Parallel Programming and 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

CS Grad Student Seminars

November 6, 2020

Outline

Solving a Linear System of Equations

$$\begin{cases} x + 3y - 2z = 6 \\ 3x + 5y + 6z = 7 \\ 2x + 4y + 3z = 8 \end{cases}$$

Step 1: triangularization

(a) by *elimination of variables*:

$$\begin{cases} x + 3y - 2z = 6 \\ 3x + 5y + 6z = 7 \\ 2x + 4y + 3z = 8 \end{cases} \xrightarrow[\text{substitue } x]{\text{solve for } x} \begin{cases} x = 5 - 3y + 2z \\ -4y + 12z = -8 \\ -2y + 7z = -2 \end{cases} \xrightarrow[\text{substitue } y]{\text{solve for } y} \begin{cases} x = 5 + 2z - 3y \\ y = 2 + 3z \\ z = 2 \end{cases}$$

(b) by *Gaussian elimination*:

$$\left[\begin{array}{ccc|c} 1 & 3 & -2 & 5 \\ 3 & 5 & 6 & 7 \\ 2 & 4 & 3 & 8 \end{array}\right] \Longrightarrow \left[\begin{array}{ccc|c} 1 & 3 & -2 & 5 \\ 0 & 1 & -3 & 2 \\ 0 & -2 & 7 & -2 \end{array}\right] \Longrightarrow \left[\begin{array}{ccc|c} 1 & 3 & -2 & 5 \\ 0 & 1 & -3 & 2 \\ 0 & 0 & 1 & 2 \end{array}\right]$$

Step 2: back-substitution to find particular values for x, y, z

Solving a Non-Linear System of Equations

Via Gröbner Basis we can "solve" a non-linear system

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

"Solving" a system is not just about finding particular values, rather:

> *"find a description of the solutions from which we can easily extract relevant data."*

Why?

→ A **positive-dimensional system** has *infinitely many solutions*

→ *Underdetermined* linear systems, and most non-linear systems

Decomposing a Non-Linear System

Many ways to "solve" a system

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

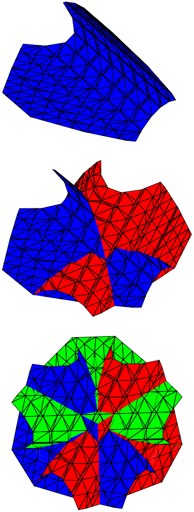
$\Downarrow$ **Triangular Decomposition**

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

Both solutions are equivalent (via a union).

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

Incremental Decomposition via Intersection



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

$$\varnothing$$

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

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

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

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

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

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

Our Goal: take advantage of different, independent components to gain performance via concurrency and **thread-level parallelism**

Motivations and Challenges

Component-level parallelism

↳ when a splitting is found during an *intermediate step*, subsequent operations can be performed on each component concurrently

Solving systems by intersection exhibits **irregular parallelism**: parallelism is **problem-dependent** and not algorithmic

↳ Finding splittings in the geometry is as difficult as solving the system

↳ Some systems never split

↳ Some split only at the final step, resulting in no concurrency

↳ Some split irregularly into one big component and many small ones

A **dynamic, adaptable** solution is needed to find, and exploit possible parallelism, without adding excessive overhead in cases where there is none.

A more interesting example (1/2)

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

$$\varnothing$$

$F[1]$

$$\{y + w\}$$

$F[2]$

$$\begin{Bmatrix} 5y+1 \\ 5w-1 \end{Bmatrix}, \quad \begin{Bmatrix} y \\ w \end{Bmatrix}$$

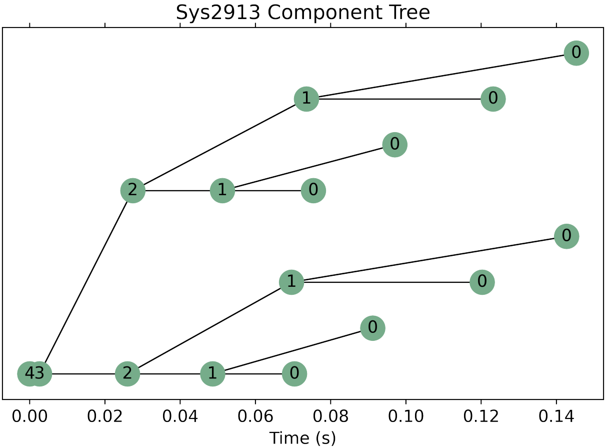
$F[3]$

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

$F[4]$

$$\begin{Bmatrix} x+z^2+1 \\ 5y+1 \\ z^8+\cdots \\ 5w-1 \end{Bmatrix}, \begin{Bmatrix} x-z \\ 5y+1 \\ z^2+z+1 \\ 5w-1 \end{Bmatrix}, \begin{Bmatrix} x \\ 5y+1 \\ z \\ 5w-1 \end{Bmatrix}, \begin{Bmatrix} x^2+1 \\ 5y+1 \\ z \\ 5w-1 \end{Bmatrix}, \begin{Bmatrix} x+z^2+1 \\ y \\ z^8+\cdots \\ w \end{Bmatrix}, \begin{Bmatrix} x-z \\ y \\ z^2+z+1 \\ w \end{Bmatrix}, \begin{Bmatrix} x^2+1 \\ y \\ z \\ w \end{Bmatrix}, \begin{Bmatrix} x \\ y \\ z \\ w \end{Bmatrix}$$

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 in computer algebra 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

- An implementation of triangular decomposition fully in C/C++
- Parallelization dynamically finds and 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

Polynomial Notations

- Let $\mathbf{k}$ be a perfect field, such as $\mathbb{Q}$ (and its extensions) or $\mathbb{C}$
- Let $\mathbf{k}[\underline{X}]$ be the set of multivariate polynomials (a *polynomial ring*) with n ordered variables, $\underline{X} = X_1 < \cdots < X_n$.
- For $p \in \mathbf{k}[\underline{X}]$:
  - ↳ the main variable of p is the maximum variable with positive degree
  - ↳ the initial of p is the leading coeff. of p with respect to its main variable
  - ↳ the tail of p is the terms leftover after setting its initial to 0

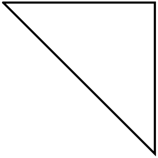
$$(2y + ba)x^2 + (by)x + a^2 \quad \in \; \mathbb{Q}[b < a < y < x]$$

- Any set of polynomials $F \subset \mathbf{k}[\underline{X}]$ can form a **system of equations** by setting $f = 0$ for each $f \in F$.
- The **algebraic variety** of F is the geometric representation of the solution set of F
  - ↳ $V(F) = \{(a_1, \ldots, a_n) \in \mathbf{k}^n \mid f(a_1, \ldots, a_n) = 0, \forall f \in F\}$

Triangular Sets and Regular Chains

A **triangular set** $T \subset \mathbf{k}[\underline{X}]$ is a collection of polynomials with pairwise different main variables.

$$T = \left\{ \begin{array}{l} T_v = h\,v^d + \mathrm{tail}(T_v) \\ T_v^- = \left\{ \quad \right\} \end{array} \right\}$$



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

Example:

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

$\subset \mathbb{Q}[b < a < y < x]$

A **regular chain** is a triangular set if:

(i) T_v^- is a regular chain, and

(ii) initial of T_v (h) is **regular** with respect to T_v^-

In triangular decomposition, **every component is a regular chain**

Regularity

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

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

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

- → y has two solutions: $y^2 - 1 = (y+1)(y-1)$
- → For $y = -1$, x has 1 solution
- → For $y = 1$, x has 2 solutions

A polynomial is **regular** (w.r.t. a particular regular chain) if it is neither:

(i) zero (e.g. y in F_1), nor

(ii) a *zero-divisor* (e.g. $(y+1)$ in F_2)

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

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

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

$y + 1 = 0$:

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

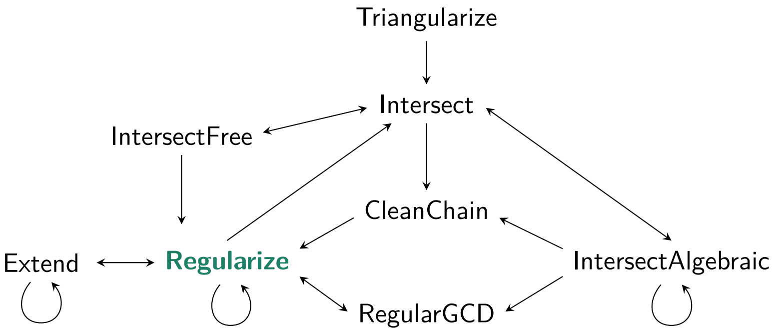
$y + 1 \neq 0$:

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

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

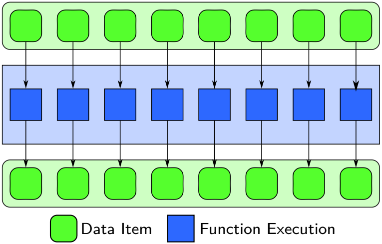
Parallel Map and Workpile

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

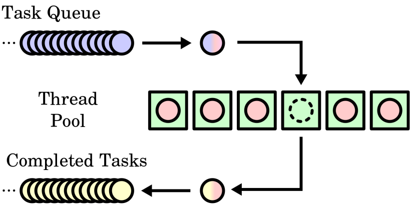
↳ execute a function on each item in a collection concurrently

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

Map Pattern [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 $T_1, \dots, T_e \subseteq \mathbf{k}[\underline{X}]$ encoding the solutions of $V(F)$

```
𝒯 := {∅}
for p ∈ F do
    𝒯′ := {}
    for T ∈ 𝒯 Map                          ▷ 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 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** $|$*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 a time, allowing the function's control flow to resume on its next execution.

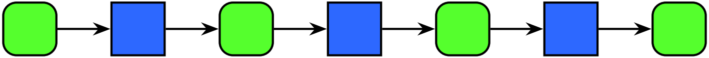
Asynchronous Generators; Producer-Consumer

→ *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*



Regularize as an Asynchronous Generator

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

Input: $p \in \mathbf{k}[\underline{X}] \setminus \mathbf{k}$, $v := \mathrm{mvar}(p)$, a regular chain $T = T_v^- \cup T_v$

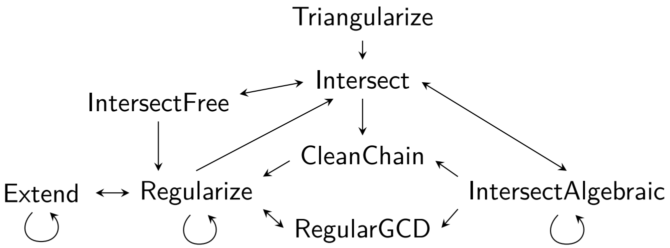
Output: regular chains $T_1, \ldots, T_e$ satisfying specs.

```
for (g_i, T_i) ∈ RegularGCD(p, T_v, T_v^-) do
    if 0 < deg(g_i, v) < deg(T_v, v) then
        yield T_i ∪ g_i
        yield T_i ∪ pquo(T_v, g_i)
        for T_{i,j} ∈ Intersect(lc(g_i, v), T_i) do
            for T' ∈ Regularize(p, T_{i,j}) do
                yield T'
    else
        yield T_i
```

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

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

Subroutine Pipeline



- → Making all subroutines generators allows a pipeline to evolve dynamically with the call stack.
- → call stack forms a **tree** if several generators invoked by one consumer
- → Asynchronous Generators, Pipelines create **fine-grained parallelism** since work diminishes with each recursive call, pipeline depth
- → In our implementation, a thread pool is used and shared among all generators; generators run synchronously if pool is empty

Outline

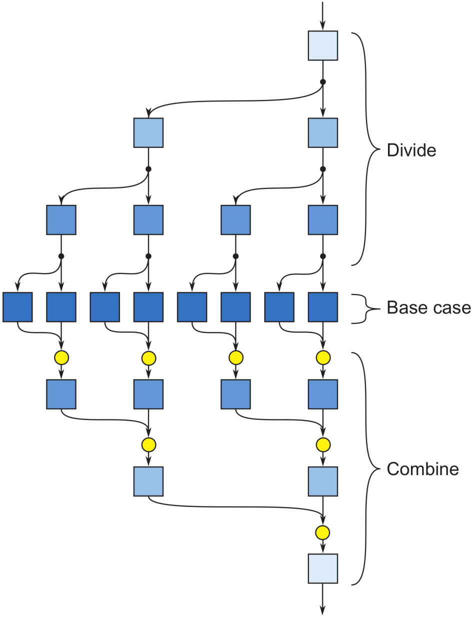
Divide-and-Conquer and Fork-Join

→ 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

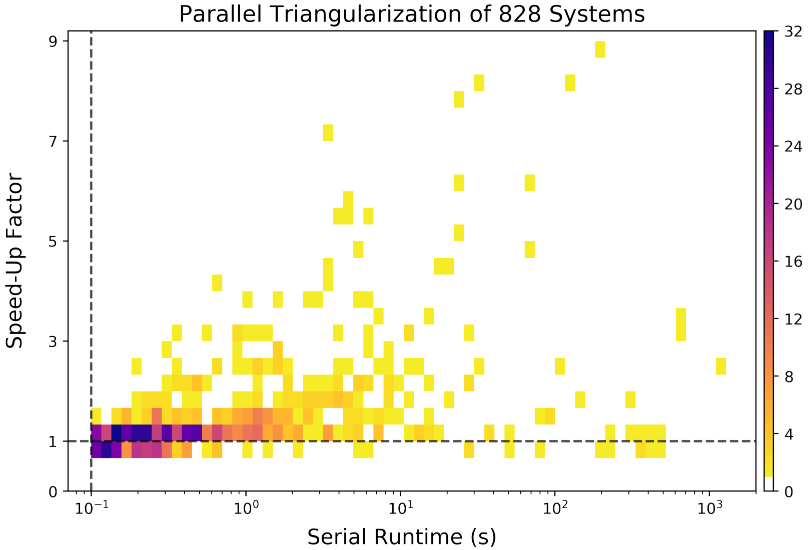
Algorithm 4 RemoveRedundantComponents($\mathcal{T}$)

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



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!

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

[9] M. Monagan and B. Tuncer. "Sparse Multivariate Hensel Lifting: A High-Performance Design and Implementation". In: *ICMS 2018*. 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.