Parallelization of The Sparse Modular GCD Algorithm for Multivariate Polynomials on Shared Memory Multiprocessors*

Mohamed Omar Rayes (rayes@mcs.kent.edu)
Department of Mathematics and Computer Science
Kent State University
Kent, Ohio 44242-0001 USA

Paul S. Wang† (pwang@mcs.kent.edu)
Sandia National Laboratories
Livermore, CA 94551-0969 USA

Kenneth Weber (kweber@mcs.kent.edu)
Department of Mathematics and Computer Science
Kent State University
Kent, Ohio 44242-0001 USA

Abstract

Reported are experiences and practical results from parallelizing the modular GCD algorithm for sparse multivariate polynomials. The strategy is to identify key computation steps in the sequential algorithm and implement them in parallel. The two major steps of the sequential algorithm—computing the GCD modulo several primes and applying the Chinese Remainder Algorithm on the integer coefficients—are easily partitioned into independent subtasks. The subtask of computing the GCD modulo one prime can be subdivided further. Several parallel strategies for the multivariate GCD modulo a prime are presented. Actual timings on a Sequent Balance with 26 processors are presented.

1 Introduction

Polynomial Greatest Common Divisor (GCD) is a basic capability in any computer algebra system. Since the early days of symbolic computation, polynomial GCD has been an active area of research. Well-known algorithms include modular GCD [4], reduced PRS (polynomial remainder sequence) [8], subresultant PRS [5], EEZ GCD, and sparse modular (SMGCD) algorithms. The reader is referred to [6] and [13] for a history and a survey of polynomial GCD algorithms.

For multivariate polynomials, the EEZ algorithm is normally very effective and provides a lifting procedure that is also important in polynomial factorization. For sparse multivariate polynomials (with many missing terms), the SMGCD algorithm can be very efficient. These procedures can become even faster by employing parallelism. For information on the use of parallelism in computer algebra, the reader is referred to [10] [17]. Here we will focus on the SMGCD algorithm.

We consider the parallelization of the SMGCD algorithm on shared-memory multiprocessors (SMP), a popular MIMD architecture that is both affordable and widely available. Our department maintains a 12-processor Encore Multimax and a 26-processor Sequent Balance. The latter is the primary machine used in our work. Our experience with parallel factorization (PFACTOR [23] and PLIFT [24]) helped much in the current investigation.

Without inventing a brand new parallel algorithm, our strategy is to parallelize key steps in the SMGCD algorithm as given by Zippel [25]. We briefly introduce the SMGCD algorithm and define necessary notations in the next section. Then we specify our parallel strategy and describe the PSMGCD package which implements the proposed parallel algorithm. Porting the PARI computer algebra library [3] to the Sequent Balance [22] allowed us to avoid coding many basic polynomial operations and to concentrate on parallelism. PSMGCD is extensively tested and timing results are included to show the effect of the parallelization.

2 The Sparse Modular GCD Algorithm

We wish to compute $G = \gcd(U, V)$ for polynomials $U, V \in \mathbb{Z}[x_0, \ldots, x_t]$, where G is sparse. Assume the variables are numbered so that x_0 has the highest degree appearing in either U or V, and that U and V are primitive with respect to x_0. An overview of the sequential SMGCD algorithm is given. The reader is referred to [25] for Zippel's original presentation of the algorithm.

SMGCD substitutes randomly chosen integer values for the variables x_1 through x_t in U and V to reduce the multivariate GCD computation to one of univariate polynomials in $\mathbb{Z}_p[x_0]$, then recovers the lost variables one at a time, us-

*Work reported herein has been supported in part by the National Science Foundation under Grant CCR-9201800

†On sabbatical leave from Kent State University

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association of Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
ISAAC 94 - 7/94 Oxford England UK
© 1994 ACM 0-89791-638-7/94/0007..$3.50

Input: $p \in \mathbb{Z}^+$, $U, V \in \mathbb{Z}[x_0, \ldots, x_t]$,
$\alpha = (\alpha_1, \ldots, \alpha_t) \in \mathbb{Z}_p^t$,

Output: $\gcd(U, V) \bmod p$ if α is good

$G_0 = \sum_{i=1}^{m} g_{0,i} x_0^{e_i} \leftarrow \gcd(U(x_0, \alpha), V(x_0, \alpha)) \bmod p$

For s from 1 to t do

$U_s \leftarrow U(x_0, \ldots, x_s, \alpha_{s+1}, \ldots, \alpha_t) \bmod p$

$V_s \leftarrow V(x_0, \ldots, x_s, \alpha_{s+1}, \ldots, \alpha_t) \bmod p$

$d \leftarrow \min\{\deg(U_s, x_s), \deg(V_s, x_s)\}$

For $i \in \{1, \ldots, m\}$ do

$n_i \leftarrow$ the number of power-product terms in $g_{s-1,i}$

$N \leftarrow \max\{n_i | 1 \le i \le m\}$

Randomly select distinct $\beta_1, \ldots, \beta_d \in \mathbb{Z}_p$

Choose distinct $\xi_1, \ldots, \xi_N \in \mathbb{Z}_p^{s-1}$

For $(j, k) \in \{1, \ldots, d\} \times \{1, \ldots, N\}$ do

$T_{j,k} \leftarrow \gcd(U_s(x_0, \xi_k, \beta_j), V_s(x_0, \xi_k, \beta_j))$ in $\mathbb{Z}_p[x_0]$

For $i \in \{1, \ldots, m\}$ do $\tau_{i,j,k} \leftarrow \text{coeff}(T_{j,k}, x_0^{e_i})$

For $(i, j) \in \{1, \ldots, m\} \times \{1, \ldots, d\}$ do

$h_{i,j} \leftarrow$ S_INTERP$(p, g_{s-1,i}, \xi_1, \ldots, \xi_{n_i}, \tau_{i,j,1}, \ldots, \tau_{i,j,n_i})$

For $i \in \{1, \ldots, m\}$ do

$g_{s,i} \leftarrow$ D_INTERP$(p, d+1, g_{s-1,i}, h_{i,1}, \ldots, h_{i,d}, \alpha_s, \beta_1, \ldots, \beta_d)$

$G_s \leftarrow \sum_{i=1}^{m} g_{s,i}(x_1, \ldots, x_s) x_0^{e_i}$

Return G_t

Figure 1: SMOD_GCD(p, U, V, α)

ing techniques that are efficient for sparse polynomials. The prime p must be large enough to guarantee that the result mod p is the same as the actual GCD. In practice, if p is too big to allow single precision computation, the algorithm instead would be carried out modulo several single-precision primes and the Chinese Remainder Algorithm (CRA) for integers [12] is used to coalesce the results.

Our implementation of SMGCD is shown in Fig.1. SMODGCD(U, V, α, p) computes the image of G modulo the prime p if given a "good" evaluation point[1]. Tests to check whether G is indeed the true GCD are not shown to keep Fig.1 simple.

We first compute

$$G_0(x_0) = \sum_{i=1}^{m} g_{0,i} x_0^{i} = \gcd(U_0, V_0) \bmod p$$

using some algorithm for the GCD of two univariate polynomials over a finite field (e.g., the Euclidean algorithm).

[1]This choice is crucial: if it causes any non-zero term to evaluate to zero an incorrect result (discovered by dividing the result into U and V) is obtained. When this occurs the process must be started all over again with a new evaluation point. The probability of this happening can be reduced arbitrarily by selecting larger primes [25]. Using the variable of highest degree as the main variable also reduces this probability.

Input: $p \in \mathbb{Z}^+$

$f = \sum_{i=1}^{n} \varphi_i x_1^{c_{i,1}} \cdots x_q^{c_{i,q}} \in \mathbb{Z}_p[x_1, \ldots, x_q]$, $\varphi_i \neq 0$

$\xi_1, \ldots, \xi_n \in \mathbb{Z}_p^q$, where $\xi_i = (\xi_{i,1}, \ldots, \xi_{i,q})$,

$\tau_1, \ldots, \tau_n \in \mathbb{Z}_p$

Output: $\bar{f} \in \mathbb{Z}_p[x_1, \ldots, x_q]$ such that $\bar{f}(\xi_j) \equiv \tau_j \bmod p$ for all $j = 1, \ldots, n$

Solve the system of equations

$y_1 \xi_{1,1}^{c_{1,1}} \cdots \xi_{1,q}^{c_{1,q}} + \cdots + y_n \xi_{1,1}^{c_{n,q}} \cdots \xi_{1,q}^{c_{n,q}} = \tau_1$

$\vdots$

$y_1 \xi_{n,1}^{c_{1,1}} \cdots \xi_{n,q}^{c_{1,q}} + \cdots + y_n \xi_{n,1}^{c_{n,q}} \cdots \xi_{n,q}^{c_{n,q}} = \tau_n$

over $\mathbb{Z}_p$ to obtain solutions $y_1 = \bar{\varphi}_1, \ldots, y_n = \bar{\varphi}_n$

Return $\sum_{i=1}^{n} \bar{\varphi}_i x_1^{c_{i,1}} \cdots x_q^{c_{i,q}}$

Figure 2: S_INTERP$(p, f, \xi_1, \ldots, \xi_n, \tau_1, \ldots, \tau_n)$

Then we successively recover

$$G_s(x_0, \ldots, x_s) = \sum_{i=1}^{m} g_{s,i}(x_1, \ldots, x_s) x_0^{i} = \gcd(U_s, V_s) \bmod p,$$

where U_s and V_s are as defined in Fig.1, from G_{s-1}, until $G_t \equiv G \bmod p$ is obtained. Notice that we are relying heavily on the assumption that G_s is the correct image of G in $\mathbb{Z}_p[x_0, \ldots, x_s]$. Each term $g_{s,i}$ of G_s is recovered from the corresponding term $g_{s-1,i}$ of G_{s-1}, by first using sparse interpolation to obtain polynomials $h_{i,j}$, each of which is congruent to $g_{s,i}$ modulo a different linear polynomial, then applying dense interpolation to $g_{s-1,i}$ and $h_{i,j}$. Subalgorithms for termwise sparse and dense interpolation are given in Figures 2 and 3, respectively. To illuminate the behavior of the algorithm we describe the steps taken to compute the gcd of two polynomials having three variables. After α_1 and α_2 are randomly chosen, the univariate GCD

$$\begin{aligned} G_0(x_0) &= G(x_0, \alpha_1, \alpha_2) \\ &= \gcd(U(x_0, \alpha_1, \alpha_2), V(x_0, \alpha_1, \alpha_2)) \\ &= g_{0,m} x_0^{e_m} + g_{0,m-1} x_0^{e_{m-1}} + \ldots + g_{0,1} x_0^{e_1}. \end{aligned}$$

is computed. The goal now is to determine the coefficient polynomials $g_{2,i}(x_1, x_2)$ for $i = 1, \ldots, m$. We assume that G_0, G_1 and G_2 all have the same nonzero terms in x_0; if the assumption is proven wrong we start over again. Compute d, the bound for the degree of x_1 in $G(x_0, x_1, x_2)$, choose β_1 through β_d, and compute all the $h_{i,j}$. Since each system of equations is of size one, the sparse interpolation step at this stage is trivial:

$$\begin{aligned} h_{i,j} &= \text{coeff}(\gcd(U(x_0, \beta_j, \alpha_2), V(x_0, \beta_j, \alpha_2)), x_0^{e_i}) \\ &= g_{2,i}(\beta_j, \alpha_2) \end{aligned}$$

Dense polynomial interpolation at $d+1$ values, $g_{0,i}$, $h_{i,1}, \ldots, h_{i,d} \in \mathbb{Z}_p$, produces $g_{1,i}(x_1)$. Assembling these we obtain

$$G_1(x_0, x_1) = G(x_0, x_1, \alpha_2) = \sum_{i=1}^{m} g_{1,i} x_0^{e_i}.$$

Let d now denote the degree bound computed for x_2 in G. We randomly select new values β_1 through β_d and compute

Input: $p, \ell \in \mathbb{Z}^+$, $f_1, \ldots, f_\ell \in \mathbb{Z}_p[x_1, \ldots, x_{s-1}]$, $\phi_1, \ldots, \phi_\ell \in \mathbb{Z}_p$

Output: $\bar{f} \in \mathbb{Z}_p[x_1, \ldots x_s]$ such that $\bar{f}(x_1, \ldots, x_{s-1}, \phi_i) = f_i$ for all $i = 1, \ldots, \ell$

Use CRA for polynomials to interpolate $\bar{f}$ from

$f_1 \equiv \bar{f} \bmod (x_s - \phi_1)$

$\vdots$

$f_\ell \equiv \bar{f} \bmod (x_s - \phi_\ell)$

Return $\bar{f}$

Figure 3: D_INTERP$(p, \ell, f_1, \ldots, f_\ell, \phi_1, \ldots, \phi_\ell)$

the images $h_{i,j}(x_1)$ using sparse interpolation. Dense interpolation is then used to construct $g_{2,i}(x_1, x_2)$, the polynomial coefficients for each x_0 term in $G_2 = \gcd(U, V)$. Let us illustrate the sparse interpolation by an example. Suppose we are recovering $G_2(x_0, x_1, x_2)$ from $G_1(x_0, x_1)$ in $\mathbb{Z}_{19}$, and that

$$\begin{aligned} U_2(x_0, x_1, x_2) &= x_0^4 + x_1 x_0^3 + (x_1^2 + x_2 x_1)\, x_0^2 + \\ &\quad (x_1^3 + x_2 x_1^2 + 4)\, x_0 + 4 x_1 \\ V_2(x_0, x_1, x_2) &= x_0^4 + x_2 x_0^3 + (x_1^2 + x_2 x_1)\, x_0^2 + \\ &\quad (x_2 x_1^2 + x_2^2 x_1 + 4)\, x_0 + 4 x_2 \\ \alpha_2 &= 7 \\ G_1(x_0, x_1) &= G_2(x_0, x_1, 7) \\ &= x_0^3 + (x_1^2 + 7\, x_1)\, x_0 + 4 \\ &= g_{1,3} x_0^3 + g_{1,2} x_0 + g_{1,1} \end{aligned}$$

In order to use dense interpolation to recover G_2, we need G_1 and $d = \min(\deg(U_2, x_2), \deg(V_2, x_2)) = 1$ more image of G_2 in $\mathbb{Z}_{19}[x_0, x_1]$. The coefficient $h_{i,1}$ of each term of the new image is obtained from $g_{1,i}$ by sparse interpolation: first $\beta_1 = 4$, and $\xi_1 = (1), \xi_2 = (8)$ are chosen as prescribed by the algorithm. Then

$$\begin{aligned} T_{1,1} &= \gcd(U(x_0, \xi_1, \beta_1), V(x_0, \xi_1, \beta_1)) = x_0^3 + 5x_0 + 4 \\ T_{1,2} &= \gcd(U(x_0, \xi_2, \beta_1), V(x_0, \xi_2, \beta_1)) = x_0^3 + x_0 + 4 \end{aligned}$$

are computed, giving us

$$\begin{aligned} \tau_{1,1,1} &= \text{coeff}(T_{1,1}, x_0^0) = 4 \quad \tau_{1,1,2} = \text{coeff}(T_{1,2}, x_0^0) = 4 \\ \tau_{2,1,1} &= \text{coeff}(T_{1,1}, x_0^1) = 5 \quad \tau_{2,1,2} = \text{coeff}(T_{1,2}, x_0^1) = 1 \\ \tau_{3,1,1} &= \text{coeff}(T_{1,1}, x_0^3) = 1 \quad \tau_{3,1,2} = \text{coeff}(T_{1,2}, x_0^3) = 1. \end{aligned}$$

Then $h_{1,1} = 4$ and $h_{3,1} = 1$ are found by solving the trivial linear systems $y_1 = \tau_{1,1,1} = \tau_{1,1,2}$ and $y_1 = \tau_{3,1,1} = \tau_{3,1,2}$. To determine $h_{2,1}$, the system of equations

$$\begin{aligned} y_2(1)^2 + y_1(1) &= \tau_{2,1,1}, \\ y_2(8)^2 + y_1(8) &= \tau_{2,1,2}, \end{aligned}$$

is solved over $\mathbb{Z}_{19}$. This gives us $h_{1,2} = x_1^2 + 4x_1$.

Now dense interpolation can be used on each term of the two polynomials

$$\begin{aligned} G_2 &\equiv x_0^3 + (x_1^2 + 7x_1)\, x_0 + 4 \mod (x_2 - 7) \\ G_2 &\equiv x_0^3 + (x_1^2 + 4x_1)\, x_0 + 4 \mod (x_2 - 4) \end{aligned}$$

to produce

$$G_2(x_0, x_1, x_2) = x_0^3 + (x_1^2 + x_1 x_2)\, x_0 + 4.$$

Input: $U, V \in \mathbb{Z}[x_0, \ldots, x_t]$

Output: $\gcd(U, V)$

Choose a set of primes P such that their product is greater than twice a predicted bound on the integer coefficients of the GCD.

For each prime $p \in P$ do in parallel

Choose at random $\alpha \in \mathbb{Z}_p^t$, and compute

$$\mathcal{G}_p = \sum_{i=1}^{r} \gamma_{i,p} x_0^{c_{i,0}} \cdots x_t^{c_{i,t}} \leftarrow \text{PSPMODX\_GCD}(U, V, \alpha, p)$$

For $i \in \{1, \ldots, r\}$ do in parallel

Use CRA to recover γ_i from $\gamma_{i,p}$ for all $p \in P$

Return $\sum_{i=1}^{r} \gamma_i x_0^{c_{i,0}} \cdots x_t^{c_{i,t}}$

Figure 4: Naive partitioning of top level

3 Parallel Sparse Modular GCD Strategies

We now consider strategies for partitioning the SMGCD algorithm into independent subtasks, keeping interaction between subtasks low yet minimize redundant calculation.

Task partitioning is possible on many levels. Shortly we shall consider three methods for partitioning SMOD_GCD, but first we shall discuss two methods to subdivide the work at the topmost level. This is only necessary when G has multiple-precision integer coefficients. We shall refer to a generic partitioning of SMOD_GCD as PSPMODX_GCD. The naive top level partition is given in Fig. 4. In this method, a parallel version of CRA for integers [21] would allow subdivision of the final recovery of the gcd from its various images mod p, although the grain-size may be too fine for this to be practical on many architectures.

An alternate partitioning of the top level is given in Fig. 5. It seems preferable for several reasons.

First, this approach computes the GCD modulo one prime at a time and terminates as soon as the true GCD is determined. On the other hand, the naive partitioning must compute the GCD modulo all of the chosen primes before it can detect the true GCD. This is inefficient since the predicted bound on the integer coefficients of the GCD is often too large [16].

Also, all available processes are employed to compute each single GCD image in the second approach while the naive partition can only use as many parallel processes as the number of images required.

Finally, the partitioning of Fig.5 uses only sparse interpolation for computing additional images of the GCD, once the first GCD image (i.e. $G \bmod p_0$) is computed. The GCD images $\mathcal{G}_p$ are computed based on the structure [2] of $G \bmod p_0$. For example, if $G \bmod p_0 = x_0^{10} + x_1^{10} + \ldots + x_t^{10}$, then $\mathcal{G}_p$ is assumed to be of the form $\mathcal{G}_p = a_0\, x_0^{10} + a_1\, x_1^{10} + \ldots + a_t\, x_t^{10}$ where the $a_i \in \mathbb{Z}_p$. The a_i are computed via sparse interpolation.

The naive partitioning, on the other hand, computes each of the $\mathcal{G}_p$ starting from scratch. None of the $\mathcal{G}_p$ is used as a model for computing other images. Hence many unnecessary computations are performed.

[2] It is possible that the image computed first is erroneous and, hence, all of the subsequent $\mathcal{G}_p$ are also erroneous. This possibility, however, can be arbitrarily small by choosing large p_0. See footnote 1

Input: $U, V \in \mathbb{Z}[x_0, \ldots, x_t]$

Output: $\gcd(U,V) = \sum_{i=1}^{r} \gamma_i x_0^{c_{i,0}} \cdots x_t^{c_{i,t}} \in \mathbb{Z}[x_0, \ldots, x_t]$

Choose P and α as in naive partitioning, select $p_0 \in P$ and let $P' = P - \{p_0\}$

$\mathcal{R} = \sum_{i=1}^{r} \gamma_i x_0^{c_{i,0}} \cdots x_t^{c_{i,t}} = \sum_{i=1}^{m} g_{t,i}(x_1, \ldots, x_t)\, x_0^{e_i}$
$\leftarrow$ PSMGCD_1(U, V, α, p_0)

If $\mathcal{R}$ divides both U and Vover $\mathbb{Z}$, then return $\mathcal{R}$
For $i \in \{1, \ldots, m\}$ do
  $n_i \leftarrow$ the number of power-product terms in $g_{t,i}$
$N \leftarrow \max\{n_i | 1 \le i \le m\}$
For each prime $p \in P'$ do (* sequential loop *)
  Choose distinct $\xi_1, \ldots, \xi_N \in \mathbb{Z}_p^t$
  For $\ell \in \{1, \ldots, N\}$ do in parallel
    $T_\ell \leftarrow \gcd(U(x_0, \xi_\ell), V(x_0, \xi_\ell))$ in $\mathbb{Z}_p$
    For $i \in \{1, \ldots, m\}$ do in parallel
      $\tau_{i,\ell} \leftarrow \text{coeff}(T_\ell, x_0^{e_i})$
  For $i \in \{1, \ldots, m\}$ do in parallel
    $\overline{g}_{t,i} \leftarrow$ S_INTERP$(p, g_{t,i}, \xi_1, \ldots, \xi_{n_i}, \tau_{i,1}, \ldots, \tau_{i,n_i})$
  $\mathcal{G}_p = \sum_{i=1}^{r} \gamma_{i,p} x_0^{c_{i,0}} \cdots x_t^{c_{i,t}} \leftarrow \sum_{i=1}^{m} \overline{g}_{t,i}(x_1, \ldots, x_t)\, x_0^{e_i}$
  For $i \in \{1, \ldots, r\}$ do in parallel
    Use CRA to update the integer coefficients of $\mathcal{R}$ given γ_i, $\gamma_{i,p}$ and p.
  Set $\mathcal{R} \leftarrow \sum_{i=1}^{r} \gamma_i x_0^{c_{i,0}} \cdots x_t^{c_{i,t}}$
  If $\mathcal{R}$ divides both U and Vover $\mathbb{Z}$, then return $\mathcal{R}$

Figure 5: Sophisticated partitioning of top level

Input: $p \in \mathbb{Z}^+$, $U, V \in \mathbb{Z}[x_0, \ldots, x_t]$,
$\alpha = (\alpha_1, \ldots, \alpha_t) \in \mathbb{Z}_p^t$,

Output: $\gcd(U,V) \bmod p$ if α is good

$G_0 = \sum_{i=1}^{m} g_{0,i} x_0^{e_i} \leftarrow \gcd(U(x_0, \alpha), V(x_0, \alpha)) \bmod p$
For s from 1 to t do (* sequential loop *)
  $U_s \leftarrow U(x_0, \ldots, x_s, \alpha_{s+1}, \ldots, \alpha_t) \bmod p$
  $V_s \leftarrow V(x_0, \ldots, x_s, \alpha_{s+1}, \ldots, \alpha_t) \bmod p$
  $d \leftarrow \min\{\deg(U_s, x_s), \deg(V_s, x_s)\}$
  For $i \in \{1, \ldots, m\}$ do
    $n_i \leftarrow$ the number of power-product terms in $g_{s-1,i}$
  $N \leftarrow \max\{n_i | 1 \le i \le m\}$
  Randomly select distinct $\beta_1, \ldots, \beta_d \in \mathbb{Z}_p$
  Choose distinct $\xi_1, \ldots, \xi_N \in \mathbb{Z}_p^{s-1}$
  For $(j,k) \in \{1, \ldots, d\} \times \{1, \ldots, N\}$ do in parallel
    $T_{j,k} \leftarrow \gcd(U_s(x_0, \xi_k, \beta_j), V_s(x_0, \xi_k, \beta_j))$ in $\mathbb{Z}_p[x_0]$
    For $i \in \{1, \ldots, m\}$ do $\tau_{i,j,k} \leftarrow \text{coeff}(T_{j,k}, x_0^{e_i})$
  For $(i,j) \in \{1, \ldots, m\} \times \{1, \ldots, d\}$ do in parallel
    $h_{i,j} \leftarrow$ S_INTERP$(p, g_{s-1,i}, \xi_1, \ldots, \xi_{n_i}, \tau_{i,j,1}, \ldots, \tau_{i,j,n_i})$
  For $i \in \{1, \ldots, m\}$ do in parallel
    $g_{s,i} \leftarrow$ D_INTERP$(p, d+1, g_{s-1,i}, h_{i,1}, \ldots, h_{i,d}, \alpha_s, \beta_1, \ldots, \beta_d)$
  $G_s \leftarrow \sum_{i=1}^{m} g_{s,i}(x_1, \ldots, x_s)\, x_0^{e_i}$
Return G_t

Figure 6: PSMGCD_1(U, V, α, p)

3.1 Parallelization of SMOD_GCD

We now turn to the parallelization of SMOD_GCD: Figures 6-8 lay out three possibilities. The same notation as given in Fig. 1 is used here.

3.1.1 PSMGCD_1

PSMGCD_1 derives $G_0, G_1, \ldots, G_t$ in sequence, but the tasks performed to get G_s from G_{s-1} are done in parallel. This method performs the sparse interpolation stage first, then synchronizes to perform the dense interpolation next. The sparse interpolation stage computes all the $h_{i,k}$ in parallel. The dense interpolation, in turn, interpolates (in parallel) the $g_{s,i}$ from the polynomials $h_{i,k}$.

3.1.2 PSMGCD_2

Similar to PSMGCD_1, PSMGCD_2 computes G_t by computing $G_0, G_1, \ldots, G_t$ in sequence. The tasks performed to get G_s from G_{s-1} are done in parallel as well. However, instead of performing sparse interpolation followed by dense interpolation, the second method divides the work so that both sparse and dense interpolation for the term $g_{s,i}$ are done independently of the sparse/dense interpolation for all other terms. This requires less synchronization among the computational threads. Notice that a thread recovering $g_{s,i}$ may further subdivide to compute $h_{i,j}$ in parallel.

3.1.3 PSMGCD_3

PSMGCD_3 recovers $g_{0,i}, \ldots, g_{t,i}$ in sequence independent of the recovery of the other terms of G_t. This is especially useful if many terms of the GCD involve only a minor subset of the variables involved. For example, if the variables occuring in the term $g_{t,i}$ are x_1, x_4, x_6, then both of the sparse and dense interpolation steps are applied to recover only these variables.

This scheme is effective since there is a way to compute degree bounds of the variables that occur in each term of the final gcd. Each term may proceed at its own pace, using more time to recover one variable and less time to recover another. This allows us to prune a large amount of extra work arising from conservative bounds on the degree.

One way to find the variables occuring in each term of the final gcd along with their corresponding degree bounds is to compute $\gcd(\overline{U}_s, \overline{V}_s) \bmod p$, $s = 1, \ldots, t$, where

$$\overline{U}_s = U(x_0, \alpha_1, \ldots, \alpha_{s-1}, x_s, \alpha_{s+1}, \ldots, \alpha_t) \bmod p$$
$$\overline{V}_s = V(x_0, \alpha_1, \ldots, \alpha_{s-1}, x_s, \alpha_{s+1}, \ldots, \alpha_t) \bmod p.$$

Input: $p \in \mathbb{Z}^+$, $U, V \in \mathbb{Z}[x_0, \ldots, x_t]$,
$\alpha = (\alpha_1, \ldots, \alpha_t) \in \mathbb{Z}_p^t$,

Output: $\gcd(U, V) \bmod p$ if α is good

$G_0 = \sum_{i=1}^{m} g_{0,i} x_0^{e_i} \leftarrow \gcd(U(x_0, \alpha), V(x_0, \alpha)) \bmod p$

For s from 1 to t do (* sequential loop *)

$U_s \leftarrow U(x_0, \ldots, x_s, \alpha_{s+1}, \ldots, \alpha_t) \bmod p$

$V_s \leftarrow V(x_0, \ldots, x_s, \alpha_{s+1}, \ldots, \alpha_t) \bmod p$

$d \leftarrow \min\{\deg(U_s, x_s), \deg(V_s, x_s)\}$

For $i \in \{1, \ldots, m\}$ do

$n_i \leftarrow$ the number of power-product terms in $g_{s-1,i}$

$N \leftarrow \max\{n_i | 1 \le i \le m\}$

Randomly select distinct $\beta_1, \ldots, \beta_d \in \mathbb{Z}_p$

Choose distinct $\xi_1, \ldots, \xi_N \in \mathbb{Z}_p^{s-1}$

For $(j, k) \in \{1, \ldots, d\} \times \{1, \ldots, N\}$ do in parallel

$T_{j,k} \leftarrow \gcd(U_s(x_0, \xi_k, \beta_j), V_s(x_0, \xi_k, \beta_j))$ in $\mathbb{Z}_p[x_0]$

For $i \in \{1, \ldots, m\}$ do $\tau_{i,j,k} \leftarrow \text{coeff}(T_{j,k}, x_0^{e_i})$

For $i \in \{1, \ldots, m\}$ do in parallel

For $j \in \{1, \ldots, d\}$ do in parallel

$h_{i,j} \leftarrow$ S_INTERP$(p, g_{s-1,i}, \xi_1, \ldots, \xi_{n_i}, \tau_{i,j,1}, \ldots, \tau_{i,j,n_i})$

$g_{s,i} \leftarrow$ D_INTERP$(p, d+1, g_{s-1,i}, h_{i,1}, \ldots, h_{i,d}, \alpha_s, \beta_1, \ldots, \beta_d)$

$G_s \leftarrow \sum_{i=1}^{m} g_{s,i}(x_1, \ldots, x_s)\, x_0^{e_i}$

Return G_t

Figure 7: PSMGCD_2(U, V, α, p)

Input: $p \in \mathbb{Z}^+$, $U, V \in \mathbb{Z}[x_0, \ldots, x_t]$,
$\alpha = (\alpha_1, \ldots, \alpha_t) \in \mathbb{Z}_p^t$,

Output: $\gcd(U, V) \bmod p$ if α is lucky

For $s \in 1, \ldots, t$ do in parallel (* Precompute bivariate gcds *)

Use any GCD algorithm to compute $\gcd(\overline{U}_s, \overline{V}_s) \bmod p$ and

set $\mathcal{G}(x_0, x_s) = \sum_{i=1}^{m} w_i^{(s)}(x_s) x_0^{e_i} \leftarrow \gcd(\overline{U}_s, \overline{V}_s)$

$d_{s,i} = \deg(w_i^{(s)}(x_s))$ for all $1 < s \le t, 1 \le i \le m$

$d \leftarrow \max\{d_{s,i} | 1 < s \le t, 1 \le i \le m\}$

Randomly select distinct $\beta_1, \ldots, \beta_d \in \mathbb{Z}_p$

Set $G_1 = \sum_{i=1}^{m} g_{1,i} x_0^{e_i} \leftarrow \mathcal{G}(x_0, x_1)$

For $i \in \{1, \ldots, m\}$ do in parallel (* Main computation *)

(* The scope of s, j, k, n, and ξ_k is limited to *)
(* current iteration of the outer parallel loop. *)

For s = 2 to t if $d_{s,i} > 0$ do (* sequential loop *)

$n \leftarrow$ the number of power-product terms in $g_{s-1,i}$

For $k \in \{1, \ldots, n\}$ do
$\xi_k \leftarrow$ GEN_EVAL_PT(p, s, k)

(* Inner loop 1 *)
For $(j, k) \in \{1, \ldots, d_{s,i}\} \times \{1, \ldots, n\}$ do in parallel

If $T_{s,j,k}$ has not yet been computed then
$T_{s,j,k} \leftarrow \gcd(U_s(x_0, \xi_k, \beta_j), V_s(x_0, \xi_k, \beta_j))$ in $\mathbb{Z}_p[x_0]$
$\tau_{i,j,k} \leftarrow \text{coeff}(T_{s,j,k}, x_0^{e_i})$

For $j \in \{1, \ldots, d_{s,i}\}$ do in parallel (* Inner loop 2 *)

$h_{i,j} \leftarrow$ S_INTERP$(p, g_{s-1,i}, \xi_1, \ldots, \xi_n, \tau_{i,j,1}, \ldots, \tau_{i,j,n})$

$g_{s,i} \leftarrow$ D_INTERP$(p, d_{s,i}+1, g_{s-1,i}, h_{i,1}, \ldots, h_{i,d_{s,i}}, \alpha_s, \beta_1, \ldots, \beta_{d_{s,i}})$

Return $\sum_{i=1}^{m} g_{t,i}(x_1, \ldots, x_t)\, x_0^{e_i}$

Figure 8: PSMGCD_3(U, V, α, p)

These bivariate gcds may be computed in parallel in the precomputation section of PSMGCD_3 using PSMGCD_1 or PSMGCD_2. This precomputation of the bivariate gcds may actually provide some speed up. To show this, observe that each $g_{t,i}$ now involves iterating *at most* $t-1$ variables. This is in contrast to blindly introducing *exactly* $t-1$ variables without bivariate gcds computations.

In addition, this approach predicts lower variable degree bounds than the conservative bounds $\min\{\deg(U_s, x_s), \deg(V_s, x_s)\}$ for $2 \le s \le t$. As a result, the sparse and dense interpolation stages are performed much faster since, as the two inner parallel loops of Fig.8 indicate, both stages depend on the degree bound used.

In S_INTERP, the solution of the systems of linear equations may be performed in parallel. Kaltofen *et al* [11] suggest a more efficient sequential approach which may also be parallelized. Similarly the CRA for polynomials in D_INTERP may be performed in parallel. It is not clear whether such a fine partitioning of the problem will afford any significant speedup on current hardware, although in the future this may prove useful.

Also in PSMGCD_3, since it is not necessary to choose the ξ_k at random, we posit a function GEN_EVAL_PT that deterministically returns the appropriate value that must be shared by other computational threads.

4 Parallel Implementation of PSMGCD

As an experiment to show the effectiveness and the practicality of the various parallel strategies presented, we have implemented the parallel approach shown in figure 6. The results are very promising. Three input samples are shown in figures 9 through 11, with timing results obtained by computing the GCD of these polynomials using different number of processes for each input.

The PSMGCD package is written in the C language with Sequent parallel extensions. It calls on routines from the Sequent parallel programming library and the PARI[3] computer algebra system (appendices A and B) and runs under

[3] We made minor modifications to PARI to put its internal stack in shared memory.

Sequent's DYNIX operating system [2].

Our implementation uses a data partition method called microtasking[2] to execute loop iterations in parallel. In this method, independent processes execute the loop body using different sets of data. For example a sequential loop of the form

```
For (i = 0; i < N; i++)
     {
      point = Generate an evaluation point
      Compute GCD of U and V at point
     }
```

can be implemented in parallel as

```
For (i = pid, i < N; i+= nprocs)
     {
      point_i = Generate an evaluation point
      g_i = Compute GCD of U and V at point_i
     }
```

where nprocs is the number of processes to use and pid ($0 \leq$ pid $<$ nprocs - 1) is a unique process ID assigned to each process.

In addition to microtasking, PSMGCD makes heavy usage of the shared memory feature to facilitate inter-processes communication. All the relevant data, such as β_1 through β_d and all the univariate gcds computed in the first parallel loop, are all kept in shared memory accessible by all running processes.

4.1 PSMGCD Usage

PSMGCD can be used in two different modes:

- Filter mode: PSMGCD takes a set of multivariate polynomials pairs from standard input and produces output on standard output. In this way PSMGCD can process several input requests in a file or from another process via a pipe connection.
- Server mode: PSMGCD runs as a network server ready to receive TCP/IP stream socket connections. This allows processes running on remote hosts to send GCD requests to PSMGCD.

The input to PSMGCD consists of the two polynomials, the number of processes to use, and an optional upper bound on the coefficients of the target GCD.

If not supplied, a very conservative *a priori* upper bound is calculated by PSMGCD. For example, to compute

$$\gcd((x_2 + x_1 + 2)^4 (3\,x_2 + 3\,x_1 - 1),\\ (-x_2 + 4\,x_1 + 2)\,(x_2 + x_1 + 2)^3)$$

using 4 processes and given the upper bound 100, one types:

```
PSMGCD 4  100
(x1+x2+2)^4*(3*x1+3*x2-1)
(x1+x2+2)^3*(4*x1 -x2+2)
```

(Note that expression expansion is done by PSMGCD.)

PSMGCD requires the use of two important parameters to control the parallel activities inside the program.

nprocs: the total number of processes to use. This parameter is provided by the user, but cannot be greater than the number of hardware processors.

np: The total number of primes to use. This parameter is computed internally and depends on the *a priori* upper bound on the integer coefficients of G.

Number of Processes	μ-seconds
1	10665959
2	7355444
3	5504141
4	5024199
5	5264585
6	5408994
7	5491863

$$\begin{aligned} G &= 7x_1x_2x_3^2 - 2x_1x_2^3 - x_1^3 - 3 \\ U &= G(-x_1x_2x_3 - x_1x_2^2 - x_1^2 - x_1 + 2) \\ V &= G(6x_1x_3^3 - x_1x_2 - x_1^3 - 3) \end{aligned}$$

Figure 9: Timings for Case 1

Number of Processes	μ-seconds
1	121094885
2	65203089
3	47711299
4	38042621
5	32697187
6	28213439
7	26844616
8	25528374
9	25435541
10	25144418
12	23724676

$$\begin{aligned} G &= x_3x_4^6 + x_4^6 + x_1^4x_2x_3^5 + x_1^4x_3^5 + x_2^4x_3^4 + x_3^4 + x_1^2x_2^2x_3^3 + x_1^2x_3^3 + \\ &\quad x_1^4x_2x_3^2 + x_1^4x_3^2 + x_3 + x_2^4 + x_1^2x_2^2 + x_1^4x_2 + x_1^4 + x_1^2 + 2 \\ U &= G(x_4 + x_3^3 + x_2 + x_1 + 1) \\ V &= G(x_4 + x_3 + x_2^2 + x_1) \end{aligned}$$

Figure 10: Timings for Case 2

Number of Processes	μ-seconds
1	160815869
2	84524491
3	60481662
4	48474882
5	46974361
6	35552337
7	35655118
8	31371559
12	28611390

$$\begin{aligned} G &= x_1x_4^4 + 20x_1x_4^3 + 150x_1x_4^2 + 500x_1x_4 + x_1^6x_3^5 + 10x_1^6x_3^4 + \\ &\quad 40x_1^6x_3^3 + 80x_1^6x_3^2 + x_1^2x_3^2 + 80x_1^6x_3 + 8x_1^2x_3 + x_1^5x_2^3 + \\ &\quad 9x_1^5x_2^2 + 27x_1^5x_2 + x_1^8 + 32x_1^6 + 27x_1^5 + 16x_1^2 + 625x_1 \\ U &= G(x_2x_3x_4 + x_3x_4 + x_2x_3 + x_1) \\ V &= G(x_3x_4 + x_2x_4 + x_1 + 1) \end{aligned}$$

Figure 11: Timings for Case 3

5 Conclusion

The investigation here indicates that the sparse modular GCD algorithm can be parallelized effectively on shared memory multiprocessors. The PSMGCD package can be used as a network server for clients requiring polynomial factorization on the same or different machine. It can be also used as a stand alone GCD computing utility for other computer algebra problems.

A final note: The authors have repeatedly found the current state of software support for parallel programming facilities (e.g., shared memory management) to be frustrating. We hope this situation will be addressed in the next generation of system software.

A Hardware Environment

Our implementations were carried out on a Sequent Balance [15][20] configured with 26 processors and 32 Mbytes of shared memory. The architecture is built around the system bus, which links all memory modules, input/output devices, and the system CPU's. The bus operates at 10 MHZ yielding a channel bandwidth of 80 Mbyte/s. 32 bit addresses and 64 bit data are multiplexed on the 64 bit bus. The system bus access the memory in data packets of 1, 2, 3, 4, and 8 bytes.

Memory read and memory write requests are stored in separate queues to increase the system bus bandwidth. Individual requests and responses are interleaved in sequential bus cycles [20].

In addition, the Balance system avoids bus congestion by providing a distribute control-mechanism in which the status of the read and write queues are checked before any read or write requests are honored. Request accesses are denied unless there is space in the corresponding queue.

The processors in the Balance system are packaged on dual processor boards which are all functionally equivalent. They can be added, removed, or replaced without affecting the operating system. Each board contains two NS32032 general-purpose processors capable of executing 0.75 MIPS. Each processor has a cache memory of 8 Kbytes, a floating point co-processor, a memory management unit, and a system link and interrupt handler chip. A small memory used to store frequently used routines is also provided [2].

B Software Environment

Our software environment consists of the PARI computer algebra system [9] as an algebraic kernel and the Sequent parallel programming library [15] and Dynix operating system [18] for parallel execution of processes.

The PARI computer algebra system is a highly small portable system capable of manipulating algebraic expressions. It can be used as a stand alone interactive system or as an ordinary library callable from any other users programs.

The system is written in C and hence it is highly portable. To increase efficiency, three versions are available: one for the MACINTOSH personal computers, another for the SUN/3 workstations, and a third version for any 32-bit machine with no memory constraints.

PARI has 18 different kinds of data types. All of the PARI data types are recursive in nature. We describe below the data types that are relevant to our work:

1. **Integers** are limited in absolute value to less than $2^{1048480}$ (roughly 315623 digits).
2. **Reals** allow precision of at most 315623 significant decimal digits and exponent absolute value of at most $2^{23} - 1 = 8388607$.
3. **Integermods and Polymods** are represented by two components: the modulus and the value.
4. **Polynomials** are completely recursive: their components (i.e. terms) can be of any type and can be intermixed.

The PARI system consists of 3 hierarchical levels. The *basic kernel* is made up of routines for performing addition, subtraction, multiplication, and division between single and multiple precision integers and reals. The speed of the whole system is highly affected by the efficiency of these routines. Part of this level is written in assembly for the SPARC workstations version. The *generic kernel* consists of a large set of routines for performing the four standard operations on all of the other PARI objects. This part is written in C. The highest level is a larger collection of routines for performing input and output and operations from linear algebra, numerical analysis and number theory. This level also includes a syntactical parser for the interactive user interface.

To implement our algorithm, we found it necessary to modify PARI to run in a parallel environment on the Sequent Balance. PARI allocates a stack which contains all its computations; the software has been modified so that several processes running PARI will have their *individual* stacks allocated in shared memory so that any one PARI process can access data in another PARI process's stack—although it is not allowed to modify another process's data, so as not to confuse the other process's garbage collector.

The Sequent parallel programming library [19] is a set of C routines which allow the programmer to execute C subprograms in parallel. The library provides simplified access to the following capabilities:

- Process creation and identification.
- Shared memory management.
- Mutual exclusion.
- Idle process suspension (useful during serial program sections).
- Process synchronization.

References

[1] Akritas, A. *Elements of Computer Algebra with Applications.* John Wiley and Sons, New York, 1989.

[2] *Balance Technical Summary.* Sequent Computer Systems, Inc., 1986.

[3] Batut, C., Bernardi, D., Cohen, H., Olivier, M. *User's Guide to PARI-GP.* Feb. 1991.

[4] Brown, W. S. *On Euclid's Algorithm and the Computation of Polynomial Greatest Common Divisors.* J. ACM. Vol. 18, 1971.

[5] Brown, W. S. and Traub, J. F. *On Euclid's Algorithm and the Theory of Subresultants.* J. ACM, Vol. 18, No. 4, 1971.

[6] Caviness, B. *Computer Algebra : Past and Future.* EUROCAL'85, Buchberger, B. (ed). Pages 1-18.

[7] Char, B., Geddes, K., and Gonnet G. *GCDHEU: Heuristic Polynomial GCD Algorithm Based on Integer GCD Computation.* J. Symbolic Computation, 7, 1989.

[8] Collins, G. E. *Subresultants and Reduced Polynomial Remainder Sequences.* J. ACM, Vol. 14, No. 1, 1967.

[9] Collins, G., Encarnacion, M., Krandick, W., and Neubacher, A. *A SACLIB Primer.*

[10] Della Dora, J. and Fitch J. *Computer Algebra and Parallelism.* Academic Press, New York, New York, 1989.

[11] Kaltofen, E. and Yagati L. *Improved Sparse Multivariate Polynomial Interpolation Algorithm.* Symbolic and Algebraic Computation, ISSAC'88. Gianni P. (ed).Springer-Verlag Lecture in Computer Science, No.358. pp. 467-474

[12] Laure, M. *Computing by Homomorphic Images.* Computer Algebra Symbolic and Algebraic Computation, Loos R. , Collins, G. , Buchberger (eds). Pages 139-168. Springer-Verlag. 1982.

[13] Loos, R. *Generalized Polynomial Remainder Sequence.* Computer Algebra Symbolic and Algebraic Computation, Loos R. , Collins, G. , Buchberger (eds). Pages 115-137. Springer-Verlag. 1982.

[14] Mathematics and Computer Science Division. *Balance 8000/21000 C Compiler User's Manual.* Argonne National Laboratory, 1989.

[15] Mathematics and Computer Science Division. *Sequent Guide to Parallel Programming.* Aragone National Laboratory, 1989.

[16] Mignotte, M. *An inequality about factors of polynomials.* Math. Comp., Vol. 28, 1974. Pages 1153-1157.

[17] Ponder, C. *Evaluation of "Performance Enhancements" in Algebraic Manipulation Systems.* (PhD Thesis) Report #88/438 UC Berkely Computer Science Division, 1988.

[18] Sequent Computer Systems. *DYNIX programmer's Manual.* 1987.

[19] Anita Osterhaug, ed., *Guide to Parallel Programming on Sequent Computer Systems*, Prentice Hall, Englewood Cliffs, NJ, 1987.

[20] Thakkar, Shreekant. *The Balance Multiprocessor System.* IEEE Micro, pp 57-69, 1988.

[21] Villard, G. *Chinese Remaindering on a MIMD Parallel Computer.* Preprint Nov. 1990.

[22] Wang, P. *A Guide to Parallel Programming on the Sequent Balance*, Technical Report, Institute for Mathematical Computation, Dept. of Math. and Comp. Sci., Kent State University, 1989.

[23] Wang, Paul S., *Parallel Univariate Polynomial Factorization on Shared-Memory Multiprocessors*, Proceedings of the ISSAC'90, Addison-Wesley (ISBN 0-201-54892-5), Aug. 1990, pp. 145-151.

[24] Wang, Paul S. *Parallel Univariate p-adic Lifting on Shared-Memory Multiprocessors.*Proceedings, ISSAC'92, July 27-29, Berkeley, California, 1992, pp. 168-176.

[25] Zippel, Richard. *Probabilistic Algorithms for Sparse Polynomials.* PhD. Thesis, Massachusetts Institute of Technology, 1979.