Optimizing and Parallelizing the Modular GCD Algorithm

Matthew Gibson
Department of Mathematics
Simon Fraser University
Burnaby, B.C., Canada. V5A 1S6
mdg583@gmail.com

Michael Monagan
Department of Mathematics
Simon Fraser University
Burnaby, B.C., Canada. V5A 1S6
mmonagan@cecm.sfu.ca

ABSTRACT

Our goal is to design and implement a high performance modular GCD algorithm for polynomial GCD computation in $\mathbb{Z}_p[x_1, x_2, ..., x_n]$ for multi-core computers which will be used to compute the GCD of polynomials over $\mathbb{Z}$.

For $n = 2$ we have designed and implemented in C a highly optimized serial code for primes $p < 2^{63}$. For $n > 2$ we parallelized in Cilk C Brown's dense modular GCD algorithm using our serial bivariate code at the base. For $n = 3$, we obtain good parallel speedup on multi-core computers with 16 and 20 cores. We also compare our code with the GCD codes in Maple and Magma.

Categories and Subject Descriptors

I.1.2 [**Symbolic and Algebraic Manipulation**]: Algorithms

Keywords

Polynomial GCD computation, Parallel Algorithms

1. INTRODUCTION

Consider the multivariate polynomial GCD problem over the integers; that is, compute $G = \mathrm{Gcd}(A, B)$ where $A, B \in \mathbb{Z}[x_1, x_2, \ldots x_n]$. The modular GCD algorithm computes the $\mathrm{Gcd}(A \bmod p, B \bmod p)$ in $\mathbb{Z}_p[x_1, x_2, \ldots, x_n]$ for primes $p_1, p_2, \ldots$ (we use 63 bit primes for convenience) then reconstructs G over $\mathbb{Z}$ using Chinese Remaindering. We could do the computation for each prime in parallel, but since we don't know how many primes we need in advance (we don't know how big the coefficients in G are) and because many large multivariate problems in practice have small integer coefficients, we need to also parallelize the GCD computation for each prime.

The only work in the literature attempting to do this that we know of is that of Rayes and Wang [10] and Wang [11] which parallelize Zippel's sparse GCD algorithm [12, 13]. Other works, e.g. Emeliyanenko [6] and Haque and Moreno Maza [7], parallelize univariate GCD computation. Let

$$G = \textstyle\sum_{i=0}^{d} c_i(x_1, \ldots, x_{n-1}) x_n^d.$$

Zippel's algorithm interpolates the coefficient polynomials $c_0, c_1, \ldots, c_d$ from univariate images in $\mathbb{Z}_p[x_n]$ using a sparse interpolation one variable at a time. Rayes and Wang [10] compute the univariate images for each variable in parallel. Letting $\#A$ denote the number of non-zero terms of a polynomial A, if $\deg G = d$ and $t = \max(\#c_0, \#c_1, \ldots, \#c_d)$ then Zippel's algorithm uses $O(ndt)$ univariate images to interpolate the coefficients. Hu and Monagan [3] propose instead to use Ben-Or/Tiwari sparse interpolation which needs only $O(t)$ images but needs special primes and larger primes.

However, ignored in these works is the computation of the contents. Let

$$A = \textstyle\sum_{i=0}^{d} a_i(x_1, \ldots, x_{n-1}) x_n^i.$$

In the modular GCD algorithm one first computes the content $C = \mathrm{GCD}(a_0, a_1, \ldots, a_d)$ and divides out A by C. Similarly for B. Then one computes the GCD of the two contents and the GCD of the leading coefficients of A and B for leading coefficient correction. All these GCD computations are in $\mathbb{Z}_p[x_1, \ldots, x_{n-1}]$. They can ruin parallel speedup when A and B are not monic in x_n. We propose instead to interpolate G from bivariate images. Let

$$G = \textstyle\sum_{0 \le j,k \le d} b_{jk}(x_1, \ldots, x_{n-2}) x_{n-1}^j x_n^k.$$

Thus we will interpolate the b_{jk} from images in $\mathbb{Z}_p[x_{n-1}, x_n]$. This simplifies the content computations because the coefficients in $\mathbb{Z}_p[x_1, \ldots, x_{n-2}]$ of A and B are likely much smaller. Let $s = \max \#b_{jk}$ and $t = \max \#c_i$. We have $s \le t$ but often s will be much smaller than t. This reduces the cost of sparse interpolation when t is large. The extra cost of computing bivariate images is offset by the reduction in number of images by a factor of t/s.

The proposed approach then is to interpolate the coefficients of G from bivariate images and to compute them as fast as possible. In this paper we compute multiple bivariate images in parallel. For each image we use an optimized serial implementation of Brown's algorithm [1].

In Section 2 we reproduce details of Brown's algorithm, including pseudo-code for Brown's algorithm so that the paper can be understood. Our contribution is a series of optimizations to the serial algorithm and a parallel implementation of a dense multivariate GCD algorithm in Cilk C in Section 3. In Section 4 we compare our parallel GCD algorithm with that of Maple and Magma for problems in $\mathbb{Z}_p[x_1, x_2, x_3]$.

PASCO '15, July 10 - 12, 2015, Bath, United Kingdom
© 2015 ACM. ISBN 978-1-4503-3599-7/15/07…$15.00
DOI: http://dx.doi.org/10.1145/2790282.2790293

A note about asymptotics.

We reduce GCD computation in $\mathbb{Z}_p[x_1, x_2, \dots, x_n]$ to bivariate GCD computations which are then reduced to univariate GCD computations in $\mathbb{Z}_p[x_1]$. Do we need to use the fast Euclidean algorithm (see Chapter 11 of [2]) which has complexity $O(d \log^2 d)$? Or is the $O(d^2)$ classical Euclidean algorithm sufficient? Since the Fast Euclidean algorithm does not break even with the classical Euclidean algorithm until $d > 1000$, and since most large multivariate GCD problems in several variables have degree $d < 100$ in practice, it seems to us that we should try to optimize the classical Euclidean algorithm rather than the fast Euclidean algorithm.

1.1 Definitions and Notations

Let R be a ring and let $A \in R[x_1, x_2, \dots, x_n]$ be non-zero. Let $\mathrm{lc}(A) \in R$ be the leading coefficient of A and $\mathrm{lm}(A)$ be the leading monomial of A. The term ordering that we use is lexicographical order with $x_1 < x_2 < \cdots < x_n$. For example, if $R = \mathbb{Z}_p$ and $A = 3x_1^2x_2 + 5x_1x_2^3$ is a polynomial in $\mathbb{Z}_p[x_1, x_2]$ then $x_1x_2^3 > x_1^2x_2$ in this ordering hence $\mathrm{lc}(A) = 5$ and $\mathrm{lm}(A) = x_1x_2^3$.

Let $A, B \in \mathbb{Z}_p[x_1, x_2, \dots, x_n]$. The GCD of A and B is unique up to a scalar in $\mathbb{Z}_p$. We impose uniqueness by fixing $G = \mathrm{Gcd}(A, B)$ to be the monic GCD with $\mathrm{lc}(G) = 1$. We also define the co-factors $\bar{A}, \bar{B}$ by $G\bar{A} = A$ and $G\bar{B} = B$.

In the algorithm we view $A \in \mathbb{Z}_p[x_1][x_2, \dots, x_n]$ as a multivariate polynomial in the variables $x_2 \dots x_n$ with coefficients in the ring $\mathbb{Z}_p[x_1]$. Then $\mathrm{lm}(A)$ is a monomial in $x_2 \dots x_n$ and $\mathrm{lc}(A) \in \mathbb{Z}_p[x_1]$. We define the content of A, denoted $\mathrm{cont}(A)$ to be the monic GCD of the coefficients of A in $\mathbb{Z}_p[x_1]$. If $\mathrm{cont}(A) = 1$ we say A is primitive. We define the primitive part of A, denoted $\mathrm{pp}(A)$, to satisfy $\mathrm{pp}(A)\,\mathrm{cont}(A) = A$.

The algorithm will evaluate A and B at $x_1 = \alpha_i \in \mathbb{Z}_p$. We use $\phi_i : \mathbb{Z}_p[x_1][x_2 \dots x_n] \to \mathbb{Z}_p[x_2, \dots, x_n]$ to denote the evaluation homomorphism $\phi_i(A) = A(\alpha_i, x_2, \dots, x_n)$.

2. BROWN'S MODULAR GCD ALGORITHM

In developing a modular GCD algorithm, several complicating issues arise. Consider the following inputs in $\mathbb{Z}_{11}[y][x]$:

$$A(x, y) = (y^2 + 3y)x^3 + (y^2 + y + 2)x^2 + (y + 8)x$$
$$B(x, y) = (y^2 + 3y)x^3 + yx^2$$

We wish to evaluate $y = \alpha_1, \alpha_2, \dots \in \mathbb{Z}_p$ and interpolate G from $\mathrm{Gcd}(A(x, \alpha_i), B(x, \alpha_i))$. The degree in y of the GCD must be no more than 2 so 3 points are sufficient. Picking $y = 1, 2, 3$ we get the results in the table below.

α_i	$A_i = \phi_i(A)$	$B_i = \phi_i(B)$	$\mathrm{Gcd}(A_i, B_i)$
1	$4x^3 + 4x^2 + 9x$	$4x^3 + x^2$	$x^2 + 3x$
2	$10x^3 + 8x^2 + 10x$	$10x^3 + 2x^2$	$x^2 + 9x$
3	$7x^3 + 3x^2$	$7x^3 + 3x^2$	$x^3 + 2x^2$

The true GCD is $G = (y + 3)x^2 + x$. Observe that simply interpolating the coefficients of the images $\mathrm{Gcd}(A_i, B_i)$ will not give us G. The first problem is that we will not be able to reconstruct the $y + 3$ coefficient of x^2 in G since the images are all monic. Recovering $y + 3$ is called the leading coefficient problem. The second problem is that the image GCD for $y = 3$ has the wrong degree in x because the cofactors $\bar{A} = yx + y + 8$ and $\bar{B} = xy$ are not relatively prime at $y = 3$. We must detect and discard this image.

In general, consider inputs A, B in $\mathbb{Z}_p[x_1][x_2 \dots x_n]$. For $\phi_i(A) = A(\alpha_i, x_2, \dots, x_n)$ we have

$$\mathrm{Gcd}(\phi_i(A), \phi_i(B)) = \mathrm{Gcd}(\phi_i(\bar{A})\phi_i(G),\ \phi_i(\bar{B})\phi_i(G))$$

If we pick $\alpha_i \in \mathbb{Z}_p$ such that $\phi_i(\mathrm{lc}(A)) \neq 0$, since the function Gcd gives a monic result, this means:

$$\mathrm{Gcd}(\phi_i(A), \phi_i(B)) = \frac{\phi_i(G)}{\mathrm{lc}(\phi_i(G))}\,\mathrm{Gcd}(\phi_i(\bar{A}), \phi_i(\bar{B})) \qquad (1)$$

We say that α_i is *unlucky* if $\mathrm{Gcd}(\phi_i(\bar{A}), \phi_i(\bar{B})) \neq 1$. Note, in order to formally prove termination, one must determine how many $\alpha_i \in \mathbb{Z}_p$ could be unlucky since we can not use them to interpolate G. We will not go into details about this here.

To solve the leading coefficient problem we will assume that the inputs $A, B \in \mathbb{Z}_p[x_1][x_2 \dots x_n]$ have been made primitive at the beginning of the algorithm by computing and dividing out by their content in $\mathbb{Z}_p[x_1]$. Further, a consequence of Gauss' Lemma is that

$$\mathrm{Gcd}(A, B) = \mathrm{Gcd}(\mathrm{pp}(A), \mathrm{pp}(B)) \times \mathrm{Gcd}(\mathrm{cont}(A), \mathrm{cont}(B))$$

so we can compute the GCD and co-factors of the content and primitive parts separately. Then we compute $\gamma = \mathrm{Gcd}(\mathrm{lc}(A), \mathrm{lc}(B))$, where $\mathrm{lc}(A)$ and $\mathrm{lc}(B)$ are in $\mathbb{Z}_p[x_1]$. Now $G \mid A$ and $G \mid B$ implies $\mathrm{lc}(G) \mid \mathrm{lc}(A)$ and $\mathrm{lc}(G) \mid \mathrm{lc}(B)$ and therefore $\mathrm{lc}(G) \mid \gamma$. Let $\gamma = \mathrm{lc}(G)\Delta$ for some $\Delta \in \mathbb{Z}_p[x_1]$. Then for each evaluation point α_i, if $\phi_i(\gamma) = 0$ we discard it. Otherwise, we multiply each image GCD by $\phi_i(\gamma) = \phi_i(\mathrm{lc}(G))\phi_i(\Delta)$. We know $\phi_i(\gamma) \neq 0$ implies $\phi_i(\mathrm{lc}(G)) = \mathrm{lc}(\phi_i(G))$, and therefore (1) implies

$$\phi_i(\gamma)\,\mathrm{Gcd}(\phi_i(A), \phi_i(B)) = \phi_i(\Delta)\phi_i(G)\,\mathrm{Gcd}(\phi_i(\bar{A}), \phi_i(\bar{B})). \qquad (2)$$

If no evaluation point is unlucky, then after computing $\deg_{x_1}(\Delta) + \max(\deg_{x_1} A, \deg_{x_1} B) + 1$ image GCDs, we will have enough information to interpolate the polynomial $\Delta \times G$, where $\deg_{x_1}(\Delta)$ is bounded by $\deg_{x_1}(\gamma)$. Since the inputs were assumed to be primitive, it follows that G is also primitive, so it can be recovered as $G = \mathrm{pp}(\Delta G)$.

2.1 Detecting unlucky evaluation points

Recall that A, B, G are in $\mathbb{Z}_p[x_1][x_2 \dots x_n]$ and that $\mathrm{lm}(A)$ is a monomial in $x_2 \dots x_n$. Thus $\mathrm{lm}(\phi_i(\Delta)) = 1$ since Δ is in $\mathbb{Z}_p[x_1]$ and $\mathrm{lm}(\phi_i(G)) = \mathrm{lm}(G)$ when $\phi_i(\mathrm{lm}(G)) \neq 0$. Then if we let $g_i^* = \phi_i(\gamma)\,\mathrm{Gcd}(\phi_i(A), \phi_i(B))$ for each α_i, then by (2)

$$\mathrm{lm}(g_i^*) = \mathrm{lm}(G)\,\mathrm{lm}(\mathrm{Gcd}(\phi_i(\bar{A}), \phi_i(\bar{B}))).$$

Then we have $\mathrm{lm}(g_i^*) > \mathrm{lm}(G)$ if and only if α_i is unlucky; otherwise, $\mathrm{lm}(g_i^*) = \mathrm{lm}(G)$. Using this fact, Brown rules out unlucky evaluation points with a high probability using the following test after we compute g_i^*:

1. If $\mathrm{lm}(g_i^*) > \mathrm{lm}(g_{i-1}^*)$, then α_i is unlucky so choose a new evaluation point.
2. If $\mathrm{lm}(g_i^*) < \mathrm{lm}(g_{i-1}^*)$, then discard all previous evaluation points; they are unlucky.

Once we obtain $\deg_{x_1}(\gamma) + \max(\deg_{x_1} A, \deg_{x_1} B) + 1$ images for g_i^* with the same leading monomial, then either all evaluation points are unlucky or none are. While the probability of all α_i being unlucky is often very small, Brown tests whether the interpolated GCD (E in the following theorem) divides the inputs. If it does then the α_i used cannot be unlucky.

Theorem 1. *Let E, G be in $\mathbb{Z}_p[x_1][x_2 \dots x_n]$, $G \neq 0$ and G primitive so* $\mathrm{cont}(G) = 1$ *in* $\mathbb{Z}_p[x_1]$. *Let ϕ be an evaluation homomorphism evaluating $x_1 = \alpha$ for some α in $\mathbb{Z}_p$ such that* $\mathrm{lc}(G)$ *does not vanish under ϕ. Then if $E \mid G$ and $\phi(G) \mid \phi(E)$, it follows that $G \mid E$.*

Proof. Recall that the leading monomial of a polynomial in $\mathbb{Z}_p[x_1][x_2 \dots x_n]$ is the monomial in $x_2 \dots x_n$ of the leading term. Since $E \mid G$, we know that $G = KE$ for some $K \in \mathbb{Z}_p[x_1][x_2 \dots x_n]$. Also, since G is primitive, either K is a unit or $\deg_{x_i}(K) > 0$ for some $2 \leq i \leq n$. Then $\phi(G) = \phi(K)\phi(E)$ so that $\mathrm{lm}(\phi(E)) \leq \mathrm{lm}(\phi(G))$. Since $\phi(G) \mid \phi(E)$ by assumption and $\phi(E) \neq 0$, we further get that $\mathrm{lm}(\phi(G)) \leq \mathrm{lm}(\phi(E))$ so that $\mathrm{lm}(\phi(G)) = \mathrm{lm}(\phi(E))$.

Since $E \mid G$, we know that the leading term of E does not vanish under ϕ, and so $\mathrm{lm}(\phi(E)) = \mathrm{lm}(E)$ and $\mathrm{lm}(\phi(G)) = \mathrm{lm}(G)$. Then it follows that:

$$\mathrm{lm}(G) = \mathrm{lm}(\phi(G)) = \mathrm{lm}(\phi(E)) = \mathrm{lm}(E)$$

Since $G = KE$, this means $\mathrm{lm}(K) = 1$ and so $K \in \mathbb{Z}_p[x_1]$. Then since G is primitive, K is a unit and $G \mid E$. $\square$

Using this theorem, if an interpolated GCD divides the true GCD and if at least one of its images is divisible by the corresponding image of the true GCD, then it must be equal to the true GCD up to a unit.

2.2 MGCD algorithm

Brown's MGCD algorithm from [1] outputs $G = \mathrm{Gcd}(A, B)$ and the cofactors $\bar{A}$ and $\bar{B}$. After computing and removing contents from the inputs it computes $\gamma = \mathrm{Gcd}(\mathrm{lc}(A), \mathrm{lc}(B))$ in $\mathbb{Z}_p[x_1]$ and a bound $bnd = 1 + \min(\deg_{x_1} A, \deg_{x_1} B) + \deg_{x_1}(\gamma)$. Then it picks evaluation points $\alpha_i \in \mathbb{Z}_p$ for which $\phi_i(\gamma) \neq 0$ and computes $g_i^* = \phi_i(\gamma)\,\mathrm{Gcd}(\phi_i(A), \phi_i(B))$.

We know that $g_i^* \mid \phi_i(\gamma A)$ and $g_i^* \mid \phi_i(\gamma B)$, which means that we can define unique $\bar{a}_i^*$ and $\bar{b}_i^*$ satisfying:

$$\begin{aligned} \phi_i(\gamma A) - g_i^* \bar{a}_i^* &= 0 \\ \phi_i(\gamma B) - g_i^* \bar{b}_i^* &= 0 \end{aligned} \tag{3}$$

Using congruence notation, this is

$$\begin{aligned} \gamma A - g_i^* \bar{a}_i^* &\equiv 0 \mod (x_1 - \alpha_i) \\ \gamma B - g_i^* \bar{b}_i^* &\equiv 0 \mod (x_1 - \alpha_i). \end{aligned} \tag{4}$$

The recursive call to MGCD computes $g_i^*, \bar{a}_i^*$ and $\bar{b}_i^*$. Using $g_i^*, \bar{a}_i^*$ and $\bar{b}_i^*$ we will iteratively build the interpolants G_k^*, $\bar{A}_k^*$ and $\bar{B}_k^*$ to satisfy the following

$$\left.\begin{aligned} G_k^* &\equiv g_i^* \mod (x_1 - \alpha_i) \\ \bar{A}_k^* &\equiv \bar{a}_i^* \mod (x_1 - \alpha_i) \\ \bar{B}_k^* &\equiv \bar{b}_i^* \mod (x_1 - \alpha_i) \end{aligned}\right\} \text{ for } i = 1 \dots k. \tag{5}$$

Further, since $(x_1 - \alpha_i)$ and $(x_1 - \alpha_j)$ are relatively prime for $i \neq j$, we can define $M_k = (x_1 - \alpha_1)(x_1 - \alpha_2) \dots (x_1 - \alpha_k)$ and get from (4) and (5) that:

$$\begin{aligned} \gamma A - G_k^* \bar{A}_k^* &\equiv 0 \mod M_k \\ \gamma B - G_k^* \bar{B}_k^* &\equiv 0 \mod M_k. \end{aligned} \tag{6}$$

As we interpolate, we know that $M_k \mid \gamma A - G_k^* \bar{A}_k^*$ and $M_k \mid \gamma B - G_k^* \bar{B}_k^*$. Then when we have interpolated at least $bnd = \deg_{x_1}(\gamma) + \max(\deg_{x_1} A, \deg_{x_1} B) + 1$ images, we know that $\deg_{x_1}(\gamma A) < bnd$ and $\deg_{x_1}(\gamma B) < bnd$, and since $\deg_{x_1}(M_k) = k \geq bnd$, we know the following:

1. If $\deg_{x_1}(G_k^* \bar{A}_k^*) < bnd$ then $\gamma A = G_k^* \bar{A}_k^*$
2. If $\deg_{x_1}(G_k^* \bar{B}_k^*) < bnd$ then $\gamma B = G_k^* \bar{B}_k^*$

If these conditions are satisfied then $G_k^* \mid \gamma G$. Since G is primitive with respect to $x_2 \dots x_n$, we know further that $\mathrm{pp}(G_k^*) \mid G$. For each α_i, we ensured that $\phi_i(\gamma) \neq 0$ so that the leading term of G doesn't vanish under ϕ_i. By (2) and since $g_i^* = \phi_i(G_k^*)$ we know that $\phi_i(G) \mid \phi_i(G_k^*)$ and therefore $\phi_i(G) \mid \phi_i(\mathrm{pp}(G_k^*))$. Then by Theorem 1 we know that $G \mid \mathrm{pp}(G_k^*)$ and so $\mathrm{pp}(G_k^*)$ is equal to G up to a unit. Further, $\mathrm{pp}(G_k^*)$ will be monic. The corresponding primitive co-factors can be found as $\bar{A} = \mathrm{pp}(\bar{A}_k^*)$ and $\bar{B} = \mathrm{pp}(\bar{B}_k^*)$. Finally the contents of the original inputs (see Steps 3 and 18) are put back on the GCD and cofactors.

2.3 Pseudocode for Non-optimized MGCD

Algorithm MGCD takes as inputs polynomials $A, B \in \mathbb{Z}_p[x_1][x_2] \dots [x_n]$ where $n \geq 1$. The output is $[G, \bar{A}, \bar{B}]$, the GCD and two co-factors.

Our implementation stores polynomials in a recursive dense representation which takes the form of a tree with n levels where the leaves of the tree are polynomials in $\mathbb{Z}_p[x_1]$. Note, in the algorithm, all GCD computations in $\mathbb{Z}_p[x_1]$ are done using the Euclidean algorithm.

```
Begin:
  1. If $n = 1$ compute $G := \mathrm{GCD}(A, B)$ using the Euclidean algorithm then compute $cA := B/G$ and $cB := B/G$ using univariate division and return $[G, cA, cB]$.
  2. Compute and remove content in $\mathbb{Z}_p[x_1]$ from $A$ and $B$:
     $cA := \mathrm{cont}(A)$, $cB := \mathrm{cont}(B)$, $A := A/cA$, $B := B/cB$ .
  3. Compute the content GCD and co-factors as
     $cG := \mathrm{GCD}(cA, cB)$, $c\bar{A} := cA/cG$ and $c\bar{B} := cB/cG$.
  4. Compute $\gamma := \mathrm{GCD}(\mathrm{lc}(A), \mathrm{lc}(B)) \in \mathbb{Z}_p[x_1]$.
  5. Set $bnd := \deg(\gamma) + \max(\deg_{x_1} A, \deg_{x_1} B) + 1$.
  6. Set $G^*, A^*, B^* := 0$ to clear interpolants and $i := 0$.
Loop:
  7. Pick a new $\alpha_i \in \mathbb{Z}_p$ at random such that $\phi_i(\gamma) \neq 0$.
  8. Evaluate at $x_1 = \alpha_i$. Compute $[g^*, \bar{a}^*, \bar{b}^*] :=$
     $MGCD(\ \phi_i(A),\ \phi_i(B)\ ) \in \mathbb{Z}_p[x_2, \dots, x_n]$.
  9. If $\mathrm{lm}(g^*) = 1$ set $G^* := 1, cA^* := A, cB^* := B$ and goto End, since $A$ and $B$ must be relatively prime.
  10. If $i > 0$ and $\mathrm{lm}(g^*) > \mathrm{lm}(G^*)$ goto Loop ($\alpha$ is unlucky).
      If $i > 0$ and $\mathrm{lm}(g^*) < \mathrm{lm}(G^*)$ set $G^*, A^*, B^* := 0$, $i := 0$ (all previous evaluations were unlucky).
  11. Multiply $g^* := \phi_i(\gamma) \times g^*$.
  12. Extend $G^*, \bar{A}^*, \bar{B}^*$ by interpolating into them the new images $g^*, \bar{a}^*$ and $\bar{b}^*$ respectively.
  13. Set $i := i + 1$.
  14. If $i < bnd$ then goto Loop.
  15. If $\deg_{x_1}(\gamma) + \deg_{x_1}(A) = \deg_{x_1}(G^*) + \deg_{x_1}(\bar{A}^*)$ and $\deg_{x_1}(\gamma) + \deg_{x_1}(B) = \deg_{x_1}(G^*) + \deg_{x_1}(\bar{B}^*)$ then goto End since the degree condition guarantees divisibility and we are done.
  16. Otherwise, set $G^*, A^*, B^* := 0$ and $i := 0$ and goto Loop as all evaluation points were unlucky.
End:
  17. Remove the content from the interpolants:
      $G^* := G^*/\mathrm{cont}(G^*)$,  $\bar{A}^* := \bar{A}^*/\mathrm{cont}(\bar{A}^*)$, $\bar{B}^* := \bar{B}^*/\mathrm{cont}(\bar{B}^*)$.
  18. Add the correct content to the GCD and co-factors:
      $G^* := cG \times G^*$,  $\bar{A}^* := c\bar{A} \times \bar{A}^*$, $\bar{B}^* := c\bar{B} \times \bar{B}^*$.
  19. Return $[G^*, \bar{A}^*, \bar{B}^*]$.
```

3. OPTIMIZATIONS TO THE ALGORITHM

3.1 Division in the Images

In the MGCD algorithm, the validity of the interpolated GCD is verified by checking divisibility. This is done by computing images and doing interpolations for $\text{bnd} = \deg_{x_1}(\gamma) + \max(\deg_{x_1} A, \deg_{x_1} B) + 1$ evaluation points. Since the degree of ΔG could be much smaller than bnd, we could use a probabilistic test for the completeness of the interpolated GCD and find another way to test divisibility.

Monagan and Wittkopf did this in [5]. They compute a tight degree bound for G_k^* in x_1 by computing one univariate image of G in x_1 and then compute enough image GCDs to interpolate only G. Then division is used to check divisibility and acquire $\bar{A}_k^*$ and $\bar{B}_k^*$. However, division is slow if a classical algorithm is used and neither the divisor nor the quotient is small.

We present an alternative optimization for the bivariate case where A and B are in $\mathbb{Z}_p[x_1, x_2]$. If, after k images, one of the interpolants G_k^*, $\bar{A}_k^*$ or $\bar{B}_k^*$ doesn't change with the addition of the next image, we say the interpolant has stabilized. Then, for each additional evaluation point α_i, we do one of the following.

1. If the stabilized interpolant is G_k^*, we evaluate $\phi_i(G_k^*)$ to obtain g_i^* and $\phi_i(\gamma)\phi_i(A)$ and $\phi_i(\gamma)\phi_i(B)$ as before. Then we use univariate division in $\mathbb{Z}_p[x_2]$ to compute $\bar{a}_i^*$ and $\bar{b}_i^*$ according to (3) and then continue interpolation of $\bar{A}_i^*$ and $\bar{B}_i^*$.

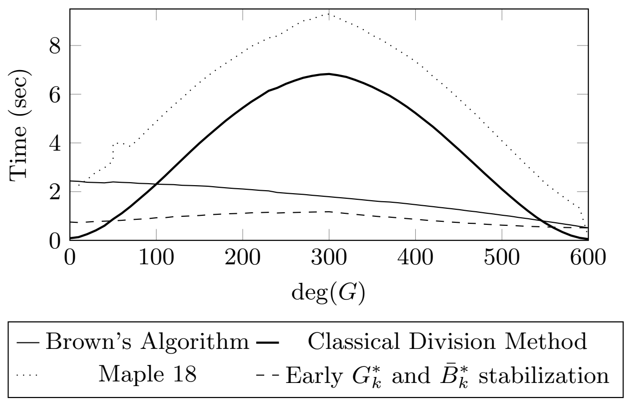
2. If the stabilized interpolant is $\bar{B}_k^*$ (or symmetrically $\bar{A}_k^*$), we evaluate $\phi_i(\gamma)\phi_i(A)$, $\phi_i(\gamma)\phi_i(B)$ and $\bar{b}_i^* = \phi_i(\bar{B}_k^*)$ and use univariate division to compute first g_i^* and then $\bar{a}_i^*$ to satisfy (3). As before, we continue interpolation of G_i^* and $\bar{A}_i^*$ with the new image.

If these univariate divisions ever have a non-zero remainder, then $G_k^* \nmid G$ so it is either built from all unlucky α's, or the stabilized interpolant was not actually complete. Otherwise, once we have performed this image division for the rest of the evaluation points up to the bound, we use the same degree check as before to verify divisibility and thereby the correctness of the GCD and co-factors.

Normally, we would have to evaluate $\phi_i(A)$, $\phi_i(B)$, compute $\text{Gcd}(\phi_i(A), \phi_i(B))$ and then use two divisions to get $\bar{a}_i^*$ and $\bar{b}_i^*$. This optimization replaces at least half of the univariate GCD computations with bivariate polynomial evaluations one of G, $\bar{A}$ and $\bar{B}$ of least degree in x_1. Optimization 3.3 will reduce this cost further.

Figure 1 compares different strategies for verifying divisibility. The plot is for dense inputs A, B of degree 600 in both x_1 and x_2, where the degree of the GCD G is $0, 10, 20, \ldots, 600$. The plot compares our implementation of the non-optimized modular GCD algorithm (Brown's Algorithm) and the early G_k^* and $\bar{B}_k^*$ stabilization here. Also shown are results of our algorithm using a classical division test and Maple 18's built-in GCD mod p command which also uses a classical division test. The two humps in Figure 1 show classical division dominating the cost. We can see that the optimization gives the most benefit compared to the non-optimized algorithm when $\deg(G)$ is small, and that classical division is very slow when $\deg(G)$ is neither very small nor very large.

Figure 1: Image Division Optimizations



3.2 Eliminating Integer Divisions

Our implementation uses 63 bit primes. Therefore a multiplication $a \times b$ in $\mathbb{Z}_p$ needs to multiply two 63 bit integers and divide a 126 bit product by a 63 bit prime p. 64 bit computers have hardware instructions for this multiplication and for the division. The cost of the hardware division instruction, however, is prohibitive. On an Intel E5-2680 v2 processor, the integer multiplication $a \times b$ costs 2.655 cycles but one division of ab by p costs 66.602 cycles. There are many papers in the literature that address this problem. The main idea is to replace division with two multiplications and other cheap operations. We are using the algorithm described by Möller and Torbjorn in [4]. Our implementation for computing $ab \mod p$ for a 63 bit prime takes 6.016 cycles. Compared with 66.602 cycles this is a huge gain, nevertheless, division is still relatively expensive compared with multiplication so we try to eliminate divisions.

There are several places in the MGCD algorithm where we need to compute a sum of products. For example, consider evaluating $B \in \mathbb{Z}_p[x_1][x_2]$ at $x_1 = \alpha \in \mathbb{Z}_p$ and let $a = \sum_{i=0}^{d} a_i x_1^i$ be a coefficient of B. Horner's rule

$$a_0 + \alpha(a_1 + \alpha(a_2 + \ldots \alpha(a_{d-1} + \alpha a_d) \ldots))$$

uses d multiplications and d additions. But because the multiplications are nested we must reduce mod p after each one to prevent overflow.

Suppose instead we pre-compute an array X of the powers of α modulo p so that $X[k] = \alpha^k \mod p$ and suppose the coefficients of a are stored in the array A. The following C code snippet assumes p is a 31 bit prime. It accumulates $-a(\alpha)$ in a signed 64 bit integer T then does one division by p at the end. We accumulate $-a(\alpha)$ instead of $a(\alpha)$ to eliminate the branch.

```
long M = (long) p << 32; // = 2^32 x p
long T = M;
for(k=0; k <= d; k++)
{
   T -= (long) A[k] * X[k];
   T += (T>>63) & M; // if(T < 0) T += M;
}
T = M-T;
return T % p;
```

One may optimize this further since we may accumulate two products without overflow. Our implementation for 63-

bit primes uses assembler macros to manage a 128-bit accumulator. The performance improvement that we obtain by doing this is a factor of 3.1 for $p = 1924145348627$. This optimization is used in univariate polynomial evaluation, interpolation, division, multiplication and gcd. For univariate gcd we explain what we did.

Algorithm MGCD computes many GCDs in $\mathbb{Z}_p[x_1]$. We use the Euclidean algorithm which performs a sequence of divisions each to acquire a new remainder. If we call the inputs to a division A of degree m and B of degree n, then it is normally the case that $m = n+1$ for all divisions except the first. We can write this as:

$$\begin{aligned} A &= a_0 + a_1 x + \cdots + a_n x^n + a_{n+1} x^{n+1} \\ B &= b_0 + b_1 x + \cdots + b_n x^n \end{aligned}$$

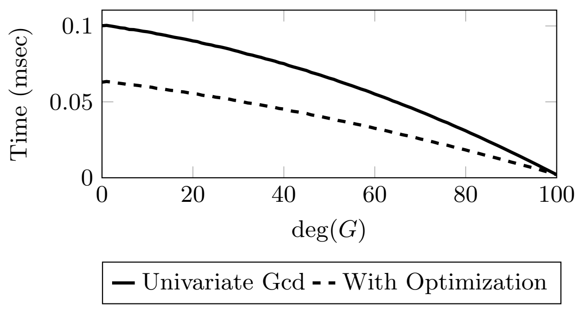
When using a standard division routine, the terms of the quotient are computed one at a time in a loop and then in a nested loop terms are subtracted from A. We optimize this computation by first computing the quotient $Q = ax + b$ of $A \div B$ and then subtracting QB from A in a simple for loop.

This following C code snippet illustrates how to do this for a 31 bit prime p using a long int (64 bits) for the accumulator. It does n divisions by p.

```
void div1( int *A, int *B, int n,
           long a, long b, int p )
{  /* compute A(x) = A(x) - (ax+b) B(x) */
   long M = (long) p << 32;
   A[0] = (M+A[0]-b*B[0]) % p;
   for( i=1; i<n; i++ )
      A[i] = (M+A[i]-a*B[i-1]-b*B[i]) % p;
   A[n] = 0;
   A[n+1] = 0;
   return;
}
```

Figure 2 shows the performance gain for our 63 bit prime implementation over the unoptimized Euclidean algorithm for inputs of degree 100 whose GCD degree varies from 0 to 100. The average savings is 43%.

Figure 2: Univariate GCD Optimization



3.3 Partial FFT

Along with the univariate GCD, the other main contribution to the time of the MGCD algorithm is evaluation and interpolation in $\mathbb{Z}_p[x_1]$. An initial optimization is to evaluate at pairs of points $\pm\alpha$. This reduces the cost of evaluation almost in half. We separate $f \in \mathbb{Z}_p[x_1]$ into coefficients with even and odd powers of x_1 and evaluate each term in f once at $x_1 = \alpha$. Then $f(\alpha)$ and $f(-\alpha)$ can be found using an addition and subtraction. Interpolation can similarly be cut almost in half. We have experimented with generalizing this to a partial FFT to further reduce the evaluation and interpolation cost which we now describe.

Given the field $\mathbb{Z}_p$, let j be a power of 2 such that $j \mid p-1$. Let $f \in \mathbb{Z}_p[x]$ be a polynomial of degree $n > j-1$. We will evaluate f in batches of size j. For each evaluation point α, we will find a polynomial f^* of degree less than j satisfying

$$f^* \equiv f \mod (x^j - \alpha^j)$$

The term $(x^j - \alpha^j)$ factors as

$$(x^j - \alpha^j) = \textstyle\prod_{i=0}^{j-1} (x - \alpha\omega_i)$$

where each ω_i is a distinct jth root of unity in $\mathbb{Z}_p$. Because each of these factors is relatively prime, we know that

$$f^* \equiv f \mod (x - \alpha\omega_i) \qquad \text{for } i = 0, 1, \ldots, j-1$$

This means that f and f^* evaluate to the same points at all of the values in $S = \{\alpha\omega_i\}_{i=0}^{j-1}$. So to evaluate f at the j distinct points in S, we compute f^* by substituting $x^j = \alpha^j$, and then evaluate at the j points using the FFT. The cost involves first computing $\{1, x^j, x^{2j} \ldots\}$ and then using $\deg(f)$ multiplications and additions to compute f^*. Then the FFT has a cost in $j \log j$.

The strategy for interpolation is similar. We interpolate $g \in \mathbb{Z}_p[x]$ using a modification of Newton's method. Let $m_k = \prod_{i=0}^{k} (x^j - \alpha_i^j)$. The kth interpolant of g satisfies $g_k \equiv g \mod m_k$ and may be written

$$g_k = c_0 + c_1 m_0 + \cdots + c_k m_{k-1}$$

where each c_i is a polynomial of degree less than j. Given a new batch of j images of g at the points $x \in \{\alpha_{k+1}\omega_i\}_{i=0}^{j-1}$, we first use the inverse FFT to get a polynomial $g^* \equiv g \mod (x^j - \alpha_{k+1}^j)$. Then because $g \equiv g_{k+1} \mod m_{k+1}$ and $g_{k+1} = g_k + c_{k+1} m_k$, and further since $m_{k+1} = m_k(x^j - \alpha_{k+1}^j)$ we get the next coefficient polynomial as:

$$c_{k+1} \equiv \frac{g^* - g_k}{m_k} \mod (x^j - \alpha_{k+1}^j)$$

according to the normal form of Newton interpolation.

The cost of interpolating a batch of size j involves first using the inverse FFT to interpolate a polynomial of degree at most $j-1$, with a cost in $j \log j$. Next we need to evaluate the basis functions $\{1, m_0, m_1 \ldots\}$ by substituting $x^j = \alpha^j$ and then use these to reduce g_k and $m_k \mod (x^j - \alpha_{k+1}^j)$. This can all be done in a cost linear in $\deg(g)$. Thus the cost *per point* for both evaluation and interpolation of a polynomial of degree n is

$$C = \frac{n}{j} + \log j \tag{7}$$

where j is a power of 2. We can see that as we increase j from 1, the linear part of this cost drops quickly, while the logarithmic part grows slowly. While it is hard to use the FFT in the GCD algorithm due to the cost of computing extra image GCDs, the partial FFT with small j can still get much of the performance gains in evaluation and interpolation while only computing a few extra image GCDs.

Table 1 shows the results for using various values of j for two different problems. Tests are for triangular dense inputs of total degree 300 with GCD of total degree 150, with $p = 167772161$. Table 1a uses blocks of size j for

Table 1: Partial FFT Optimizations

j	Time for evaluation and interpolation	for image GCDs in $\mathbb{Z}_p[x_2, x_3]$	Total Time
1	7.5481	28.9039	36.5883
2	3.9039	27.8351	31.8288
4	2.9636	27.9806	31.0109
8	2.5036	27.9426	30.5034
16	2.1192	27.9491	30.1215
32	2.1517	29.4514	31.6592

Table 1a: GCD timings (seconds) for $A, B \in \mathbb{Z}_p[x_1][x_2, x_3]$

j	Time for evaluation and interpolation	for image GCDs in $\mathbb{Z}_p[x_3]$	Total time
1	0.0538	0.0703	0.1277
2	0.0267	0.0687	0.0975
4	0.0196	0.0699	0.0910
8	0.0189	0.0702	0.0903
16	0.0160	0.0753	0.0923
32	0.0159	0.0811	0.0981

Table 1b: GCD timings (secconds) for $A, B \in \mathbb{Z}_p[x_2][x_3]$

evaluation and interpolation of the variable x_1 only. Table 1b shows the results of blocks of size j for the sub-problem in $\mathbb{Z}_p[x_2, x_3]$. Using $j = 8$ for the variable x_1 in the trivariate problem gives a performance gain of about 20 %, and using $j = 8$ for the variable x_2 in the bivariate problem gives a gain of 41%. The results suggest that we use $j = 2$ or $j = 4$ which realizes most of the benefit.

3.4 Parallelization

The modular GCD algorithm is well suited to parallel computation provided one does not use explicit multivariate polynomial division checks which is what Maple and Magma do. This was implemented using the CILK framework which uses a work stealing algorithm and a fixed number of worker threads.[1] CILK uses two basic commands. `cilk_spawn` branches a thread of execution into two, creating a new job, while `cilk_sync` collects multiple threads back together, ending any branched jobs. We do not make use of the third CILK command, `cilk_for`.

Our implementation only uses parallelization for 3 or more variables, providing a fairly large bivariate problem at the base of parallelization. In the parallel version of MGCD we perform the *bnd* evaluations and image GCDs in parallel. We do this by first spawning $\lceil bnd/j \rceil$ jobs of blocks of size j. For each block the j input evaluations are performed in serial (using the partial FFT), and then j more jobs are spawned for the recursive calls to MGCD on the input images. This call will involve further parallelization if it is also in 3 or more variables. When all of this is complete, a `cilk_sync` command waits for the parallel threads to terminate then interpolation begins.

To parallelize interpolation if we think of the interpolation image $\phi_i(G)$ as being in $\mathbb{Z}_p[x_2 \ldots x_{n-1}][x_n]$, then $\deg_{x_n} \phi_i(G)$ jobs are spawned on the coefficients in x_n, which are interpolated in $\mathbb{Z}_p[x_1 \ldots x_{n-1}]$. Further, the GCD and two co-factors are interpolated in parallel.

An issue when trying to parallelize large memory-intensive programs is the fact that implementations of `malloc` for memory allocation in C use mutexes to protect against memory allocation contention. Since the modular multivariate GCD algorithm breaks a large problem into many smaller problems, if all memory allocations were performed using `malloc` throughout the recursive algorithm, there is potential for thread contention. To prevent this situation, and to avoid the somewhat costly calls to `malloc`, we attempt to pre-allocate memory for the algorithm as much as possible.

To do this, pre-computations were performed on the inputs to get memory usage bounds, also making some basic decisions for execution of the recursive algorithm that would affect memory usage. We pre-allocate a large contiguous block of memory in the form of an array. During execution, memory is drawn from this pool in the recursive algorithm by treating it as a stack.

To avoid making assumptions about the number of worker threads available to CILK, memory needed to be allocated to allow all image GCDs of a set of inputs to be calculated simultaneously. The result was that the amount of memory required often could not be pre-allocated in one block for reasonably large inputs. Therefore we used a compromise by allocating large blocks of memory several times dynamically throughout the MGCD algorithm. Each call to MGCD for $n > 2$ variables was given enough memory for the computations in n variables, but not for the recursive calls in $n - 1$ variables. When $n = 2$, each call to MGCD was given enough memory for the entire bivariate problem.

4. PERFORMANCE TIMINGS

Timings were obtained on two Intel Xeon servers `gaby` and `jude`. Both servers are running CentOS Unix. `gaby` has two E5-2660 CPUs, each with 8 cores running at 2.2 GHz (3.0 GHz turbo). `jude` has two E5-2680 v2 CPUs, each with 10 cores running at 2.8 GHz (3.6 GHz turbo). So the theoretical maximum parallel speedup on `gaby` is $2.2/3.0 \times 16 = 11.7\times$ and on `jude` is $2.8/3.6 \times 20 = 15.5\times$.

Parallel Performance and Optimization Tests

The first set of tests in Tables 2, 3 and 4 investigate the effectiveness of the optimizations in sections 3.1, 3.2 and 3.3 and the performance of our parallel code. These tests were run on the `jude` machine, and are on monic trivariate inputs A and B of total degree d in which $G, \bar{A}$ and $\bar{B}$ are dense with random coefficients from $\mathbb{Z}_p$. In each set of tests, the total degree of G varies while the total degree d of the inputs A and B is held constant. There are three sets of tests, one for $p = 2^{30} - 35$ and $d = 200$, and one for $p = 2^{62} - 57$ and $d = 200$ and one for $p = 2^{62} - 57$ and $d = 400$.

For each generated input, tests were run on a fully optimized non-parallel version of code (column No CILK). Then three separate tests were run with each of the optimizations turned off (columns 3.1,3.2,3.3). Further tests were used to find what percentage of No CILK time was spent on the univariate GCD part of the recursive algorithm. Finally, tests were run on the parallel version of the code with 1,2,4,8,16 and 20 worker threads.

For each of these tests, the MGCD algorithm was compiled as a Maple extension and called from Maple. When run in this way, the input and output of the MGCD algorithm needs to be converted between Maple's internal integer polynomial structure called POLYDAG and the recursive array structure used in the MGCD algorithm. Timings

[1]typically set to the number of cores on the machine.

for this conversion were taken separately, and are shown in the columns POLYDAG:In/Out. To get the real times of each Maple call, the POLYDAG conversion times should be added to the other columns.

Notes on Tables 2, 3 and 4

1. Optimization 3.1 is the most significant of the three. It is greatest when $\deg(G)$ is small, since this is when the univariate GCD problems are most expensive. In this case the gain is a speedup of a factor of 2 to 3.

2. Optimization 3.2 is most effective when G is mid-sized, that is when $\deg(G) = \deg(\bar{A}) = \deg(\bar{B})$, which is when the problem requires the most univariate GCDs. This is because optimization 3.1 reduces the number of needed univariate GCDs to approximately $\min(\deg G, \deg \bar{A}, \deg \bar{B})$.

3. The benefit of doing evaluation and interpolation in blocks using the partial FFT method (optimization 3.3) was more than expected. This is partly because optimization 3.1 replaces univariate GCDs with evaluations. The gain is most when % UniGcd is small and more time is spent on evaluation and interpolation.

4. Comparing the No CILK results to the parallel version with 1 core indicates that the CILK overhead is minimal for this problem.

5. Column % UniGcd shows that a surprisingly small percentage of the time is in the univariate GCD problem. This is due to optimization 3.1 and optimization 3.2. Notice that the portion of time reduces to just a few percent as $\deg(G)$ increases, since the univariate GCD problem is least expensive when the GCD is large.

6. Using multiple cores, not counting input and output conversions, we obtain parallel speedups of 6.01/0.48 = 12.5 and 6.72/0.50 = 13.4 when $\deg(G)$ is mid-sized. This is a good result for inputs of degree 200. For inputs of degree 400 parallel speedup improves to 86.89/5.93 = 14.6 which is close to the maximum theoretical speedup of 15.55 for this machine.

Comparison with Maple and Magma

The second set of tests in Tables 5 and 6 compares the parallel MGCD algorithm to the modular GCD algorithms in Maple 18 and Magma 2.19-6. These tests were run on the `gaby` machine. Inputs were generated in the same manner as for those in Tables 2 and 3.

The Maple and Magma GCD timings include a Mult and Div column. The Mult column measures the polynomial multiplication cost of generating the two inputs as $A = G\bar{A}$ and $B = G\bar{B}$, while the Div column measures the time to compute the co-factors as $\bar{A} = A/G$ and $\bar{B} = B/G$ using polynomial division. These allow us to compare the relative cost of a GCD with multiplication and division. Note, our MGCD algorithm constructs and returns the co-factors without a division.

The Magma tests are run twice. In the columns under MagmaR, the tests are run on a Magma ring constructed in the recursive polynomial structure $GF(p)[x][y][z]$ as follows.

```
F := GaloisField(p);
S<x> := PolynomialRing(F);
T<y> := PolynomialRing(S);
P<z> := PolynomialRing(T);
```

The columns under MagmaM instead run the tests in the multivariate polynomial ring $GF(p)[x, y, z]$ constructed with

```
F := GaloisField(p);
P<x,y,z> := PolynomialRing(F,3);
```

Notes on Tables 5 and 6

1. Maple 18 uses Hensel Lifting to compute $\mathrm{Gcd}(A, B) \bmod p$ with 3 or more variables. Maple 18 is using the parallel multivariate polynomial multiplication and division algorithms of Monagan and Pearce [8, 9] which improve the performance of the Hensel Lifting. This is also the reason why Maple's multiplication timings (column Mult) are much faster than Magma's.

2. The time for MGCD on one core is typically 20-50 times faster than Maple's Gcd time and is faster than multiplication and division in Maple. This is partly because we are using a modular Gcd algorithm which has complexity is $O(d^4)$ for the case $\deg G = \deg \bar{A} = \deg \bar{B} = d/2$ whereas Maple's multiplication is $O(d^6)$.

3. Both Maple and Magma perform better when coefficients are less than 2^{31}. MGCD performed well on the 62 bit prime. This can mostly be attributed to the optimized integer division in section 3.2.

4. Maple and Magma perform best when G has very low degree or the cofactors have very low degree. This is because they use trial divisions $G \,|\, A$ and $G \,|\, B$ to terminate and to obtain the cofactors $\bar{A}$ and $\bar{B}$.

5. The POLYDAG conversion time for the MGCD inputs remains constant for each test set, since input degrees are not changing. Since there are two co-factors, output conversion time when $\deg(\bar{A}) = \deg(\bar{B})$ is large is roughly twice that of when $\deg(G)$ is large.

5. REFERENCES

[1] W. S. Brown. On Euclid's Algorithm and the Computation of Polynomial Greatest Common Divisors. *J. ACM* **18** (1971), 478-504.

[2] J. von zur Gathen and J. Gerhard, *Modern Computer Algebra*, 3rd ed., Cambridge University Press, 2013.

[3] J. Hu and M. B. Monagan. A Parallel Algorithm to Compute the Greatest Common Divisor of Sparse Multivariate Polynomials Extended Abstract. *Comms. in Comp. Algebra* **47**:3, 108−109, 2013.

[4] Niels Möller and Torbjorn Granlund. Improved division by invariant integers IEEE Transactions on Computers, **60**, 165–175, 2011.

[5] M. B. Monagan and A. D. Wittkopf. On the Design and Implementation of Brown's Algorithm over the Integers and Number Fields. *Proceedings of ISSAC '2000*, ACM Press, pp. 225–233, 2000.

Table 2: `jude` Tests, $p = 2^{30} - 35$, $\deg A = \deg B = 200$, inputs have 1373701 terms

		Without Opt					MGCD, #CPUs						POLYDAG	
$\deg(G)$	$\deg(\bar{A})$	3.1	3.2	3.3	No CILK	% UniGcd	1	2	4	8	16	20	In	Out
10	190	10.86	4.90	8.57	4.38	10.4	4.43	2.31	1.26	0.74	0.47	0.42	0.13	0.13
40	160	10.30	6.53	8.90	5.19	25.0	5.27	2.70	1.43	0.82	0.49	0.44	0.13	0.08
70	130	9.42	7.69	9.18	5.82	33.0	5.78	2.98	1.56	0.88	0.50	0.46	0.13	0.05
100	100	8.41	8.40	9.20	6.00	37.8	6.01	3.09	1.62	0.90	0.51	0.48	0.13	0.03
130	70	7.29	6.18	8.00	4.84	25.0	4.83	2.49	1.31	0.74	0.42	0.37	0.13	0.03
160	40	5.75	4.44	7.00	3.82	11.9	3.84	1.99	1.06	0.60	0.36	0.32	0.13	0.04
190	10	3.93	3.36	6.65	3.13	1.8	3.17	1.67	0.90	0.52	0.33	0.29	0.13	0.07

Table 3: `jude` Tests, $p = 2^{62} - 57$, $\deg A = \deg B = 200$, inputs have 1373701 terms

		Without Opt					MGCD, #CPUs						POLYDAG	
$\deg(G)$	$\deg(\bar{A})$	3.1	3.2	3.3	No CILK	% UniGcd	1	2	4	8	16	20	In	Out
10	190	13.10	5.39	8.79	4.77	11.9	4.79	2.53	1.39	0.84	0.54	0.48	0.13	0.24
40	160	12.39	7.22	9.42	5.73	28.8	5.79	3.00	1.61	0.92	0.55	0.49	0.13	0.14
70	130	11.29	8.26	9.74	6.42	36.9	6.47	3.33	1.76	0.99	0.56	0.49	0.13	0.08
100	100	9.93	9.00	9.87	6.74	41.0	6.72	3.45	1.82	1.00	0.57	0.50	0.13	0.05
130	70	8.38	6.58	8.19	5.29	27.5	5.29	2.73	1.44	0.80	0.46	0.40	0.13	0.05
160	40	6.52	4.71	7.14	4.14	14.4	4.16	2.16	1.16	0.66	0.39	0.34	0.13	0.07
190	10	4.50	3.59	6.58	3.42	1.8	3.44	1.82	0.99	0.58	0.37	0.33	0.13	0.12

Table 4: `jude` Tests, $p = 2^{62} - 57$, $\deg A = \deg B = 400$, inputs have 10827401 terms

		Without Opt					MGCD, #CPUs						POLYDAG	
$\deg(G)$	$\deg(\bar{A})$	3.1	3.2	3.3	No CILK	% UniGcd	1	2	4	8	16	20	In	Out
20	380	157.34	69.68	126.29	60.09	12.0	60.42	32.39	17.74	10.54	6.80	6.17	1.04	1.96
80	320	150.20	93.84	130.85	73.94	27.5	73.68	38.77	20.67	11.55	6.88	5.91	1.05	1.15
140	260	140.62	109.51	132.04	83.87	35.6	82.86	43.16	22.61	12.33	7.08	6.49	1.04	0.67
200	200	125.47	119.31	124.88	87.81	39.8	86.69	44.96	23.38	12.80	7.15	5.93	1.05	0.42
260	140	104.64	87.65	106.24	72.44	25.9	68.85	35.93	18.85	10.41	5.83	5.30	1.04	0.41
320	80	82.53	64.89	93.16	57.32	13.1	61.46	28.79	15.24	8.52	4.90	4.17	1.06	0.60
380	20	55.38	45.37	99.01	47.33	3.2	44.01	23.79	12.86	7.38	4.54	4.02	1.04	0.98

Table 5: `gaby` Tests, $p = 2^{30} - 35$, $\deg A = \deg B = 200$, inputs have 1373701 terms

		Maple			MagmaR		MagmaM		MGCD, #CPUs					POLYDAG	
$\deg(G)$	$\deg(\bar{A})$	Mult	GCD	Div	Mult	GCD	Mult	GCD	1	2	4	8	16	In	Out
10	190	2.21	41.22	3.66	61.20	9.60	31.16	1.83	5.82	3.05	1.64	0.94	0.62	0.17	0.17
40	160	15.12	120.80	19.75	301.71	20.44	1100.0	4.83	7.03	3.64	1.92	1.06	0.67	0.17	0.10
70	130	15.21	220.25	41.95	350.77	46.81	2971.7	9.72	7.89	4.05	2.12	1.14	0.71	0.17	0.06
100	100	15.00	234.40	65.43	315.78	102.00	4454.7	16.31	8.25	4.24	2.19	1.18	0.71	0.17	0.04
130	70	15.34	217.53	47.08	336.02	3118.2	3129.7	5544.3	6.53	3.36	1.75	0.95	0.61	0.17	0.04
160	40	15.15	118.54	17.13	282.25	3159.4	1037.0	5249.6	5.12	2.65	1.40	0.78	0.48	0.17	0.05
190	10	2.23	33.42	5.38	41.18	2050.2	33.75	3578.1	4.18	2.20	1.18	0.67	0.43	0.17	0.08

Table 6: `gaby` Tests, $p = 2^{62} - 57$, $\deg A = \deg B = 200$, inputs have 1373701 terms

		Maple			MagmaR		MagmaM		MGCD, #CPUs					POLYDAG	
$\deg(G)$	$\deg(\bar{A})$	Mult	GCD	Div	Mult	GCD	Mult	GCD	1	2	4	8	16	In	Out
10	190	2.22	70.98	3.51	77.22	33.34	62.65	10.03	6.35	3.34	1.83	1.06	0.71	0.17	0.30
40	160	25.65	267.16	20.24	920.48	159.71	2436.5	39.64	7.75	4.01	2.13	1.18	0.75	0.17	0.18
70	130	25.62	439.80	42.44	1624.6	462.09	6567.4	85.97	8.72	4.48	2.35	1.27	0.75	0.17	0.11
100	100	25.43	453.27	64.85	1526.2	900.65	10425	85.97	9.11	4.67	2.43	1.32	0.79	0.17	0.07
130	70	25.69	436.11	50.46	1559.2	14254.	7096.7	11050.	7.11	3.66	1.92	1.04	0.62	0.17	0.06
160	40	25.44	282.04	17.18	934.45	7084.3	2393.0	6608.8	5.63	2.89	1.52	0.83	0.51	0.17	0.09
190	10	2.23	77.28	4.29	90.30	2229.8	72.63	2075.2	4.69	2.41	1.29	0.74	0.47	0.17	0.15

[6] P. Emeliyanenko. High-performance Polynomial GCD Computations on Graphics Processors. *Proceedings of HPCS 2011*, IEEE, pp. 215–224, 2011.

[7] S. A. Haque and M. Moreno Maza. Plain Polynomial Arithmetic on GPU. *J. Physics Conference Series* **385** 2012.

[8] M. B. Monagan and R. Pearce. Parallel Sparse Polynomial Multiplication using Heaps. *Proceedings of ISSAC '09*, pp. 263-269, ACM Press, 2009.

[9] M. B. Monagan and R. Pearce. Sparse Polynomial Division Using a Heap. *J. Symb. Cmpt.* **46** (7) 807–822, 2011.

[10] M. O. Rayes and P. S. Wang. Parallelization of the Sparse Modular GCD Algorithm for Multivariate Polynomials on SMP. *Proceeding of ISSAC 1994*, ACM Press, 66–73.

[11] Paul S. Wang. Parallel Polynomial Operations on SMPs: An Overview. *J. Symb. Cmpt.* **21** 397–410, 1996.

[12] R. Zippel. Probabilistic Algorithms for Sparse Polynomials, *Proceedings of EUROSAM '79*, Springer-Verlag LNCS, **2** pp. 216–226, 1979.

[13] R. Zippel. Interpolating Polynomials from their Values. *J. Symb. Cmpt.* **9**, 3 (1990), 375-403.