = 1 \\ (y + z - 1)(y - z) = 0 \\ z^2 \left(z^2 + 2y - 1\right) = 0 \\ z^2 \left(z^2 + 2z - 1\right)(z - 1)^2 = 0 \end{cases}$$

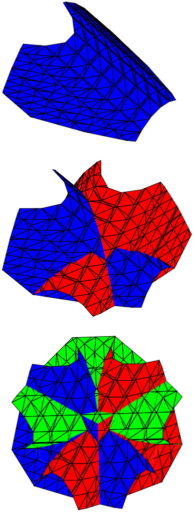
$\Downarrow$ Triangular Decomposition

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

Both solutions are equivalent (via a union).

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

Incremental Decomposition of a Non-Linear System



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

$$\varnothing$$

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

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

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

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

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

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

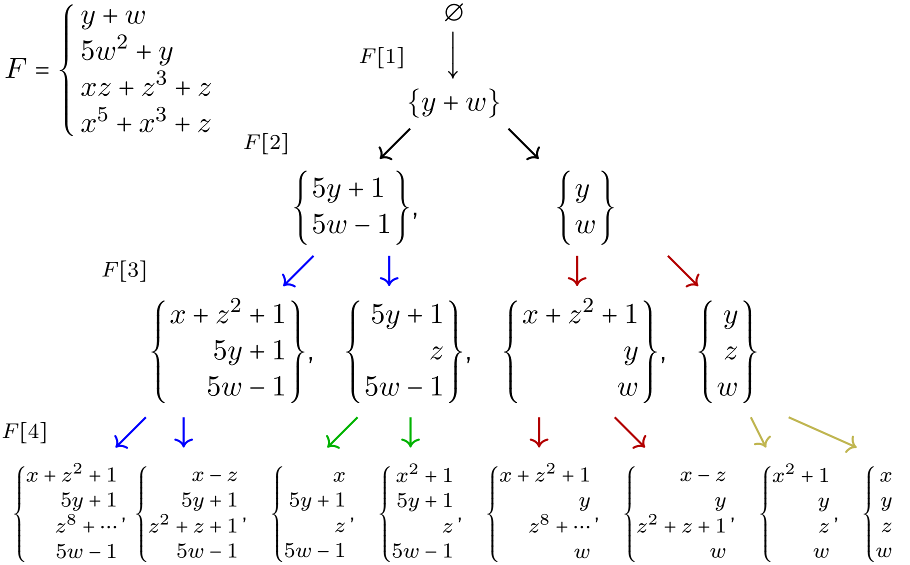
Our Goal: take advantage of different components to gain better performance in high-level decomposition algorithms via **parallelism**

Motivations and Challenges

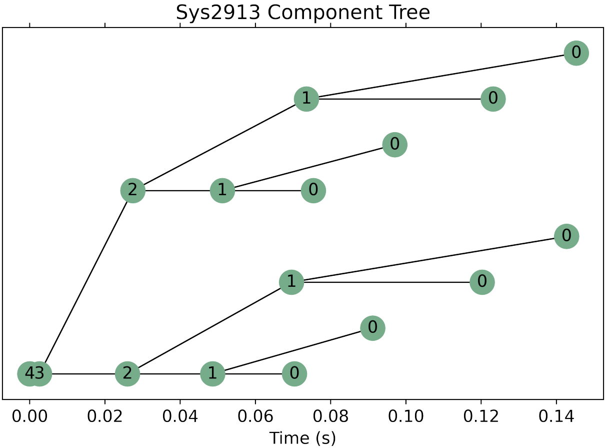
- Many challenges exist in parallelizing triangular decompositions:
  - ↳ Some systems never split
  - ↳ Some split only at the final step, leaving very little concurrency
  - ↳ Some split into one "main" component and several degenerative cases
- Potential **parallelism is problem-dependent** and not 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
- A solution must exploit all possible parallelism, without adding too much overhead in the cases where there is none

A more interesting example (1/2)

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



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 [10]
- Recent work on parallelism has been on *low-level* routines with *regular parallelism*:
  - ↳ Polynomial arithmetic [5, 7]
  - ↳ Modular methods for GCDs and Factorization [6, 8]
- 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 [9]

Main Results

Basic Polynomial Algebra Subprograms

- 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

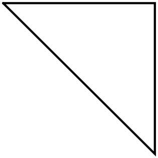
Regular Chains, Notations

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 h is regular modulo $\mathrm{sat}(T_v^-)$ and T_v^- is a regular chain

$T = \Big\{\ T_v = h\,v^d + \mathrm{tail}(T_v)$

$T_v^- = \Big\{$



$\Big\}\ \Big\}$

$\subset \mathbf{k}[\underline{X}]$

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:

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

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:

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

Parallel Programming Patterns

Parallel Map, Workpile

- → Map a function to each item in a collection, executing each function call simultaneously. Requires *lockstep threads*.
- → Workpile generalizes map to a "pile" of tasks and a set of workers. Allows intermediate tasks to add more tasks, enables **load-balancing**

Asynchronous Generators, Pipeline

- → A generator function (a.k.a iterator, coroutine) which produces data to be consumed in parallel; special-case of *producer-consumer problem*
- → Async generators calling other async generators create a **pipeline**

Divide-and-Conquer, Fork-Join Parallelism

- → Divide a problem, solve recursively, then combine sub-solutions.
- → When >1 recursive call *fork* computations, perform each recursive call concurrently, then *join* before combining sub-solutions.

Outline

Triangularize: incremental triangular decomposition

Algorithm 1 Triangularize(F)

Input: a finite set $F \subseteq \mathbf{k}[\underline{X}]$

Output: regular chains $T_1, \ldots, T_e \subseteq \mathbf{k}[\underline{X}]$ such that $V(F) = W(T_1) \cup \cdots \cup W(T_e)$

```
𝒯 := {∅}
for p ∈ F do
    𝒯′ := {}
    for T ∈ 𝒯 do                         ▷ map Intersect over the current components
        𝒯′ := 𝒯′ ∪ Intersect(p, T)
    𝒯 := 𝒯′
return RemoveRedundantComponents(𝒯)
```

- **Coarse-grained parallelism**: each Intersect represents substantial work
- At each "level" there $|\mathcal{T}|$ components with which to intersect, yielding $|\mathcal{T}| - 1$ additional threads
- Performs a *breadth-first search*, with synchronization at each level

Triangularize: a task-based approach

Algorithm 2 TriangularizeByTasks(F)

Input: a finite set $F \subseteq \mathbf{k}[\underline{X}]$

Output: regular chains $T_1, \ldots, T_e \subseteq \mathbf{k}[\underline{X}]$ such that $V(F) = W(T_1) \cup \cdots \cup W(T_e)$

```
Tasks ← { (F, ∅) };  𝒯 ← {}
while |Tasks| > 0 do
    (P, T) ← pop a task from Tasks
    Choose a polynomial p ∈ P; P′ ← P ∖ {p}
    for T′ in Intersect(p, T) do
        if |P′| = 0 then 𝒯 ← 𝒯 ∪ {T′}
        else Tasks ← Tasks ∪ {(P′, T′)}
return RemoveRedundantComponents(𝒯)
```

- *Tasks* is essentially an underlying data structure for a **task scheduler**
- Use a **thread pool** of workers, each executing the body of the while loop
- Tasks create more tasks, workers pop Tasks until none remain.
- Adaptive to load-balancing, no inter-task synchronization

Outline

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'

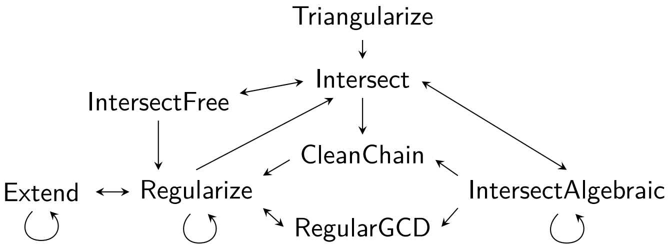
Algorithm 4 **Regularize**(p, T)

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*.

Subroutine Pipeline



- → All subroutines as generators allows pipeline to evolve dynamically with the call stack.
- → call stack forms a **tree** if several generators invoked by one consumer
- → 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 pool is empty

Outline

Divide-and-Conquer and Fork-Join

Remove redundancies from a list of regular chains with DnC:

- → Recursively and concurrently obtain two irredundant lists, then merge.
- → **Cilk** is used to **fork**/**spawn** and **join**/**sync**

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 𝒯
ℓ ← ⌈e/2⌉; 𝒯_{≤ℓ} ← {T_1, ..., T_ℓ}; 𝒯_{>ℓ} ← {T_{ℓ+1}, ..., T_e}
𝒯_1 := spawn RemoveRedundantComponents(𝒯_{≤ℓ})
𝒯_2 := RemoveRedundantComponents(𝒯_{>ℓ})
sync
𝒯_1' := ∅;  𝒯_2' := ∅
for T_1 ∈ 𝒯_1 do
    if ∀T_2 in 𝒯_2 IsNotIncluded (T_1, T_2) then 𝒯_1' := 𝒯_1' ∪ {T_1}
for T_2 ∈ 𝒯_2 do
    if ∀T_1 in 𝒯_1' IsNotIncluded (T_2, T_1) then 𝒯_2' := 𝒯_2' ∪ {T_2}
return 𝒯_1' ∪ 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.

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

→ Non-trivial systems to warrant the overheads of parallelism

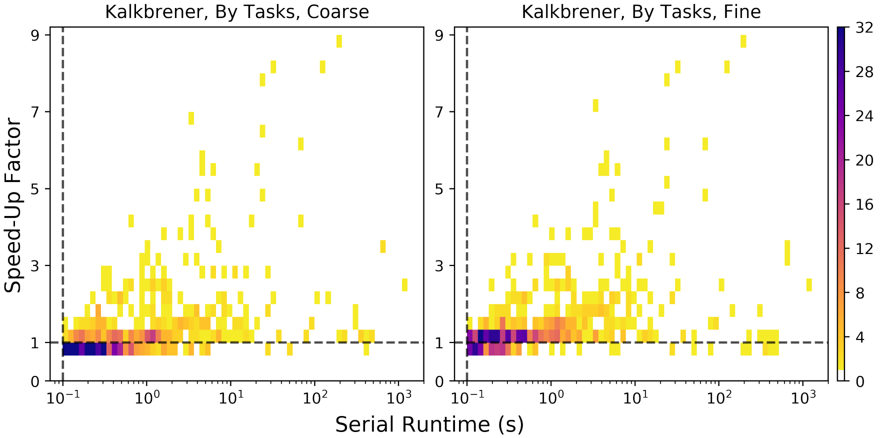
203 of these 828 systems (25%) **do not split** at all

→ 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)

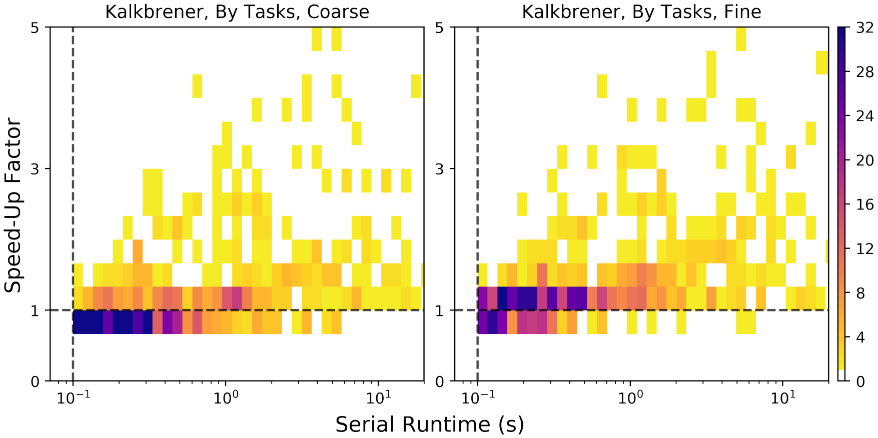
Speed-ups on Non-Trivial Systems (1/2)



→ "Coarse": task manager only, "Fine": tasks and generators

→ Adding generators **increases parallelism**: *streaming* components allows Triangularize to create a new task as early as possible

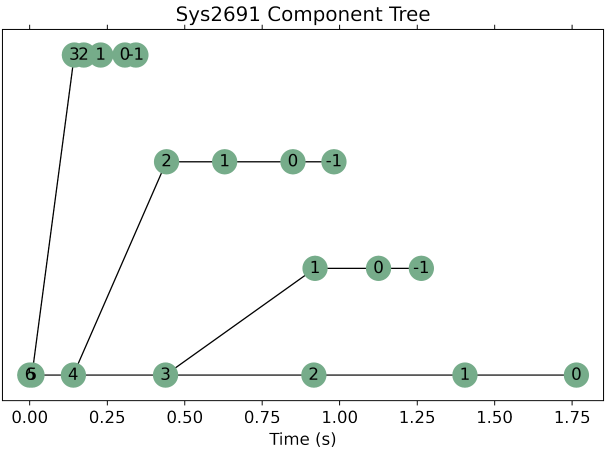
Speed-ups on Non-Trivial Systems (2/2)



→ "Coarse": task manager only, "Fine": tasks and generators

→ Adding generators **increases parallelism**: *streaming* components allows Triangularize to create a new task as early as possible

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

Conclusion & Future Work

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

- → our solution dynamically finds and exploits possible parallelism
- → uses dynamic parallel task management, async. generators, and DnC

Further parallelism can be found through:

- → evaluation/interpolation schemes for subresultant chains
- → solving over a prime field produces more splittings; then lift solutions

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

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

Thank You!

I look forward to your questions:

→ during the live Q/A session,

→ at a Zoom Meeting 18:00-19:00 EEST July 21 2020:
https://westernuniversity.zoom.us/j/93900888047
(Meeting ID: 93900888047)

→ via email:
Alex Brandt <abrandt5@uwo.ca>,
Ali Asadi <masadi4@uwo.ca>,
Marc Moreno Maza <moreno@csd.uwo.ca>

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. Monagan and R. Pearce. "Parallel sparse polynomial multiplication using heaps". In: *ISSAC*. 2009, pp. 263–270.

[8] M. Monagan and B. Tuncer. "Sparse Multivariate Hensel Lifting: A High-Performance Design and Implementation". In: *ICMS 2018*. 2018, pp. 359–368.

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

[10] 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.