On the Parallelization of Triangular Decompositions

Mohammadali Asadi
University of Western Ontario
London, Canada
masadi4@uwo.ca

Alexander Brandt
University of Western Ontario
London, Canada
abrandt5@uwo.ca

Robert H. C. Moir
University of Western Ontario
London, Canada
rmoir3@uwo.ca

Marc Moreno Maza
University of Western Ontario
London, Canada
moreno@csd.uwo.ca

Yuzhen Xie
University of Western Ontario
London, Canada
yuzhenxie@yahoo.ca

ABSTRACT

We discuss the parallelization of algorithms for solving polynomial systems by way of triangular decomposition. The Triangularize algorithm proceeds through incremental intersections of polynomials to produce different components (points, curves, surfaces, etc.) of the solution set. Independent components imply the opportunity for concurrency. This "component-level" parallelization of triangular decompositions, our focus here, belongs to the class of dynamic irregular parallelism. Potential parallel speed-up depends only on geometrical properties of the solution set (number of components, their dimensions and degrees); these algorithms do not scale with the number of processors. To manage the irregularities of component-level parallelization we combine different concurrency patterns, namely, workpile, producer-consumer, and fork/join. We report on our implementation in the freely available BPAS library. Experimentation with thousands of polynomial systems yield examples with up to 9.5× speed-up on a 12-core machine.

CCS CONCEPTS

• **Computing methodologies** → **Symbolic and algebraic manipulation**; **Parallel algorithms**; • **Mathematics of computing** → **Solvers**; **Mathematical software performance**;

KEYWORDS

polynomial system solving, parallel processing, triangular decomposition, fork-join model, producer-consumer problem, regular chains, dynamic irregular parallel applications

Reference Format:
Mohammadali Asadi, Alexander Brandt, Robert H. C. Moir, Marc Moreno Maza, and Yuzhen Xie. 2020. On the Parallelization of Triangular Decompositions. 8 pages.

1 INTRODUCTION

Solving a polynomial system by means of triangular decomposition entails computing a collection of regular chains which together encode the zero set of the input system. Where triangular decomposition proceeds incrementally, that is, by solving one equation after the other, a splitting of the quasi-component of a regular chain may be discovered when intersecting the next polynomial of the input system and the current partial solution. Concurrency is possible as the decomposition proceeds independently on each branch.

Parallelization of high-level procedures for algebraic and geometric computation is not new, receiving much attention in the '80s and '90s, for example see [2, 8, 9, 12, 22]. In recent years parallelization has again seen attention but in low-level operations like polynomial arithmetic [5, 13, 19], GCDs [14], and Factorization [20]. Parallelization in these low-level routines is more natural, being known as *regular parallelism* [18], since the task decomposes in a static way into consistently sized units of work. Taking advantage of the *irregular parallelism* in high-level geometric computations is more challenging, where splitting, and thus parallelism, is dependent only on the geometry of the input system, and must be found dynamically. For example, in the normalization algorithm of [6], components are first found serially and then processed by a *parallel map* over the components. In our proposed technique, we both discover components and process them in parallel. Indeed, finding splittings in the geometry is as difficult as solving the system itself.

Parallel triangular decomposition was first addressed in [21]. There, parallelism was facilitated by multi-processor shared memory and inter-process communication. The overhead associated with this parallel implementation is drastic and only suited for extremely large problems. It also relied on solving systems modulo a prime in order to generate extra splittings and provide opportunities for parallelism. Solving instead over the rationals provides less opportunity for parallelism but is of more practical importance.

Despite these challenges, we investigate opportunities for thread-level parallelism in triangular decomposition algorithms over the rational numbers. In particular, we discuss three different categories of concurrency to be exploited: (1) high-level parallelism via independent intersection tasks, (2) finer-grained parallelism by means of *asynchronous generators* between subroutines, and (3) a divide-and-conquer scheme for the removal of redundant components. The parallel schemes are independent but their implementations are designed to work cooperatively if needed. This is particularly important to combat the work imbalance and inherent irregular parallelism of triangular decomposition algorithms. As we will discuss, we find that the use of generators in addition to a top-level parallelization scheme is an effective sort of dynamic load-balancing.

Our implementation is extensive, leading to 12 possible configurations of the Triangularize algorithm. This includes solving in the sense of Kalkbrener or Lazard and Wu, two organizations of the top-level Triangularize algorithm, and three different levels of parallelization. Our algorithms have been implemented in the C/C++ language and extensively evaluated using a collection of over 3000

polynomial systems. The results are encouraging, yielding up to 9.5× parallel speed-up on a 12-core machine.

We begin in Section 2 with a brief review of regular chain theory, the Triangularize algorithm, and parallel patterns. Section 3 examines opportunities for parallelism in Triangularize via those parallel patterns. We report on our implementation in Section 4. Finally, we conclude in Section 5 with discussion on experimental results, the effectiveness of our techniques, and areas for future work.

2 PRELIMINARIES

This section is a short review of concepts and algorithms for triangular decomposition and parallel programming. The first two sections deal with the former, for which details can be found in [10]. Throughout this paper, let $\mathbf{k}$ be a perfect field, $\mathbf{K}$ be its algebraic closure, and $\mathbf{k}[\underline{X}]$ be the polynomial ring with $\underline{X} = X_1 < \cdots < X_n$.

2.1 Regular chain theory

Let $p \in \mathbf{k}[\underline{X}]$. Assume that $p \notin \mathbf{k}$ holds. Denote by $\mathrm{mvar}(p)$, $\mathrm{init}(p)$, $\mathrm{mdeg}(p)$, and $\mathrm{tail}(p)$, respectively, the greatest variable appearing in p (called the *main variable* of p), the leading coefficient of p w.r.t. $\mathrm{mvar}(p)$ (called the *initial* of p), the degree of p w.r.t. $\mathrm{mvar}(p)$ (called the *main degree* of p) and the reductum of p w.r.t. $\mathrm{mvar}(p)$ (called the *tail* of p). For $F \subseteq \mathbf{k}[\underline{X}]$, we denote by $\langle F \rangle$ and $V(F)$ the ideal generated by F in $\mathbf{k}[\underline{X}]$ and the algebraic set of $\mathbf{K}^n$ consisting of the common roots of the polynomials of F, respectively.

Triangular set. Let $T \subseteq \mathbf{k}[\underline{X}]$ be a *triangular set*, that is, a set of non-constant polynomials with pairwise distinct main variables. Denote by $\mathrm{mvar}(T)$ the set of main variables of the polynomials in T. A variable $v \in \underline{X}$ is called *algebraic* w.r.t. T if $v \in \mathrm{mvar}(T)$, otherwise it is said *free* w.r.t. T. For $v \in \mathrm{mvar}(T)$, we denote by T_v and T_v^- (resp. T_v^+) the polynomial $f \in T$ with $\mathrm{mvar}(f) = v$ and the polynomials $f \in T$ with $\mathrm{mvar}(f) < v$ (resp. $\mathrm{mvar}(f) > v$). Let h_T be the product of the initials of the polynomials of T. We denote by $\mathrm{sat}(T)$ the *saturated ideal* of T: if $T = \emptyset$ holds, then $\mathrm{sat}(T)$ is defined as the trivial ideal $\langle 0 \rangle$, otherwise it is the ideal $\langle T \rangle : h_T^\infty$. The *quasi-component* $W(T)$ of T is defined as $V(T) \setminus V(h_T)$. For $f \in \mathbf{k}[\underline{X}]$, we define $Z(f, T) := V(f) \cap W(T)$. The Zariski closure of $W(T)$ in $\mathbf{K}^n$, denoted by $\overline{W(T)}$, is the intersection of all algebraic sets $V \subseteq \mathbf{K}^n$ such that $W(T) \subseteq V$ holds; moreover we have $\overline{W(T)} = V(\mathrm{sat}(T))$.

Regular chain. A triangular set $T \subseteq \mathbf{k}[\underline{X}]$ is a *regular chain* if either T is empty, or letting v be the largest variable occurring in T, the set T_v^- is a regular chain, and the initial of T_v is regular (that is, neither zero nor zero divisor) modulo $\mathrm{sat}(T_v^-)$. The *dimension* of T, denoted by $\dim(T)$, is by definition the dimension of its saturated ideal and, as a property, equals $n - |T|$, where $|T|$ is the number of elements of T. The saturated ideal $\mathrm{sat}(T)$ of the regular chain T enjoys important properties, in particular the following, proved in [7]. Let $u_1, \ldots, u_d$ be all the free variables of T. Then $\mathrm{sat}(T)$ is unmixed of dimension d. Moreover, we have $\mathrm{sat}(T) \cap \mathbf{k}[u_1, \ldots, u_d] = \langle 0 \rangle$. Another property is the fact that a polynomial p belongs to $\mathrm{sat}(T)$ if and only if p reduces to 0 by pseudo-division w.r.t T, see [3].

Regular GCD. Let i be an integer with $1 \le i \le n$, $T \subseteq \mathbf{k}[\underline{X}]$ be a regular chain, $p, t \in \mathbf{k}[\underline{X}] \setminus \mathbf{k}$ be polynomials with the same main variable X_i, and $g \in \mathbf{k}$ or $g \in \mathbf{k}[\underline{X}]$ with $\mathrm{mvar}(g) \le X_i$. Assume that

(1) $X_i > X_j$ holds for all $X_j \in \mathrm{mvar}(T)$, and
(2) both $\mathrm{init}(p)$ and $\mathrm{init}(t)$ are regular w.r.t. $\mathrm{sat}(T)$.

Denote by $\mathbb{A}$ the total ring of fractions of the residue class ring $\mathbf{k}[X_1, \ldots, X_{i-1}]/\sqrt{\mathrm{sat}(T)}$. Note that $\mathbb{A}$ is isomorphic to a direct product of fields. We say that g is a *regular GCD* of p, t w.r.t. T whenever:

(G_1) the leading coefficient of g in X_i is a regular element of $\mathbb{A}$;
(G_2) g belongs to the ideal generated by p and t in $\mathbb{A}[X_i]$; and
(G_3) if $\deg(g, X_i) > 0$, then g pseudo-divides both p and t in $\mathbb{A}[X_i]$, that is, both $\mathrm{prem}(p, g)$ and $\mathrm{prem}(t, g)$ belong to $\sqrt{\mathrm{sat}(T)}$.

When Conditions (G_1), (G_2), (G_3) and $\deg(g, X_i) > 0$ hold, we have:

(G_4) if $\mathrm{mdeg}(g) = \mathrm{mdeg}(t)$, then $\sqrt{\mathrm{sat}(T \cup t)} = \sqrt{\mathrm{sat}(T \cup g)}$ and $W(T \cup t) \subseteq Z(h_g, T \cup t) \cup W(T \cup g) \subseteq \overline{W(T \cup t)}$ both hold,
(G_5) if $\mathrm{mdeg}(g) < \mathrm{mdeg}(t)$, let $q = \mathrm{pquo}(t, g)$, then $T \cup q$ is a regular chain and the following two relations hold:
  (a) $\sqrt{\mathrm{sat}(T \cup t)} = \sqrt{\mathrm{sat}(T \cup g)} \cap \sqrt{\mathrm{sat}(T \cup q)}$,
  (b) $W(T \cup t) \subseteq Z(h_g, T \cup t) \cup W(T \cup g) \cup W(T \cup q) \subseteq \overline{W(T \cup t)}$,
(G_6) $W(T \cup g) \subseteq V(p)$,
(G_7) $V(p) \cap W(T \cup t) \subseteq W(T \cup g) \cup V(p, h_g) \cap W(T \cup t) \subseteq V(p) \cap \overline{W(T \cup t)}$.

Triangular decomposition. Let $F \subseteq \mathbf{k}[\underline{X}]$. Regular chains $T_1, \ldots, T_e$ of $\mathbf{k}[\underline{X}]$ form a *triangular decomposition* of $V(F)$ in the sense of Kalkbrener (resp. Wu and Lazard) whenever we have $V(F) = \cup_{i=1}^e \overline{W(T_i)}$ (resp. $V(F) = \cup_{i=1}^e W(T_i)$). Hence, a triangular decomposition of $V(F)$ in the sense of Wu and Lazard is necessarily a triangular decomposition of $V(F)$ in the sense of Kalkbrener, while the converse is not true. One important issue in the implementation of algorithms decomposing polynomial ideals and algebraic sets is the removal of redundant components. In the context of triangular decompositions, this issue implies being able to decide whether $W(T_i) \subseteq W(T_j)$ holds or not, for any two regular chains $T_i, T_j \subseteq \mathbf{k}[\underline{X}]$.

2.2 Specification of the main algorithms

Triangularize. Let $F \subseteq \mathbf{k}[\underline{X}]$. The function call Triangularize(F) computes regular chains $T_1, \ldots, T_e \subseteq \mathbf{k}[\underline{X}]$ forming a triangular decomposition of $V(F)$ in the sense of either Kalkbrener, or Wu and Lazard. An algorithm for Triangularize(F) is presented in [10].

Regularize. For $p \in \mathbf{k}[\underline{X}]$ and $T \subseteq \mathbf{k}[\underline{X}]$ a regular chain, Regularize(p, T) computes regular chains $T_1, \ldots, T_e \subseteq \mathbf{k}[\underline{X}]$ such that:

(R_1) for $i = 1, \ldots, e$, either $p \in \mathrm{sat}(T_i)$ or p is regular w.r.t. $\mathrm{sat}(T_i)$,
(R_2) we have $W(T) \subseteq W(T_1) \cup \cdots \cup W(T_e) \subseteq \overline{W(T)}$.

RegularGcd. Let i, T, p, t, g be as above in the definition of a regular GCD. The function call RegularGcd(p, t, X_i, T) computes a set of pairs $\{(g_1, T_1), \ldots, (g_e, T_e)\}$ such that:

(1) for $i = 1, \ldots, e$, if $\dim(T_i) = \dim(T)$ holds, then g_i is a regular GCD of p, t w.r.t. T_i,
(2) we have $W(T) \subseteq W(T_1) \cup \cdots \cup W(T_e) \subseteq \overline{W(T)}$.

Intersect. Let $p \in \mathbf{k}[\underline{X}]$ and let $T \subseteq \mathbf{k}[\underline{X}]$ be a regular chain. The function call Intersect(p, T) computes regular chains $T_1, \ldots, T_e \subseteq \mathbf{k}[\underline{X}]$ such that: $V(p) \cap W(T) \subseteq W(T_1) \cup \cdots \cup W(T_e) \subseteq V(p) \cap \overline{W(T)}$.

2.3 Parallel Programming Patterns

The algorithms shown in the previous subsection already hint at their parallel opportunities where each either take a list as an argument or return a list. These opportunities are explained in detail

in Section 3, while here we review some parallel programming patterns which will be employed by those opportunities.

The first key observation is that the Triangularize algorithm itself only presents parallelism when the solution set can be separated into multiple components. The existence of such components is not an algorithmic property but rather one subject to the system being solved. Even if computations do split, the work is likely to be unbalanced. This describes *irregular parallelism*. In contrast, *regular parallelism* exists where problems algorithmically decompose into sub-problems of roughly equal size. Despite these challenges, parallel patterns can be employed for irregular parallelism [18].

We are concerned with *thread parallelism*, and thus with minimizing *parallel overheads*, as well as effectively managing inter-thread dependencies and communication. The former deals with the cost of spawning threads, and *over-subscription*—where software threads outnumber hardware resources to drastically reduce performance. The latter can be addressed through parallel design patterns [18].

Parallel Map and Workpile. The *map* pattern maps a function to each item in a collection, simultaneously executing the function on each independent data, scaling well with increasing data and threads. But, threads must operate in lockstep, and are thus limited by the slowest in the group, working best with regular parallelism.

The *workpile* pattern generalizes map to handle both irregular amounts of work and an unknown number of tasks for *load-balancing*. Tasks are collected into a pile (or queue) and one thread executes one task from the pile in parallel, repeating until the pile is empty. This pattern allows in-flight tasks to add additional tasks to the pile, allowing new tasks to be launched immediately by an idle thread. Threads are thus uncoupled, making load balancing possible. Tasks in the pile may also be ordered so that tasks can create new tasks earlier in the computation to exploit further parallelism.

Asynchronous Generators, Producer-Consumer, and Pipeline. A *generator* function is one yielding data items one a time rather than many as a collection. Concurrency arises if items are generated asynchronously while the caller processes a generated item; hence an *asynchronous generator*. This yields the classic *producer-consumer problem* (see [4, Ch. 6]). Using a collection of producer-consumer pairs in a sequence (or directed acyclic graph), where interior nodes are both producers and consumers, is one way of describing the *pipeline* pattern. Pipeline's greatest asset is its ability to begin processing before all input data items are ready (cf. the map pattern). If the producer-consumer pairs are implemented using generators, one can construct a tree—rather than a strict pipeline— which dynamically grows and shrinks as functions are called and return. A tree arises where a producer consumes multiple generators.

Divide-and-Conquer and Fork-Join. Divide-and-Conquer (DnC) is a ubiquitous algorithmic technique based on recursion. A problem is divided into sub-problems, each solved (conquered) recursively, and then sub-solutions are combined to provide a solution to the original problem. Where there are multiple recursive calls per level, the *fork-join* pattern can be employed where each recursive branch is executed in parallel (forked) and then joined together before returning. In a parallel DnC it is important to avoid too many parallel recursive calls to reduce parallel overheads and over-subscription.

3 CONCURRENCY OPPORTUNITIES

In this section, we highlight the opportunities for concurrent execution offered by the algorithms for computing triangular decompositions presented in [10]. To do so, we review the key ideas underlying those algorithms and show how concurrency can be exposed.

3.1 Parallel Map and the Triangularize procedure

Algorithm 1 states a simple procedure implementing the Triangularize procedure. Lines 1 to 5 in Algorithm 1 compute a triangular decomposition of $V(F)$ in the sense of Wu and Lazard; this follows easily from the specification of the Intersect algorithm given in Section 2. Line 5 ensures the decomposition is free of redundant components; we shall discuss this step in detail in Section 3.3.

One can organize the regular chains computed in the loop of Algorithm 1 as a tree with an edge going from node T to node T' if T' is returned by Intersect(p, T) for some $p \in F$, (e.g., see Figure 2 in Section 5). Let (T, T') be such an edge: observe that we have $|T| \leq |T'|$. Algorithm 1 traverses this tree in a breadth-first search manner. Using this algorithm, a Kalkbrener decomposition can be computed by simply pruning branches of the tree, for which the *height* of a regular chain, i.e., the number of polynomials in the chain, exceeds the number of input polynomials in F. This is a consequence of *Krull's height theorem*, see [10], for details.

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 $V(F) = W(T_1) \cup \cdots \cup W(T_e)$
1: $\mathcal{T} := \{\emptyset\}$
2: **for** $p \in F$ **do**
3: $\quad \mathcal{T}' := \bigcup_{T \in \mathcal{T}}$ Intersect(p, T)
4: $\quad \mathcal{T} := \mathcal{T}'$
5: Remove from $\mathcal{T}$ any T_1 where there exists $T_2 \in \mathcal{T}$ such that $W(T_1) \subseteq W(T_2)$ and $T_1 \neq T_2$ both hold.
6: **return** $\mathcal{T}$

It follows from Algorithm 1 that whenever Intersect(p, T) returns more than one regular chain, there is an opportunity for concurrent execution. Indeed, the branches of the breadth-first search are independent and can be continued concurrently. Referring to the celebrated parallel map pattern [18, Ch. 4], one can see Line 3 as a map-step where Intersect maps each current regular chain. Moreover, this can be seen as coarse-grained parallelism as each Intersect call represents substantial work. We now turn our attention to parallel opportunities in the core subroutines of Triangularize.

3.2 Asynchronous generators with Intersect, RegularGcd and Regularize

Let $p \in \mathbf{k}[\underline{X}]$ and $T \subseteq \mathbf{k}[\underline{X}]$ be a regular chain. The operation Intersect(p, T) is quite complicated in general. Yet, for the purpose of discussing opportunities concurrency, it is sufficient to consider the most common scenario. Let us assume $p \notin \mathbf{k}$, $v = \text{mvar}(p)$, init(p) is regular w.r.t sat(T) (calling Regularize can assure this), and that T_v^+ is empty (by proceeding by induction on the number of variables). Algorithm 2 implements Intersect(p, T) under these conditions. It follows from the applications of Formulas (G_6) and (G_7) from Section 2 together with induction on $\dim(T_v^-)$.

Algorithm 2 Intersect(p, T)

Input: $p \in \mathbf{k}[\underline{X}]$, $p \notin \mathbf{k}$, $v := \mathrm{mvar}(p)$, a regular chain $T \subseteq \mathbf{k}[\underline{X}]$ such that $\mathrm{init}(p)$ regular w.r.t. $\mathrm{sat}(T)$ and $T_v^+ = \emptyset$.
Output: regular chains $T_1, \ldots, T_e \subseteq \mathbf{k}[\underline{X}]$ such that $V(p) \cap W(T) \subseteq W(T_1) \cup \cdots \cup W(T_e) \subseteq V(p) \cap \overline{W(T)}$.

```
1:  if v ∉ mvar(T) then
2:      yield T ∪ {p}
3:      for S in Intersect(init(p), T) do
4:          for U in Intersect(tail(p), S) do
5:              yield U
6:  else
7:      for (g_i, T_i) ∈ RegularGcd(p, T_v, v, T_v^-) do
8:          if dim(T_i) ≠ dim(T_v^-) then
9:              for T_{i,j} ∈ Intersect(p, T_i) do
10:                 yield T_{i,j}
11:         else
12:             if g_i ∉ k and mvar(g_i) = v then
13:                 yield T_i ∪ {g_i}
14:             for T_{i,j} ∈ Intersect(lc(g_i, v), T_i) do
15:                 for T_{i,j,k} ∈ Intersect(p, T_{i,j}) do
16:                     yield T_{i,j,k}
```

Note that Algorithm 2 is a *generator function*, also called an *iterator*, a special type of co-routine, see Chapter 8 in [23]. In our pseudo-code, the keyword **yield** outputs a value to the generator's caller and then resumes execution. In contrast, **return** is used to return a value and terminate the function. Each yield in Intersect is an opportunity for concurrency where the caller may execute in parallel with the one yielded regular chain, meanwhile Intersect continues. Hence, Intersect may be implemented as a so-called *asynchronous generator*, a concept described in Section 2.3.

Observe now that the function call RegularGcd(p, T_v, v, T_v^-), when it returns more than one pair, provides additional opportunities for concurrency. Let us actually see how this latter function call is performed in [10]. The subresultant chain S of p and T_v, regarded as univariate in v, is computed. Let $\lambda = \min(\mathrm{mdeg}(p), \mathrm{mdeg}(T_v))$ and let i be in the range $0, \ldots, \lambda + 1$. Denote by S_i the subresultant (from S) of index i and by s_i the principal subresultant coefficient of S_i. Recall that we have $S_{\lambda+1} = p$, $S_\lambda = T_v$, and that S_0 is simply the resultant of p and T_v in v; moreover, we have $S_0 = s_0$. Let j be an integer, with $1 \leq j \leq \lambda + 1$, such that s_j is a regular modulo $\mathrm{sat}(T_v^-)$ and such that for any $0 \leq i < j$, we have $s_i \in \mathrm{sat}(T_v^-)$. Then S_j is a regular GCD of p and T_v w.r.t. T_v^-. By calling Regularize on s_k, $k = 0, \ldots, j$ it is always possible to find such an S_j, up to splittings of the regular chain. This again suggests that RegularGCD could be implemented as an asynchronous generator for Intersect.

We now consider Regularize, focusing on the most common scenario as with Intersect. Algorithm 3 presents this case, stating the assumptions which follow from Formulas (G_4) and (G_5) in Section 2, together with a reasoning by induction on $\dim(T_v^-)$. Just as in the previous two algorithms, Regularize may both be implemented as an asynchronous generator and use generators as it calls Intersect.

The above discussion of Intersect, RegularGcd and Regularize shows that each of those routines can be implemented as a generator function. Each top-level call to Intersect thus creates a tree of generator function calls, most of which being both producers and consumers of values. This hints towards using the *producer-consumer* and *pipeline* patterns, as discussed in Section 2.3.

Algorithm 3 Regularize(p, T)

Input: $p \in \mathbf{k}[\underline{X}]$, $p \notin \mathbf{k}$, $v := \mathrm{mvar}(p)$, a regular chain $T \subseteq \mathbf{k}[\underline{X}]$ such that $\mathrm{init}(p)$ regular w.r.t. $\mathrm{sat}(T_v^-)$ and $T_v^+ = \emptyset$.
Output: regular chains $T_1, \ldots, T_e \subseteq \mathbf{k}[\underline{X}]$ such that (R_1), (R_2) hold.

```
1:  if v ∉ mvar(T) then return T
2:  for (g_i, T_i) ∈ RegularGcd(p, T_v, v, T_v^-) do
3:      if dim(T_i) < dim(T_v^-) then
4:          for T_{i,j} ∈ Regularize(p, T_i) do
5:              yield T_{i,j}
6:      else
7:          if g_i ∈ k or mvar(g_i) < v or mdeg(g_i) = mdeg(T_v) then
8:              yield T_i
9:          else
10:             yield T_i ∪ {g_i}
11:             q_i := pquo(T_v, g_i, v)
12:             yield T_i ∪ {q_i}
13:             for T_{i,j} ∈ Intersect(lc(g_i, v), T_i) do
14:                 for T_{i,j,k} ∈ Regularize(p, T_{i,j}) do
15:                     yield T_{i,j,k}
```

These concurrency opportunities represent more fine-grained parallelism as the amount of work diminishes with each recursive call. Further, it is worth noting that the work between splittings is likely unbalanced. For instance, the polynomials g_i and q_i, returned with the regular chain T_i at Lines 10 and 12 of Algorithm 3, may have very different degrees. These irregular parallelism challenges are addressed through cooperation between the generators and the coarse-grained parallelism in Triangularize (see Section 4.3).

3.3 Fork-join approach for removing redundant components

To remove redundant components efficiently we must address two issues: how to efficiently test single inclusions, e.g. $W(T_i) \subseteq W(T_j)$ and how to efficiently remove redundant components from a large set. The first issue is addressed by taking advantage of the heuristic algorithm IsNotIncluded (see [24, pp. 166–169]) which is very effective in practice. Handling large sets of regular chains is possible by structuring the computation as a divide-and-conquer algorithm.

Given a set $\mathcal{T} = \{T_1, \ldots, T_e\}$ of regular chains, Algorithm 4, RemoveRedundantComponents($\mathcal{T}$), removes redundant chains by dividing $\mathcal{T}$ into two subsets, producing two irredundant sets by means of recursion. Then, the two sets are merged by checking for pair-wise inclusions between the two sets. The divide-and-conquer nature of RemoveRedundantComponents is undoubtedly admissible to ubiquitous fork-join parallelism, Particularly, one *forks* the computation to compute one of the recursive calls in parallel, and then *joins* upon return. These are indicated by the keywords **spawn** and **sync**, respectively. The merge-step is also embarrassingly parallel and can use the map pattern for each of the inner loops.

4 IMPLEMENTATION

Our implementation of regular chains and the Triangularize algorithm follows that of [10], hence, here we look only at the implementation of parallel aspects. Our implementation is written in the C and C++ languages. For the simple fork-join parallelism in

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

Input: a finite set $\mathcal{T} = \{T_1, \ldots, T_e\}$ of regular chains
Output: a set of regular chains forming an irredudant decomposition of the same algebraic set as $\mathcal{T}$

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

the removal of redundant components, we simply use Cilk [15], built-in to the GCC compiler, mirroring Algorithm 4 and requires no additional explanation. In all other cases we implement our own parallel constructs using the Thread Support Library of C++11. Our implementation is freely available in source as part of the Basic Polynomial Algebra Subprograms (BPAS) library [1] at www.bpaslib.org. We begin by describing two reorganizations of the Triangularize algorithm and then describe the underlying parallelization.

The first reorganization of Triangularize is "by level". It is simple restructuring of Algorithm 1 to move the removal of redundant components inside the loop, thus removing redundancies after each incremental step ("level"). We apply the map pattern to the inner for loop, thus spawning $|\mathcal{T}| - 1$ additional threads to execute the $|\mathcal{T}|$ independent calls to Intersect. As previously described for the map pattern, if the intersections at a particular level are unbalanced then the program must wait for the slowest, reducing parallelism.

The second reorganization of Triangularize is "by tasks" (Algorithm 5), and makes use of the workpile pattern to combat the issues incurred by applying the map pattern to irregular parallelism. Here, we essentially invert the loops of Triangularize to first iterate over the current collection of regular chains and then iterate over polynomials in the input system. Since the former is actually of a variable and unknown size, this is achieved by creating tasks, one per regular chain, with a list of polynomials associated to each task. Splittings create new tasks to be added to the work pile. Once a task has finished intersecting its list of polynomials, it is complete and added to a list of results. In this scheme, the potential parallelism is greater than TriangularizeByLevel but the potential amount of work is also greater since redundancies are no longer removed at each step. We discuss these differences later in Section 5.

4.1 Executor Thread Pool

A fundamental structure of most parallel systems is a thread pool. Thread pools maintain a collection of long-running threads which wait to be given a task, execute that task, and then return to the pool. This avoids the overhead of repeatedly spawning threads and limits the number of threads to avoid over-subscription. When tasks outnumber threads the pool also maintains a queue of tasks.

Often threads in a pool execute a predetermined task. However, the many different subroutines in the Triangularize algorithm all need attention. We thus make use of *functors* (e.g. `std::function`), objects which encapsulate a function as a first-class object, to model generic tasks. Our `ExecutorThreadPool` then maintains a queue of functor tasks and a pool of `ExecutorThreads`, threads capable of executing any functor. For genericity, our implementation requires `void` functors, hence returning values by reference. Moreover, values can be returned one at a time if the functor is a generator.

Algorithm 5 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 $V(F) = W(T_1) \cup \cdots \cup W(T_e)$

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

4.2 Asynchronous Generators & Object Streams

Following the object-oriented nature of C++, and much like functor objects, we look to encapsulate generators as objects, providing a generic interface for generators producing many different kinds of objects. We have created a generic `AsyncGenerator` class, where objects are created very simply by passing it a functor whose underlying function creates a collection of objects. The caller then requests data from the generator object itself instead of the functor.

Serially, a generator object could be implemented by collecting the objects returned by the functor in a queue and yielding them one at a time to the caller. To achieve parallelism the `AsyncGenerator` facilitates the producer-consumer pattern; the caller is the consumer, the functor is the producer, and the `AsyncGenerator` itself works as the intermediary and common interface between the two. The interface of `AsyncGenerator` can be seen in Listing 1.

In practice, the subroutines like Intersect, RegularGCD and Regularize, are mutually recursive and simultaneously act as both producers and consumers, using multiple `AsyncGenerator` objects.

```
template <class Object>
class AsyncGenerator {

  /* CONSUMER: create generator to encapsulate a function call. */
  template<class Function, class... Args>
  AsyncGenerator(Function&& f, Args&&... args);

  /* PRODUCER: Add a new Object to be retrieved later. */
  virtual void generateObject(Object& obj) = 0;

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

  /* CONSUMER: Obtain the next generated Object by reference.
  returns false iff no more objects available and setComplete() */
  virtual bool getNextObject(Object& obj) = 0;
};
```

Listing 1: The `AsyncGenerator` interface which implements an asynchronous producer-consumer pattern.

The `AsyncGenerator` fulfills producer-consumer by first inserting itself as a parameter to the functor, so that the producer has

a handle on the generator object, and then invokes the functor as follows. The generator requests a thread from the `ExecutorThreadPool` and, where one is available, asynchronously executes the functor on that thread. Otherwise, the generator acts serially, as just explained, maintaining a queue of objects returned by the functor.

The final detail to the `AsyncGenerator` is a mechanism to sleep the consumer when no object is available to consume. The solution is provided generically as the `AsyncObjectStream` class which provides a thread-safe queue interface and an internal mechanism to sleep the consumer until an object is ready to be consumed (i.e. a condition variable or semaphore, see [4, Ch. 6]). Ultimately, the object stream is completely encapsulated by the `AsyncGenerator` and may or may not be used depending on if the generator is truly executing asynchronously.

4.3 A "Cooperative" Task Scheduler

A task scheduler is one possible implementation of the workpile pattern. We look to facilitate the scheduling of an unknown number tasks where the tasks themselves can produce more tasks. This is exactly the case as in TriangularizeByTasks. Our `TaskScheduler`, much like the `ExecutorThreadPool`, encapsulates tasks as a queue of functors, where each functor has a reference to the scheduler in order to schedule more tasks.

The scheduler internally makes use of an `ExecutorThreadPool` to launch new tasks immediately (if the pool is not empty) and otherwise add it to the queue of tasks. To reap the most benefits of workpile, active threads should add tasks to the scheduler as early as possible to expose more parallelism. For tasks which produce exactly one task, instead of adding the new task to the scheduler, the producing task should execute the new task directly in order to avoid synchronization overhead.

Consider also that we want to simultaneously use generators and a task scheduler. However, they both use an `ExecutorThreadPool`, which may lead to too many active threads and thus resource contention. One solution would be to limit the sum of the number of threads in both pools to be the number of hardware threads. However, this static solution is not receptive to dynamic load-balancing. In particular, if the number of scheduler tasks is low then more threads should be made available to generators, or vice-versa. This leads to a "cooperative" task scheduler and generators.

The cooperation begins by sharing a single thread pool between the `TaskScheduler` and all `AsyncGenerators`. However, this alone is not enough. The tasks to be scheduled (i.e. the calls to Intersect from Triangularize) represent a larger amount of work than any subroutine generator. Hence, to support more coarse-grained parallelism and less parallel overhead, the scheduler should dynamically be given more resources, as needed. Simply stated, the thread pool creates a new "high priority" thread if the thread pool is empty when a new task becomes available. Although the number of active threads may now exceed hardware resources, this is only temporary until a generator finishes. As threads are returned to the thread pool, they may be terminated to account for new high priority threads, keeping the total number of threads within hardware limits. To avoid a runaway task scheduler, the pool limits the number of priority threads that can be created. Naturally, this pattern also works in reverse, when there are few tasks, as more threads will be available in the pool to support more asynchronous generators.

5 EXPERIMENTATION AND DISCUSSION

The preceding two sections have explored various opportunities for parallelism within triangular decomposition algorithms and their implementations. In particular, we have described coarse-grained parallelism where Triangularize calls Intersect in parallel, and the more fine-grained parallelism brought by generators. We now look to evaluate the effectiveness of the different *configurations* of Triangularize. A configuration is parameterized by (1) the type of decomposition being computed (Lazard-Wu or Kalkbrener), (2) the organization of the top-level algorithm (TriangularizeByLevel or TriangularizeByTasks), and (3) the level of parallelization employed (serial, coarse-grained, or coarse- and fine-grained together).

We test the 12 possible configurations of our implementation by considering a suite of over 3000 real-world polynomial systems coming from the scientific literature as well as user-data and bug reports provided by MapleSoft and the RegularChains [16] library. In particular, we look at non-trivial systems taking at least 100ms to solve, in order to warrant the overhead of parallelism. This yields 828 systems. Our results herein are a median of 3 trials and were collected on a node running Ubuntu 14.04.5 with two Intel Xeon X5650 processors each with 6 cores (12 physical threads with hyperthreading) at 2.67 GHz, and a 12x4GB DDR3 memory configuration at 1.33 GHz.

On our test suite, speed-ups of up to 9.5× were found. Further, 824 of the 828 systems saw at least some parallel gains from at least one of the parallel configurations, with 133 having at least 2.0× speed-up. Considering that 203 of those systems contain only a single component, and thus have no potential for parallelism, implies that our implementation limits parallel overheads well. Table 1 presents some examples from the literature with timings and speed-up factors for all 12 configurations. We also compare timings there against the RegularChains package of Maple 2019.

Figure 1 summarizes the data collected for Kalkbrener decomposition as a two-dimensional histogram (the trends are the same for Lazard-Wu decomposition). For each subplot, the x-axis is serial execution time while the y-axis is parallel speed-up factor. It may appear that the task-based method incurs some slow-downs, but, this is mainly for cases running in less than 1s. There, some parallel overhead is expected, particularly as 203 systems have no potential parallelism. Nonetheless, if we consider more substantial examples—those requiring at least 1s to solve—then only 9 examples in the *Kalkbrener-Tasks-Fine* configuration have a speed-up less than 0.9, with the minimum being 0.84. From the trends in the data, we make two observations: (i) TriangularizeByTasks has the potential for higher parallelism than TriangularizeByLevel, and (ii) TriangularizeByLevel is, in general, slowed down by the use of generators while the performance TriangularizeByTasks has improved performance.

From our discussions in the preceding sections, (i) should be very apparent. Our task scheduler, built using the workpile parallel pattern is more receptive to the irregular parallelism present in triangular decomposition. Specifically, it allows new tasks to be taken up immediately by the scheduler's threads. This contrasts with the level-wise scheme where threads must operate in lockstep

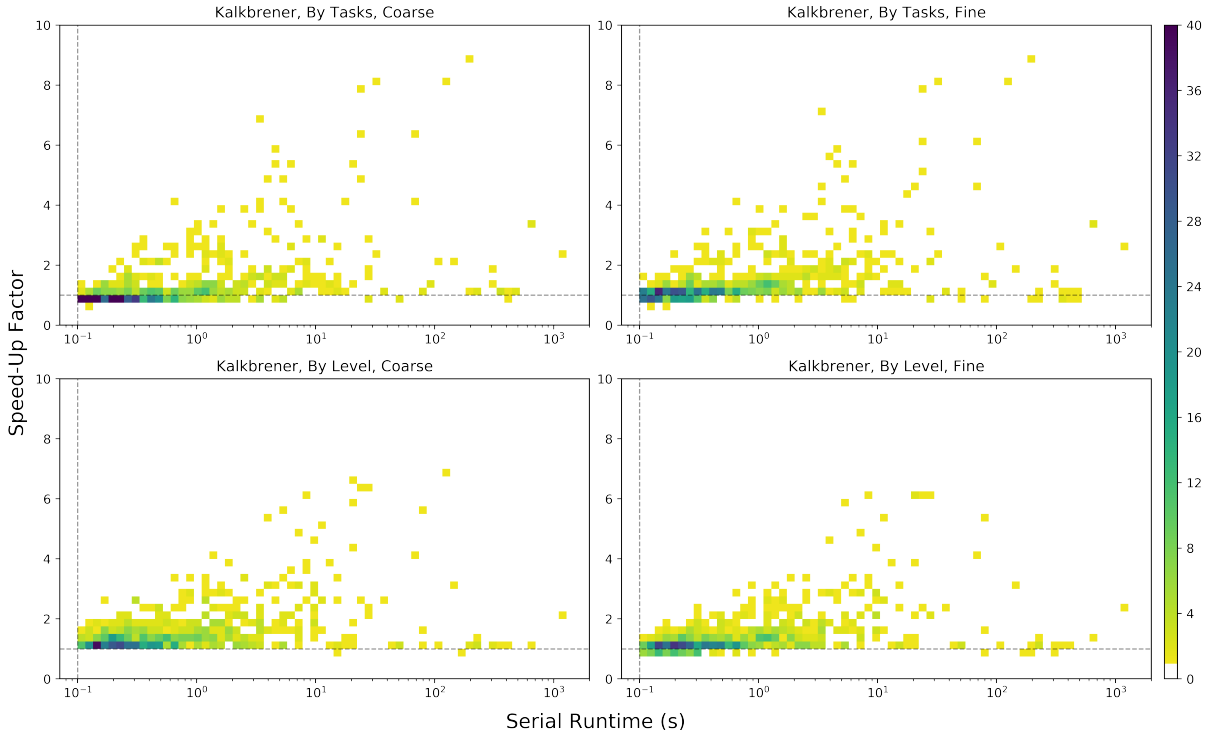

Figure 1: Histograms illustrating the distribution of serial runtime against speed-up factor for Kalkbrener decompositions.

for each map step. Note (ii) also follows from our discussion of over-subscription and resource contention. The task scheduler was implemented to specifically cooperate with the generators, their interplay acting as a sort of dynamic load balancing depending on the number of components discovered during the decomposition. On the other hand, the rather naïve implementation of the map pattern used by TriangularizeByLevel created resource contention with the generator threads and worsened parallel performance compared to using map alone. While TriangularizeByLevel was the weaker performer in terms of parallel speed-up, there still exists examples where intermediate removal of redundant components is an important optimization step (e.g. W41 in Table 1, where TriangularizeByLevel is twice as fast). One should not forgo intermediately removing redundant components just for the parallel benefits of workpile. Indeed, combining redundant component removal alongside the task scheduler is an important area for future work.

Lastly, we consider two specific examples, Systems 2691 and 3295, illustrated as trees in Figure 2. These plots show the evolution of the decomposition as Intersect is called in parallel on independent components in the *Kalkbrener-Tasks-Fine* configuration. Two typical patterns are shown. For 2691, each call to Intersect creates two components. The dynamic and irregular nature of the parallelism is highlighted where each branch is discovered at different times, with each having different workloads. For 3295, the first component splits into several relatively equal branches, except for one branch with considerably more work. Despite the overall high number of components, computations in all branches overlap only briefly

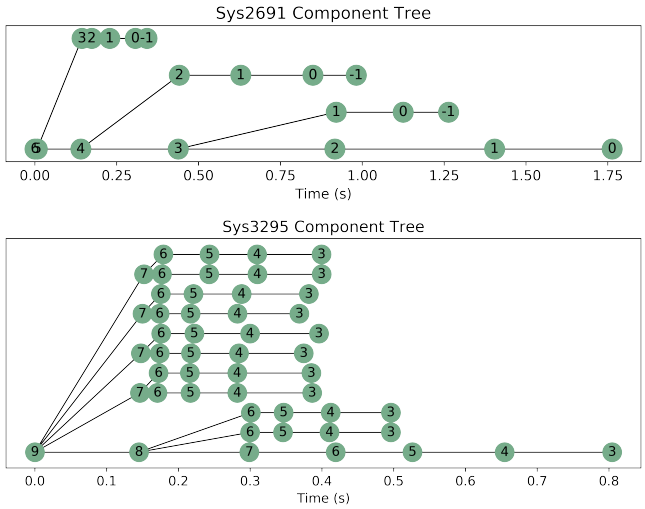

Figure 2: The *component tree* of two systems, showing that components and independent branches of computation are found dynamically during the decomposition. Each node depicts a component, and the node's label is the component's dimension (-1 being the empty set). Edges are drawn where a call to Intersect **on the parent component returned the child.**

since the split in the bottom branch is not found until later, further highlighting the irregularity in parallelism.

Using these trees and the terminology of fork-join parallelism, we may consider a crude upper-limit on potential speed-up as the

System Name	Kalkbrener Decomposition							Lazard-Wu Decomposition						
	Level			Tasks			Maple 19	Level			Tasks			Maple 19
	S. Time	C	C+F	S. Time	C	C+F		S. Time	C	C+F	S. Time	C	C+F	
8-3-config-Li	8.188	3.81	3.14	8.275	2.80	2.84	5.836	36.299	2.93	2.89	38.825	3.07	3.06	26.660
dessin-2	46.974	1.12	1.08	37.090	1.11	1.11	126.008	50.185	1.19	1.14	36.557	1.11	1.07	125.276
dgp6	80.134	5.66	5.47	69.147	6.31	6.06	54.368	78.067	4.02	3.98	67.460	6.12	5.72	49.496
Ducos-7-5	49.001	1.04	1.02	49.542	0.99	1.00	1520.692	48.136	1.00	0.97	50.151	1.03	1.01	1537.293
Gerdt	3.939	3.01	2.88	3.259	3.91	4.57	0.932	3.755	2.65	2.62	3.172	4.03	4.28	0.952
Gonnet	1.406	2.42	2.43	1.683	2.57	2.46	1.924	1.399	2.26	2.09	1.680	2.25	2.77	1.984
Hereman-2	1.178	2.56	2.24	1.117	2.24	2.43	0.472	1.248	2.52	2.30	1.145	2.57	2.32	0.592
Issac97	3502.410	1.40	1.40	311.231	1.34	1.32	445.312	3571.450	1.40	1.42	318.640	1.37	1.34	450.880
Leykin-1	7.194	2.48	2.20	6.376	1.77	2.13	3.316	10.043	2.29	2.10	9.053	1.96	1.85	5.424
lhlp3	0.254	1.39	1.24	0.247	1.09	1.21	0.016	0.193	0.98	0.94	0.192	0.95	0.92	0.016
MacLane	1.137	2.04	1.74	1.170	1.92	1.71	1.748	3.816	1.50	1.42	4.042	1.49	1.41	4.828
MontesS16	2.233	2.33	2.11	2.650	2.30	2.32	2.400	2.177	2.11	2.09	2.592	2.40	2.30	2.488
Pappus	1.839	2.13	1.59	1.940	2.34	1.77	2.704	6.255	2.95	2.30	10.671	3.53	2.87	16.312
Pavelle	1.178	1.62	1.44	1.165	1.28	1.43	0.259	33.179	1.02	1.21	39.707	1.15	1.39	5.352
Reif	20.547	4.33	3.91	20.382	5.34	4.58	10.899	18.465	3.77	3.60	18.703	5.00	4.23	10.691
SEIT	0.593	1.89	1.49	0.635	1.69	1.54	0.448	4.275	3.14	2.48	4.606	3.07	2.64	4.368
W1	10.137	3.08	2.93	10.525	2.79	3.26	2.304	9.806	2.81	2.73	10.160	2.88	3.08	2.500
W41	12.960	3.94	3.80	25.266	4.82	5.16	18.644	12.645	3.65	3.57	24.757	4.86	4.72	15.892
W44	5.294	3.44	3.23	4.251	4.92	5.64	1.132	5.226	3.10	3.01	4.175	4.92	5.11	1.184
W5	20.247	5.98	6.20	22.435	6.38	6.15	14.260	20.248	5.90	6.16	21.951	6.19	6.07	13.180
YangBaxterRosso	0.675	2.42	2.17	0.674	4.03	4.18	0.348	0.638	2.18	1.98	0.663	4.32	4.13	0.371

Table 1: A comparison of timings for the 12 configurations of Triangularize**. Here, serial timings are given along with speed-up factors for coarse (C) and coarse and fine (C+F). Timings for solving using** RegularChains **in Maple 2019 are also included.**

ratio of work (i.e. sum of edges) to span (i.e. the overall decomposition time). This gives 2.13 and 4.97 for Systems 2691 and 3295, respectively, and an "efficiency" (the ratio of actual to potential speed-up) as 87.8% and 74.4%, respectively. This suggests that our implementation indeed exploits the irregular parallelism available, and is able to exploit more parallelism when more is available.

Despite the inherent challenges of irregular parallelism in triangular decomposition, our implementation effectively utilizes that which is available through task parallelism and asynchronous generators. The use of generators in computer algebra is something which we hope to see applied elsewhere to improve upon irregular parallelism. For example, generators could also be applied to polynomial factorization, where factors could be produced and consumed along a pipeline consisting of square free factorization, distinct degree factorization, and equal degree factorization.

For the future of triangular decompositions, we hope to include methods which support more regular parallelism, for example, evaluation/interpolation schemes for the computation of subresultant chains [17] needed by RegularGCD. We also look to perform some of the solving over a prime field, where computations are more likely to split, and then lift the solutions [11].

Acknowledgments

The authors would like to extend thanks to IBM Canada Ltd (CAS project 880), NSERC of Canada (CRD grant CRDPJ500717-16, award PGSD3-535362-2019), and John P. May (Maplesoft).

REFERENCES

[1] M. Asadi, A. Brandt, C. Chen, S. Covanov, F. Mansouri, D. Mohajerani, R. H. C. Moir, M. Moreno Maza, Lin-Xiao Wang, Ning Xie, and Yuzhen Xie. 2018. Basic Polynomial Algebra Subprograms (BPAS). http://www.bpaslib.org.

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

[3] Philippe Aubry, Daniel Lazard, and Marc Moreno Maza. 1999. On the Theories of Triangular Sets. *J. Symb. Comput.* 28, 1-2 (1999), 105–124.

[4] Mordechai Ben-Ari. 1990. *Principles of concurrent and distributed programming.* Prentice Hall.

[5] Francesco Biscani. 2012. Parallel sparse polynomial multiplication on modern hardware architectures. In *ISSAC 2012, Grenoble, France, 2012.* 83–90.

[6] Janko Böhm, Wolfram Decker, Santiago Laplagne, Gerhard Pfister, Andreas Steenpaß, and Stefan Steidel. 2013. Parallel algorithms for normalization. *J. Symb. Comput.* 51 (2013), 99–114.

[7] F. Boulier, F. Lemaire, and M. Moreno Maza. 2006. Well known theorems on triangular systems and the D5 principle. In *Proc. of* Transgressive Computing 2006. Granada, Spain.

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

[9] Reinhard Bündgen, Manfred Göbel, and Wolfgang Küchlin. 1994. A fine-grained parallel completion procedure. In *Proceedings of ISSAC.* ACM, 269–277.

[10] C. Chen and M. Moreno Maza. 2012. Algorithms for computing triangular decomposition of polynomial systems. *J. Symb. Comput.* 47, 6 (2012), 610–642.

[11] Xavier Dahan, Marc Moreno Maza, Éric Schost, Wenyuan Wu, and Yuzhen Xie. 2005. Lifting techniques for triangular decompositions. In *ISSAC 2005, Beijing, China, 2005, Proceedings.* 108–115.

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

[13] M. Gastineau and J. Laskar. 2015. Parallel sparse multivariate polynomial division. In *Proceedings of PASCO 2015.* 25–33.

[14] Jiaxiong Hu and Michael B. Monagan. 2016. A Fast Parallel Sparse Polynomial GCD Algorithm. In *ISSAC 2016, Waterloo, ON, Canada, July 19-22, 2016.* 271–278.

[15] C. E. Leiserson. 2011. Cilk. In *Encyclopedia of Parallel Computing.* 273–288.

[16] F. Lemaire, M. Moreno Maza, and Y. Xie. 2005. The RegularChains library in MAPLE. *ACM SIGSAM Bulletin* 39, 3 (2005), 96–97.

[17] Xin Li, Marc Moreno Maza, and Wei Pan. 2009. Computations modulo regular chains. In *ISSAC 2009, Seoul, Republic of Korea, Proceedings.* 239–246.

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

[19] M. Monagan and R. Pearce. 2010. Parallel sparse polynomial division using heaps. In *Proceedings of PASCO 2010.* ACM, 105–111.

[20] Michael B. Monagan and Baris Tuncer. 2018. 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.* 359–368.

[21] Marc Moreno Maza and Yuzhen Xie. 2007. Component-level parallelization of triangular decompositions. In *PASCO 2007 Proceedings.* ACM, 69–77.

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

[23] Michael L. Scott. 2009. *Programming Language Pragmatics (3. ed.).* Academic Press.

[24] Y. Xie. 2007. *Fast Algorithms, Modular Methods, Parallel Approaches, and Software Engineering for Solving Polynomial Systems Symbolically.* Ph.D. Dissertation.