Programming Research Group

PARALLEL ABSOLUTE IRREDUCIBILITY TESTING VIA POLYTOPES

Fatima Abu Salem*

PRG-RR-04-14

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD

*Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford OX1 3QD, U.K. E-mail: fkas@comlab.ox.ac.uk. The author is supported by the EPSRC.

Abstract

A heuristic algorithm for testing absolute irreducibility of multivariate polynomials over arbitrary fields using Newton Polytopes was proposed in [8]. A preliminary implementation in [9] established a wide range of families of low degree and sparse polynomials for which the algorithm works efficiently and with a high success rate. In this paper, we develop a BSP variant of the absolute irreducibility testing via polytopes, with the aim of producing a more memory and run time efficient method that can provide wider ranges of applicability, specifically in terms of the degrees of the input polynomials. In the bivariate case, we describe a balanced load scheme and a corresponding data distribution leading to a parallel algorithm whose efficiency can be established under reasonably realistic conditions. This is later incorporated in a doubly parallel algorithm in the multivariate case that achieves similar scalable performance. Both parallel models are analysed for efficiency, and the theoretical analysis is compared to the performance of our experiments. In the empirical results we report, we achieve absolute irreducibility testing for bivariate and trivariate polynomials of degrees up to 30000, and for low degree multivariate polynomials with more than 3000 variables.

1 Introduction

Absolute irreducibility testing of polynomials is of importance in various fields in algebra and geometry (see for instance [3, 13, 18, 24, 25]). Convex polytopes have been studied in connection with multivariate polynomials over arbitrary fields. In particular, recent work has examined the connection they bear to factorising bivariate polynomials [1], and to testing absolute irreducibility of multivariate polynomials [6, 8], over arbitrary fields. In the latter case, the problem of testing indecomposability of Newton polytopes formed part of a pseudo-polynomial time algorithm (see [10]) for bivariate absolute irreducibility testing, and of a heuristic and randomised algorithm in the multivariate case. The fact that the non-zero coefficients of the input polynomial do not matter in the testing process makes it possible to show absolute irreducibility of families of polynomials, rather than single polynomials [8]. The empirical sequential tests and timing results reported in [9], on "Fast absolute irreducibility testing via Newton polytopes", indicated a high success rate for a large class of low degree and sparse multivariate polynomials, whose number of terms is bounded by $O(nd)$, where n is the number of variables and d is the upper bound on the degree in each variable. In the instances when the algorithm is highly successful and the input polynomials have a number of non-zero terms bounded by a constant factor of their total degree, the run times have been shown to be at most cubic in the total degree of the input polynomial [9]. For a degree q polynomial with n variables and coefficients from a finite field, for instance, the input size of the irreducibility problem is $N = O(q^n)$, ignoring logarithmic factors. Expressing the complexity of the absolute irreducibility testing algorithm of [8] in terms of the input size, this requires $O(N^{\frac{3}{n}})$, which for polynomials with more than 3 variables implies a run time that is almost linear in the input size. As such, the algorithm can be used as a fast pretest before any of the infallible yet slower irreducibility tests are invoked [5, 7, 11, 14, 16, 17].

Motivated by these original findings and the special feature which makes absolute irreducibility testing dependent in large upon the shape and the size of Newton polytopes, we investigate a parallel scheme mainly set to widen the range of applicability of the algorithm, by making it possible to tackle significantly higher degree, sparse polynomials, and by allowing a more efficient performance for low degree yet denser polynomials. We show that the algorithm can be efficiently parallelised, thanks to some of its inherent features: First, we discuss a balanced load scheme which can be constructed using the pattern of computations in the sequential case in [8]. Second, we discuss how a corresponding data distribution representing lattice points inside polytopes in $\mathbb{R}^2$ can be constructed, adhering to the balanced load scheme, and allowing a scalable parallel algorithm aimed at high degree irreducibility testing. We adopt the BSP model for parallel computation [4, 19, 20], and we analyse the conditions necessary for an efficient parallel performance in the bivariate case. This then serves as a sub-problem for the multivariate case, where a model involving parallelism at two different levels is described. The BSP algorithm makes it possible to test significantly higher degree polynomials than can be allowed in the sequential case in [8], and hence, to the best of our knowledge, using any other known absolute irreducibility testing algorithm. We further study both parallel models for issues relating to efficiency, and establish conditions under which a good performance is guaranteed. Our empirical results agree closely with the theoretical estimates, and we report on our implementation in this respect.

The remainder of this paper is organised as follows. In Section 2 we recall the absolute irreducibility test on which our work is based from [8]. In Section 3 we study a parallel scheme for bivariate polynomial irreducibility testing, and in Section 4 we address the multivariate case. Finally, empirical results are presented and analysed in Section 6.

2 Background

2.1 Polynomials and Newton polytopes

For an extensive review on the theory of convex polytopes we refer the reader to [12]. Let $\mathbb{R}$ denote the field of real numbers and $\mathbb{R}^n$ the Euclidean n-space. A *convex polytope* in $\mathbb{R}^n$ is the smallest convex set containing a finite set of points in $\mathbb{R}^n$. Let $\mathbb{F}[X_1, X_2, \dots, X_n]$ be the ring of polynomials in n variables over an arbitrary field $\mathbb{F}$. For any vector $e = (e_1, \dots, e_n)$ of non-negative integers, define $X^e := X_1^{e_1} \dots X_n^{e_n}$. Let $f \in \mathbb{F}[X_1, \dots, X_n]$ be given by

$$f := \sum_e a_e X^e \tag{1}$$

where the sum is over finitely many points e in $\mathbb{N}^n$ called support vectors of f, and $a_e \in \mathbb{F}$. The Newton polytope of f is the polytope in $\mathbb{R}^n$ obtained as the convex hull of all exponents e for which the corresponding coefficient a_e is non-zero. It has integer vertices, since all the e are integral points. We call such polytopes *integral*. Given two polytopes Q and R, their *Minkowski sum* is defined to be the set

$$Q + R := \{q + r \mid q \in Q, r \in R\}. \tag{2}$$

When Q and R are integral polytopes, so is $Q + R$. If we can write an integral polytope P as a Minkowski sum $Q + R$ for integral polytopes Q and R then we call this an *integral decomposition*. The decomposition is *trivial* if Q or R has only one point, and P is *integrally decomposable* if it has at least one non-trivial decomposition. If a polytope has no non-trivial decompositions then it is *integrally indecomposable*. The Minkowski sum of two convex polytopes is also a convex polytope. A polytope of dimension 2 is a *polygon*, where the only proper faces are edges and vertices.

Let $f \in \mathbb{F}[X_1, ..., X_n]$ be a non-constant polynomial and let $Supp(f)$ denote the set of all its support vectors. The *total degree* of f when $f \neq 0$ is defined to be the maximum value of $\sum_{1 \leq i \leq n} e_i$ over all $(e_1, ..., e_n) \in Supp(f)$. We call f *absolutely irreducible* over $\mathbb{F}$ if it has no non-trivial factors over $\overline{\mathbb{F}}$, the algebraic closure of $\mathbb{F}$. Absolute irreducibility forms a strong irreducibility criterion because f has no irreducible factors over $\mathbb{F}$ if f is absolutely irreducible. The following theorem is at the heart of an absolute irreducibility criterion:

Theorem 1 (Ostrowski) *[22] Let $f, g, h \in \mathbb{F}[X_1, \ldots, X_n]$. If $f = gh$ then* $\text{Newt}(f) = \text{Newt}(g) + \text{Newt}(h)$.

Corollary 2 (Irreducibility Criterion) *[6] Let $f \in \mathbb{F}[X_1, ..., X_n]$ with f not divisible by any non-constant X_i, for $1 \leq i \leq n$. If* $\text{Newt}(f)$ *is integrally indecomposable, then f is absolutely irreducible.*

For proof, see [6].

2.2 Testing indecomposability of polytopes

2.2.1 Polygons

Following the discussion above, testing absolute irreducibility of multivariate polynomials over arbitrary fields is thus reduced to deciding whether a given polytope is integrally decomposable. Assuming that the polytope is given as a list of its vertices, the input size of this problem consists in the length of the binary representation of the coordinates of the vertices. It was established in [8] that deciding polygon indecomposability (and hence indecomposability of higher dimensional polytopes) is NP-complete, and thus it remains an open problem to develop an efficient, polynomial time, deterministic or even randomised algorithm for testing general integral polytopes for indecomposability. Gao and Lauder developed a pseudo-polynomial time algorithm with a run time complexity that is polynomial in the lengths of the sides of the polygon, rather than in the logarithm of the lengths [8]. At this stage we shall need few more terminologies. Let P be a convex polygon in $\mathbb{R}^2$, and let $v_0, ..., v_{m-1}$ denote its vertices ordered cyclically in a counter-clockwise direction. The edges of P are vectors of the form $E_i = v_{i+1} - v_i = (a_i, b_i)$, for

$0 \leq i \leq m-1$, where $a_i, b_i \in \mathbb{Z}$ and the indices are taken modulo m. A vector $v = (a, b) \in \mathbb{Z}^2$ is called a primitive vector if $\gcd(a, b) = 1$. If $n_i = \gcd(a_i, b_i)$ and $e_i = (a_i/n_i, b_i/n_i)$, then $E_i = n_i e_i$, where e_i is a primitive vector, for $0 \leq i \leq m-1$. The sequence of vectors $\{n_i e_i\}_{0 \leq i \leq m-1}$ is called the edge sequence or polygonal sequence and uniquely identifies the polygon up to translation determined by v_0. Since the boundary of a polygon forms a closed path, we have that $\sum_{0 \leq i \leq m-1} n_i e_i = (0, 0)$. For convenience, an edge sequence can be identified with that obtained by extending the sequence by inserting an arbitrary number of zero vectors, and so we can assume that the edge sequence of a summand of P has the same length as that of P. We now recall the polygon indecomposability testing algorithm from [8]:

Algorithm 3 Input: The edge sequence $\{n_i e_i\}_{0 \leq i \leq m-1}$ of an integral convex polygon P starting at a vertex v_0 where $e_i \in \mathbb{Z}^2$ are primitive vectors.
Output: Whether P is decomposable.

Step 1: Compute the set IP of all the integral points in P, and set $A_{-1} = \emptyset$.

Step 2: For $i = 0, ..., m-2$, compute the set of points in IP that are reachable via the vectors $e_0, ..., e_i$:

2.1: For each $k = 1, ..., n_i$, if $v_0 + ke_i \in IP$, then add it to A_i.
2.2: For each $u \in A_{i-1}$ and $k = 0, ..., n_i$, if $u + ke_i \in IP$, then add it to A_i.

Step 3: Compute the last set A_{m-1}: For each $u \in A_{m-2}$ and $k = 0, ..., n_{m-1} - 1$, if $u + ke_{m-1} \in IP$ and $u + ke_{m-1}$ is not already in A_{m-1}, then add it to A_{m-1}.

Step 4: Return "Decomposable" if $v_0 \in A_{m-1}$ and "Indecomposable" otherwise.

Theorem 4 *[8] The above algorithm decides decomposablitiy correctly in $O(t'mN)$ vector operations where t' is the number of integral points in P, m is the number of its edges, and N is the maximum number of integral points on an edge.*

For a detailed a proof of the above theorem we refer the reader to [8] and we sketch only the basic idea for the sake of clarity. All the points in A_{m-1} that are points in IP reachable via the vectors $e_0, ..., e_{m-1}$ are constructed to be of the form $v_0 + \sum_{i=0}^{m-1} k_i e_i$, $0 \leq k_i \leq n_i$. If one of the points in A_{m-1} is equal to v_0, then $\sum_{i=0}^{m-1} k_i e_i = (0, 0)$, and so the sequence $\{k_i e_i\}$ forms the edge sequence of a proper integral summand Q of P. On the other hand, it can be easily shown that the edge sequence of any proper integral summand of P will be detected by the above algorithm.

2.2.2 Higher dimensional polytopes

This work can be carried further to deal with general polytopes in $\mathbb{R}^n$, and the above algorithm can be integrated into a heuristic randomised test for higher dimensional polytope indecomposability [8]. The main idea behind this approach relies on the use of random integral linear maps which project a given polytope into a polygon in a plane. If the projected polygon is indecomposable, and under certain conditions presented in the lemma below, one deduces that the original polytope is indecomposable.

Lemma 5 *[8] Let S be a set of integral points in $\mathbb{R}^n$ and let A be a $2 \times n$ integral matrix. Define the set $A(S) = \{As | s \in S\}$ where each point in $\mathbb{R}^n$ is viewed as a column vector. If the polygon $conv(A(S))$ is indecomposable, and each of its vertices has exactly one pre-image under A in S, then the polytope $conv(S)$ is indecomposable.*

For proof, see [8].

The above result forms the basis of an algorithm for testing indecomposability of polytopes in $\mathbb{R}^n$ (and hence absolute irreducibility of multivariate polynomials) as follows:

Algorithm 6 [8]
Input: $f \in \mathbb{F}[X_1, ..., X_n]$ with no non-constant monomial factors, and S_f the set of exponent vectors of non-zero terms of f of cardinality c.
Output: Absolutely irreducible or Failure, where the latter case means that decomposability of $conv(S_f)$ (and hence absolute irreducibility of f) is not decided.

Step 1: Re-arrange the points in S_f as an $n \times c$ matrix S. If $n = 2$, let A be the trivial projection and go to Step 4. Else, choose positive integers b and e. Let $M(b)$ denote the set of all $2 \times n$ matrices with integer coefficients bounded in absolute value by b. Repeat Steps 2-4 up to e times.

Step 2: Select a matrix A uniformly at random from $M(b)$ and compute the set of points in $\mathbb{R}^2$ defined by $A(S) := \{As | s \in S\}$.

Step 3: Compute the convex hull, $conv(A(S))$, of $A(S)$ and check that each vertex of $conv(A(S))$ has only one pre-image in S under the projection A. If this condition is not met, return to Step 2.

Step 4: Call Algorithm 3 above using the edge sequence of $conv(A(S))$. If this polygon is integrally indecomposable, output "Absolutely Irreducible" and halt. Else, if $n > 2$, return to Step 2. Else, if $n = 2$, output "Failure".

Step 5: Output "Failure".

The following theorem appears in [9]:

Theorem 7 *The above algorithm works correctly and requires*

$$eO(((nbd)^3 + c(c + n)) \log^2(nbd))$$

binary operations and $O((nbd)^2 \log(nbd))$ bits of storage for a polynomial in n variables, with c non-zero terms, and degree at most d in each variable. If f has no more that $c = O(nd)$ non-zero terms, the run time becomes cubic in the total degree of the input polynomial.

For proof, see [9].

The implications of the above theorem make it possible to avoid working with convex hulls of points in $\mathbb{R}^n$ and testing them for integral indecomposability. Instead, it often becomes sufficient to project these points onto the plane and invoke a polygon indecomposability testing algorithm. Though promisingly efficient for polynomials whose number of terms is not much greater than their total degree (see [9]) for an empirical study in the case of sparse polynomials), the above method is still considered a heuristic for the following reasons: Although the probability that the condition in Step 3 is satisfied can be determined, a rigorous analysis is still missing concerning the probability with which the algorithm will output indecomposable if $conv(S)$ is so, but the shadow polygon is decomposable [8]. In other words, it still needs to be determined how likely it is that the algorithm will establish indecomposability if $\mathrm{Newt}(f)$ is indecomposable, since it is possible that there are indecomposable polytopes whose shadow polygons are always decomposable [8]. On the other hand, since it has been proven that most polytopes in $\mathbb{R}^n$, for $n \geq 3$, are homothetically indecomposable [23], a direct consequence of Proposition 12 of [8] is that most random integral polytopes may be expected to be indecomposable.

3 Parallel bivariate absolute irreducibility testing

In all the following we shall assume that nd fits in a machine word for an input polynomial with n variables and upper bound d on each variable. We will thus measure run time in bit operations and space in bits.

Since the Newton polytope of a bivariate polynomial already lies in $\mathbb{R}^2$, a strong feature of Algorithm 6 is that it will always decide indecomposability of polygons, and hence will always establish absolute irreducibility of the input bivariate polynomial f if $\mathrm{Newt}(f)$ is indecomposable. Initial empirical results in [9] indicate that the algorithm has a high success rate when the number of terms is bounded by a constant multiple of the total degree, that it detects the indecomposability of their Newton polytopes quickly very often, and hence that it should be particularly effective for random sparse polynomials [8]. While these observations still stand as conjectures based on empirical results, it is important to extend the range of applicability of the algorithm for higher degrees, and to support initial arguments related to the rate of success of the algorithm. However, as degrees of the input bivariate polynomials increase, so do the sizes of their Newton polytopes, and hence one major difficulty limiting attempts to improve the performance of Algorithm 3 becomes the space requirement. Yet, the fact that one can shift the focus from a data structure and distribution representing bivariate polynomials to one that represents shapes of Newton polytopes in $\mathbb{R}^2$ suggests that a parallel re-construction of the irreducibility testing algorithm which exploits the shapes of the polygons can achieve considerable improvements, both in the operational and spatial complexities. In the remainder of this section, we discuss our first BSP model for a bivariate absolute irreducibility testing based on a parallel polygon indecomposability testing algorithm.

Let $f \in \mathbb{F}[x, y]$ be of total degree q with no non-constant monomial factors and let

IP denote the set of integral points belonging to Newt(f). Since all vectors in Supp(f) have positive integral coordinates that are at most equal to q, Newt(f) can be embedded within a square of dimension $O(q^2)$. Moreover, Algorithm 3 requires a description of IP in two different ways. First, one has to have, at the beginning of the algorithm, a structure by which one can test an arbitrary point for inclusion in IP. Second, the algorithm re-constructs integral points in IP following paths of the form $v_0 + \sum_{0 \leq i \leq j} k_i e_i$, for $0 \leq k_i \leq n_i$ and $j = 0, ..., m-1$. In the following discussion, we aim to show how the problems associated with each of the two representations can be circumvented by reducing the quadratic spatial factor.

3.1 Computing the set of all integral points in a polygon

Often in the innermost loops of Steps 2 and 3 of Algorithm 3, one will have to test for inclusion of arbitrary points in IP. Computing and then storing the set of all integral points belonging to the polytope requires about $O(q^2)$ bits of storage, which can become highly restrictive even for moderately large input degrees. Another alternative approach which does not require that we store any lattice points but performs a test of inclusion based on the "leftedness" of an arbitrary point with respect to all directed edges of the polytope has a cost of $O(m)$ integer operations (see [21], Ch. 3). In our application, it can be costly to invoke this test very frequently, incurring a cost of $O(m^2)$ for each loop iteration in Steps 2 and 3 of Algorithm 3 above, and hence a cost of $O(t'm^3N)$ integer operations in total. Alternatively, we adopt a strategy which requires that we store only a "useful" subset of IP with no more than $O(q)$ points. Let x_{min}, x_{max}, y_{min} and y_{max} denote respectively the lowest and highest x coordinates, and the lowest and highest y coordinates, of all $e \in$ Supp(f). We then store only the intersection points between all horizontal (or vertical) lines $y = k$, for $k = y_{min}, ..., y_{max}$, using only $y_{max} - y_{min} + 1 = O(q)$ bits of memory. However, unlike the case in [2] where computations are performed such that all output is integral, we need a weaker condition affecting the following. The intersection of the polytope with a horizontal straight line is obtained by intersecting the polytope with at most all of its edges (until two points of intersection are found, not necessarily distinct). If $ux + vy + w = 0$ denotes the generic equation of a line ℓ' defining an edge of Newt(f), then finding the intersection of ℓ' with $y = k$ requires that we solve for x in

$$ux + vk + w = 0 \text{ or } x = \frac{-w - vk}{u}, \tag{3}$$

when $u \neq 0$. Although this division operation involves only integral values, the quotient itself may not be an integer, in which case x has to be declared as a float or a double, for otherwise, the division might produce a rounded-off integral value $\lfloor x \rfloor$, which may not correspond to a lattice point in IP. Note that computing this subset of IP must be performed as a pre-computation, requiring at most $O(qm)$ floating point operations.

3.2 Constructing sets of points along paths of edges

Obviously, the above strategy of reducing the description of IP by preserving only a smaller subset of it cannot be extended when dealing with the main computations of Algorithm 3. Specifically, one needs to have all points of IP that are reachable via a subset of edges $e_0, ..., e_i$, for $0 \leq i \leq m-2$, in order to find a larger subset reachable via $e_0, ..., e_{i+1}$. Eventually, the last path requires that one has available almost all of the points in IP, and hence, the best that one can attempt is to distribute the points in IP among a fixed number of processors. Since this task is never so immediate, we need to address several of the following issues: Whether there exists at all any potential parallelism in the main computations of Algorithm 3 that can make such an approach possible, whether there exists a balanced load scheme that ensures all processors are almost equally engaged in the independent computations, and finally, whether there exists a data distribution which not only ensures that a given number of n_p processors, say, store at most $O(q^2)/n_p$ lattice points in IP, but also adheres to the pattern of the proposed balanced load scheme.

3.3 Detecting independent computations

Recall that the main computations in Algorithm 3 build up iteratively by constructing subsets of IP in each step of the iteration. This may make it easy to detect a straightforward parallelism as the algorithm progresses forward. However, it is almost immediate to see that the inner-most computations of the iterative loop can themselves be independent, generating a "horizontal" inherent parallelism across each iteration of the main loop over edges of the input polygon. In particular, we have:

Lemma 8 *Algorithm 3 for polygon indecomposability testing contains two patterns of computations, one which describes a sequence of inter-dependent iterative steps for constructing new subsets of IP using previous subsets, and another pattern describing completely independent tasks for vector operations across a fixed iterative step.*

Proof: During a fixed stage $i = 0, ..., m-2$ of the main loop iteration (Step 2) of Algorithm 3, one finds all points $u' \in A_i$ satisfying the following:

1. $u' = v_0 + ke_i$, for $0 < k \leq n_i$, in which case the computations over all k require no information from the previous loop iteration of index $i-1$, and each computation per fixed k requires no information apart from e_i.
2. $u' = u + ke_i$, for $0 \leq k \leq n_i$ and for all $u \in A_{i-1}$, in which case the computations over all u require information from the previous loop iteration of index $i-1$, each computation per fixed u and over all $k = 0, ..., n_i$ requires no information apart from n_i and e_i, and each computation per fixed u and fixed k requires no information apart from e_i.

For $i = m-1$, one finds all points $u' \in A_i$ satisfying $u' = u + ke_{m-1}$, for $0 \leq k < n_{m-1}$ and $u \in A_{m-2}$, which involves the dependencies described in 2 above.

3.4 Constructing a balanced load scheme

We will now attempt to examine the geometric pattern of the computations above, on which we can base a possible load balancing scheme. To this end, we will consider a slight modification of Algorithm 3 to produce sets of points B_i in $\mathbb{R}^2$, for $0 \le i \le m-1$, constructed as follows:

1. Initialise $B_i \leftarrow \emptyset$, for $i = -1, ..., m-1$.
2. For $i = 0, ..., m-1$, compute the set of points of the plane that are reachable via the vectors $e_0, ..., e_i$:
3. For each $u \in B_{i-1}$ and $k = 0, ..., n_i$, add $u + ke_i$ to B_i.

The differences between all the sets A_i and B_i are that v_0 is in every single set B_i, that B_i contains points reachable via $e_0, ..., e_i$ which are not necessarily in IP, and that the points $v_0 + ke_{m-1}$, for $k = 0, ..., n_{m-1}$, do lie in B_{m-1}. As such, it is clear that $A_i \subseteq B_i$ for every i. Because the sets B_i have weaker conditions characterising their points, it will be simpler to describe the geometric pattern they follow. Since $A_i \subseteq B_i$ for every i, any such pattern will apply to elements of A_i. Moreover, we will define every region B_i to be an *active* region in the sense that all computations and possible communications across a main loop iteration of index i in Algorithm 3 are restricted to only those points belonging to B_i, but not to $\mathbb{R}^2 - B_i$.

Let P denote a convex polygon and w a vector in $\mathbb{R}^2$, and let $Tr_w(P)$ denote the image of P under translation by w. The following result gives an explicit geometric description of B_i, for $0 \le i \le m-1$.

Lemma 9 *Let P be a convex polygon with vertices $v_0, ..., v_{m-1}$ and edge sequence $\{n_i e_i\}$, for $0 \le i \le m-1$, where $e_i \in \mathbb{Z}^2$ are primitive vectors. For each iterative step $i = 0, ..., m-1$ of the polygon indecomposability test in Algorithm 3, the computations are restricted to active regions B_i of the plane representing points reachable from v_0 via $e_0, ..., e_i$. Furthermore, the regions can be defined inductively as follows:*

1. *For $i = 0$, $B_0 = conv(v_0, v_1)$,*
2. *For $1 \le i \le m-1$, $B_i = conv(\cup\{Tr_{ke_i}(B_{i-1})\}_{0 \le k \le n_i})$.*

Moreover, $\mathrm{Newt}(f) \subseteq B_{m-1}$.

Proof: We prove the assertion by induction on i. For $i = 0$, B_0 consists only of the points $v_0 + ke_0$, for $k = 0, ..., n_0$. But this spans all integral points along the first edge $E_0 = n_0 e_0$, so that $B_0 = conv(v_0, v_1)$. We now assume the assertion is true for $i \le m-2$, i.e., that the set of points in B_i of the form $v_0 + \sum_{0 \le i \le m-2} k_i e_i$, for $k_i = 0, ..., n_i$, constitutes a convex polygon as defined in 2 above. Since all the points $u \in B_i$ lead to points u' in B_{i+1} obtained as

$$u' = u + je_{i+1} = (v_0 + \sum_{0 \le i < m-1} k_i e_i) + je_{i+1}, \tag{4}$$

for $k_i = 0, ..., n_i$ and $j = 0, ..., n_{i+1}$, this reduces to translating all points of B_i by je_{i+1}, for all j. Since B_i is convex, its image is also a convex set, whose vertices are defined by the images under the translation $Tr_{je_{i+1}}$ of vertices of B_i. Let C denote the convex hull of all vertices in the union of the sets $Tr_{je_{i+1}}(B_i)$ over all $j = 0, ..., n_{i+1}$. We shall show that $C = B_{i+1}$. Since C contains all possible points in the sets $Tr_{je_{i+1}}(B_i)$, we have that $C \subseteq B_{i+1}$. On the other hand, and by Eq. (4) above, any point in B_{i+1} belongs to some set $Tr_{je_{i+1}}(B_i)$, so that $B_{i+1} \subseteq C$. This establishes our inductive proof. Since $A_i \subseteq B_i$, for $i = 0, ..., m-1$, and since the union of all the A_i's comprises the set of all points in IP, we have $\text{Newt}(f) \subseteq \cup_{0 \leq i \leq m-1} B_i$. But $B_{m-1} = \cup_{0 \leq i \leq m-1} B_i$. This establishes our proof.

3.5 Constructing a balanced data distribution

The above lemma provides the general guidelines under which a balanced load scheme can be chosen. In particular, it is immediate that the bulk of the work during any iterative step of Algorithm 3 takes place in well-defined active zones of the plane. One should thus avoid any form of data distribution whereby the polygon is triangulated into zones and each zone is exclusively assigned to one single processor. Specifically, this risks having some processors completely idle when others are engaged in the active zones. Instead, we propose the following:

Lemma 10 *Let B_i, for $0 \leq i \leq m-1$, denote an active region of the plane as defined in Lemma 9 above, and let b_i denote the total number of lattice points belonging to the smallest rectangle containing B_i. Let n_p denote the total number of processors operating in parallel such that $n_p < \sqrt{b_i}$. The data distribution of integral points in IP which allocates every point $(k, k') \in B_i$ to the processor with identification number $id \equiv (k + k') \bmod n_p$ allows for a balanced load scheme as required by Lemma 9 above, and assigns to each processor $O(b_i)/n_p$ integral points, where b_i represents the upper bound on the number of integral points in B_i.*

Proof: Consider each of the convex active regions B_i of the plane, for $i = 0, ..., m-1$, defined in Lemma 9 above. Let x_1, y_1, x_2 and y_2 denote respectively the lowest x and y coordinates and the highest x and y coordinates appearing in any point belonging to B_i. Assume without loss of generality that B_i is translated so that $x_1 = y_1 = 0$. Then the maximum of $(k + k')$ over all $k = x_1, ..., x_2$ and $k' = y_1, ..., y_2$ is equal to $\sqrt{b_i}$. Let t'_{id} denote the total number of lattice points of B_i assigned to a processor with identification number $id = 0, ..., n_p - 1$. For any integer $z \geq 0$ define the class associated with z:

$$[z] = \{(x, y) | x, y \in \mathbb{Z}, x, y \geq 0, \text{ and } x + y = z\}, \tag{5}$$

and let $\#[z]$ denote the number of elements in this class. Then $\#[z] = z + 1$. Given an arbitrary point (k, k') of B_i and $0 \leq id < n_p$, the distribution which maps (k, k') to processor $id \equiv (k + k') \bmod n_p$ assigns to it all classes of the form $[id + hn_p]$, such that

$$id \leq id + hn_p \leq \sqrt{b_i}. \tag{6}$$

Since $n_p < \sqrt{b_0}$, we have that $n_p < \sqrt{b_i}$, for $i = 1, ..., m-1$, and we can require

$$0 \leq h \leq \left\lfloor (\sqrt{b_i} - id)/n_p \right\rfloor. \tag{7}$$

Hence, the total number of points in B_i that are assigned to processor id is at most

$$t'_{id} = \sum_{h=0}^{\lfloor (\sqrt{b_i}-id)/n_p \rfloor} (id + hn_p + 1). \tag{8}$$

For $0 \leq id \leq n_p - 1$, we have

$$\begin{array}{rcl} t'_{id} & \leq & \sum_{h=0}^{\lfloor (\sqrt{b_i}-id)/n_p \rfloor} (n_p + hn_p + 1) \\ & \leq & \sum_{h=0}^{\lfloor \sqrt{b_i}/n_p \rfloor} (n_p + hn_p + 1) \\ & = & \frac{\left\lfloor \frac{\sqrt{b_i}}{n_p} \right\rfloor (\left\lfloor \frac{\sqrt{b_i}}{n_p} \right\rfloor + 1)}{2} n_p + (n_p + 1)(\left\lfloor \frac{\sqrt{b_i}}{n_p} \right\rfloor + 1) \\ & \leq & \frac{(\frac{\sqrt{b_i}}{n_p} + 1)(\frac{\sqrt{b_i}}{n_p} + 2)}{2} n_p + (n_p + 1)(\frac{\sqrt{b_i}}{n_p} + 2) \\ & \leq & \frac{1}{2n_p}(\sqrt{b_i} + 2n_p)^2 + (n_p + 1)(\sqrt{b_i} + 2n_p) \\ & < & \frac{1}{2n_p}(3\sqrt{b_i})^2 + 3\sqrt{b_i}(\sqrt{b_i} + 1) \\ & = & \frac{15b_i}{2n_p} + 3\sqrt{b_i}. \end{array} \tag{9}$$

This establishes $t'_{id} = \frac{O(b_i)}{n_p}$.

In practice, and even though the condition $n_p < \sqrt{b_0}$ may not easily hold (as B_0 is simply the first edge of $\mathrm{Newt}(f)$, in which case b_0 denotes the number of integral points on that edge), we note that the sizes of the sets B_i, for $i > 0$, start growing fast immediately afterwards; specifically, since the number of lattice points in B_1 is at least four, we expect all processors to be engaged in their assigned computations as soon as the first edge of the input polygon is examined. Another relevant implication of the above lemma is the following:

Corollary 11 *Let $f \in \mathbb{F}[x, y]$ be a non-constant polynomial with no non-constant monomial factors and IP denote the set of integral points in $\mathrm{Newt}(f)$. Let n_p denote the total number of processors operating in parallel such that $n_p < q$. The data distribution of integral points in IP which allocates every point $(k, k') \in \mathrm{Newt}(f)$ to the processor with identification number $id \equiv (k + k') \bmod n_p$ assigns to each processor $O(q^2)/n_p$ integral points, where q^2 is an upper bound on the number of lattice points in $\mathrm{Newt}(f)$.*

Proof: By Lemma 9, $\mathrm{Newt}(f) \subseteq B_{m-1}$, and so $q^2 \leq b_{m-1}$, where b_{m-1} denotes an upper bound on the number of lattice points in B_{m-1}. In particular, we have $q^2 = kb_{m-1}$ for some $0 < k \leq 1$. If $n_p < q$, then $n_p < \sqrt{b_{m-1}}$ and by Lemma 10, each processor will be allocated $O(b_{m-1})/n_p$ lattice points, and so $O(q^2)/n_p$ integral points of $\mathrm{Newt}(f)$.

3.6 Removing redundancies in computation

Recall that in Algorithm 3 above, one has to test for redundancies in appending points u to each set A_i. The reason that this needs to be done is that it may be possible to find two points u and u' in A_{i-1}, and two positive integers $k, k' \leq n_i$, such that $u + ke_i = u' + k'e_i \in IP$. If left unchecked, this may produce up to $O(N)$ copies of the same point, so that in the worst case scenario, each multi-set A_i will have $O(t'N^i)$ points, and the run time of polygon indecomposability testing will become exponential in the edges of the input polygon. Checking for redundancies in the sequential case can be made at a cost not exceeding that of an integer operation per point. In particular, one can use a double array of integers, say $Flag$, of total size t', and whose entries are all initialised to PASS. $Flag[k][k']$ is then set to FAIL when a point (k, k') is first added to A_i, so that any future attempts to add another copy of the point are halted upon a simple check of the value in $Flag[k][k']$.

The above strategy, however, can become extremely inefficient in the parallel setting as follows. Let $A_i(id)$ denote the set of points in IP that are reachable via $e_0, ..., e_i$ and that are assigned to processor id. Assume that $i < m - 1$ and that some processor p_a computes the vector operation $v = u + ke_i$, for some $u \in A_i(a)$, and some $k = 0, ..., n_i$. Suppose further that all processors have their own copy of a $Flag$ array, $Flag(id)$, labelling points that have already been added to their sets $A_i(id)$. If $v \in IP$ and v has to be assigned to some other processor p_b, then p_a has to read the information in the entry of array $Flag(b)$ corresponding to point v. This involves a communication step for every such point v. Moreover, reading from remote locations requires a synchronisation barrier for all processors to update their communicated values. This also requires a synchronisation step for every such point v. Consequently, a process like the above will require extremely expensive communication and synchronisation costs which can even overwhelm the computational cost of the sequential algorithm. Alternatively, we introduce the following iterative strategy that will later be shown to come at a very modest cost: For any processor $id = 0, ..., n_p - 1$, we know that $A_0(id)$ does not contain any redundant points. For $i > 0$, we proceed as follows:

- Start with an irredundant set $A_{i-1}(id)$.
- Add all relevant points to their corresponding locations in $A_i(id')$, for some processor id' different from or equal to id, without any check on redundancies.
- When the computations across the iterative step of index i are over, remove all redundancies in $A_i(id)$.

More details illustrating where communication and synchronisation steps should be invoked in the above will be given next in our parallel algorithm for polygon indecomposability testing.

3.7 A BSP algorithm for testing polygon indecomposability

We are now ready to present our parallel algorithm based on the BSP model for parallelisation. We shall recall a brief description of the model and refer the reader to [4, 19, 20, 26] for a more extensive review.

A BSP computer consists of a set of p processors each with its own private memory, and having remote access to other processors' private memories through a communication network. A BSP algorithm consists of a sequence of parallel steps, denoted by *supersteps*. A computation superstep is a series of computations performed on local data available to the processor before the superstep. A communication superstep is a series of communications in the form of sending or receiving a number of non-local data between processors that are needed to perform local computations. Communication supersteps are followed by *synchronisation barriers*, whereby all transferred data is updated. A BSP computer can be described by the following four parameters:
- n_p, the number of processors available;
- s, the processor speed in flop/sec;
- $g(n_p)$, the time (in flop time units) it takes one processor to communicate (send or receive) a data element when n_p processors are operating in parallel;
- $\ell(n_p)$, the time (in flop time units) it takes all n_p processors to synchronise.

We distinguish between the BSP cost of an algorithm and its expected running time. The BSP cost is established using the parameters g and ℓ and the estimate of the execution time is obtained by dividing the BSP cost in flop time units by s, the single processor speed. The BSP cost of an algorithm is simply the sum of the BSP costs of its supersteps. The complexity of a superstep is defined as

$$w_{max} + g(n_p).h_{max} + \ell(n_p), \tag{10}$$

where w_{max} is the maximum number of flops performed, and h_{max} is the maximum number of messages sent or received by any one processor during that superstep.

Our algorithm is designed as a SPMD model, in which a single program with multiple data is encountered by all processors, which then execute their own version of the program, as distinguished by their own identification number, $id = 0, ..., n_p - 1$.

Algorithm 12 (Parallel Polygon Indecomposability Test)
Input: The edge sequence $\{n_i e_i\}_{0\leq i\leq m-1}$ of an integral convex polygon P starting at a vertex v_0 where $e_i \in \mathbb{Z}^2$ are primitive vectors, and a number n_p of processors such that $n_p \geq 1$. Let y_{min}, y_{max}, x_{min} and x_{max} denote the respective smallest and largest y coordinates, and the smallest and largest x coordinates, of vertices of P, and assume further that

$$n_p \leq \min((x_{max} - x_{min} + 1), (y_{max} - y_{min} + 1)). \tag{11}$$

Output: Whether P is decomposable.

Step 1: Define a double integer array Int of size $2(y_{max} - y_{min} + 1)$. Let id denote the processor identification number, and N denote the maximum over all n_i's, for $i = 0, ..., m-1$.

Step 2: Set $h \leftarrow y_{min} + id$, and while $h \leq y_{max}$, do:

2.1: Compute the points of intersection of the line $y = h$ with P.

2.2: Broadcast the x coordinates of points of intersection to all processors at the entries $Int[h][0]$ and $Int[h][1]$ respectively.

2.3: Set $h \leftarrow h + n_p$.

Step 3: $bsp\_sync()$.

Step 4: Set $A_i \leftarrow \emptyset$, for $i = -1, ..., m-1$, and $Result \leftarrow Indecomposable$. Let $M = (y_{max} - y_{min}).(x_{max} - x_{min})$. Define an auxiliary array of integer vectors, Aux, of size $O(MN)/n_p$, an index array of integers, $Index$, of size n_p, and a flag double array of integers, $Flag$, of size $O(M)/n_p$.

Step 5: For $i = 0, ..., m-2$, compute the set $A_i(id)$ of points (k, k') in IP that are reachable via the vectors $e_0, ..., e_i$ and satisfying $(k + k') \equiv id \bmod n_p$:

5.1: For $h = 0, ..., n_p - 1$, set $Index[h] \leftarrow 0$.

5.2: Set $k \leftarrow id + 1$, and while $(k \leq n_i)$ do:

5.2.1: If $v_0 + ke_i = (k, k') \in IP$, set $id' \leftarrow (k + k') \bmod n_p$, and broadcast (k, k') to processor id' in the array Aux at location $Index[id']$.

5.2.2: Set $k \leftarrow k + n_p$ and $Index[id'] \leftarrow Index[id'] + 1$.

5.3: For each $u \in A_{i-1}(id)$ and $k = 0, ..., n_i$, if $u + ke_i = (k, k') \in IP$, set $id' \leftarrow (k + k') \bmod n_p$, broadcast (k, k') to processor id' in the array Aux at location $Index[id']$, and set $Index[id'] \leftarrow Index[id'] + 1$.

5.4: $bsp\_sync()$.

5.5: Initialise all flags in array $Flag$ to $PASS$. For each $u = (k, k') \in Aux$ do:

5.5.1: Find the smallest integer $j \geq y_{min}$ such that $k + j \equiv id \bmod n_p$.

5.5.2: Set $h \leftarrow (k' - j)/n_p$; if $Flag[k - x_{min}][h] \neq FAIL$, add u to $A_i(id)$ and set $Flag[k - x_{min}][h]$ to $FAIL$.

Step 6: Compute the last set $A_{m-1}(id)$:

6.1: For $h = 0, ..., n_p - 1$, set $Index[h] \leftarrow 0$.

6.2: For each $u \in A_{m-2}(id)$ and $k = 0, ..., n_{m-1} - 1$: if $u + ke_{m-1} = (k, k') \in IP$, set $id' \leftarrow (k + k') \bmod n_p$ and broadcast (k, k') to id' in the array Aux at location $Index[id']$, and set $Index[id'] \leftarrow Index[id'] + 1$.

Step 7: $bsp\_sync()$.

Step 8: Remove the redundancies in Aux as described in 5.5 above and store the unique points in $A_{m-1}(id)$.

Step 9: Let $v_0 = (k, k')$. If $id = (k + k') \bmod n_p$ and if $v_0 \in A_{m-1}(id)$, set $Result \leftarrow Decomposable$ and broadcast $Result$ to all processors.

Step 10: $bsp\_sync()$.

Step 11: Return $Result$.

Theorem 13 *The above algorithm works correctly as specified, has a BSP cost equal to*

$$\frac{O(t'mN)}{n_p} + \left(\frac{O(t'mN)}{n_p} + O(\sqrt{t'} + n_p)\right) g(n_p) + (m+2).\ell(n_p) \tag{12}$$

flops, and requires $\frac{O(t'N)}{n_p}$ bits of storage per processor, assuming that the maximum of absolute values of all coordinates of vertices of P fits in a machine word. Here, m denotes the number of edges in P, N denotes the maximum number of integral points along any edge in P, and t' denotes the total number of integral points in P.

Proof: The algorithm is executed by all processors which implement their own copy of the ensuing instructions. In the first step, all processors define a global array that will be used to store the x coordinates of points of intersection between horizontal lines $y = y_{min}, ..., y_{max}$ and P. We assume that there are always two points to be stored even if they were identical, in which case they designate a vertex. The amount of memory required to keep this global information about IP is $2(y_{max} - y_{min} + 1)$ bits. Step 2 is done in parallel where processors perform almost an equal number of intersections between horizontal lines and the polygon. Since each such intersection consists in at most $m(y_{min} - y_{max} + 1)$ floating point operations followed by a communication of two integers to n_p processors, the computation and communication costs for this step are

$$\lceil m(y_{min} - y_{max} + 1)/n_p \rceil + 2n_p \lceil (y_{min} - y_{max} + 1)/n_p \rceil .g(n_p) \tag{13}$$

flops. Step 3 is a synchronisation point needed for all processors to update the values of the above points of intersection and hence for each to have a complete description of IP. Note that $t' = O((x_{max} - x_{min})(y_{max} - y_{min}))$, so that $\sqrt{t'} = O(max((x_{max} - x_{min}), (y_{max} - y_{min})))$, and hence Steps 1-3 require

$$O(m\sqrt{t'})/n_p + O(\sqrt{t'})g(n_p) + \ell(n_p) \tag{14}$$

flops and $O(\sqrt{t'})$ bits of storage.

In the remaining steps, each processor id computes its own subset of A_i, for $i = 0, ..., m-1$, denoted by $A_i(id)$. According to our data distribution in Corollary 11, each processor will be in charge of $O(t')/n_p$ points, but makes use of a private copy of an auxiliary array, Aux, which temporarily stores the points as they build up in one iterative step, even when they occur redundantly up to N times. The inner-most computations in each iteration in Steps 5 and 6 require that each processor id performs a vector sum (finding a new point $v = u + ke_i$), two integer comparisons (checking for inclusion of v in IP), an integer division (determining the relevant processor id'), and a communication of two machine words representing the coordinates of the communicated point. Note that the communication is done only to the relevant processor id' in the auxiliary array Aux, and that processor id keeps an updated index on the address in $Aux(id')$ where it can communicate v to its relevant processor id'. By Lemma 8, all the computations within a single iteration of the loops over the edges of the polygon are independent, and hence, a synchronisation barrier is needed only at the end of the sequence of computations and communications detecting paths along $e_0, ..., e_i$ for $i = 0, ..., m-1$. This ensures that all processors update the values of the newly added points in the array Aux. When this is done, each processor can then remove the redundancies in its storage as follows. For $k = x_{min}, ..., x_{max}$, processor id stores all points (k, k') such that $k + k' \equiv id \bmod n_p$, and $y_{min} \leq k' \leq y_{max}$. In removing redundancies, each point (k, k') has to have a flag associated with it, which is initially set to $PASS$ but then permanently set to $FAIL$ signalling that it has been copied from Aux to $A_i(id)$ exactly once. We have seen in Corollary 11 that this distribution allocates $O(t')/n_p = O(M)/n_p$ integral points per processor, hence the size of the array $Flag$. The flag of (k, k') has to occupy an address in $Flag$ that is dependent solely on both the order of the entries and their values in the pair (k, k'). Flags also have to occupy entries in the double array successively starting from the location $(0, 0)$. This justifies the index $k - x_{min}$, specifying the row in which the flag of (k, k') will be found, where $k - x_{min} = 0, ..., x_{max} - x_{min} = O(\sqrt{t'})$. For a fixed such k, the choice of the integer j such that j is the first integer greater than or equal to y_{min} (and of course less than or equal to y_{max} since $n_p \leq y_{max} - y_{min} + 1$ and $y_{min} \leq j \leq y_{min} + n_p - 1$) and satisfying $k + j \equiv id \bmod n_p$ implies that any k' such that $(k, k') \in A_i(id)$ has to satisfy

$$k' = j + hn_p, \qquad \text{for some integer } h = 0, ..., \left\lfloor \frac{y_{max} - j}{n_p} \right\rfloor \tag{15}$$

(note that $j \leq k' \leq y_{max}$ for (k, k') to belong to IP). For $h = (k' - j)/n_p$, we have that h uniquely identifies k' such that $k + k' \equiv id \bmod n_p$ and that the flags of (k, k') for a fixed k and increasing values of k' occupy columns $h = 0, ..., \lfloor (y_{max} - j)/n_p \rfloor$ of array $Flag$. Since $j \geq y_{min}$, this ensures that the number of columns per row of the array $Flag$ is at most $O(\sqrt{t'})/n_p$, and hence the total size of the array does not exceed $O(\sqrt{t'}\sqrt{t'})/n_p = O(t')/n_p$.

The BSP cost of Steps 5-8 can then be computed as follows. The loop over all edges of the polygon iterates m times. In each iteration, every processor performs at most $O(t'N)/n_p$ vector operations, communicates a set of possibly redundant $O(t'N)/n_p$ points, each to one processor only, removes the redundancies in its auxiliary array using at most $O(t'N)/n_p$ integer operations, and synchronises with other processors once. The BSP cost of Steps 5-8 is then

$$\frac{O(t'mN)}{n_p} + \frac{O(t'mN)}{n_p} g(n_p) + m.\ell(n_p) \tag{16}$$

flops.

By the end of all computations, the processors assume the result to be "Indecomposable". The processor which is assigned the pivot v_0 checks whether v_0 is in its copy of $A_{m-1}(id)$. Only then does it broadcast to all other processors the result "Decomposable" (say in the form of a bit word $FAIL = 0$ or $PASS = 1$). A synchronisation barrier is finally met to update the ultimate result. The total cost of this superstep is $n_p.g(n_p) + \ell(n_p)$. Summing up the above sub-costs, the final estimate in the theorem is established. Since

$$n_p \leq \min((x_{max} - x_{min} + 1), (y_{max} - y_{min} + 1)), \tag{17}$$

the spatial complexity is dominated by the space required by the array Aux, which is $O(t'N)/n_p$ bits, assuming that the maximum over absolute values of all coordinates of vertices of P fits in the machine word.

Corollary 14 *Given a bivariate polynomial f over $\mathbb{F}$ of degree q with no non-constant monomial factors and with c non-zero terms, absolute irreducibility testing can be performed in parallel using*

$$\frac{O(cq^3)}{n_p} + \left(\frac{O(cq^3)}{n_p} + O(\sqrt{c} + n_p)\right) g(n_p) + O(c).\ell(n_p) \tag{18}$$

flops and $\frac{O(q^2)}{n_p}$ bits of storage, assuming q fits in a machine word, and $1 \leq n_p = O(q)$.

Proof: Let y_{min}, y_{max}, x_{min} and x_{max} be as defined above. Since

$$y_{max} - y_{min} = O(q), \quad \text{and } x_{max} - x_{min} = O(q), \tag{19}$$

Newt(f) can be embedded in a $q \times q$ square whose leftmost and lowest vertex is the origin of coordinates. Let u denote some unit of length on the horizontal or vertical axes. Then, the longest edge of Newt(f) will have length that is bounded by $\sqrt{2}qu$ representing the length of the diagonal of the square, and hence will have $O(q)$ integral points. We also have that $\#Newt(f) = O(q^2)$, and the number of edges of Newt(f) bounded by the number of terms c of f. The input to Algorithm 12 will then satisfy $N = O(q)$, $t' = O(q^2)$ and $m = O(c)$, and by Theorem 13 above, the result follows immediately.

Let T_s denote the sequential run time in flops of Algorithm 3 and T_{n_p} denote the parallel run time in flops using n_p processors of the parallel version 12. Let E_{n_p} denote the absolute efficiency (see [15]) defined by $E_{n_p} = T_s/(n_p T_{n_p})$, measuring the scalability of the above parallel algorithm. The following result establishes the conditions under which our BSP algorithm can achieve linear speed-up, where efficiency approaches 1.

Corollary 15 *Algorithm 12 for testing indecomposability of a convex polygon P in parallel achieves efficiency $E_p \geq 1/2$ under the conditions*

1. $g(n_p) = O(1)$,
2. $n_p < \sqrt{t'}mN$,
3. $n_p < \sqrt{t'}\left(\frac{\sqrt{t'}mN}{n_p} - 1\right)$,

4. $n_p \ell(n_p) = O(t'mN)$.

Proof: Recall from Theorem 13 that

$$T_{n_p} = T_s/n_p + \left(O(t'mN)/n_p + O(\sqrt{t'} + n_p)\right) g(n_p) + (m+2)\ell(n_p). \tag{20}$$

For $g(n_p) = O(1)$ we have

$$T_{n_p} = T_s/n_p + O(\sqrt{t'} + n_p) + (m+2)\ell(n_p). \tag{21}$$

If $n_p < \sqrt{t'}mN$, $\frac{\sqrt{t'}mN}{n_p} - 1 > 0$ and hence we can require $n_p < \sqrt{t'}\left(\frac{\sqrt{t'}mN}{n_p} - 1\right)$ or $\sqrt{t'} + n_p < t'mN/n_p$, so that

$$T_{n_p} = T_s/n_p + (m+2)\ell(n_p). \tag{22}$$

If we also have $n_p \ell(n_p) = O(t'mN)$, then $(m+2)\ell(n_p) = O(t'mN)/n_p = T_s/n_p$ and hence

$$T_{n_p} = \frac{T_s}{n_p}(1 + O(1)) \tag{23}$$

from which one deduces that

$$E_p = \frac{T_s}{n_p T_{n_p}} \geq 1/2. \tag{24}$$

4 Parallel multivariate absolute irreducibility testing

As discussed earlier, multivariate polynomial absolute irreducibility testing through polytopes can be performed only heuristically with varying rates of success. For a multivariate polynomial of n variables and degree bound equal to d on each variable, the empirical results in [9] reflect a very high probability of success for polynomials whose number of terms is bounded by $O(nd)$. For polynomials whose number of terms exceeds this bound, the probability of success can be increased by loosening the bound on the absolute values of the randomly chosen matrices, or the bound on the number of projections that one can try before the algorithm outputs success. The first strategy obviously comes at the expense of a larger running time, since it implies increasing the sizes of the shadow polygons and hence the run time of the polygon indecomposability test. The second option also increases the run time of the algorithm, simply because it involves many more shadow polygons (of roughly the same size) to be tested for indecomposability. Unlike the case for bivariate absolute irreducibility testing, improving the sequential algorithm not only involves extending the range of success for higher degree polynomials, but also investigating how parallel techniques can improve upon the performance when any of the two above strategies is invoked, with the aim of increasing the chances of success even for polynomials that are denser than those for which the randomised algorithm is generally successful. A parallel approach to multivariate absolute irreducibility testing will naturally depend on the parallel bivariate case as a sub-problem; however, we shall emphasise the role that a parallel environment can have in the interplay between the number of projections and their "size", as one manipulates the two parameters to improve the success rate.

4.1 A BSP algorithm for testing polytope indecomposability

In the following, let n_p denote the maximum number of processors that can be made available, and let p denote the number of processors that are actually invoked. We then have $p \leq n_p$. As sizes of shadow polygons may change (according to the matrix bound one chooses in Step 2 of Algorithm 6), so can their numbers, and accordingly we identify the two parameters governing the behaviour of our parallel algorithm: First, the number p_r of processors that we assign for testing indecomposability of one common shadow polygon using the parallel bivariate version, and the total number j of shadow polygons to which one applies Algorithm 12 simultaneously in parallel. Since $p_r \leq p$, one will have $j \geq 1$ blocks of p_r processors, each block performing a parallel polygon indecomposability test, resulting in a "doubly" parallel scheme aimed at improving the sequential performance at the two levels of size and number of projections. Before presenting our algorithm, we shall need a few more notations. Let p_{min} denote the minimum number of processors required to test indecomposability of one shadow polygon using the parallel algorithm in 12. Note that, given a polynomial f with total degree at most nd, the projections of the support vectors of f into the plane using a matrix whose entries are bounded by b have coordinates that are equal to at least $-2nbd$ and at most $2nbd$. Consequently, the shadow polygons are embedded in a square of dimension $O(nbd \times nbd)$ and hence have at most $O((nbd)^2)$ integral points. The number p_{min} thus corresponds to the minimum number of processors required to store all these integral points using the data distribution in Corollary 11.

Let n_p, p and p_r be as defined above, and assume in the following that more than p_{min} processors can be allowed to test one shadow polygon. In particular, let $k \in \{0, ..., n_p - p_{min}\}$ be such that $p_r = p_{min} + k$, and let $j \in \{1, ..., \left\lfloor \frac{n_p}{p_{min}+k} \right\rfloor\}$ be such that $jp_r = p$. With this structure we can have blocks of j shadow polygons, each of which is tested for indecomposability by Algorithm 12 using $j(p_{min} + k) = jp_r = p \leq n_p$ processors. Taking the maximum over all shadow polygons, let N denote the maximum number of integral points along any edge, E denote the maximum number of edges, and I denote that maximum number of integral points belonging to any shadow polygon.

Algorithm 16 (Parallel Polytope Indecomposability Test)
Input: Let $f \in \mathbb{F}[X_1, ..., X_n]$, with $n > 2$, be a polynomial with c terms and no non-constant monomial factors, and let S_f denote the set of exponent vectors of non-zero

terms of f.
Output: Absolutely irreducible or Failure, where the latter case means that indecomposability of $conv(S_f)$ (and hence absolute irreducibility of f) is not decided.

Step 1: Re-arrange the points in S_f as an $n \times c$ matrix S. Choose positive integers b and e. Let $M(b)$ denote the set of all $2 \times n$ matrices with integer coefficients bounded in absolute value by b.

Step 2: Determine the minimum number of processors p_{min} necessary to test indecomposability of a shadow polygon of size $O((nbd)^2)$.

Step 3: Choose a parameter $k \in \{0, ..., n_p - p_{min}\}$ such that any shadow polygon is tackled by $p_{min} + k$ processors. Set $p_r = p_{min} + k$.

Step 4: Choose a parameter $j \in \{1, ..., \left\lfloor \frac{n_p}{p_r} \right\rfloor\}$ such that $jp_r \leq n_p$.

Step 5: Invoke $p = jp_r$ processors to operate in parallel. If $e \bmod j \neq 0$, set $e \leftarrow e + j - (e \bmod j)$.

Step 6: Define an auxiliary array of integer vectors, Aux, of size $O(IN)/p_r$, an index array of integers, $Index$, of size p_r, and a flag double array of integers, $Flag$, of size $O(I)/p_r$.

Step 7: Processors are divided into j blocks according to indices as follows:

$$\begin{aligned} block_0 &= \{id | id = 0, ..., p_r - 1\}, \\ block_1 &= \{id | id = p_r, ..., 2p_r - 1\}, \\ &.... \\ block_{j-1} &= \{id | id = (j-1)p_r + jp_r - 1\}. \end{aligned} \tag{25}$$

All processors in block w, for some $w = 0, ..., j-1$, perform Steps 8-18 repeatedly up to e/j times:

Step 8: Select a common matrix Mat_w uniformly at random from $M(b)$ and compute the set of points in $\mathbb{R}^2$ defined by $Mat_w(S) := \{Mat_w.s | s \in S\}$.

Step 9: Compute the convex hull and the edge sequence $\{n_i e_i\}_{0 \leq i \leq m-1}$ of $Mat_w(S)$. Check that each vertex of $conv(Mat_w(S))$ has only one pre-image in S under the projection Mat_w. If this condition is not met, all processors in block w return to Step 8.

Step 10: $bsp\_sync()$.

Step 11: Compute a description of the set IP of all the integral points in $Mat_w(S)$ as described in Steps 1 and 2 of Algorithm 12. Let $A_i^{(w)}(id)$ denote the set of points in $conv(Mat_w(S))$ that are reachable via $e_0, ..., e_i$ and that are assigned to processor id in block w. Set $A_i^{(w)} = \emptyset$, for $i = -1, ..., m-1$, and $Result^{(w)} \leftarrow Indecomposable$. Let id denote the processor's identification number.

Step 12: For $i = 0, ..., E-2$ do:

12.1: If $i \leq m-2$, compute the set $A_i^{(w)}(id)$ of points (k, k') in IP that are reachable via the vectors $e_0, ..., e_i$ and satisfying $(k+k') \equiv id \bmod p_r$:

12.1.1: For $h = w.p_r, ..., (w+1).p_r - 1$, set $Index[h] \leftarrow 0$.

12.1.2: Set $k \leftarrow id - (w.p_r) + 1$, and while $(k \leq n_i)$ do:

. If $v_0 + ke_i = (k, k') \in IP$, set $id' \leftarrow [(k+k') \bmod p_r] + w.p_r$ and broadcast (k, k') to processor id' in the array Aux at location $Index[id']$.

. Set $k \leftarrow k + p_r$ and $Index[id'] \leftarrow Index[id'] + 1$.

12.1.3: For each $u \in A_{i-1}^{(w)}(id)$ and $k = 0, ..., n_i$, if $u + ke_i = (k, k') \in IP$, set $id' \leftarrow [(k+k') \bmod p_r] + w.p_r$, broadcast (k, k') to processor id' in the array Aux at location $Index[id']$ and set $Index[id'] \leftarrow Index[id'] + 1$.

12.2: $bsp\_sync()$.

12.3: Initialise all flags in array $Flag$ to $PASS$. For each $u = (k, k')$ $\in Aux$ do:

12.3.1: Find the smallest integer $j \geq y_{min}$ such that $k + j \equiv id \bmod p_r$.
12.3.2: Set $h \leftarrow (k' - j)/p_r$; if $Flag[k - x_{min}][h] \neq FAIL$, add u to $A_i^{(w)}(id)$ and set $Flag[k - x_{min}][h]$ to $FAIL$.

Step 13: Compute the last set $A_{m-1}^{(w)}(id)$:

13.1: For $h = wp_r, ..., (w+1)p_r - 1$, set $Index[h] \leftarrow 0$.
13.2: For each $u \in A_{m-2}^{(w)}(id)$ and $k = 0, ..., n_{m-1} - 1$: if $u + ke_{m-1}$ $= (k, k') \in IP$, set $id' \leftarrow [(k+k') \bmod p_r] + w.p_r$ broadcast (k, k') to id' in the array Aux at location $Index[id']$, and set $Index[id'] \leftarrow Index[id'] + 1$.

Step 14: $bsp\_sync()$.

Step 15: Remove the redundancies in Aux as described in 12.3 above and store the unique points in $A_{m-1}^{(w)}(id)$.

Step 16: Let $v_0 = (k, k')$. If $id = [(k+k') \bmod p_r] + w.p_r$ and if $v_0 \in A_{m-1}^{(w)}(id)$, set $Result \leftarrow Decomposable$ and broadcast $Result$ to all processors.

Step 17: $bsp\_sync()$.

Step 18: If this polygon is integrally indecomposable, output "Absolutely irreducible" and halt. Else, all processors return to Step 8.

Step 19: All processors output "Failure".

Theorem 17 *Let $f \in \mathbb{F}[X_1, ..., X_n]$, with $n > 2$, be a polynomial with c terms and no non-constant monomial factors. Let d denote the upper bound on the degree in each variable of f, and let b denote the upper bound on the absolute values of integer coefficients of $2 \times n$ matrices representing random projections. Let p_{min} denote the minimum number* *of processors needed to store $O((nbd)^2)$ integral points using the distribution in Corollary 11. Let $k \in \{0, ..., n_p - p_{min}\}$ and $j \in \{1, ..., \left\lfloor \frac{n_p}{p_{min}+k} \right\rfloor\}$ such that $p = j(p_{min} + k)$ processors are operating in parallel and $p \leq n_p$, $n_p = O(nbd)$. Then Algorithm 16 can decide absolute irreducibility of f correctly or else produce "failure" using at most*

$$\frac{e}{j}\left(O(c^2 + cn) + \frac{O(IEN)}{p_r}\right) + \frac{e}{j}\left(\frac{O(IEN)}{p_r} + O(p + \sqrt{I})\right) g(p) + \frac{e}{j} O(E)\ell(p) \tag{26}$$

flops, and requires $\frac{O(IN)}{p_r}$ bits of storage per processor, assuming that nbd fits in machine word. Here, E denotes the maximum over all shadow polygons of total number of edges, N denotes the maximum over all shadow polygons of number of integral points along any edge, and I denotes the maximum over all shadow polygons of total number of interior integral points.

Proof: The first four steps of the algorithm are performed sequentially, whereby a matrix bound b is chosen, which determines p_{min} according to the data distribution in Corollary 11. The two parameters k and j are also chosen such that $p_{min} + k = p_r \leq n_p$ processors can be assigned for any shadow polygon, and j shadow polygons can be tested simultaneously. The algorithm then invokes $p = jp_r \leq n_p$ processors to operate in parallel. The minor modification to the value of the number of projections e ensures that j divides e, so that all processors are made to enter the loop starting at Step 8. The reason we enforce such full participation of processors is that synchronisation barriers will be met throughout that particular loop, which causes a run time error if any of these supersteps is not met by all processors. In Step 6, all arrays that will be used for removing redundancies in the upcoming computations are declared. Note that the amount of memory per array is dependent on the number of processors p_r operating in one block rather than the total number of processors. This is because only p_r processors will be allowed to share the work in the parallel polygon indecomposability testing. In Step 7, all p processors re-cluster into j blocks as determined by their identification number. This re-grouping ensures processors within one block $w = 0, ..., j - 1$ compute the same random projection $Mat_w(S)$, and perform the rest of the Steps 8-18 using this common input.

The loop starting at Step 8 iterates at most e/j times. We analyse each step in the iteration as follows. Step 8 is only a computation whereby all processors perform c matrix vector multiplications using their assigned matrix Mat_w. Each such multiplication requires $2n$ multiplications and $2(n - 1)$ additions of integers bound in absolute value by nbd, and so Step 8 has a BSP cost of $O(cn)$ flops. In Step 9, all processors in block w compute the edge sequence of the shadow polygon $conv(Mat_w(S))$. This is only a computation step with BSP cost $O(c^2)$ flops. A synchronisation barrier is met at Step 10 to ensure that processors in a particular block w which have found a successful projection wait for others in different blocks still searching for a good candidate projection. Without this barrier, one risks having some but not all processors entering the loop of the parallel polygon indecomposability testing phase (which in turn contains a synchronisation barrier that should be met by all p processors). In Steps 11-17, each processor joins the others in its block to test indecomposability of their common shadow polygon. In Step 12, we enforce an upper bound of $E-2$ rather than $m-2$, since E is a global maximum of the number of edges belonging to any shadow polygon, whereas m is a private copy representing the number of edges of $conv(Mat_w(S))$. This is again to ensure that all processors enter the loop within which a synchronisation barrier is to be met in Step 12.2. However, the relevant computations and communications are performed only when the processors in block w can do so (as indicated by the condition $i \leq m-2$ in Step 12.1). The vector computations and the redundancy checkings in Steps 12.1, 12.3 and 13 are similar to those in the parallel polygon indecomposability testing algorithm. However, we note the following essential differences. Note that any processor in the above scheme has two labels attached to it, one describing its identification number $id = 0, ..., p-1$, and another describing its index ind within its block, for $ind = 0, ..., p_r - 1$. Moreover, a processor id operates within block $w = \lfloor id/p_r \rfloor$, and has index $ind \equiv id \bmod p_r$ in that block. Conversely, a processor with index ind in block w has id equal to $w.p_r + ind$. The data allocation in the present algorithm should assign arbitrary points (k, k') of the polygon $conv(Mat_w(S))$ only to p_r processors. Thus, one checks for the value of $(k+k')$ modulo p_r rather than p, the total number of processors in action. But this gives the index of the processor to which (k, k') should be allocated. The actual id can then be retrieved as $[(k+k') \bmod p_r] + w.p_r$. As seen previously, the BSP cost required by Steps 11-15 is at most

$$\frac{O(IEN)}{p_r}) + \left(\frac{O(IEN)}{p_r} + O(\sqrt{I})\right) g(p) + (E+2)\ell(p) \tag{27}$$

flops and $O(I)/p_r$ bits of storage. Note that we make explicit the dependence of g and ℓ on p, since their cost depends on the total number of processors invoked despite the fact that the computations and communications are shared between blocks of p_r processors only.

In Step 16, all p processors resume contact to be able to know which of the w shadow polygons have been shown indecomposable. The processor in block w which is in charge of the pivot v_0 of $Conv(Mat_w(S))$ decides whether the polygon is indecomposable, and if so, signals to all processors in its block and other blocks to halt the algorithm. Else, all processors repeat Steps 8-18 choosing a different projection Mat_w. This involves p communications of a boolean representing "Indecomposable", and brings the total cost of the entire algorithm to

$$\begin{aligned}\frac{e}{j}\left(O(c^2+cn) + \frac{O(IEN)}{p_r}\right) + \frac{e}{j}\left(\frac{O(IEN)}{p_r} + O(p+\sqrt{I})\right) g(p) \\ + \frac{e}{j}O(E)\ell(p)\end{aligned} \tag{28}$$

flops. The memory requirement is dominated by $O(IN)/p_r$ bits needed per processor to store its subset of integral points belonging to any shadow polygon.

Corollary 18 *Algorithm 16 for absolute irreducibility testing of multivariate polynomials achieves efficiency*

$$E_p \geq \frac{1}{p_r} \tag{29}$$

under the conditions

1. $g(p) = O(1)$,
2. $p < c^2 + cn + (IEN/p_r) - \sqrt{I}$,
3. $\ell(p) < \frac{c^2+cn}{E} + \frac{IN}{p_r}$.

Proof: Recall that the sequential time for testing absolute irreducibility of a random multivariate polynomial is given by

$$T_s = eO(c^2 + cn + IEN) \tag{30}$$

bit operations, assuming *nbd* fits in a machine word. Rewrite this as

$$T_s = T_1 + T_2 \tag{31}$$

where $T_1 = eO(c^2 + cn)$ denotes the cost of that part of the algorithm in which the projected points and their convex hull are computed, and $T_2 = eO(IEN)$ denotes the cost of testing integral indecomposability of the shadow polygons. We also have the parallel run time T_p given by

$$\begin{aligned} T_p = \frac{e}{j}\left(O(c^2 + cn) + \frac{O(IEN)}{p_r}\right) + \frac{e}{j}\left(\frac{O(IEN)}{p_r} + O(p + \sqrt{I})\right) g(p) \\ + \frac{e}{j} O(E)\ell(p) \end{aligned} \tag{32}$$

flops. For $g(p) = O(1)$,

$$T_p = \frac{e}{j}\left(O(c^2 + cn) + \frac{O(IEN)}{p_r}\right) + \frac{e}{j}O(p + \sqrt{I}) + \frac{e}{j}O(E)\ell(p). \tag{33}$$

By the condition $p < c^2 + cn + (IEN/p_r) - \sqrt{I}$, $p + \sqrt{I} = O(c^2 + cn) + O(IEN)/p_r$ and so

$$T_p = \frac{e}{j}\left(O(c^2 + cn) + \frac{O(IEN)}{p_r}\right) + \frac{e}{j}O(E)\ell(p), \tag{34}$$

and if $\ell(p) < \frac{c^2+cn}{E} + \frac{IN}{p_r}$,

$$T_p = \frac{e}{j}\left(O(c^2 + cn) + \frac{O(IEN)}{p_r}\right) \tag{35}$$

or

$$T_p = \frac{1}{j}\left(T_1 + \frac{T_2}{p_r}\right) \tag{36}$$

for T_1 and T_2 as defined above. We then have

$$\begin{aligned} pT_p &= \tfrac{p}{j}\left(T_1 + \tfrac{T_2}{p_r}\right) \\ &= p_r T_1 + T_2 \\ &\leq p_r(T_1 + T_2) \\ &= p_r T_s \end{aligned} \tag{37}$$

from which one concludes that

$$\frac{T_s}{pT_p} \geq \frac{1}{p_r}. \tag{38}$$

The above discussion investigates the parallel efficiency for an increasing number of processors. However, the implications play an important role in the choice we have to make of the parameters k and j. In particular, the lower bound on E_p can be improved for decreasing values of p_r, which indicates that the best realistic performance is achieved by choosing $k = 0$, so that $p_r = p_{min}$. Since one cannot invoke Algorithm 16 without less than p_{min} processors per shadow polygon, we expect this to be the best case scenario describing the parallel scalability of the algorithm.

5 Implementation and Run Times

All programs were written in C and extended using the standard BSP library [19, 20]. The work was carried out at the Oxford University Supercomputing Centre (OSC) using the Oswell machine. Oswell is a Sun cluster of 84 processors and a shared memory system where the processors are arranged in three groups of 24 processors (having 48 GBytes of memory each) and a group of 12 processors (having 24 GBytes of memory each). Each of the UltraSPARC III processors is capable of performing an average of 122.2 Mflop/sec. With this scheme, any work submitted to the machine is queued to one of those four boxes, and the number of processors available for use by any one job is at most 24. Table 1 contains the machine values of the BSP parameters †, which are in flops for ℓ and in flops/word for g.

In the following, n denotes the number of variables in the input polynomial f, D denotes its total degree, d denotes the upper bound on the degrees in each of its variables, and c denotes the number of its terms. Also, E and N denote the number of edges and the maximum number of integral points along any edge of Newt(f) if f is bivariate. If f is multivariate, E and N denote the maximum over the number of edges and the

†See http://www.bsp-worldwide.org/implmnts/oxtool/params.html for the benchmark tests.

maximum number of integral points along any edge over all shadow polygons of Newt(f). S denotes the number of cases (out of 100) in which Newt(f) is integrally indecomposable. In the case that $n > 2$, MB denotes the matrix upper bound on absolute values of random coefficients of the projections, PB denotes the upper bound on the number of projections per polytope, AP denotes the average number of projections required to show that the input polynomial is absolutely irreducible, and p_r denotes the number of processors allocated per shadow polygon in the parallel multivariate algorithm. T_1 denotes the sequential running time in seconds, and T_p, for $p > 1$, denotes the parallel running time in seconds using p processors, to show absolute irreducibility successfully for one case that uses about the average number of projections. An empty column location appearing before the first reported running time T_p indicates that there is not enough memory using less than p processors to tackle the input polygon or shadow polygons of the input polytopes. An empty column location appearing after the last reported running time T_p indicates that there are no more processors available in the system for our use. The absolute efficiencies E_p are shown in parentheses below their corresponding parallel times. Note that when the algorithm cannot be run using one processor for memory constraints, we are contended with calculating absolute efficiency using p processors as $p'T_{p'}/pT_p$, where $T_{p'}$ is the first reported parallel running time.

The input to the two parallel algorithms was generated as follows. A choice is first made on the parameters n, D or d, c, E and N. A hundred random polynomials satisfying the above parameters is then chosen: In the bivariate case, those polynomials should also satisfy the conditions governing their Newton polytopes (in terms of the number of edges E and the maximum number N of integral points appearing along any of their edges). In the multivariate case, the projections chosen for these 100 polynomials should produce shadow polygons satisfying the parameters E and N.

In Table 2 we examine relatively small degree bivariate polynomials with 300 terms and whose Newton polytopes have no more than 6 edges, and we study the effect of variations in the average number N of integral points along any edge of the polygon. The performance generally has good efficiency up to 4 processors, but it can be improved for more processors by increasing the degree (and hence the total number I of integral points in the polygon) or increasing values of N for a fixed degree. Specifically, and according to Corollary 15 and the values in Table 1, we see that the first three conditions are easily satisfied. For small degree polynomials whose Newton polytopes have a small value for the parameter N, the fourth condition might not hold as we increase the number of processors. In this case efficiency can be improved by increasing the maximum number of integral points along any edge of Newt(f). The rate of success is generally high, since as indicated previously in [9], one expects the algorithm to perform well for sparse polynomials whose number of terms is bounded by $O(nd)$. However, we note that the rate of success decreases with increasing values of N.

In Table 3 we examine larger degree bivariate polynomials with varying numbers of terms. Mostly, efficiency improves significantly for larger degree polynomials. As noted above, and for a fixed degree polynomial whose Newton polytope has a fixed number of edges, the run time increases and efficiency improves when N increases for a fixed D or when D increases. Unlike the examples in Table 2 though, varying the number of terms

c indicates that the rate of success decreases for increasing ratios N/c rather than simply for increasing values of N. Fixing the degree, the number of terms, and the parameter N, we also note an increase in the run time and improvement in efficiency when the number of edges increases, which is to be expected from Corollary 15.

In Table 4 we study the performance of the parallel algorithm in the multivariate case and for small degree polynomials. The number of edges belonging to the shadow polygons is fixed. For all unstarred rows, the number of processors in each block is set to be the minimum required. Efficiency improves for a fixed degree as the number of terms increases. This relates to conditions 2 and 3 of Corollary 18. For a fixed degree and a fixed number of terms, efficiency improves when the matrix bound increases, since this implies larger shadow polygons and hence larger values for I and N in conditions 2 and 3 of Corollary 18. In the starred rows we compare the performance of the algorithm when only p_r is increased beyond the minimum required, but all other parameters remain fixed. As predicted at the end of Section 4, and for a fixed total number of processors p, efficiency is better maintained when $p_r = p_{min}$. As earlier noted in [9], the success rate decreases with increasing numbers of terms, and can be improved by either increasing the projection bound or the matrix bound.

In Table 5 we examine the performance for large degree trivariate polynomials. Here, the number of terms is fixed, and is significantly less than the total degree of the input. Also, E and N are fixed for all shadow polygons, and so are the matrix bound and projection bound. We note a much better parallel performance than in the case of small degree trivariate polynomials in Table 4. This also improves upon increasing degrees (as indicated by conditions 2 and 3 of Corollary 18). The algorithm has never failed for examples using only one projection per case and a very small matrix bound. This emphasises the expected high success rate of the algorithm for sparse polynomials.

Finally, in Table 6, we examine the performance for multivariate polynomials all with the same small bound on the degree in each of their variables. Also fixed are the number of terms, E and N of the shadow polygons, and the matrix bound and projection bound. Here, p_r is set to be the minimum number of processors required to tackle a shadow polygon. We increase the number of blocks to be tested in parallel by increasing the number of processors available. Efficiency is very good in all cases, even for polynomials whose total degree is less than others. This is to be expected since the sizes of shadow polygons are relatively large (conditions 2 and 3 of Corollary 18). The success rate is also high for these sparse polynomials, and so are the number of projections needed to produce a successful experiment.

Table 1: BSP parameters.

p	g	ℓ
1	0.34	55
2	1.64	1496
4	2.48	1683
8	2.34	2562
16	4.83	3431

Table 2: $n = 2$, $c = 300$, $E = 6$

D	N	S	T_1	T_2	T_4	T_8	T_{16}
1500	10	100	11	7 (0.7)	9 (0.3)	14 (0.1)	15 (0.1)
1500	50	100	68	42 (0.8)	24 (0.7)	16 (0.5)	10 (0.4)
1500	100	97	83	46 (0.9)	26 (0.8)	17 (0.6)	10 (0.5)
1500	900	89	828	427 (1)	216 (1)	115 (0.9)	52 (1)
2500	10	100	35	20 (0.9)	12 (0.7)	7 (0.6)	4 (0.5)
2500	50	100	185	94 (1)	58 (0.8)	37 (0.8)	19 (0.6)
2500	100	100	240	122 (0.9)	67 (0.9)	37 (0.8)	25 (0.6)

6 Conclusion

In this paper we have revisited a fast irreducibility testing algorithm for multivariate polynomials over arbitrary fields. The algorithm works deterministically in the bivariate case but heuristically and randomly for polynomials with more than two variables. Although finding a polynomial time algorithm for multivariate polynomial irreducibility testing remains an open problem, the work in [8] gives a pseudo-polynomial time algorithm, which can be applied as a fast pre-test before any of the rigorous yet slower algorithms are invoked. Motivated by the original empirical findings in [9] which provide various ranges of applicability of the heuristic algorithm, we investigated potential parallelism with the aim of extending these ranges, both for large degree bivariate polynomials and for multivariate polynomials of all degrees. A crucial aspect of our work exploited the fact that absolute irreducibility testing can be reduced to polytope indecomposability testing in $\mathbb{R}^n$. For $n = 2$, we addressed the two important issues of load balancing and data distribution. Having set the parallel framework we provided an algorithm whose communication cost can be easily bounded by the computation, and whose synchronisation cost has a factor that is a constant multiple of the number of edges in the polygon. This immediately implies highly efficient parallel performance for sparse bivariate polynomials whose Newton polytopes have few edges. Empirical results in this case achieve overall efficiency under

Table 3: $n = 2$

D	c	E	N	S	T_1	T_2	T_4	T_8	T_{12}	T_{16}
3000	1000	7	2000	72	108	56 (1)	30 (0.9)	17 (0.8)	10 (0.9)	7 (1)
3000	1000	7	3000	6	129	65 (1)	36 (0.9)	18 (0.9)	12 (0.9)	9 (0.9)
5000	500	6	10	100	72	37 (1)	20 (0.9)	13 (0.7)	9 (0.7)	8 (0.6)
5000	500	10	10	98	168	84 (1)	47 (0.9)	24 (0.9)	18 (0.8)	15 (0.7)
5000	2000	10	4000	81	...	115	60 (1)	37 (0.8)	27 (0.7)	25 (0.6)
5000	2000	8	5000	5	...	230	117 (1)	64 (0.9)	48 (0.8)	37 (0.7)
10000	500	6	100	100	...	...	60	32 (0.9)	22 (0.9)	18 (0.8)
10000	500	10	100	100	...	...	190	95 (0.8)	79 (0.8)	50 (1)
10000	1000	8	3000	96	...	...	133	73 (0.9)	55 (0.8)	40 (0.8)
10000	3000	8	5000	86	...	...	371	206 (0.9)	155 (0.8)	116 (0.8)
20000	2000	8	1000	100	...	...	...	...	131	112 (0.9)
20000	5000	8	7000	90	...	...	...	...	177	142 (0.9)
30000	3000	8	3000	100	...	...	...	...	329	235 (1)

Table 4: $E = 8$, $N = 50$

n	D	c	MB	PB	S	AP	p_r	T_1	T_2	T_4	T_8	T_{16}
3	75	100	2	100	61	16	1	7	6 (0.6)	4 (0.4)	5 (0.2)	3 (0.1)
3	75	100	2	200	82	80	1	49	33 (0.7)	20 (0.6)	13 (0.5)	31 (0.1)
3	75	100	6	100	80	14	1	127	74 (0.9)	33 (1)	24 (0.7)	17 (0.5)
3^*	75	100	6	100	80	14	2	...	74 (0.9)	66 (0.5)	29 (0.5)	26 (0.3)
3	75	200	2	100	30	18	1	7	5 (0.7)	5 (0.4)	6 (0.1)	4 (0.1)
3	75	200	2	200	45	127	1	232	129 (0.9)	72 (0.8)	41 (0.7)	24 (0.6)
3	75	200	6	100	51	8	1	51	25 (1)	18 (0.7)	12 (0.5)	7 (0.5)
3^*	75	200	6	100	51	8	2	...	26 (1)	19 (0.7)	22 (0.3)	15 (0.2)

reasonable parametric conditions that are implied by our theoretical analysis of the algorithm, and by significantly higher degree absolute irreducibility testing up to degree 30000.

Table 5: $n = 3$, $c = 500$, $E = 8$, $N = 100$, $MB = 1$, $PB = 100$

D	S	AP	T_1	T_2	T_4	T_8	T_{16}
1500	100	1	19	10 (1)	6 (0.8)	3 (0.8)	2 (0.6)
3000	100	1	...	65	28 (1)	16 (1)	8 (1)
15000	100	1	...	...	138	74 (0.9)	39 (0.9)
21000	100	1	...	...	150	83 (0.9)	47 (0.8)
30000	100	1	...	...	...	237	118 (1)

Table 6: $d = 10$, $c = 500$, $E = 8$, $N = 50$, $MB = 1$, $PB = 100$

n	S	AP	p_r	T_1	$T_{1.p_r}$	$T_{2.p_r}$	$T_{3.p_r}$	$T_{4.p_r}$	$T_{5.p_r}$	$T_{6.p_r}$	$T_{8.p_r}$
500	100	20	2	...	2460	1250 (1)	828 (1)	617 (1)	517 (1)	410 (1)	313 (1)
1000	100	24	3	...	3004	1501 (1)	885 1.1	800 (1)	680 (1)	...	...
2000	100	24	5	...	854	427 (1)	280 (1)	...	...	...	...
3000	100	26	6	...	3224	1543 (1)	...	...	...	...	...

We incorporated the above for the parallel multivariate case into a doubly parallel scheme where several shadow polygons are tested in parallel by blocks of processors. This was done by identifying two parameters reflecting the size as well as the number of the shadow polygons. Conditions under which this algorithm achieves good efficiency were also studied, and reflected in the empirical results for the multivariate case. Those exhibited a good efficiency for both small degree polynomials and for high degree polynomials, where parallelism could be exploited not only for speeding up computations but also for increasing the rate of success of the algorithm. The algorithm was used to test absolute irreducibility of trivariate polynomials with degree up to 30000 and of low degree multivariate polynomials with up to 3000 variables.

The algorithm we provided not only serves as a fast pre-test before invoking other well known algorithms, but also achieves absolute irreducibility testing for degrees which could not have been tackled otherwise. Still, it would be of great interest to determine whether a theoretical justification can be found at all, explaining why the probability of success of the absolute irreducibility testing via polytopes is at best when the number of nonzero terms of a multivariate polynomial is bounded by some constant multiple of its total degree.

Acknowledgments

The author is grateful to the Oxford Supercomputing Centre for allowing the use of its facilities to perform the reported experiments.

References

[1] F. Abu Salem, S. Gao, and A.G.B. Lauder. Factoring polynomials via polytopes. To appear in *International Symposium on Symbolic and Algebraic Computation*, ACM Press, 2004.

[2] F. Abu Salem. An efficient sparse adaptation of the polytope method over $\mathbb{F}_p$ and a record-high binary bivariate factorisation. Report PRG-RR-04-13, Oxford University Computing Laboratory, 2004, submitted to *Journal of Symbolic Computation*.

[3] L. M. Adleman. The function field sieve. In L. M. Adleman and M. -D. Huang, eds., *Algorithmic Number Theory*, Lecture Notes in Computer Science, Vol. 877, pp. 108–121, Springer-Verlag, Berlin, 1994.

[4] R. H. Bisseling, *Parallel Scientific Computation: A structured Approach using BSP and MPI*, Oxford University Press, 2004.

[5] D. Duval. Absolute factorization of polynomials: a geometric approach. *SIAM Journal on Computing*, 20: 1–21, 1991.

[6] S. Gao. Absolute irreducibility of polynomials via Newton polytopes. *Journal of Algebra*, 237: 501–520, 2001.

[7] S. Gao. Factoring multivariate polynomials via partial differential equations. *Mathematics of Computation*, 72: 801–822, 2003.

[8] S. Gao and A.G.B. Lauder. Decomposition of polytopes and polynomials. *Discrete and Computational Geometry*, 26: 89–104, 2001.

[9] S. Gao and A.G.B. Lauder. Fast absolute irreducibility testing via Newton polytopes, preprint 2003.

[10] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman, New York, 1979.

[11] J. von zur Gathen and E. Kaltofen. Factorization of multivariate polynomials over finite fields. *Mathematics of Computation*, 45: 251–261, 1985.

[12] B. Grünbaum, *Convex Polytopes*, John Wiley and Sons, New York, 1967.

[13] J. W. P. Hirschfeld, *Projective Geometries over Finite Fields*, Clarendon Press, Oxford, 1979.

[14] E. Kaltofen. Polynomial-time reductions from multivariate to bi- and univariate integral polynomial factorization. *SIAM Journal on Computing*, 14: 469–489, 1985.

[15] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to Parallel Computing: Design and Analysis of Algorithms*, The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1994.

[16] A. K. Lenstra. Factoring multivariate polynomials over finite fields. *Journal of Computer and System Sciences*. 30: 235–248, 1985.

[17] A. K. Lenstra. Factoring multivariate polynomials over algebraic fields". *SIAM Journal on Computing*, 16: 591–598, 1987.

[18] R. Lidl and H. Niederreiter, *Finite Fields*, Encyclopedia of Mathematics and its Applications, vol. 20, Addison-Wesley, Reading, Mass., 1983.

[19] J. M. D. Hill, W. F. McColl, and D. B. Skillicorn. Questions and Answers about BSP". Report PRG-TR-15-96, Oxford University Computing Laboratory, 1996.

[20] J. M. D. Hill, W. F. McColl, D. C. Stefanescu, M. W. Goudrea, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, R. H. Bisseling. BSPlib: The BSP Programming Library. *Parallel Computing*, 24: 1947–1980, 1998.

[21] J. O'Rourke, *Computational Geometry in C*, second edition, Cambridge University Press, Cambridge, New York, 1998.

[22] A. M. Ostrowski. Über die Bedeutung der Theorie der konvexen Polyeder für die formale Algebra". *Jahresberichte Deutsche Math. Verein*, 30: 98–99, 1921.

[23] R. Schneider, *Convex Bodies: the Brunn-Minkowski Theory*, Encyclopedia of Mathematics and its Applications, Vol. 44, Cambridge University Press, Cambridge, 1993.

[24] H. Stichtenoth, *Algebraic Function Fields and Codes*, Universitext, Springer-Verlag, Berlin, 1993.

[25] T. Szönyi. Some applications of algebraic curves in finite geometry and combinatorics. In R. A. Bailey, ed., *Surveys in Combinatorics*, pp. 197–236, London Mathematical Society Lecture Notes Series, Vol. 241, Cambridge University Press, 1997.

[26] L. G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33: 103–111, 1990.