Optimizing and Parallelizing Brown's Modular GCD Algorithm

Matthew Gibson, Michael Monagan

October 7, 2014

1 Introduction

Consider the multivariate polynomial problem over the integers; that is, $\mathrm{Gcd}(A, B)$ where $A, B \in \mathbb{Z}[x_1, x_2, \ldots x_n]$. We can solve this problem by solving the related Gcd problem in $\mathbb{Z}_p[x_1, x_2, \ldots x_n]$ for several primes p, and then reconstructing the solution in the integers using Chinese Remaindering. The question we address in this paper is how fast can we solve the problem $\mathrm{Gcd}(A, B)$ in $\mathbb{Z}_p[x_1, x_2, \ldots x_n]$ using either 31 or 63 bit primes, and how well can we make use of parallel processing to do so? To this end, we implemented a modular algorithm using evaluations and interpolations, and parallelized it using the CILK framework. Several optimizations for the algorithm were found, and a few parallelization strategies were attempted. Our final implementation preformed significantly better than both maple and magma on most test cases without using multiple processors, and with parallelization we acheive a speedup of a factor of about 11 on 16 processors.

The modular method of solving the GCD problem was first explained by Brown in 1971 [1]. Further, the methods used here are very similar to those used in a 2000 paper by Monagan and Witkopf [2]. A proof of Brown's algorithm will be included in this paper to aid in the explenations of the optimizations developed. However, one may wish to reference either the 1971 or 2000 papers for an alternative and more detailed proof.

The MGCD (modular GCD) algorithm is going to compute the GCD G and the corresponding co-factors $\bar{A}, \bar{B}$ of the inputs $A, B \in \mathbb{Z}_p[x_1, x_2 \ldots x_n]$. The algorithm is modular and recursive. We will use an evaluation homomorphism to remove the variable x_1, solve the problem in $\mathbb{Z}_p[x_2 \ldots x_n]$ several times recursively, and use the results to reconstruct the solution to the original n-variable problem.

Definitions

If $\mathbb{Z}$ is the ring of integers and p is a prime in $\mathbb{Z}$, then $p\mathbb{Z}$ is an ideal and $\mathbb{Z}_{/p\mathbb{Z}} = \mathbb{Z}_p$ is a field of charecteristic p. Then $\mathbb{Z}_p[x_1 \ldots x_n]$ is the ring of multivariate polynomials in the variables $x_1, x_2 \ldots x_n$ with coefficients in $\mathbb{Z}_p$.

Definition. The terms of $A \in \mathbb{Z}_p[x_1, x_2 \ldots x_n]$ have a monomial part of the form $x_1^{e_1} x_2^{e_2} \ldots x_n^{e_n}$. A lexicographical ordering with $x_1 < x_2 < \cdots < x_n$ orders the monomials according to e_n, and in case of a tie according to e_{n-1}, and in case of a tie continueing to e_1. If all the exponents are the same, then the terms are equal lexicographically. A polynomial is in lexicographical order if its terms are ordered lexicographically according to their monomials. For example, if $x > y > z$ then the polynomial $A = 3x^2yz^2 + xy^2z - xz^2 + 2y + z^2$ is in lexicographical order.

The univariate polynomial ring $\mathbb{Z}_p[x]$ is a Euclidean Domain, so that given any $a, b \in \mathbb{Z}_p[x]$, $b \neq 0$, we can find a corresponding $q, r \in \mathbb{Z}_p[x]$ such that $a = bq + r$ where $r = 0$ or $\deg(r) < \deg(b)$. As a result, we can use the Euclidean Algorithm to obtain the GCD of polynomials $A, B \in \mathbb{Z}_p[x]$ through a sequence of univariate divisions in $\mathbb{Z}_p[x]$. The multivariate polynomial domain $\mathbb{Z}_p[x_1, x_2 \ldots x_n]$ is not a Euclidean Domain whenever $n > 1$, but it is a Unique Factorization Domain (UFD).

Definition. Let D be any UFD. Then for $A, B \in D$, we say that $G \in D$ is the Greatest Common Divisor of A and B if and only if

1. $G \mid A$ and $G \mid B$
2. for any $f \in D$, $f \mid A$ and $f \mid B$ implies $f \mid G$

For $A, B \in \mathbb{Z}_p[x_1, x_2 \ldots x_n]$ we will define $G = \operatorname{Gcd}(A, B)$ to be the unique monic polynomial GCD for terms in lexicographical order. We will also define the co-factors $\bar{A}, \bar{B}$ as satisfying $G\bar{A} = A$ and $G\bar{B} = B$.

Definition. In this proof, we will see $A \in \mathbb{Z}_p[x_1][x_2 \ldots x_n]$ as a multivariate polynomial in the variables $x_2 \ldots x_n$ with coefficients in the ring $\mathbb{Z}_p[x_1]$. Then the leading term of A is the term with highest lexicographical order in $x_2 \ldots x_n$, composed of the leading monomial $\operatorname{lm}(A)$ in $x_2 \ldots x_n$ and the leading coefficient $\operatorname{lc}(A)$ in $\mathbb{Z}_p[x_1]$.

Definition. Considering $A \in \mathbb{Z}_p[x_1][x_2 \ldots x_n]$, we say that the content of A is the monic GCD of the $\mathbb{Z}_p[x_1]$ coefficients, and we write this as $\operatorname{cont}(A)$. If the content of a polynomial is 1, then the polynomial is primitive. We define the primitive part of A, $\operatorname{pp}(A)$, as satisfying $\operatorname{pp}(A)\operatorname{cont}(A) = A$. It could be proved that the polynomial $A \in \mathbb{Z}_p[x_1][x_2 \ldots x_n]$ is primitive in $x_2 \ldots x_n$ if and only if, for any factor $f \mid A$, either f is a unit or $\deg_{x_i}(f) > 0$ for some $2 \leq i \leq n$.

Definition. Consider $A \in \mathbb{Z}_p[x_1][x_2 \ldots x_n]$ with coefficients in the ring $\mathbb{Z}_p[x_1]$. Then let ϕ_i be the homomorphism corresponding to the evaluation $x_1 = \alpha_i$ for some point $\alpha_i \in \mathbb{Z}_p$. Then $\phi_i(A)$ maps $\mathbb{Z}_p[x_1][x_2 \ldots x_n] \to \mathbb{Z}_p[x_2 \ldots x_n]$. It will be helpful later on to use the fact that this homomorphism is equivalent to the natural homomorphism which maps A to its congruence class mod $x_1 - \alpha_i$.

2 Modular Algorithm Proof

2.1 An example of the complicating issues

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

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

We wish to evaluate $y = \alpha_i$ for multiple points, and use the results to reconstruct the GCD. The degree in y of the GCD must be no more than 2, so we only need to evaluate and interpolate at 3 points. Picking $y = 1, 2, 3$, we get the following results:

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 of A and B is $G = x^2(y + 3) + x$. Notice that we could not simply interpolate the GCD's of the images of the inputs, as two problems arise. The first problem is that we will not be able to reconstruct the $y + 3$ coefficient of x^2 in the GCD since the image GCD's are all monic. This is because the GCD is unique only up to a unit, and is called the leading coefficient problem. The second problem, the problem of unlucky evaluation points, is that the image GCD for $\alpha_i = 3$ has the wrong degree in x since the co-factors of the inputs are not relatively prime under ϕ_3.

2.2 How to solve leading term problem

In general, consider the inputs A, B in $\mathbb{Z}_p[x_1][x_2 \ldots x_n]$, and ϕ_i the homomorphism which evaluates x_1 at some point α_i. We have that $\mathrm{Gcd}(\phi_i(A), \phi_i(B)) = \mathrm{Gcd}(\phi_i(G), \phi_i(G))\,\mathrm{Gcd}(\phi_i(\bar{A}), \phi_i(\bar{B}))$, and since the function Gcd gives a monic result, this means:

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

We say that α_i is *unlucky* whenever $\mathrm{Gcd}(\phi_i(\bar{A}), \phi_i(\bar{B})) \neq 1$.

To solve the problem of the leading coefficients, we will assume that the inputs $A, B \in \mathbb{Z}_p[x_1][x_2 \ldots x_n]$ have been made primitive with respect to the variables $x_2 \ldots x_n$ at the beginning of the algorithm. This can be done by computing the content of each input in $\mathbb{Z}_p[x_1]$ and dividing this from its coefficients. Further, by Gauss' Lemma, we know that $\mathrm{Gcd}(A, B) = \mathrm{Gcd}(\mathrm{pp}(A), \mathrm{pp}(B))\,\mathrm{Gcd}(\mathrm{cont}(A), \mathrm{cont}(B))$ so we can compute the GCD and cofactors of the content and primitive parts seperately and recombine them at the end of the algorithm.

Next, we compute $\gamma = \mathrm{Gcd}(\mathrm{lc}(A), \mathrm{lc}(B))$, where $\mathrm{lc}(A)$ and $\mathrm{lc}(B)$ are in $\mathbb{Z}_p[x_1]$. For the true GCD G we know that $\mathrm{lc}(G) \mid \mathrm{lc}(A)$ and $\mathrm{lc}(G) \mid \mathrm{lc}(B)$, and so it must be that $\mathrm{lc}(G) \mid \gamma$ and $\gamma = \mathrm{lc}(G)\Delta$ 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 $\phi_i(\gamma) = \phi_i(\mathrm{lc}(G))\phi_i(\Delta)$. Since $\phi_i(\gamma) \neq 0$ we know that $\phi_i(\mathrm{lc}(G)) = \mathrm{lc}(\phi_i(G))$ and so:

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

If we knew that no evaluation points we use are unlucky, then after computing $\deg_{x_1}(\Delta) + \max(\deg_{x_1}(A), \deg_{x_1}(B))$ image GCD's, we will have enough information to interpolate the polynomial ΔG, where $\deg_{x_1}(\Delta)$ is bounded by $\deg_{x_1}(\gamma)$.[1] Since the inputs were assumed to be primitive with respect to $x_2 \ldots x_n$, it follows that G is also primitive, and so it can be recovered as $G = \mathrm{pp}(\Delta G)$.

2.3 How to detect most unlucky alphas

The second problem is that of detecting unlucky evaluation points. Recall that A, B, G are in $\mathbb{Z}_p[x_1][x_2 \ldots x_n]$ and that $\mathrm{lm}(A)$ is the monomial in $x_2 \ldots x_n$ of the leading term. We know that

[1]At the end of section 2.5 we will show that the final algorithm needs $\deg_{x_1}(\gamma) + \max(\deg_{x_1}(A), \deg_{x_1}(B)) + 1$ images.

$\mathrm{lm}(\phi_i(\Delta)) = 1$ since Δ is in $\mathbb{Z}_p[x_1]$. Then if we let $g_i^* = \phi_i(\gamma)\,\mathrm{Gcd}(\phi_i(A), \phi_i(B))$ for each α_i, by (2) we see that:

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

Then we can see that $\mathrm{lm}(g_i^*) > \mathrm{lm}(G)$ if and only if α_i is unlucky; otherwise, $\mathrm{lm}(g_i^*) = \mathrm{lm}(G)$. Using this fact, we can rule out unlucky evaluation points with a high probability using the following method. For each of a sequence of points α_i, ensure that $\phi_i(\gamma) \neq 0$. Then compute g_i^* and do the following:[2]

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 as unlucky.

Once we have computed an image g_i^* for $\deg_{x_1}(\gamma) + \max(\deg_{x_1}(A), \deg_{x_1}(B))$ different points and discarded unlucky points using the above method, we will have that either all of the evaluation points used are unlucky or none of them are. If we could ensure that not all of the points chosen is unlucky, then we could interpolate ΔG as described above.

2.4 Divisibility Theorem

While the probability of all α_i being unlucky is often very small, this possibility is ruled out by Brown by checking if the interpolated GCD divides the inputs. The basis of this idea can be seen in the following theorem, in which E can be seen as an interpolated GCD which we are testing for correctness.

Theorem 1. *Let E, G be in $\mathbb{Z}_p[x_1][x_2 \ldots x_n]$, $G \neq 0$ and G primitive with respect to $x_2 \ldots x_n$. Let ϕ be an evaluation homomorphism evaluating $x_1 = \alpha$ for some α in $\mathbb{Z}_p$ such that the leading term of 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 $\mathbb{Z}_p[x_1][x_2 \ldots x_n]$ is the monomial in $x_2 \ldots x_n$ of the leading term. Since $E \mid G$, we know that $G = KE$ for some $K \in \mathbb{Z}_p[x_1][x_2 \ldots x_n]$. 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]The final MGCD implementation comapres the total degree of the leading terms rather than comparing monomials. When $\phi_i(\gamma) \neq 0$ these tests are equivalent since the leading term doesn't vanish under the evaluation.

2.5 MGCD algorithm

Assuming $A, B \in \mathbb{Z}_p[x_1][x_2 \ldots x_n]$ have been made primitive with respect to $x_2 \ldots x_n$, we wish to compute $\operatorname{Gcd}(A, B)$. The MGCD algorithm will also give the cofactors $\bar{A}$ and $\bar{B}$. At the beginning of the algorithm, we compute $\gamma = \operatorname{Gcd}(\operatorname{lc}(A), \operatorname{lc}(B))$ and choose a bound $bnd = \min(\deg(A), \deg(B)) + \deg(\gamma) + 1$. Then we begin choosing evaluation points α_i. We discard any point for which $\phi_i(\gamma) = 0$, and for the others we compute $g_i^* = \phi_i(\gamma) \operatorname{Gcd}(\phi_i(A), \phi_i(B))$. We use the leading monomial comparison from section 2.3 to detect and rule out as many unlucky α_i as possible.

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

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

If A and B are univariate, then we can acquire $\bar{a}_i^*$ and $\bar{b}_i^*$ above using univariate division. Otherwise, they will be computed in the recursive call to the MGCD algorithm. 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 for the first k points α_i:

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

Further, since $(x_1 - \alpha_i)$ and $(x_1 - \alpha_j)$ are relatively prime for $i \neq j$, we can define $M_k = (x_1 - \alpha_1)(x_1 - \alpha_2) \ldots (x_1 - \alpha_k)$ and get from (4) 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 likewise $M_k \mid \gamma B - G_k^* \bar{B}_k^*$. 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 $x_2 \ldots x_n$, we know further that $\operatorname{pp}(G_k^*) \mid G$. 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 $\phi_i(G) \mid \phi_i(\operatorname{pp}(G_k^*))$. Then by Theorem 1 we know that $G \mid \operatorname{pp}(G_k^*)$ and so $\operatorname{pp}(G_k^*)$ is equal to G up to a unit. Further, $\operatorname{pp}(G_k^*)$ will be monic. The corresponding primitive co-factors can be found as $\bar{A} = \operatorname{pp}(\bar{A}_k^*)$ and $\bar{B} = \operatorname{pp}(\bar{B}_k^*)$.

2.6 Psuedocode for Non-optimized MGCD

The algorithm MGCD takes as inputs n-variable polynomials $A, B \in \mathbb{Z}_p[x_1][x_2]\dots[x_n]$, where $n \geq 2$. The inputs are stored computationally in a recursive dense representation, which takes the form of a tree with n levels and where the leaves of the tree are polynomials in $\mathbb{Z}_p[x_1]$. Calls in the algorithm to GCD on univariate inputs will use the Euclidean Algorithm. The output of the MGCD algorithm is $[G, \bar{A}, \bar{B}]$, the GCD and two co-factors.

```
begin:
   1. Determine the content of the inputs across the $\mathbb{Z}_p[x_1]$ leafs as $cA := \mathrm{cont}(A)$, $cB := \mathrm{cont}(B)$.
   2. Compute the content GCD and co-factors as $cG := \mathrm{GCD}(cA, cB)$, $c\bar{A} := cA/cG$ and $c\bar{B} :=$
      $cB/cG$
   3. Set $A := A/cA$, $B := B/cB$ to make the inputs primitive.
   4. Compute $\gamma := \mathrm{GCD}(\mathrm{lc}(A), \mathrm{lc}(B))$
   5. Set $bnd = \deg(\gamma) + \max(\deg_{x_1}(A), \deg_{x_1}(B)) + 1$
   6. Set $G^*, A^*, B^* := 0$ to clear the interpolants, and set $e := \infty$, $cnt := 0$
Loop:
   7. $\alpha :=$ a random element in $\mathbb{Z}_p$ such that $\phi(\gamma) \neq 0$
   8. Get $\phi(A)$, $\phi(B)$ by evaluating the leaves of $A$ and $B$
   9. If $n > 2$ then compute $[g^*, \bar{a}^*, \bar{b}^*] := \mathrm{GCD}(\phi(A), \phi(B))$ by a recursive call to MGCD with n-1
      variables. Otherwise, use the Euclidean Algorithm to get $g^*$, and do univariate division to
      get $\bar{a}^*$ and $\bar{b}^*$.
  10. Get the total degree of the leading term, $d := \mathrm{ldeg}(g^*)$
      If $d = 0$ then set $G^* := 1, cA^* := A, cB^* := B$ and goto End, since $A$ and $B$ must be relatively
      prime.
      If $d > e$ then goto Loop, since $\alpha$ must be unlucky
      If $d < e$ then set $G^*, A^*, B^* := 0$, $e := d$, $cnt := 0$ as all previous evaluations were unlucky
  11. Multiply $g^* := \phi(\gamma) \times g^*$
  12. Extend $G^*, \bar{A}^*, \bar{B}^*$ by interpolating into them the new images $g^*, \bar{a}^*$ and $\bar{b}^*$ respectively.
  13. Set $cnt := cnt + 1$
  14. If $cnt < bnd$ 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 $G^*, A^*, B^* := 0$, $e := \infty$, $cnt := 0$ 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 of the Algorithm

3.1 Division in the Images

In the algorithm presented above, Theorem 1 proves the correctness of an interpolated GCD by testing divisibility. While the bound $\deg_{x_1}(\gamma) + \max(\deg_{x_1}(A), \deg_{x_1}(B)) + 1$ is needed to ensure that the interpolated G_k^* divides the inputs A and B, the actual degree of G_k^* in the variable x_1 could be much smaller then this bound. This suggests that if we could either find a smaller bound for the degree of G_k^* in x_1 or detect when it is complete in the iterative interpolation, we could then verify its correctness only by checking if it divides the inputs.

Monagan and Wittkopf used this method in [2] by finding a reasonably tight degree bound for G_k^* in x_1 and using it to only compute enough image GCD's for interpolation. They then used division to check divisibility and to acquire $\bar{A}_k^*$ and $\bar{B}_k^*$. However, division can be slow if a classical algorithm is used and neither the divisor nor the quotient is small. An alternative optimization will be presented as follows.

We will consider the specific case where the inputs A and B are in two variables, x_1 and x_2. In this case, the image GCD's are computed using the Euclidean algorithm, and the corresponding co-factors are acquired through univariate division. If, after k images, we notice that one of the interpolants G_k^*, $\bar{A}_k^*$ or $\bar{B}_k^*$ doesn't change with the addition of the next image, we set a flag to indicate an assumption that it is complete. Then, for each additional image, we do one of the following.

1. If the flagged interpolant is G_k^*, we evaluate $g_i^* = \phi_i(G_k^*)$ as well as $\phi_i(\gamma)\phi_i(A)$ and $\phi_i(\gamma)\phi_i(B)$ in $\mathbb{Z}_p[x_2]$. Then we use univariate division to compute $\bar{a}_i^*$ and $\bar{b}_i^*$ according to (3), and we interpolate $\bar{A}_i^*$ and $\bar{B}_i^*$ as before. If the divisions ever have a non-zero remainder, than we know that $G_k^* \nmid G$ so that it is either built from all unlucky α's, or it's interpolation was not actually complete.

2. If the flagged interpolant is $\bar{B}_k^*$ (or symmetrically $\bar{A}_k^*$), then we evaluate $\bar{b}_i^* = \phi_i(\bar{B}_k^*)$, and we use univariate polynomial division to acquire first g_i^* and then $\bar{a}_i^*$ to satisfy (3). As before, we interpolate G_i^* and $\bar{A}_i^*$ with the new image. If the univariate divisions ever have a non-zero remainder, then we again know that either our evaluation points were unlucky or the interpolant was not actually complete.

Once we have preformed this image division for the rest of the evaluation points up to the bound *bnd*, we use the same degree check as before to verify the correctness of the GCD and co-factors.

This optimization effectively replaces at least half of the univariate GCD computations with univariate polynomial evaluations. 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 $\bar{b}_i^*$. With the optimization, we can avoid the GCD computation and instead evaluate one of the stabilized interpolants with degree in x_1 the least degree in x_1 of G, $\bar{A}$ and $\bar{B}$. For inputs A, B of degree n and GCD G of degree g, the cost of each univariate GCD is in the order of $n^2 - g^2$. Using the optimization, we instead do an evaluation, each of which has a cost in the order of $\min(g, n-g)$.

Figure 1 compares different strategies for verifying divisibility for a range of inputs. The plot is for inputs A, B of degree 600 in both x_1 and x_2, where the degree of the GCD G varies between 60 sampled degree sizes from 0 to 600. The plot compares the non-optimized modular GCD algorithm (Brown's Algorithm) against the early G_k^* and $\bar{B}_k^*$ stabilization described above, and also shows a version which uses classical division (written in C) and the results of using Maple 18's built-in

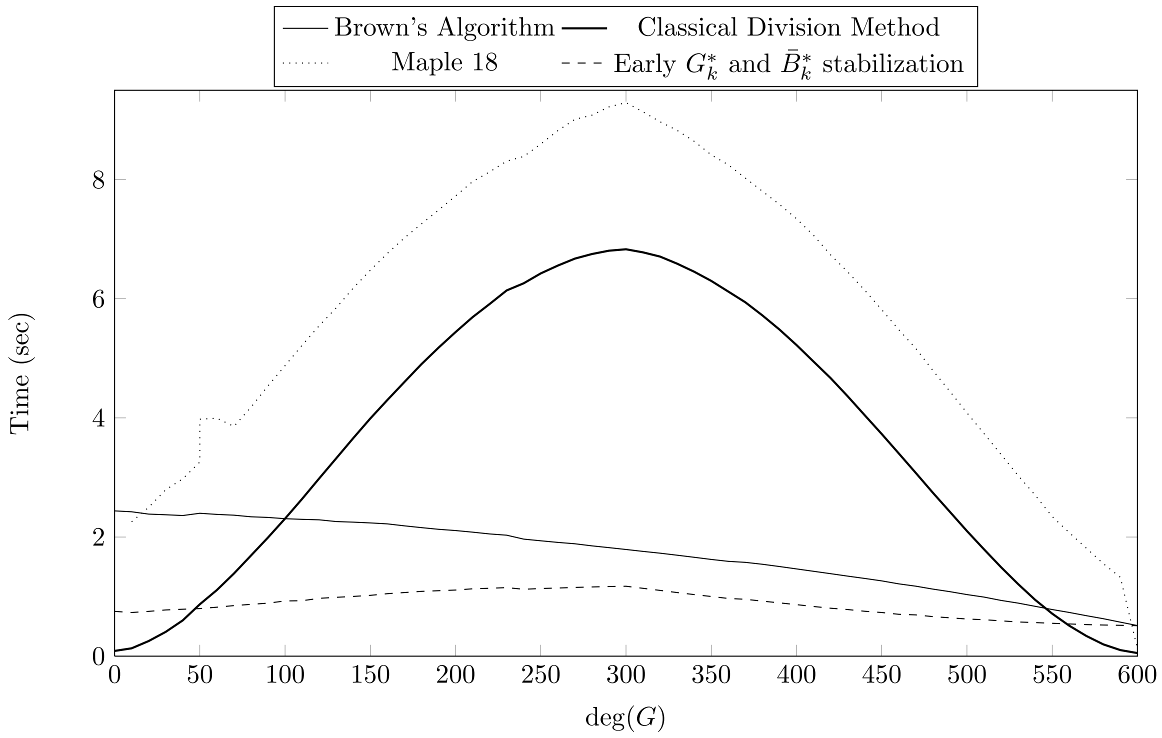

Figure 1: Image Division Optimizations

GCD mod p method. We can see that the optimization gives the most benefit compared to the non-optimized algorithm when deg(G) is small. The classical division method preforms better when deg(G) is either very small or very large.

3.2 Univariate GCD Streamlining

Consider a bivariate GCD problem in which $A, B \in \mathbb{Z}_p[x][y]$ have degree d in both x and y. Then when the degree of the GCD is mid-sized, we can divide the total MGCD cost into a number of parts, each of which are cubic in d:

$$C \in \overbrace{O(d^3)}^{\text{Content Computation}} + \overbrace{O(d^3)}^{\text{Content Division}} + \overbrace{O(d^3)}^{\text{Evaluation}} + \overbrace{O(d^3)}^{\text{Univariate GCD}} + \overbrace{O(d^3)}^{\text{Interpolation}}$$

In the bivariate problem, a significant portion of time is spent on the univariate GCD problem. This means that any optimizations for the univariate problem will immediately translate into preformance gains for the multivariate problem.

Univariate GCD's are computed using the Euclidean Algorithm, where the coefficients of the univariate inputs come from evaluations of multivariate polynomials at random points. As a result, it will often be the case that each step of the Euclidean Algorithm we will be dividing two polynomials, $R0$ of degree n and $R1$ of degree m, such that $n = m + 1$. We can write this as:

$$R0 = a_0 + a_1x + \cdots + a_{n-1}x^{n-1} + a_nx^n$$
$$R1 = b_0 + b_1x + \cdots + b_mx^m$$

For an optimization, we streamlined the computation of the next remainder $R2$ by expressing the quotient directly as $(c + \frac{a_n}{b_m}x)$ where $c = a_{n-1} - \frac{a_n}{b_m}b_{m-1}$. Then we can find the next remainder as:

$$R2 = R0 - (c + \frac{a_n}{b_m}x)R1$$

where $\deg(R2) < \deg(R1)$. This remainder can be computed with one modular reduction per coefficient of $R0$. The results are shows in figure 2 which tests the univariate GCD for inputs of degree 100 whose GCD degree varies from 0 to 100. The preformance gain is about 64.5%.

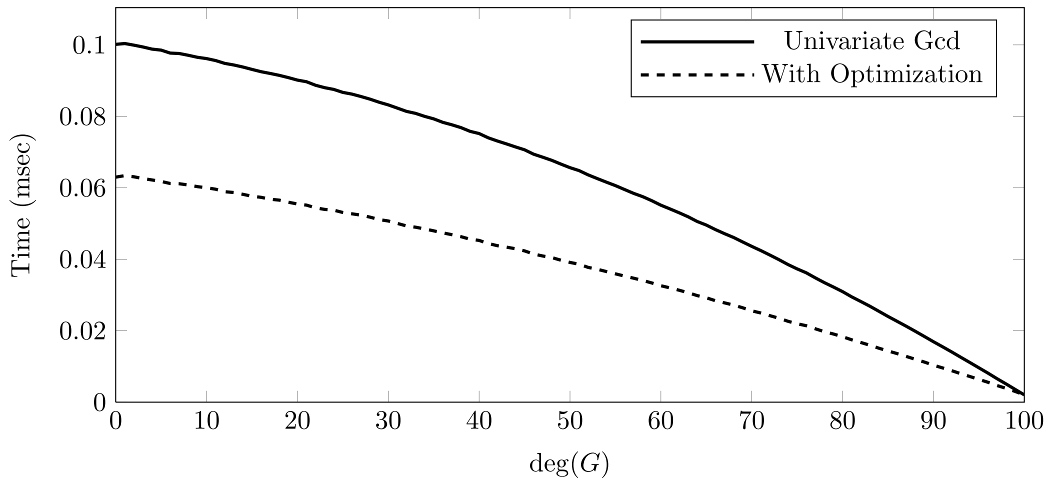

Figure 2: Univariate GCD Optimization

3.3 Partial FFT

Along with the univariate GCD, the other high contribution to the computational time of the MGCD algorithm is multivariate polynomial evaluations and interpolations. In this algorithm, all of these evaluations and interpolations are preformed in the $\mathbb{Z}_p[x_1]$ leaves of a recursively defined polynomial structure.

The cost of evaluating or interpolating a univariate polynomial A at a new point $x = \alpha_i$ is $n = \deg(A)$. An initial optimization can be found by the fact that in a field $\mathbb{Z}_p$ with $p > 2$, for any point α there exists a corresponding point $-\alpha$ such that $\alpha^2 = (-\alpha)^2$. We can use this fact to reduce the cost of evaluation and interpolation almost in half by separating A into coefficients with even and odd powers of x and evaluating each term in A once for each $(\alpha, -\alpha)$ pair. Then the two distinct evaluations of A can be found using an addition and subtraction.

This method makes use of some of the optimizations of the Fast Fourier Transform for evaluation and interpolation. Because of the nature of the modular GCD algorithm, it would be difficult to use the full Fast Fourier Transform. Since we would have to evaluate and interpolate slightly more points than the bound when it is not a power of two, and the high cost of the image GCD's quickly overcomes the performance gain in evaluation and interpolation. However, it may be beneficial to generalize the partial FFT strategy to evaluate more than two points at a time, which is done as follows.

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

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

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

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

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

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

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

The strategy for interpolation is similar. Let $M_k = \prod_{i=0}^{k} (x^j - \alpha_i^j)$. We interpolate using a modification of Newton's method. The kth interpolant of G satisfies $G_k \equiv G \mod M_k$ and is of the form:

$$G_k = g_0 + g_1 M_0 + \cdots + g_k M_{k-1}$$

where each g_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 Fast Fourier Transform 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 $G_{k+1} = G_k + g_{k+1} M_k$, and further since $M_{k+1} = M_k (x^j - \alpha_{k+1}^j)$ we get the next coefficient polynomial as:

$$g_{k+1} \equiv \frac{G^* - G_k}{M_k} \mod (x^j - \alpha_{k+1}^j)$$

according to the normal form of Newton interpolation.

The cost of interpolating a batch of size j involved first using the IFFT to interpolate a polynomial of degree $j-1$, with a cost in $j \log j$. Next we need to evaluate the basis functions $\{1, M_0, M_1 \dots\}$ 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 can be expressed as:

$$C = \frac{\deg(A)}{j} + \log j \tag{7}$$

where j is a power of 2. We can see that as we increase j from 1, the linear part of this cost diminishes quickly, while the logarithmic part grows slowly. While it is hard to use the FFT in the GCD algorithm due to extra cost associated with computing extra image GCD's, the partial FFT

j	Time: Eval and Interp	Time: Image GCDs in $\mathbb{Z}_p[x2,x3]$	Time: Total
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: $A, B \in \mathbb{Z}_p[x_1][x_2, x_3]$

j	Time: Eval and Interp	Time: Image GCDs in $\mathbb{Z}_p[x3]$	Time: Total
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: $A, B \in \mathbb{Z}_p[x_2][x_3]$

Table 1: Partial FFT Optimizations

with a small choice of j can still get some of the preformance gains in evaluation and interpolation while only computing a few extra image GCD's.

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 eval and interp of the variable x_1 only. Table 1b shows the results of blocks of size j for the subproblem in $\mathbb{Z}_p[x_2, x_3]$. Using $j = 8$ for the variable x_2 in the trivariate problem gives a preformance gain of about 20 %, and using $j = 8$ for the variable x_1 in the bivariate problem gives a gain of 41%. The final implementation in this paper selects j according to a fraction of the input degrees; however the results indicate most of the benefits of this method are realized when j is still small, and alternative strategy would have been to simply use $j = 2$ or $j = 4$ whenever possible.

3.4 Dividing Content from the Images

For the multivariate problem, we can express each input as a set of univariate polynomials in x_1 and monomials M_i in $x_2 \dots x_n$ as follows:

$$A(x_1 \dots x_n) = \sum_{i=0}^{t} (a_{i0} + a_{i1}x_1 + \cdots + a_{id}x_1^d)M^i$$

where d is $\deg_{x_1}(A)$. Then we can see that computing the content of A involves t univariate GCD computations with inputs of the order d, for a total cost of $O(td^2)$. Further, dividing the content from A will also cost $O(td^2)$. We can reduce this cost somewhat by not dividing the content from the inputs. Instead, we evaluate the content along with the evaluation of A at each α_i, and divide it out of $\phi_i(A)$ as a scalar. The number of terms in $\phi_i(A)$ is of the order t and the algorithm bound is of the order d, so the cost of doing this is $O(td)$.

3.5 Accumulation Buffer Computational Strategy

There are several places in the MGCD algorithm where we need to compute a sum of products. For example, consider univariate evaluation of a polynomial without using the FFT method described above. Given a list of precomputed powers of x in an array X, and a corresponding list of coefficients in an array A, we need to multiply the coefficients with their corresponding powers of x and add the results. All of this needs to be done in the field $\mathbb{Z}_p$ for some machine-word sized prime p. The naive method is to reduce each product modulo p in order to ensure that integer overflow is not occuring, as seen in the following C-code snippet:

```
r = A[0];
for(k=1; k <= d;k++) {
    r = r + mulmodinv(A[k],X[k],qp); // r = r + A[k] * X[k] % p
    neg64(&r,qp.p);                  // if(r>0) r = r - p
}
pos64(&r,qp.p);                      // if(r<0) r = r + p
return r;
```

However, the modular reduction in mulmodinv is expensive. To optimize this code, we use a temporary buffer T which is 2 machine-words long. Then we can store the intermediate results and ensure that overflow doesn't occur without computing a remainder. Further, p is limited to one bit less than the wordsize, so that at least two products can be added before we need to check for overflow:

```
c128(T, A[0]);              // T = A[0]
for(k=1; k < d;) {
    fadd128(T, A[k], X[k]); // T = T + A[k] * X[k]
    neg128(T, qp.p);        // if(T>0) T = T - 2^64 * p
    k++;
    fadd128(T, A[k], X[k]); // T = T + A[k] * X[k]
    k++;
}
if(k == d) { // one more
    fadd128(T, A[k], X[k]); // T = T + A[k] * X[k]
}
// reduce accumulator mod p
pos128( T, qp.p );          // if(T<0) T = T + 2^64 * p
return mod( T, qp );        // T % p
```

The results of this optimization can be seen in table 2, in $\mathbb{Z}_p[x]$ with $p = 1924145348627$. The results show a 228% preformance gain. This optimization is used in univariate polynomial evaluation, interpolationa and division, and in content multiplication.

deg(A)	Original	Optimized
20	0.10	0.03
50	0.22	0.07
100	0.44	0.14
500	2.21	0.66
1000	4.48	1.30

Table 2: Timings in milliseconds for using evaluation of A

4 Parallelization

The modular GCD algorithm is well suited to parallel computation. This was implemented using the CILK framework which uses a work stealing algorithm and a fixed number of worker threads.[3] 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 implemention only uses parallelization for 3 or more variables, providing a fairly large bivariate problem at the base of parallelization. In the parallel version, we don't follow the iterative method as described by the algorithm proof in section 2. Instead, we preform all of the *bnd* evaluations and image GCD's at the same time 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 preformed 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 recollects the parallel threads, and interpolation begins, using another parallelization strategy. If we think of the interpolation image $\phi_i(G)$ as being in $\mathbb{Z}_p[x_2 \dots x_{n-1}][x_n]$, then $\deg_{x_n}(\phi_i(G))$ jobs are spawned on the coefficients in x_n, which are interpolated in $\mathbb{Z}_p[x_1 \dots x_{n-1}]$. Further, the GCD and two cofactors are interpolated in parallel.

5 Preformance Timings

Timings were obtained on the two Intel Xeon servers `gaby` and `jude` in the CECM. Both servers are running CentOS Unix. The `gaby` server has two Intel E5-2660 cpus, each with 8 cores running at 2.2 GHz (3.0 GHz in turbo). The `jude` server has two Intel E5 2680 v2 cpus, each with 10 cores running at 2.8 GHz (3.6 GHz turbo).

Tables 3: Optimization Testing

The first set of tests in Tables 3 investigate the effectiveness of the optimizations in sections 3.1, 3.2 and 3.3. These tests were run on the `jude` machine, and are on trivariate inputs A and B in which the GCD and cofactors are built to be dense in maximum degree[4] with random coefficients in the field $\mathbb{Z}_p$. In each set of tests, the degree of G varies while the degrees of the inputs A and B are constant. There are 4 of these sets of tests. Seperate sets of tests were run for input degree 200 and 400, and for each of these seperate tests were run for the primes $p = 2^{30} - 35$ and $p = 2^{62} - 57$.

For each generated input, tests were run on a fully optimized non-parallel version of code (column No CILK). Then three seperate tests were run with each of the optimizations turned off (columns 3.1,3.2,3.3). Further tests were used to find the percentage of No CILK time was spent on the univeriate 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 for this conversion were taken seperately, and are

[3]typically set to the number of cpu threads on the machine

[4]terms of total degree $\leq d$ non-zero

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 3

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 $\deg(G) = \deg(\bar{A}) = \deg(\bar{B})$, which is when the problem requires the most univariate GCD's. This is because optimization 3.1 reduces the number of needed univariate GCD's to approximately $\min(\deg(G), \deg(\bar{A}), \deg(\bar{B}))$. The result is that optimization 3.2 saves about $\frac{1}{3}$.

3. The benefit of doing evaluation and interpolation in blocks using the partial FFT method was more than expected. This is partly because optimization 3.1 replaces univariate GCD's with evaluations. The gain is a over a factor or 3 when $\deg(\bar{A})$ and $\deg(\bar{B}) \ll \deg(G)$. This is because, as $\deg(G)$ increases, the cost of univariate GCD's decreases so that more time is spent on evaluation and interpolation.

4. A comparison of the No CILK results to the parallel version of the code with 1 cpu indicates that the overhead of the CILK infastructure is minimal for this problem.

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

6. The parallelization results indicate that, not including the conversion time, we get a speedup of about a factor of 12 on 16 cores, or about 13 times on 20 cores.

7. There is about a 10 % increase in timings going from $p = 2^{30} - 35$ to $p = 2^{2^62} - 57$ for the 200-degree problem, and about a 50 % increase for the 400-degree problem.

Tables 4: Maple and Magma Comparison

The second set of tests in Tables 4 compares the parallel MGCD algorithm to the modular GCD algorithms in Maple and Magma. These tests were run on the `gaby` machine. Inputs were generated in the same manner as for those in Tables 3, except that in this case the maximum degrees of the inputs were 100 and 200.

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 cofactors as $\bar{A} = \frac{A}{G}$ and $\bar{B} = \frac{B}{G}$ using a polynomial division. These allow us to compate the relative cost of a GCD with multiplication and division. Note, our MGCD algorithm constructs and returns the cofactors 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 $GF(p)[x][y][z]$, shown below. The columns under MagmaM instead run the tests in the multivariate polynomial structure, $GF(p)[x, y, z]$.

Listing 1: Recursive

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

Listing 2: Multivariate

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

Once again, parallelized results of the MGCD function are provided. These timings are run from a compiled maple extension, and once again the conversion times are not included in the MGCD timings but are listed seperately in the columns POLYDAG:In/Out.

Notes on tables 4

1. Maple uses the Hensel Lifting algorithm to compute $\text{Gcd}(A, B) \mod p$ with 3 or more variables. Maple also has parallelized and optimized multivariate polynomial multiplication and division procedures, which improves the preformance of the Gcd as well which uses these in the Hensel Lifting.

2. Both Maple and Magma preform better when coefficients are less than 2^{31}. Maple's timings roughly double when moving to a larger prime, while magma's timings generally increase more, depending on $\deg(G)$.

3. Magma's GCD preformance using the MagmaM ring structure for $\deg(G)$ small is very good. However, the corresponding multiplication and division preformance is much worse. In many practical uses of the GCD problem, this may defeat the gain of having a faster GCD algorithm. Multiplication and division in MagmaR are better.

4. The time for MGCD on one core is typically 20-50 times faster than Maple's Gcd time, and faster than multiplication and division in Maple. This is parly because we are using a modular Gcd algorithm of complexity $O(n^4)$ instead of $O(n^6)$ for these trivariate problems.

5. The MagmaM Gcd algorithm is faster than the MGCD algorithm in the 100 and 200 degree problems for a small prime only when $\deg(G)$ is small. for the 31-bit prime, when $\deg(G) = \deg(\bar{A}) = \deg(\bar{B})$ the MGCD algorithm is about 2 times faster than the MagmaM algorithm in both the 100- and 200-degree problem. For the 62-bit prime, the MGCD algorithm is about 18 times faster in the 100-degree problem and 9 times faster for 200-degree problem. When $\deg(G) > \deg(\bar{A})$, the MagmaM and MagmaR preformance drops off considerably, with MagmaM timings increasing to 17 and 340 times as much.

6. Using multiple CPUs, if you do not take into account the input and output conversions, then the MGCD algorithm has about a 11.4 times speedup on 16 cpu cores when $\deg(G)$ is midsized. Taking into account the input and output conversions, the average speedup is for the midsized 100-degree problem is 8.5, and for the midsized 200-degree problem is 9.1.

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

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

Table 3a: `jude` Tests, $p = 2^{30} - 35$, inputs have 1373701 terms

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

Table 3b: `jude` Tests, $p = 2^{62} - 57$, inputs have 1373701 terms

		Without Opt					MGCD, #cpus						POLYDAG	
deg(G)	deg($\bar{A}$)	3.2	3.1	3.3	No CILK	% UniGcd	1	2	4	8	16	20	In	Out
20	380	51.30	126.20	115.86	43.51	13.7	43.32	23.19	12.58	7.41	4.76	4.21	1.05	1.13
80	320	81.80	125.57	123.61	57.16	30.4	57.22	30.04	15.93	8.89	5.30	4.50	1.04	0.64
140	260	99.49	117.16	132.67	67.29	39.0	67.04	34.68	18.27	9.97	5.70	4.84	1.04	0.37
200	200	110.23	107.82	132.78	71.46	42.8	71.34	36.98	19.25	10.53	5.89	5.40	1.05	0.23
260	140	76.81	90.10	114.79	55.49	29.7	59.80	28.97	15.17	8.32	4.68	4.18	1.06	0.23
320	80	50.55	68.59	104.17	41.25	15.0	41.50	21.81	11.63	6.48	3.75	3.35	1.05	0.34
380	20	33.05	40.34	97.75	30.23	4.2	30.52	16.46	8.89	5.09	3.21	2.82	1.05	0.56

Table 3c: `jude` Tests, $p = 2^{30} - 35$, inputs have 10827401 terms

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

Table 3d: `jude` Tests, $p = 2^{62} - 57$, inputs have 10827401 terms

		Maple			MagmaR			MagmaM			MGCD, #cpus					POLYDAG	
deg(G)	deg$(\bar{A})$	Mult	GCD	Div	Mult	GCD	Div	Mult	GCD	Div	1	2	4	8	16	In	Out
10	90	0.26	6.01	0.42	6.33	0.91	5.64	3.55	0.27	88.63	0.56	0.30	0.16	0.09	0.06	0.02	0.02
20	80	0.56	9.36	0.72	18.80	1.48	7.55	15.42	0.40	109.75	0.61	0.32	0.17	0.10	0.06	0.02	0.01
30	70	1.56	12.78	1.10	18.44	2.53	6.94	39.08	0.69	234.90	0.65	0.34	0.18	0.10	0.07	0.02	0.01
40	60	1.55	14.90	1.29	19.12	4.31	6.67	56.41	0.97	178.03	0.68	0.35	0.19	0.10	0.06	0.02	0.01
50	50	1.55	15.34	2.05	15.61	6.93	6.22	64.75	1.31	122.72	0.69	0.36	0.19	0.10	0.06	0.02	0.00
60	40	1.56	15.88	3.11	18.21	53.09	6.43	58.38	80.76	112.56	0.59	0.31	0.16	0.09	0.06	0.02	0.00
70	30	1.57	13.08	1.69	18.71	53.32	6.32	40.03	114.85	65.51	0.51	0.27	0.14	0.08	0.05	0.02	0.00
80	20	0.54	9.50	0.83	17.90	52.78	6.97	19.12	73.73	23.88	0.45	0.24	0.13	0.07	0.05	0.02	0.01
90	10	0.26	6.00	0.63	5.06	43.56	5.19	3.90	47.04	4.55	0.40	0.21	0.11	0.07	0.04	0.02	0.01

Table 4a: `gaby` Tests, $p = 2^{30} - 35$, inputs have 176851 terms

		Maple			MagmaR			MagmaM			MGCD, #cpus					POLYDAG	
deg(G)	deg$(\bar{A})$	Mult	GCD	Div	Mult	GCD	Div	Mult	GCD	Div	1	2	4	8	16	In	Out
10	90	0.26	10.68	0.40	9.08	4.50	9.54	8.22	1.63	100.94	0.61	0.33	0.18	0.10	0.07	0.02	0.04
20	80	0.54	18.28	0.72	48.77	11.85	15.35	41.81	3.24	194.31	0.68	0.36	0.19	0.11	0.07	0.02	0.03
30	70	0.87	24.88	1.11	64.96	25.24	16.35	91.40	5.29	405.33	0.74	0.39	0.20	0.11	0.07	0.02	0.02
40	60	2.37	30.12	1.29	70.27	42.77	16.98	209.87	9.33	527.10	0.77	0.40	0.21	0.11	0.07	0.02	0.01
50	50	2.37	30.45	1.97	82.36	72.78	16.43	265.92	14.32	507.11	0.78	0.41	0.21	0.12	0.07	0.02	0.01
60	40	2.38	31.37	2.60	70.65	322.10	16.48	191.68	254.32	405.77	0.66	0.35	0.18	0.10	0.06	0.02	0.01
70	30	0.86	26.48	1.39	64.62	248.58	16.10	95.48	211.09	243.09	0.56	0.29	0.15	0.09	0.05	0.02	0.01
80	20	0.55	19.64	0.84	49.34	127.28	15.15	40.21	85.24	109.01	0.47	0.25	0.13	0.07	0.05	0.02	0.01
90	10	0.55	11.64	0.63	9.83	56.51	10.85	8.23	53.63	24.20	0.42	0.22	0.12	0.07	0.05	0.02	0.02

Table 4b: `gaby` Tests, $p = 2^{62} - 57$, inputs have 176851 terms

		Maple			MagmaR			MagmaM			MGCD, #cpus					POLYDAG	
deg(G)	deg$(\bar{A})$	Mult	GCD	Div	Mult	GCD	Div	Mult	GCD	Div	1	2	4	8	16	In	Out
10	190	2.21	41.22	3.66	61.20	9.60	46.34	31.16	1.83	4607.3	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	151.85	1099.97	4.83	10198	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	131.82	2971.70	9.72	16428	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	123.89	4454.7	16.31	8802.7	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	121.45	3129.7	5544.3	4521.8	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	135.60	1037.0	5249.6	1726.6	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	44.49	33.75	3578.1	49.11	4.18	2.20	1.18	0.67	0.43	0.17	0.08

Table 4c: `gaby` Tests, $p = 2^{30} - 35$, inputs have 1373701 terms

		Maple			MagmaR			MagmaM			MGCD, #cpus					POLYDAG	
deg(G)	deg$(\bar{A})$	Mult	GCD	Div	Mult	GCD	Div	Mult	GCD	Div	1	2	4	8	16	In	Out
10	190	2.22	70.98	3.51	77.22	33.34	79.26	62.65	10.03	4351.3	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	327.56	2436.5	39.64	14666	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	307.40	6567.4	85.97	29550	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	274.31	10425	85.97	27612	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	276.84	7096.7	11050	17270	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	333.45	2393.0	6608.8	5750.2	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	95.77	72.63	2075.2	225.56	4.69	2.41	1.29	0.74	0.47	0.17	0.15

Table 4d: `gaby` Tests, $p = 2^{62} - 57$, inputs have 1373701 terms