SMP-based parallel algorithms and implementations for polynomial factoring and GCD are overviewed. Topics include polynomial factoring modulo small primes, univariate and multivariate p-adic lifting, and reformulation of lift basis. Sparse polynomial GCD is also covered.

J. Symbolic Computation (2001) **11**, 1–000

Parallel Polynomial Operations on SMPs: An Overview

Paul S. Wang[†]

Department of Mathematics and Computer Science

Kent State University

Kent, Ohio 44242-0001 USA

pwang@mcs.kent.edu

http://monkey.mcs.kent.edu/∼pwang

(Received 13 March 2001)

1. Introduction

Research at Kent (**?**) and elsewhere (**?**, **?**) applies parallelism to key symbolic computation algorithms for higher performance and implements software to take advantage of advances in parallel computers. One particular focus at Kent is parallel polynomial factoring and GCD computations. In this area, the research at Kent has been conducted mainly on *symmetric multiprocessors* (SMP) where all processing elements (pe) access a global shared memory in a symmetric fashion. Early implementations and performance measuring were carried out on a 12-pe Encore Multimax and/or a 26-pe Sequent Balance. The work paves the way for constructing a parallel computer algebra system kernel to take real advantage of multiprocessor workstations that are becoming increasingly available to scientists and engineers.

Among the many competing parallel architectures, it seems that the *shared memory* model has the best chance of becoming commonly adopted and widely available. Newer parallel machines offer faster CPUs, quicker memory access, and better *scalability*, the ability to increase the number of pe's, by supporting a *distributed global store* at the hardware level.

Investigations conducted at Kent include polynomial factoring modulo small primes, univariate p-adic lifting, detection of true factors, reformulation of lift basis, multivariate p-adic lifting, and sparse multivariate GCD. Highlights of this work are reviewed here[†].

Parallel procedures take an input parameter, np, the number of parallel tasks to be used. It is of course important to fully utilize the specified number of tasks in the parallel procedures. Thus, *load balancing* is an important consideration in parallel algorithms.

2. Univariate Factoring Mod p

The univariate factoring algorithm consists of several major tasks: factoring modulo small integer primes, EEZ lifting (**?**), and recovery of true factors. A C-coded system PFACTOR (**?**) implements factoring modulo small primes. PFACTOR is a stand-alone parallel factorizer that can take input from a file, a pipe or a socket connection over a network. It can also be used interactively as a UNIX command.

[†] Work reported herein has been supported in part by the National Science Foundation under Grant CCR-9503650

[†] This paper is a revised and updated version of (**?**).

0747–7171/90/000000 + 00 $03.00/0

The parallel PFACTOR package takes an arbitrary univariate polynomial $U(x)$ with integer coefficients of any size and produces a prime p, a set of irreducible factors $u_i(x)$ mod p, and information on grouping of extraneous factors for lifting. PFACTOR implements:

1 Parallel selection of primes – Several small primes are selected in parallel to preserve the squarefreeness of the polynomial and to minimize the number of modulo factors.
2 Automatic balancing of work – If factoring modulo several primes is required, the number of tasks assigned to each finite field factorization is made proportional to the amount of work required in the Berlekamp algorithm.
3 Parallel Berlekamp algorithm – Parallel formation of the $(n \times n)$ matrix Q−I (**?**); parallel triangularization of Q−I to produce a basis of its null space; and parallel extraction of factors with greatest common divisor computations.
4 Parallel reconciliation of degrees of factors modulo different primes – The number and degrees of factors modulo different primes can be used to deduce irreducibility or to identify extraneous factors.

2.1. Load Balancing

The parameters np (the number of processes) and k (the number of primes) are important in controlling the parallel activities of PFACTOR. For example, $np = 9, k = 3$ means "factor $U(x)$ mod three different primes, all in parallel with nine processes". Setting $np = 1$ forces sequential processing. The following cases are distinguished.

1 If $np = k$ then np factorizations are performed in parallel each with one process and a different prime.
2 If $np < k$ then the first np factorizations are performed in parallel each with one process; then k is set to $k - np$.
3 If $np > k$ then all k factorizations are carried out in parallel each with one or more processes.

In case 3, if $k = 1$ then all processes are used for the parallel Berkekamp algorithm with the given prime. If $k > 1$, the number of processes assigned to each finite field factorization is made proportional to the amount of work required in the Berlekamp algorithm which is roughly $p_i\ n^2\ \log n + n^3$. Thus, the number of processes ps_i for factoring mod p_i is set to the maximum of 1 and

$$\text{round}\left(\frac{(n+p_i)\ np}{k\,n+\sum_{i=1}^{k} p_i\ \log n}\right)$$

for all but the last (largest) prime which gets all the remaining processes. For example, distributing 11 processes for the four primes 7, 11, 23, and 37, with $n = 8$ under this scheme results in 1, 2, 3, and 5 processes for each prime factoring respectively.

2.2. A Parallel Berlekamp Algorithm

The input to this algorithm is a prime p, a polynomial $u(x) = U(x) \mod p$ of degree n, and ps the number of processes assigned to this part of the computation.

To form Q−I, the computation of $x^{p\,i} \mod u(x)$ for $1 \le i \le n$ is done by a parallel shift-add procedure. The null space basis polynomial computation is done by a parallel column elimination on Q−I. The dimension r of the null space is produced, which equals the number of factors mod p. If $r = 1$, a shared global flag is set to cause all parallel processes, including those performing other prime factorings to terminate.

With $r > 1$ basis polynomials $v_i(x),\ 1 \le i \le r$, in increasing degree with $v_1 = 1$, we now perform the GCD computations

$$\gcd\left(f\left(x\right), v_j\left(x\right) - s\right)$$

for $f(x)$ a divisor of $u(x)$, all $v_j(x)$, $1 < j \leq r$ and all $0 \leq s < p$.

To parallelize the GCD computations, two lists, *facs* and *nfacs*, of factors of $u(x)$ are kept in shared memory. Initially *facs* contains only $u(x)$ and *nfacs* is empty. For each basis polynomial $v_j, j > 1$ the following is done.

One factor, called the *current factor*, is removed from *facs*. Each of the *ps* processes computes GCD's of the current factor (initially $u(x)$) with the current $v_j(x) - s$ for a distinct subset of s values in parallel. The union of the subsets covers all possible s values. Any factors found are deposited in the shared list *nfacs* until either the current factor is reduced to 1 or all *ps* processes are finished. By the end of this procedure, one or more factors whose product is equal to the current factor will have been put on *nfacs*. Now if *facs* is not empty then a new current factor is removed from *facs* and the procedure repeats. Otherwise, if *facs* is empty, then the values of *facs* and *nfacs* are interchanged and used with the next $v_j(x)$. The entire process is repeated until r factors are found.

2.3. Parallel Degree Reconciliation

After factoring modulo several different primes, a *degree compatibility* analysis can infer irreducibility (**?**) and deduce grouping of extraneous factors.

Let the *irreducible degree set* D_i be the set of degrees of the irreducible factors of $U(x)$ found modulo the prime p_i. From each D_i the *degree set* V_i of degrees of all *divisors* of $U(x)$ mod p_i can be formed. This is easily done by combining zero or more elements in D_i. For example, if D_1={1, 3, 4} then $V_1 = \{0, 1, 3, 4, 4, 5, 7, 8\}$. Because the irreducible degree set of $U(x)$ over $\mathbf{Z}$ is a subset of any V_i, it is contained in the intersection of all V_i. Let $\tilde{\mathrm{V}}$ be the intersection. If $\tilde{\mathrm{V}}$ is $\{0, n\}$ then $U(x)$ is irreducible over $\mathbf{Z}$ and our factoring algorithm terminates. Otherwise, $\tilde{\mathrm{V}}$ contains degrees of all irreducible factors of $U(x)$ over $\mathbf{Z}$. The $\tilde{\mathrm{V}}$ can then be used in an attempt to group extraneous factors mod p_i. For example, V_2={2,2,4} gives $\tilde{\mathrm{V}}$ ={0, 4, 8} which means the factors corresponding to 1 and 3 in D_1, or 2 and 2 in D_2, should be multiplied together into one factor for the subsequent lifting stage.

According to (**?**), the mean number of primes needed to establish the irreducibility of a random polynomial of degree up to 200 is less than 5. And the number of additional primes needed for larger polynomials grows very slowly with the degree. When applying this procedure in practice, just a few primes and a low number of factors are involved. Thus it costs very little and can enhance the overall factoring scheme.

3. Univariate p-adic Lifting

For univariate p-adic lifting, the overall strategy is to identify key computations in the best sequential univariate lifting procedure (**?**) and find ways to parallelize them. The input to the lifting procedure, consisting of a prime p, a primitive and squarefree polynomial $U(x)$ in $\mathbf{Z}[x]$, and $r \geq 2$ pairwise relatively prime polynomials $g_{i,0}(x)$ in $\mathbf{Z}_p[x]$, satisfying the congruence

$$U(x) = g_{1,0}(x)g_{2,0}(x)\cdots g_{r,0}(x) \quad mod\ p$$

The p-adic lifting outputs r factors $g_{i,k}(x)$ and a *final modulus* p^{k+1} such that

1 $U(x) = g_{1,k}(x)g_{2,k}(x)\cdots g_{r,k}(x) \quad mod\ p^{k+1}$
2 $g_{i,k}(x) = g_{i,0}(x) \quad mod\ p$
3 The final modulus exceeds a certain bound B (**?**) that is either specified in the input or derived from $U(x)$.

The results are obtained through a sequence of lifting steps each producing a congruence modulo a higher modulus.

$$U(x) = g_{1,j}(x)g_{2,j}(x)\cdots g_{r,j}(x) \quad mod\ p^{j+1},\ j > 0$$

Let's list the major computations involved in the lifting procedure.

1 *Lift Basis*: Obtain polynomials $\alpha_i(x)$ over $\mathbf{Z}_p$, with $deg(\alpha_i(x)) < deg(g_{i,0})$, such that

$$\alpha_1 F_1 + \alpha_2 F_2 + \cdots + \alpha_r F_r = 1 \quad mod\ p$$

where $F_i(x) = (g_{1,0} \cdots g_{r,0})/g_{i,0}$.

2 *Residue*: For lifting step $j > 0$, obtain the difference

$$C(x) = (g_{1,j-1} \cdots g_{r,j-1} - U)/p^j \quad mod\ p^{j+1}$$

Note that it can be arranged that $C(x)$ has coefficients in $\mathbf{Z}_p$ and $deg(C) < deg(U)$.

3 *Correction Coefficients*: Obtain $cc_i(x)$ in $\mathbf{Z}_p[x]$ by computing, in the field $\mathbf{Z}_p$,

$$cc_i(x) = C(x)\alpha_i \quad mod\ g_{i,0}(x), \quad i = 1, ..., r$$

4 *Updated Factors*: Compute $g_{i,j}(x)$ by correcting $g_{i,j-1}(x)$ for $i = 1, ..., r$

$$g_{i,j}(x) = g_{i,j-1} - p^j cc_i(x) \quad mod\ p^{j+1}$$

5 *True Factors*: As lifting proceeds, certain factors can lead directly to actual factors of $U(x)$ over $\mathbf{Z}$. Such factors can be detected and removed from the lifting process.

Step 1 is done only once. Then steps 2–4 are repeated to lift the modulus. When p^j becomes large enough Step 5 also joins the loop. Some key parallel steps are described. But this is only the *linear lifting* procedure. A *quadratic lifting* procedure which squares the modulus with each iteration also involves the *lifting of the lift basis* α_i, another subprocedure that is parallelized.

3.1. Parallel Polynomial Arithmetic

The regular multiplication algorithm for two polynomials of degree n is $O(n^2)$. The Karatsuba's algorithm is $O(n^{1.58})$ (**?**) but seems hard to parallelize effectively. The FFT-based polynomial multiplication is $O(n\ log(n))$ but is only suited for polynomials of high degree (say over 150). By using a parallel FFT/DFT scheme, the corresponding polynomial multiplication algorithm can also be parallelized.

It is often important in practice to have a parallel algorithm that involves low overhead and can be effective for multiplying smaller polynomials. An algorithm well-suited for implementation on SMPs involves computing the terms of the product polynomial in parallel using multiple tasks. The amount of work done by each task must also be balanced as much as possible. For dense polynomial with few missing terms, a static scheduling method can be used. Fig. **??** illustrates the idea using two quadratic polynomials, $A(x)$ and $B(x)$, and three parallel tasks. Each parallel task has a unique integer id `myid`.

$A(x) \times B(x)$				
myid=0	1	2	0	1
		$a_2 * b_0$	$a_1 * b_0$	$a_0 * b_0$
	$a_2 * b_1$	$a_1 * b_1$	$a_0 * b_1$	
$a_2 * b_2$	$a_1 * b_2$	$a_0 * b_2$		
x^4	x^3	x^2	x^1	x^0

Figure 1. Parallel Polynomial Multiplication

In general, multiplying $p(x)$ of degree dp and $q(x)$ of degree dq with np tasks, each

task (with $0 \leq myid < np$) computes its fair share of the terms in the product $ans(x)$. Assuming the coefficients of $ans(x)$ are initially zero, `parptimes` is a parallel task for multiplying $p(x)$ and $q(x)$.

```
Task parptimes (p(x), q(x), dp, dq)
{    n = dp + dq; /* degree of product */
     for ( deg = myid;  deg <= n; deg = deg + np )
         for ( i = MAX(0, deg - dq); i <= MIN(deg, dp); i++ )
             ans_deg = p_i * q_deg-i + ans_deg ;
}
```

Note p_i denotes the degree-i coefficient of $p(x)$. The parallel tasks compute disjoint sets of terms in $ans(x)$ and can run independently with no need for any synchronization. If we assume $dp >= dq$ then, for `parptimes`, the minimum grain size is dq coefficient multiplications and additions. This is the minimum amount of work for a task no matter how large np is. In fact, the optimal value is $np = dp + dq + 1$. Additional tasks don't increase the speed of `parptimes`. Because we don't have a large number of pe's and problems with small polynomials are done quickly anyway, `parptimes` can be used effectively in univariate p-adic lifting.

Modified slightly, the strategy also applies to sparse polynomials. Consider polynomials A and B in a sparse representation: $B = (e_1 \;\; P_1 \;\; ... \;\; e_m \;\; P_m)$ where e_i are exponents and P_i are coefficients or polynomials in other variables. The set of all possible terms are deduced dynamically in a *bag of terms to compute* for all parallel task. The scheme can be readily extended to compute a sum of polynomial products in the form:

$$\sum_i P_i \; * \; Q_i$$

The method is useful in multivariate p-adic lifting.

Another frequently used arithmetic operation in the lifting procedure is polynomial division to obtain the quotient and/or the remainder. Polynomial division repeats a sequence of multiplication (the divisor by a coefficient C) and subtract operations until the remainder's degree falls below that of the divisor. One effective way to parallelize polynomial division is to apply all processes to perform the multiplication and subtraction, synchronizing before each new C value.

3.2. The Lift Basis

Computing the *lift basis* $\alpha_i(x)$ is the first major step for both the linear and the quadratic lifting. The lifting basis is set up at the beginning and used throughout linear lifting. However, it must be updated at each stage of the quadratic lifting algorithm.

If there are r factors to lift, then the products

$$P_i = g_{i+1,0} \cdots g_{r,0}, \quad i = 0, ..., r-1$$

are needed to compute polynomials $a_i(x)$, $b_i(x)$ such that

$$a_i \, g_{i,0} + b_i \, P_i = 1 \qquad mod \; p.$$

Because $g_{i,0}$ are relatively prime, the a_i and b_i can be computed by a well-known polynomial extended gcd (`pxgcd`) algorithm (**?**). At present, the best way to parallel the lift basis computation is to use parallel polynomial arithmetic operations to compute the α_i sequentially.

3.3. Correction Coefficients

Because the degree of the residue $C(x)$ can be as high as $deg(U) - 1$, we can reduce its degree and simplify subsequent computation by first computing in parallel

`for (i = myid; i < r; i++)` $tmp_i(x) = C(x) \mod g_{i,0}(x)$ *in* $\mathbf{Z}_p$

then use `parptimes` to obtain the products

`for (i = 1; i < r; i++)` $tmp_i(x) =$ `parptimes`$(tmp_i(x), \alpha_i(x))$ *in* $\mathbf{Z}_p$

All r of the $cc_i(x)$ can then be obtain in parallel

`for (i = myid; i < r; i++)` $cc_i(x) = tmp_i(x) \mod g_{i,0}(x)$ *in* $\mathbf{Z}_p$

A mixture of two different kinds of parallelism is used: (A) doing all r items in parallel, and (B) employing all available tasks to perform polynomial arithmetic. The general strategies are applicable in many other situations. Strategy A is dependent on the number of factors r. If r is small, 2 or 3 say, then the parallelism is limited. Approach B avoids the r limitation but parallelizes at a finer grain size. It can be very effective if the polynomials involved in the arithmetic operations are not too small. A mixed approach combines the advantages of both by dividing all available processes into r groups.

3.4. Adjusting Lift Basis

Updating the factors and testing for the formation of true factors can be done in parallel by processing all factors at once. When true factors are detected, they are removed and the remaining factors are lifted further. For the reduced lifting problem, a different lift basis is needed. An algorithm is devised to compute a new lift basis by simply deriving it from the old one. This new algorithm has also been parallelized.

Before describing the general algorithm for adjusting the lift basis, let's consider a simple example. Consider lifting three factors ($r = 3$) with the lift basis

$$\alpha_1 g_{2,0} g_{3,0} + \alpha_2 g_{1,0} g_{3,0} + \alpha_3 g_{1,0} g_{2,0} = 1 \mod p$$

Suppose a true factor corresponding to a lifted image of $g_{3,0}(x)$ is found. Now we remove the third factor from the picture and continue to lift the other two. Thus a new basis, $a(x)$ and $b(x)$ is needed such that

$$a(x) g_{2,0}(x) + b(x) g_{1,0}(x) = 1 \mod p$$

The $a(x)$ and $b(x)$ can be derived from $\alpha_1(x)$ and $\alpha_2(x)$ as follows.

$$a(x) = \alpha_1(x) g_{3,0}(x) \mod g_{1,0}(x)$$

$$b(x) = \alpha_2(x) g_{3,0}(x) \mod g_{2,0}(x)$$

with the computations done modulo p. In general, taking both linear and quadratic lifting into account, the current lift basis $\alpha_i(x)$ satisfies a congruence

$$\alpha_1 F_1 + \alpha_2 F_2 + \cdots + \alpha_r F_r = 1 \mod p^s$$

where $s \geq 1$ and $F_m(x)$ is the product of all lifted factors $g_{i,s}(x)$, $i \neq m$. Without loss of generality assume the true factors found correspond to factors h through r ($r > h > 1$). So α_i for $1 \leq i < h$ will be adjusted. Let $GP(x)$ denote the product of $g_{h,s}$ through $g_{r,s}$. We have

$$\alpha_i(x) = \alpha_i(x)\, GP(x) \mod g_{i,s}(x) \tag{3.1}$$

for $i = 1, ..., h-1$. A proof based on the same principle as that for the example involving three factors can show that these $\alpha_i(x)$ indeed form the new lift basis. When $U(x)$ is not monic, special care must be taken to treat the leading coefficients in the adjustment procedure.

The method represented by Eq. **??** can easily be parallelized by finding the $\alpha_i(x)$ in parallel and by parallel polynomial arithmetic.

4. Multivariate p-adic lifting

The parallelization of multivariate p-adic lifting is useful in both factoring and GCD. The variable-by-variable EEZ lifting algorithm with a recursive correction coefficient (**?**) procedure is parallelized. Key parallel steps include computing the residue, extracting coefficients of terms, building correction coefficients, and updating factors.

Let $U(x_1, x_2, ..., x_k) \in \mathbf{Z}[x_1, x_2, ..., x_k]$ be an integral polynomial in k variables. The *main variable* is $x = x_1$. U is squarefree and primitive with respect to x. Let $\{a_2, ..., a_k\}$ be a set of integers such that the univariate polynomial $U_1 = U(x, a_2, ..., a_k)$ stays squarefree and $deg(U_1) = deg(U)$ in x. In general, U_i denotes $U(x, ..., x_i, a_{i+1}, ...a_k)$.

Let n_i be the degree of U in x_i, S_i the ideal $((x_i - a_i)^{n_i+1})$, and s_i the ideal $(x_i - a_i, ..., x_k - a_k)$. Given the congruence

$$U(x, ..., x_k) \equiv u_{1,1}(x)\ u_{1,2}(x)\ \cdots\ u_{1,r}(x)\ \ mod\ s_2, \tag{4.1}$$

where the $u_{1,j}$ are $r \geq 2$ factors of U_1, the EEZ algorithm (**?**) lifts all factors, adding one variable at a time, through a sequence of congruences,

$$U \equiv u_{2,1}(x, x_2)\ \cdots\ u_{2,r}(x, x_2)\ \ mod\ \ (S_2, s_3),$$

$$U \equiv u_{3,1}(x, x_2, x_3)\ \cdots\ u_{3,r}\ \ mod\ \ (S_2, S_3, s_4),$$

and so on until $u_{i,j}$ satisfying

$$U = u_{k,1}(x, ..., x_k)\ \cdots\ u_{k,r}\ \ mod\ \ (S_2, ..., S_k) \tag{4.2}$$

are obtained. For all valid indices i and j, the relation

$$u_{i,j} = u_{i+1,j}(x, .., x_i, a_{i+1})$$

holds. To simplify computations, coefficient arithmetic is performed modulo a suitably selected large prime.

4.1. Key Parallel Steps

The input to the lifting procedure, consisting of the multivariate polynomial $U(x_1, ..., x_k)$, the univariate polynomials $u_{1,j}$, and the evaluation values a_i, satisfies Eq. **??**. Lifting outputs the multivariate polynomials $u_{k,j}$ satisfying the congruence **??**. The congruences as given above, are computed sequentially. This is dictated by the *variable-by-variable* strategy and seems hard to improve. However, much parallelism can be found within each step.

The main computations involved in lifting all r factors $u_{i-1,j}$ to $u_{i,j}$ (introducing the next variable) are:

1 Computing the *residue*: (initially $u_{i,j} \leftarrow u_{i-1,j}$)

$$R(x_1, ..., x_i) = \prod_{j=1}^{r} u_{i,j}\ \ -\ \ U_i$$

2 Extracting the *term coefficient*: The coefficient of the $(x_i - a_i)^e$ term ($i > 1$ and $e > 0$) in $R(x_1, ..., x_i)$ is the polynomial C

$$C(x_1, ..., x_{i-1})\ \ =\ \ \frac{1}{e!}\ \frac{\partial^e R}{\partial x_i^e}\ \ at\ x_i = a_i \tag{4.3}$$

3 Obtaining the *correction polynomials*: To calculate r polynomials $\alpha_j(x_1, ..., x_{i-1})$ such that

$$\alpha_1 F_{i-1,1} + \cdots + \alpha_r F_{i-1,r} = C(x_1, ...x_{i-1}) \tag{4.4}$$

where

$$F_{i,m}\ \ =\ \ \prod_{j \neq m} u_{i,j},\quad 1 \leq m \leq r$$

The α_i are also computed with a variable-by-variable *p-adic* lifting performed on the congruence **??**.

4 Updating the factors: Each factor is updated

$$u_{i,j} \leftarrow u_{i,j} \ - \ (x_i - a_i)^e \alpha_j$$

And now the $u_{i,j}$ satisfy

$$U_i \equiv \prod_{j=1}^{r} u_{i,j} \ \ mod \ \ (S_2, ..., S_{i-1}, (x_i - a_i)^{e+1}).$$

The steps are performed for $e = 1, 2, ..., n_i$.

In practice, steps 1 and 2 are performed at once. R is computed by parallel arithmetic without generating terms lower than x_i^e. Meanwhile, differentiation and evaluation (Eq. **??**) are carried out on terms to produce C. All factors in step 4 can be updated in parallel as well. Let's consider step 3 next.

4.2. Parallel Computation of Correction Polynomials

Of all the main steps in multivariate p-adic lifting, computing multivariate correction polynomials, (Eq. **??**), is by far the most dominant step. In general, the computation itself involves further p-adic liftings. Given r polynomials $F_{i,j}$ and C, to compute α_j such that

$$\alpha_1 F_{i,1} + \cdots + \alpha_r F_{i,r} = C(x_1, ..., x_i) \tag{4.5}$$

we use a recursive algorithm (RC) (**?**) that builds the desired α_j in i variables by first solving the same problem in $i-1$ variables:

$$\alpha_1 F_{i-1,1} + \cdots + \alpha_r F_{i-1,r} = C(x_1, ..., x_{i-1}, a_i) \tag{4.6}$$

Then the α_j in Eq. **??** can be lifted to obtain those satisfying Eq. **??**. Of the two ways to lift for the α_j (**?**), the iterative method (algorithm IC) is easy to parallelize but grossly inefficient even in parallel. The recommended procedure to parallelize is algorithm RC.

Key Steps to Parallelize in Algorithm RC

RC-1 Difference: $W \leftarrow C(x_1, ..., x_i) - \sum_j (\alpha_j * F_{i-1,j})$

RC-2 Term Coefficient: $Tc(x_1, ..., x_{i-1}) \ = \ \frac{1}{e!} \ \frac{\partial^e W}{\partial x_i^e} \ \ at \ x_i = a_i$, where $e > 0$.

RC-3 Corrections for α_j: By a recursive call to RC, compute polynomials cc_j such that

$$cc_1 F_{i-1,1} + \cdots + cc_r F_{i-1,r} = Tc(x_1, ...x_{i-1})$$

RC-4 Updating α_j: $\alpha_j \leftarrow \alpha_j + cc_j(x_1, ..., x_{i-1})(x_i - a_i)^e$

RC-5 Updating W: $W \leftarrow W - \sum_j (cc_j * (x_i - a_i)^e * F_{i,j})$

Step RC-1 is a parallel sum of products computation (Section **??**). Step RC-2 is a parallel term coefficient calculation (Section **??**). Step RC-3 involves a recursive call, so if algorithm RC is parallelized this call is executed in parallel as well. Now we can focus on steps RC-4 and RC-5.

For step RC-4 we again update all α_j in parallel using all np tasks. But the parallel computation in this step also sets new values for cc_j:

$$cc_j \leftarrow cc_j(x_1, ..., x_{i-1})(x_i - a_i)^e$$

in preparation for step RC-5.

Updating W now involves parallel sum of products (Section **??**).

5. Sparse Multivariate GCD

Mohamed Rayes worked on a new parallel GCD algorithm (BSMGCD), using a divide-and-conquer strategy for sparse multivariate polynomials. The computation of an n-variable GCD is recursively divided into two independent GCD subproblems each with half the number of variables. Parallel solutions of the subproblems combine to give the desired GCD. BSMGCD efficiently exploits sparseness in all the variables early in the solution process.

5.1. Exploiting Sparseness

Consider computing $G = \gcd(U, V)$ for polynomials U, $V \in \mathbf{Z}[x_1, \ldots, x_t]$, where G is sparse. Assume that U, V are squarefree and primitive with respect to all variables.

Zippel (**?**) first suggested the key idea for exploiting sparseness in multivariate GCD computations. Basically, when the polynomials are sufficiently sparse, the probability of terms dropping is very small when substituting randomly generated values for a subset of the variables.

Zippel's algorithm (SMGCD) begins by selecting an *evaluation point* $(a_1, a_2, \cdots, a_t)$ and proceeds to produce the sequence of polynomials:

$$G_1 = G(x_1, a_2, a_3, \cdots, x_t), G_2 = G(x_1, x_2, a_3, \cdots, a_t), \cdots, G_t = G = G(x_1, x_2, \cdots, x_t).$$

If G is sparse, then these polynomials have many missing terms. A missing term in G_i is either absent in G or vanishes at the evaluation point $(a_1, a_2, \cdots, a_t)$. A point $(a_1, a_2, \cdots, a_t)$ is *a good evaluation point* if for all $1 \leq i < v$,

$$G_i = G(x_1, x_2, \cdots, x_i, a_i + 1, \cdots, a_t)$$

contains the maximum number of distinct terms of all possible evaluations. Informally, a good evaluation point preserves the structure of the target GCD. This is the key idea in the SMGCD algorithm. If the starting point $(a_1, a_2, \cdots, a_v)$ is chosen at random from a large set of distinct elements, then the possibility that the initial chosen point is a good evaluation point is quite large (**?**).

Thus, 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 $\mathbf{Z}_p[x_1]$, then recovers the lost variables one at a time, using 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 is used to coalesce the results.

Various strategies to parallelize SMGCD have been presented in (**?**). We now consider the new BSMGCD parallel algorithm.

5.2. The BSMGCD Parallel Algorithm

Given the polynomials U and V in t variables, the BSMGCD algorithm (**?**) performs the following steps for any suitable large (single-precision) prime p:

B-1 Evaluation: generating random t-tuples $[a_1, ..., a_t]$ in $\mathbf{Z}_p^t$
B-2 Univariate GCD: computes one-variable GCDs mod p
B-3 Interpolation: recovers lost variables through sparse interpolation (Fig. **??**).
B-4 True GCD: when p is not large enough, the final GCD, with all t variables recovered, is used to recover the actual GCD over $\mathbf{Z}$.

The univariate GCDs are computed in parallel and assumed to have *no missing terms*. The one-variable answers are combined in pairs (in parallel) by sparse interpolation to form answers with two variables, etc. Having established which coefficients are non-zero, sparse interpolation uses a number of point-value pairs equal to the number of non-zero coefficients to determine the coefficients by solving a linear system in parallel.

Multiple primes can be used if necessary and incremental Chinese Remaindering can be applied to obtain the actual GCD.

Let's illustrate BSMGCD by an example involving four variables. Suppose we want to compute

$$G = \gcd(U(x, y, z, w), V(x, y, z, w)) \text{ over } \mathbf{Z}$$

as shown in Fig.**??**. The algorithm computes the polynomials

$$G_x = G(x, y_0, z_0, w_0) \mod p, \qquad G_y = G(x_0, y, z_0, w_0) \mod p,$$

$$G_z = G(x_0, y_0, z, w_0) \mod p, \qquad G_w = G(x_0, y_0, z_0, w) \mod p,$$

where the evaluation tuple $[x_0, y_0, z_0, w_0]$ is randomly chosen from the set $\mathbf{Z}_p^4$. These polynomials can be computed using a straightforward algorithm for two univariate polynomials over a finite field (e.g., the Euclidean algorithm).

We assume that no terms in G vanish at y_0, z_0, w_0. and the polynomial G_x has k_x terms. The same assumption is made for G_y, G_z, and G_w. Next, the algorithm computes the bivariate images $G_{x,y} = G(x, y, z_0, w_0)$ and $G_{z,w} = G(x_0, y_0, z, w)$. To compute

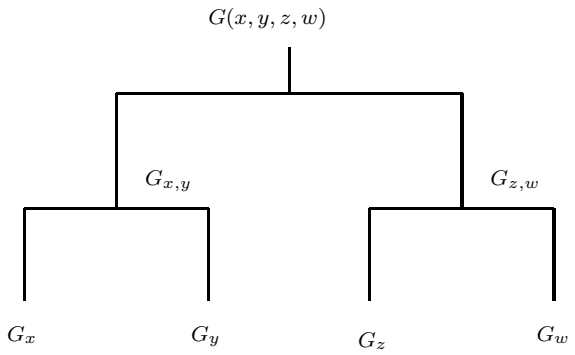

Figure 2. Parallel Computation of $G(x, y, z, w)$

$G_{x,y}$, for instance, observe that the monomials of $G_{x,y}$ must be some combination of the individual monomials of G_x and G_y. This is so since G_x and G_y have no missing terms. For example, if $G_x = x^3 + 1$, and $G_y = y^2 + 2$, it follows that $G_{x,y}$ must be of the form

$$G_{x,y} = G(x, y, z_0, w_0) = x^3\,(a_1\,y^2 + a_2) + a_3\,y^2 + a_4,$$

where the a_i are unknown integer coefficients. These unknowns can be determined by setting up a linear system whose left hand sides are coefficients of $G_{x,y}$, evaluated at a number of points (in this example $y = y_1$ and $y = y_2$, say) and whose right hand sides are the integer coefficients of terms in the corresponding GCD's. For this example, let

$$D_1 = gcd(U(x, y_1, z_0, w_0), V(x, y_1, z_0, w_0))$$

and

$$D_2 = gcd(U(x, y_2, z_0, w_0), V(x, y_2, z_0, w_0)).$$

Then, the following linear systems are solved mod p.

$$y_1^2\ a_1 + a_2 = \text{coeff}(D_1, x^3), \quad y_2^2\ a_1 + a_2 = \text{coeff}(D_2, x^3)$$

$$y_1^2\ a_3 + a_4 = \text{coeff}(D_1, x^0), \quad y_2^2\ a_3 + a_4 = \text{coeff}(D_2, x^0)$$

Note it is possible to use the points $y = y_0$ and $x = x_0$ to save computation in this case.

The same process is applied to determine the coefficients of $G_{z,w}$. From $G_{x,y}$ and $G_{z,w}$, G can be computed similarly.

6. Summary and Conclusions

Key aspects of parallel polynomial factoring and GCD are reviewed. The algorithms have been designed and implemented for SMPs and the investigations focused on medium to coarse grain parallelism. In many cases, critical steps in suitable sequential algorithms have been parallelized. The BSMGCD, however, involves a new parallel algorithm. Sometimes, as in *adjusting the lift basis*, the parallelization effort can lead to improvements in the basic sequential algorithm.

Experiences and timing data obtained in (**?**, **?**, **?**) as well as other reports indicate that SMP-based implementations produce good speed up when np is small. As the number of processes gets larger the parallel programs often become less effective. The relative high cost of process creation and memory access contention are the main problems. Key factors for improved SMP performance are

1 Efficient operating system support for parallel light-weight processes or threads
2 Larger per-pe memory cache and faster access to shared memory
3 Improved parallel programming environments that allow simple and direct coding of *recursive parallel routines* and *fluid grouping, regrouping, and subgrouping of parallel processes* for different parts of the problem.

7. Future Work

The univariate cases have been more fully investigated. For multivariate polynomials, aspects yet to be tackled include: selecting good evaluation points, determining leading coefficients, early detection of extraneous factors, deducing other coefficients while lifting.

A special p-adic lifting technique is being investigated which applies to factors with no missing terms (NMT). The method lifts coefficients of the terms by solving linear equations. A successful NMT p-adic lifting procedure can avoid multiple CRAs in BSMGCD, for example.

Another promising area of parallelism for computer algebra is offered by a network of heterogeneous processors consisting of high-speed workstations, small-scale parallel processors of different architectures, massively parallel processors, and even super computers. Investigations at Kent are looking into this from two points of view:

The Multi Protocol(**?**): establishing a standard way for mathematical compute engines to exchange mathematical data and cooperate on a distributed basis.

PVM (parallel virtual machine)(**?**): utilizing PVM as a convenient way to launch distributed applications and achieve coarse-grain parallelism for symbolic and scientific computations.

Software tools in these areas are being completed (**?**, **?**) to make parallel and distributed computing in symbolic computation easier and to allow symbolic systems to run as servers in powerful problem solving environments.

References

Beauzamy, B., Trevisan, V. and Wang, P. (1993). *Polynomial Factorization: Sharp Bounds, Efficient Algorithms*. Journal of Symbolic Computation, Vol. 15, 393–413.

Buchberger, B. Collins, G. E. et el. (1993) *SACLIB 1.1 User's Guide*. RISC-Linz Technical Report No. 93-19, Research Institute for Symbolic Computation, Kepler University, A-4040 Linz, Austria.

Char, B. Geddes, K. and Gonnet, G. (1989) GCDHEU: Heuristic Polynomial GCD Algorithm Based on Integer GCD Computation. *J. Symb. Comp.*, **7**, 31–48,

Dora, D. and Fitch, J. ed. (1988) *Computer Algebra and Parallelism*, San Diego, CA: Academic Press,

Geist, A., et el. (1993) *PVM 3 User's Guide And Reference Manual*. ORNL/TM-12187, Oak Ridge National Laboratory.

Granlund, T. (1991) *GNU MP: The GNU Multiple Precision Arithmetic Library*, Free Software Foundation.

Gray, S., Kajler, N., and Wang, P., (1996) "Design and Implementation of MP, a Protocol for Efficient Exchange of Mathematical Expressions", *J. Symb. Comp.*, to appear.

Knuth, D. (1980) *The Art of Computer Programming.* Vol. 2:*Semi-numerical Algorithms,* Addison-Wesley, Reading, Mass.

Moenck, R. (1976) Practical Fast Polynomial Multiplication. Proceedings, SYMSAC'76, 136–145.

Moore, P. and Norman, A. (1981) Implementing a Polynomial Factorization and GCD Package. Proceedings, 1981 ACM Symposium on Symbolic and Algebraic Computation, 109–116.

Moses, J. and Yun, D. (1973) The EZGCD Algorithm. Proceedings, ACM National Conference, 159–166.

Musser D. (1976) Multivariate Polynomial Factorization. JACM **22**, 291–308.

Rayes, M., Weber, K., and Wang, P. (1994) Parallelization of The Sparse Modular GCD Algorithm for Multivariate Polynomials on Shared Memory Multiprocessors, Proceedings, ISSAC'94, Oxford, UK, 66–73,

Rayes, M. and Wang P. (1994) Parallel GCD for Sparse Multivariate Polynomials]on Shared Memory Multiprocessors, Proceedings, PASCO'94 RISC/Linz, 326–335,

Serpette, B., Vuillemin, J, and Hervé, J. (1985) BigNum: A Portable and Efficient Package for Arbitrary-Precision Arithmetic. Digital Equipment Corp., Paris Research Laboratory, Av. Victor Hugo. 92563

Wang, P. and Trager, B. (1979) New Algorithms for Polynomial Square-free Decomposition over the Integers. SIAM J. Computing, **8.3**, 300–305.

Wang, P. (1978) An Improved Multivariate Polynomial Factoring Algorithm. Mathematics of Computation, Vol. 32, No. 144, 1215–1231.

Wang, P. (1979) Analysis of the p-adic Construction of Multivariate Correction Coefficients in Polynomial Factorization: Iteration vs. Recursion. Lecture Notes in Computer Science, **72**, , Springer-Verlag, 291–300.

Wang, P. (1990) Parallel Univariate Polynomial Factorization on Shared-Memory Multiprocessors. Proceedings of the ISSAC'90, Addison-Wesley , 145–151.

Wang, P. (1991) Symbolic Computation and Parallel Software. Proceedings, First International Conference of the Austrian Center for Parallel Computation. Springer-Verlag Lecture Notes in Computer Science, *Parallel Computation*, **591**, 316–337.

Wang, P., (1992) Parallel Univariate p-adic Lifting on Shared-Memory Multiprocessors. Proceedings, ISSAC'92, 168–176.

Wang, P. (1994) Parallel Polynomial Operations: a progress report. Proceedings, PASCO'94, RISC/Linz, 394–404.

Wang, P. (1996) Tools to Aid PVM Users, PVM Users Meeting, Feb. 26–27, Albuquerque, New Mexico, (`http://monkey.mcs.kent.edu/ pwang/`).

Zassenhaus, H. (1969) On Hensel Factorization. *J. Number Theory* **1** 291–311.

Zippel, R. ed. (1990) *Computer Algebra and Parallelism*, Proceedings, Second International Workshop, Ithaca, USA, Springer-Verlag.

Zippel, R. (1979) *Probabilistic Algorithms for Sparse Polynomials*. PhD. Thesis, Massachusetts Institute of Technology.