Parallel Computation of Modular Multivariate Polynomial Resultants on a Shared Memory Machine *

Hoon Hong and Hans Wolfgang Loidl

Research Institute for Symbolic Computation
Johannes Kepler University, A-4040 Linz, Austria
e-mail: {hhong,hwloidl}@risc.uni-linz.ac.at

Abstract. This paper reports our experience in parallelizing a modular algorithm for computing multivariate polynomial resultants over $\mathbb{Z}_p$. The modular algorithm has the well-known scheme of "divide-conquer-combine" where the "conquer" phase is straightforwardly parallelizable. But the "combine" phase is structurally sequential, and requires certain modifications for efficient parallelization. We describe and compare various different parallelization schemes (in particular for the combine phase). The variants of the algorithm have been implemented on top of the PACLIB kernel which provides C-primitives for task creation and non-deterministic wait on a shared memory machine.

1 Overview

Multivariate polynomial resultants play a fundamental role in commutative algebra, and it is one of the main operations in computation with algebraic numbers [10], topological study of algebraic curves [4], geometric modelling, etc. In this paper, we report our work on computing resultants over $\mathbb{Z}_p$ on a shared memory machine, by using a modular algorithm with dense interpolation [3].

This method has the basic structure of "divide-conquer-combine". Two multivariate polynomials are mapped into sufficiently many homomorphic images by evaluating them at certain points. The resultants in the homomorphic images are computed by recursively applying this "divide-conquer-combine" algorithm until the univariate case is reached. The "combine" is done by interpolating the results in the homomorphic images to the one in the original domain.

The "conquer" phase is straightforwardly parallelizable since it consists of several independent tasks. But the "combine" phase of Collins' algorithm [3] is structurally sequential due to a certain restriction (one of the modules must be linear) imposed by his efficient interpolation method.

In this paper we compare four versions of the algorithm, mainly differing in the parallelization of the "combine" phase:

* Supported by Austrian Science Foundation on Parallel Symbolic Computation (S5302-PHY)

- The **Binary-Tree Combine** algorithm (Sect. 5) uses a binary-tree structure of interpolations in the "combine" phase.
- The **Vandermonde Combine** algorithm (Sect. 6) uses Vandermonde (also called Lagrange) interpolation to compute the coefficients of the resultant in parallel.
- The **Coefficient-wise Combine** algorithm (Sect. 7) uses parallelized binary interpolation algorithms in the "combine" phase.
- The **Parallel Divide** algorithm (Sect. 8) also performs the "divide" phase in parallel by using speculative parallelism. The "combine" phase is the same as in the **Vandermonde Combine** algorithm.

Our work reported here has been influenced and also benefited by the previous works by Küchlin [8] on multivariate polynomial greatest common divisors on a shared memory machine, the works by Paul Wang [16] on polynomial factorization on a shared memory machine, and the work of Seitz [14] on multivariate polynomial resultant computation on workstation networks.

2 Problem Statement

Let I be an integral domain, and let $A = \sum_{i=0}^{m} a_i x^i$ and $B = \sum_{i=0}^{n} b_i x^i$ be non-zero polynomials over I. The *Sylvester matrix* of A and B, written as $M_{A,B}$, is the $m+n$ by $m+n$ matrix

$$M_{A,B} = \begin{bmatrix} a_m & a_{m-1} & \cdots & a_0 & & & \\ & \cdots & \cdots & \cdots & \cdots & & \\ & & & a_m & a_{m-1} & \cdots & a_0 \\ b_n & b_{n-1} & \cdots & b_0 & & & \\ & \cdots & \cdots & \cdots & \cdots & & \\ & & & b_n & b_{n-1} & \cdots & b_0 \end{bmatrix}$$

where the upper part of M consists of n rows of the coefficients of A, the lower part consists of m rows of the coefficients of B, and all entries not shown are zero. Then the *resultant* of A and B, written as $\mathrm{res}(A, B)$, is defined by

$$\mathrm{res}(A, B) = \det(M_{A,B}) \ .$$

Now we are ready to state the problem precisely.

Problem 1. Devise a parallel algorithm or parallelize a sequential algorithm with the following specification:

INPUT: $A, B \in \mathbb{Z}_p[x_1, \cdots, x_{r-1}][x_r]$, p a prime.

OUTPUT: $\mathrm{res}(A, B) \in \mathbb{Z}_p[x_1, \cdots, x_{r-1}]$. □

3 Sequential Algorithm

In this section, we shortly review the modular resultant computation algorithm by giving the theorems upon which it is based. For the details and proofs, see [3].

Let $A, B \in \mathbb{Z}_p[x_1, \ldots, x_r]$. If $r = 1$, then we can easily obtain their resultant from their natural polynomial remainder sequence (**ResultantUniVar**). From now on, we assume that $r \geq 2$. The main idea is to divide the resultant computation in $\mathbb{Z}_p[x_1, \ldots, x_r]$ into several resultant computations in $\mathbb{Z}_p[x_2, \ldots, x_r]$, to conquer these sub-problems recursively, and then to combine them into the final result ("recursive divide-conquer-combine"). Below we elaborate the idea.

Definition 2 (Evaluation). *Let $v \in \mathbb{Z}_p$. A mapping*

$$\Phi_v : \mathbb{Z}_p[x_1, \ldots, x_r] \longrightarrow \mathbb{Z}_p[x_2, \ldots, x_r]$$

such that $\Phi_v(A(x_1, \ldots, x_r)) = A(v, x_2, \ldots, x_r)$ is called an evaluation *on v.* □

Definition 3 (Lucky evaluation). *An evaluation Φ is called* lucky *for $A \in \mathbb{Z}_p[x_1, \ldots, x_r]$ iff $\deg_{x_r}(A) = \deg_{x_r}(\Phi(A))$.* □

Lemma 4 (Commutativity). *Let $A, B \in \mathbb{Z}_p[x_1, \ldots, x_r]$ and let Φ be a lucky evaluation for A and B. Then Φ and the resultant operation commute, that is,* $\mathrm{res}(\Phi(A), \Phi(B)) = \Phi(\mathrm{res}(A, B))$. □

Lemma 5 (Interpolation). *Let $v_1, \ldots, v_k$ be distinct elements of $\mathbb{Z}_p$ and let $C_1, \ldots, C_k \in \mathbb{Z}_p[x_2, \ldots, x_r]$, $k \geq 1$. Then there exists a unique element $C \in \mathbb{Z}_p[x_1, \ldots, x_r]$ such that $\Phi_{v_i}(C) = C_i$ for $1 \leq i \leq k$ and $\deg_{x_1}(C) < k$. We will denote C by $\Omega(C_1, \ldots, C_k; v_1, \ldots, v_k)$.* □

Lemma 6 (Degree bound). *Let $\mathbf{Bound}(A, B)$ denote the following expression: $\deg_{x_r}(A) \deg_{x_1}(B) + \deg_{x_r}(B) \deg_{x_1}(A)$. Then $\mathbf{Bound}(A, B)$ is an upper bound for the degree of $\mathrm{res}(A, B)$ i.e. $\mathbf{Bound}(A, B) \geq \deg_{x_1}(\mathrm{res}(A, B))$.* □

Putting these three lemmas together, we immediately obtain the following theorem, which gives the general structure of an algorithm for computing resultants.

Theorem 7 (Modular method). *Let $A, B \in \mathbb{Z}_p[x_1, \ldots, x_{r-1}][x_r]$, $r \geq 2$. Let $k = 1 + \mathbf{Bound}(A, B)$. Let $v_1, \ldots, v_k$ be distinct elements of $\mathbb{Z}_p$ such that Φ_{v_i}'s are lucky evaluations for A and B. Let $A_i = \Phi_{v_i}(A)$ and $B_i = \Phi_{v_i}(B)$. Let $C_i = \mathrm{res}(A_i, B_i)$. Let $C = \Omega(C_1, \ldots, C_k; v_1, \ldots, v_k)$. Then $C = \mathrm{res}(A, B)$.* □

An iterative application of the following theorem (a constructive version of Lemma 5) immediately provides an efficient algorithm for the function Ω.

Theorem 8 (Special Binary Interpolation). *Let C_i, v_i, and Ω be as defined in Lemma 5. Let $C' = \Omega(C_1, \ldots, C_{k-1}; v_1, \ldots, v_{k-1})$. Let $M = \prod_{i=1}^{k-1}(x_1 - v_i)$. Then we have*

$$\Omega(C_1, \ldots, C_k; v_1, \ldots, v_k) = C' + M \frac{C_k - \Phi_{v_k}(C')}{\Phi_{v_k}(M)} \quad .$$

□

We put this theorem into a function **SpecialInterpolate**, which takes C', C_k, M, v_k as input and yields $\Omega(C_1,\ldots,C_k; v_1,\ldots,v_k)$ as output. The interpolation is called "special" because it restricts one of the modules to be linear, $x_1 - v_k$.

4 Programming Environment

As programming environment we assume a dynamic memory manager providing lists and automatic garbage collection. We also assume a task (light-weight process) manager providing the following parallel primitives:

- **CreateTask**(*task-list*, $f, x_1, \ldots, x_n$)
  Creates a new parallel task, appends it to *task-list*, and immediately returns. The newly created task will eventually compute the function $f(x_1,\ldots,x_n)$.
- $y \leftarrow$ **WaitResult**(*task-list*)
  Waits non-deterministically until a task among *task-list* completes computation, removes the task from *task-list* and returns the result as y.

The current implementation uses the PACLIB kernel [6, 5], which is a parallel interface to the SACLIB computer algebra library [1]. Thus, all the algorithms were programmed in SACLIB, and parallelized using the PACLIB kernel. We believe that the algorithms can be easily ported to other platforms providing the above functionalities such as PARSAC [7], PAC [12], ||MAPLE|| [15] and Maple/Linda [2].

In the following sections we describe various parallelizations of a sequential algorithm outlined in Sect. 3 with a parallel "combine" phase (Figs. 2 to 5).

The left picture (the *task picture*) of the two pictures at the bottom of Figs. 2 to 5 shows exactly one possibly interrupted horizontal line for each parallel task. This is independent of the processor that computes this task. The x-axis in this picture represents time and the y-axis shows the tasks in the order of their creation. The right picture (the *utilization picture*) shows the utilization of the processors during the execution. Thus, the x-axis again represents time and the y-axis represents the number of active processors at a certain moment.

5 Binary-Tree Combine

A straightforward idea for parallelizing the "combine" phase is to replace the sequential linear structure of the computation by a binary-tree structure. For that structure a general and therefore more costly binary interpolation method must be used.

Theorem 9 (Binary Interpolation). *Let*

$$\begin{aligned} C^{(1)} &= \Omega(C_1,\ldots,C_\ell; v_1,\ldots,v_\ell) \\ C^{(2)} &= \Omega(C_{\ell+1},\ldots,C_k; v_{\ell+1},\ldots,v_k) \\ M^{(1)} &= (x_1 - v_1)\cdots(x_1 - v_\ell) \\ M^{(2)} &= (x_1 - v_{\ell+1})\cdots(x_1 - v_k) \ . \end{aligned}$$

Then we have

$$\Omega(C_1,\ldots,C_k;v_1,\ldots,v_k) = C^{(2)} + M^{(2)}\left[\frac{C^{(1)}-C^{(2)}}{M^{(2)}}\right]_{\bmod M^{(1)}} .$$

□

We put this theorem into the function **Interpolate**, which takes $C^{(1)}, C^{(2)}, M^{(1)}$, $M^{(2)}$ as input and yields $(\Omega(C_1,\ldots,C_k;v_1,\ldots,v_k), M^{(1)}M^{(2)})$ as output.

The **Binary-Tree Combine** algorithm in Fig. 2 implements this binary-tree structure in Step 5. Although this structure offers a high degree of parallelism in the first levels, the sequential computation in the root of the tree is a severe bottleneck, as the inputs for this last call to **Interpolate** are very big and all other **Interpolate** tasks have to terminate before the last task can start.

The task and utilization pictures of Fig. 2 show $\lceil \log_2 19 \rceil = 5$ levels of **Interpolate** tasks of increasing length, for the 19 evaluation points. The final interpolation requires more than 60 % of the overall computation time. This final bottleneck and the usage of the less efficient general interpolation algorithm **Interpolate** are the main reasons for the poor efficiency of this algorithm.

6 Vandermonde Combine

Another idea for parallelizing the "combine" phase is based on Theorem 10. Due to this theorem it is possible to compute each of the coefficients of the resultant in parallel, if one has computed the inverse Vandermonde matrix, which is built from the powers of the evaluation points v_j. The i-th coefficient of the resultant is then computed by forming the inner product of the vector of the homomorphic resultants with the i-th column in the inverse Vandermonde matrix.

Theorem 10 (Vandermonde Interpolation). *Let C_i, v_i, and Ω be the same as in Lemma 5. Let V be the k by k (Vandermonde) matrix $[V_{i,j}]$ over $\mathbb{Z}_p$ where $V_{i,j} = v_j^{i-1}$. Let $[g_0,\ldots,g_{k-1}] = [C_1,\ldots,C_k]V^{-1}$. Then we have*

$$\Omega(C_1,\ldots,C_k;v_1,\ldots,v_k) = \sum_{i=0}^{k-1} g_i x_1^i .$$

□

The **Vandermonde Combine** algorithm in Fig. 3 uses this idea for parallelizing the "combine" phase (Steps 5 to 7).

The main draw back of this algorithm is the synchronization barrier in Step 6, which causes a drastic degradation of the processor utilization. However, this disadvantage is alleviated by the fact that the modular computations require roughly the same time. This can be seen in the task picture of Fig. 3 where the slope of the block of parallel tasks at the beginning and at the end of the "conquer" phase are almost the same.

The gap during the parallel inner product computation is due to global garbage collection. We are currently studying how to improve the garbage collection algorithm to make the gap narrower.

7 Coefficient-wise Combine

Still another way of parallelizing the "combine" phase is to keep the linear structure of the computation but to parallelize the special binary interpolation in the nodes of this structure. This idea was already used by Küchlin [8] while parallelizing the combine phase of the greatest common divisor computation.

The **Coefficient-wise Combine** algorithm in Fig. 4 is based on this idea. All parallel tasks are only created inside the parallelized special interpolation algorithm (**Par-SpecialInterpolate**) in Step 5.

This algorithm yields a big number of blocks of parallel tasks of increasing length as the accumulating parameter in the linear computation structure is growing for each call to the interpolation algorithm. Due to this big number of parallel blocks the overhead for the creation and synchronization of parallel tasks has a bigger influence in this algorithm than in the other variants. Nevertheless, the utilization picture shows that the overall utilization in this variant is better than in all other variants. Generally, this overhead can be minimized by choosing an appropriately big value for *Cut-off*.

8 Parallel Divide

After having parallelized the "conquer" and the "combine" phases of the modular resultant algorithm, the only remaining sequential part is the "division" phase.

The reason for the sequential execution of the "division" phase is that only lucky homomorphic images have to be regarded. However, by introducing speculative parallelism in the division phase one can defer the test of luckiness until a parallel task has already been started. In the rather unlikely case that the chosen evaluation point is unlucky, the parallel task immediately terminates, reporting the unluckiness to the main task, which then takes another evaluation point and starts a new parallel task in substitution for the unlucky one.

The **Parallel Divide** algorithm in Fig. 5 uses this form of speculative parallelism for performing also the "division" phase in parallel. The evaluation and the test for luckiness are performed together with the modular resultant computation in the **Par-EvalResultant** function. For the parallel "combine" phase the Vandermonde parallelization that has been described in Sect. 6 is used.

Because the "division" phase is quite short, the task and the utilization pictures in Fig. 5 are quite similar to those in Fig. 3. As the major difference, the slope of the block of parallel tasks at the beginning of the execution is significantly steeper than in the other variant.

9 Summary

The experimental results from the previous sections are summarized in Fig. 1. The x axis is the number of processors, and the y axis the speedup factor.

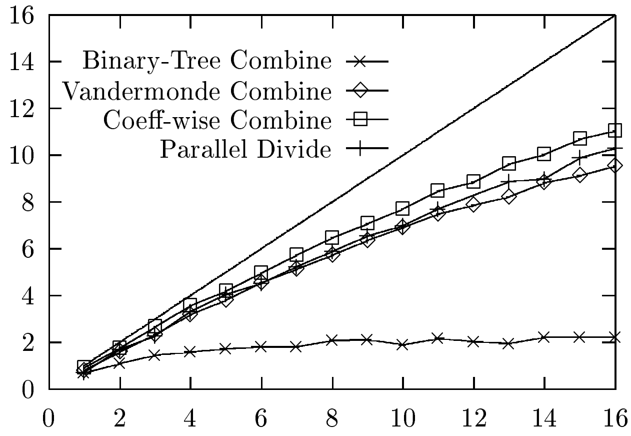


Fig. 1. Speedups for Variants of the Modular Resultant Computation

The speedups that were achieved by **Vandermonde Combine** fall short to linear speedups because the amount of computation for Vandermonde interpolation is larger than that for repeated special binary interpolation, and also because the huge memory consumption during parallel inner products causes an extremely time-consuming garbage collection. The **Parallel Divide** shows a little improvement to **Vandermonde Combine** because only the cheap division phase is parallelized.

The amount of the computation done by **Coefficient-wise Combine** is exactly the same as that done by the sequential algorithm, and the processor utilization is about 90 percent. Thus, one would expect almost linear speedup. We think that the worse behaviour is due to the task management overhead, which is very severe for this algorithm since it generates a large number of short parallel tasks. Since the time taken for task management depends on the number of tasks not on the number of processors, the speedup does not increase linearly with the number of processors.

In concluding, we believe that there is still much room for improvements and experiments. For instance, one could eliminate the "cut-off" step using the concept of virtual threads (tasks) introduced by [9, 11] and also investigated by [13]. We also plan to investigate the parallelization of other methods for computing multivariate polynomial resultants.

$(C, N) \leftarrow$ **Par-Resultant**(A, B, N)

In: $A, B \in \mathbb{Z}_p[x_1, \ldots, x_{r-1}][x_r]$. $N \in \mathbb{Z}_p[x_1]$.
Out: $C = \mathrm{res}(A, B) \in \mathbb{Z}_p[x_1, \ldots, x_{r-1}]$. $N \in \mathbb{Z}_p[x_1]$.
(1) If $r = 1$ then
  $C \leftarrow$ **ResultantUniVar**(A, B). Return (C, N).
(2) If $r <$ *Cut-off* then
  $C \leftarrow$ **Resultant**(A, B). Return (C, N).
(3) $k \leftarrow 1 +$ **Bound**(A, B).
  $C \leftarrow 0$. $M \leftarrow 1$. $v \leftarrow 0$. $d \leftarrow 0$. *task-list* $\leftarrow ()$.
(4) While $d < k$ do
  $A^* \leftarrow \Phi_v(A)$. $B^* \leftarrow \Phi_v(B)$.
  If Φ_v is lucky then
    CreateTask(*task-list*, **Par-Resultant**, $A^*, B^*, x_1 - v$).
    $d \leftarrow d + 1$.
  $v \leftarrow v + 1$.
(5) While Length(*task-list*) ≥ 2 do
  $(C^{(1)}, M^{(1)}) \leftarrow$ **WaitResult**(*task-list*).
  $(C^{(2)}, M^{(2)}) \leftarrow$ **WaitResult**(*task-list*).
  CreateTask(*task-list*, **Interpolate**, $C^{(1)}, C^{(2)}, M^{(1)}, M^{(2)}$).
  $(C, M) \leftarrow$ **WaitResult**(*task-list*).
  Return (C, N). □

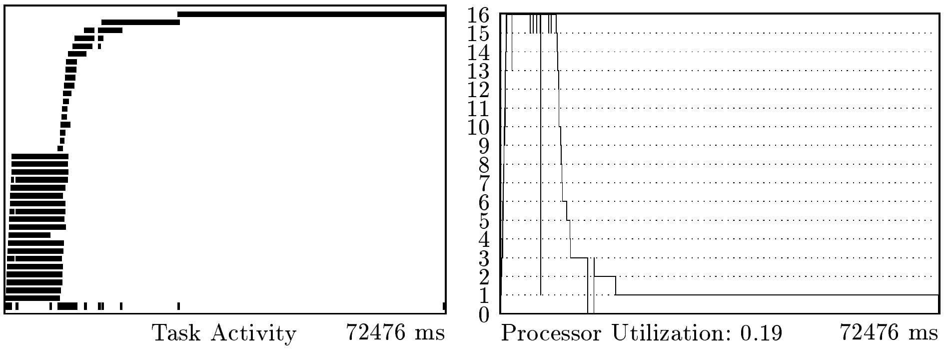


Fig. 2. Binary-Tree Combine — Sequential Divide, Parallel Conquer, Parallel Binary Combine

$(C, w) \leftarrow$ **Par-Resultant**(A, B, w)

In: $A, B \in \mathbb{Z}_p[x_1, \ldots, x_{r-1}][x_r]$. $w \in \mathbb{Z}_p$.
Out: $C = \mathrm{res}(A, B) \in \mathbb{Z}_p[x_1, \ldots, x_{r-1}]$. $w \in \mathbb{Z}_p$.

(1) If $r = 1$ then
  $C \leftarrow$ **ResultantUniVar**(A, B). Return (C, w).
(2) If $r <$ *Cut-off* then
  $C \leftarrow$ **Resultant**(A, B). Return (C, w).
(3) $k \leftarrow 1 +$ **Bound**(A, B).
  $C \leftarrow 0$. $M \leftarrow 1$. $v \leftarrow 0$. $\bar{v} \leftarrow ()$. $d \leftarrow 0$. *task-list* $\leftarrow ()$.
(4) While $d < k$ do
  $A^* \leftarrow \Phi_v(A)$. $B^* \leftarrow \Phi_v(B)$.
  If Φ_v is lucky then
    CreateTask(*task-list*, **Par-Resultant**, A^*, B^*, v).
    Append v to $\bar{v}$. $d \leftarrow d + 1$.
  $v \leftarrow v + 1$.
(5) $V' \leftarrow$ inverse of the Vandermonde matrix associated with $\bar{v}$.
(6) Wait for all the results $C_1, \ldots, C_k$ from the tasks in *task-list*.
(7) For $i = 1, \ldots, k$ do **in parallel**
  $g_{i-1} \leftarrow$ inner product of $[C_1, \ldots, C_k]$ and the i-th column of V'.
(8) Set $C \leftarrow \sum_{i=0}^{k-1} g_i x_1^i$.
  Return (C, w). □

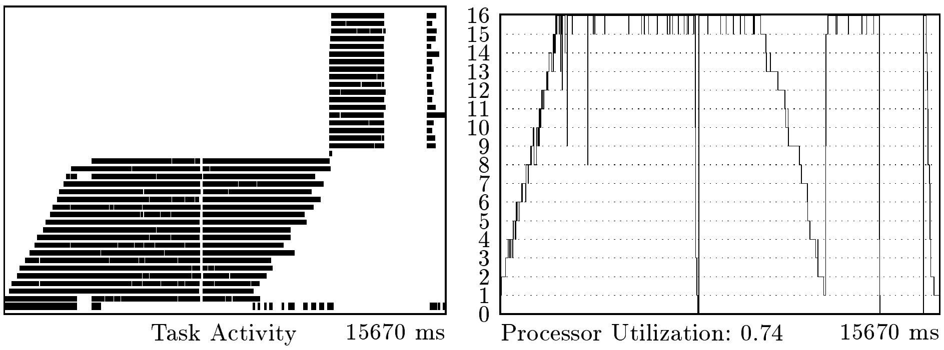
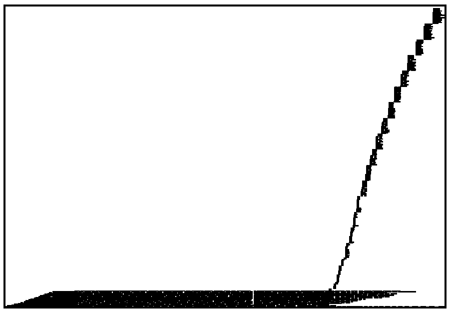


Fig. 3. Vandermonde Combine — Sequential Divide, Parallel Conquer, Parallel Vandermonde Combine

$(C, w) \leftarrow$ **Par-Resultant**(A, B, w)

In: $A, B \in \mathbb{Z}_p[x_1, \ldots, x_{r-1}][x_r]$. $w \in \mathbb{Z}_p$.
Out: $C = \mathrm{res}(A, B) \in \mathbb{Z}_p[x_1, \ldots, x_{r-1}]$. $w \in \mathbb{Z}_p$.

(1) If $r = 1$ then
    $C \leftarrow$ **ResultantUniVar**(A, B). Return (C, w).
(2) If $r <$ *Cut-off* then
    $C \leftarrow$ **Resultant**(A, B). Return (C, w).
(3) $k \leftarrow 1 +$ **Bound**(A, B).
    $C \leftarrow 0$. $M \leftarrow 1$. $v \leftarrow 0$. $d \leftarrow 0$. *task-list* $\leftarrow ()$.
(4) While $d < k$ do
    $A^* \leftarrow \Phi_v(A)$. $B^* \leftarrow \Phi_v(B)$.
    If Φ_v is lucky then
        CreateTask(*task-list*, **Par-Resultant**, A^*, B^*, v).
        $d \leftarrow d + 1$.
    $v \leftarrow v + 1$.
(5) While *task-list* is not empty do
    $(C^*, v) \leftarrow$ **WaitResult**(*task-list*).
    $C \leftarrow$ **Par-SpecialInterpolate**(C, C^*, M, v).
    $M \leftarrow M(x_1 - v)$.
    Return (C, w). □



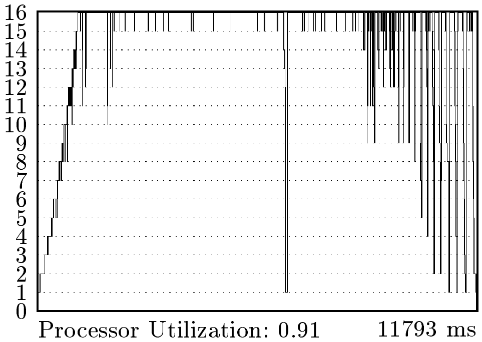


Fig. 4. Coefficient-wise Combine — Sequential Divide, Parallel Conquer, Parallel Coeff-wise Combine

$(C, w) \leftarrow$ **Par-Resultant**(A, B, w)

In: $A, B \in \mathbb{Z}_p[x_1, \ldots, x_{r-1}][x_r]$. $w \in \mathbb{Z}_p$.
Out: $C = \mathrm{res}(A, B) \in \mathbb{Z}_p[x_1, \ldots, x_{r-1}]$. $w \in \mathbb{Z}_p$.

(1) If $r = 1$ then
$C \leftarrow$ **ResultantUniVar**(A, B). Return (C, w).

(2) If $r <$ *Cut-off* then
$C \leftarrow$ **Resultant**(A, B). Return (C, w).

(3) $k \leftarrow 1 +$ **Bound**(A, B).
$C \leftarrow 0$. $M \leftarrow 1$. $v \leftarrow 0$. $\bar{v} \leftarrow ()$. $d \leftarrow 0$. *task-list* $\leftarrow ()$.

(4) While $d < k$ do
CreateTask(*task-list*, **Par-EvalResultant**, A, B, v).
Append v to $\bar{v}$. $d \leftarrow d + 1$. $v \leftarrow v + 1$.

(5) $V' \leftarrow$ the inverse of the Vandermonde matrix associated with $\bar{v}$.

(6) Wait for all results $C_1, \ldots, C_k$ from *task-list*. If a task reports unlucky evaluation, then start a new **Par-EvalResultant** task with another evaluation point and recompute the inverse of the modified Vandermonde matrix.

(7) For $i = 1, \ldots, k$ do **in parallel**
$g_{i-1} \leftarrow$ inner product of $[C_1, \ldots, C_k]$ and the i-th column of V'.

(8) Set $C \leftarrow \sum_{i=0}^{k-1} g_i x_1^i$. Return (C, w). □

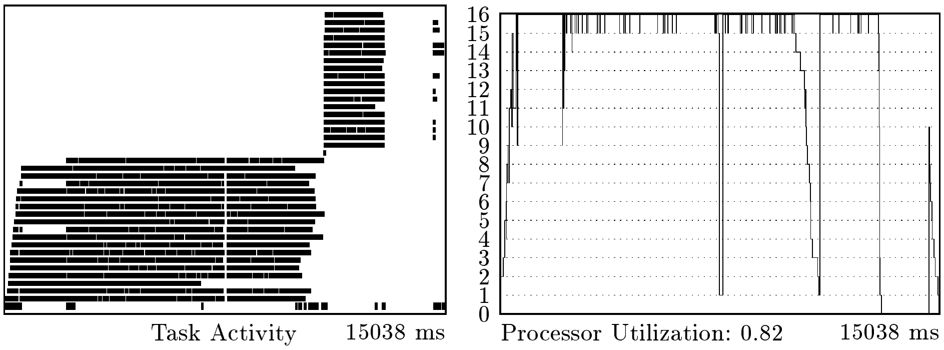


Fig. 5. Parallel Divide — Parallel Divide, Parallel Conquer, Parallel Vandermonde Combine

References

1. B. Buchberger, G. Collins, M. Encarnación, H. Hong, J. Johnson, W. Krandick, R. Loos, A. Mandache, A. Neubacher, and H. Vielhaber. A SACLIB Primer. Technical Report 92-34, RISC-Linz, Johannes Kepler University, Linz, Austria, 1992.

2. B. W. Char. Progress Report on a System for General-Purpose Parallel Symbolic Algebraic Computation. In *Int. Symposium on Symbolic and Algebraic Computation (ISSAC 90)*, pages 96–103, Tokyo, August 20–24, 1990. ACM Press.
3. G. E. Collins. The Calculation of Multivariate Polynomial Resultants. *Journal of the ACM*, 18:515–532, 1971.
4. H. Hong. Efficient Method for Analyzing Topology of Plane Real Algebraic Curves. In *Proceedings of IMACS-SC 93*, Lille, June 1993.
5. H. Hong, A. Neubacher, and W. Schreiner. The Design of the SACLIB/PACLIB Kernels. In A. Miola, editor, *Int. Symposium on Design and Implementation of Symbolic Computation System (DISCO 93)*, volume 722 of *Lecture Notes in Computer Science*, pages 288 – 302, Gmunden, September 15–17, 1993. Springer.
6. H. Hong, W. Schreiner, A. Neubacher, K. Siegl, H.-W. Loidl, T. Jebelean, and P. Zettler. PACLIB User Manual. Technical Report 92-32, RISC-Linz, Johannes Kepler University, Linz, Austria, May 1992.
7. W. Küchlin. PARSAC-2: A Parallel SAC-2 Based on Threads. In S. Sakata, editor, *Eighth Int. Symposium on Applied Algebra, Algebraic Algorithms, and Error Correcting Codes (AAECC 8)*, volume 508 of *Lecture Notes in Computer Science*, pages 206 –217, Tokyo, August 1990. Springer.
8. W. Küchlin. On the Multi-Threaded Computation of Integral Polynomial Greatest Common Divisors. In S. M. Watt, editor, *Int. Symposium on Symbolic and Algebraic Computation (ISSAC 91)*, Bonn, July 15 – 17, 1991. ACM Press.
9. W. Küchlin and J. Ward. Experiments with Virtual C Threads. In *4th IEEE Symposium on Parallel and Distributed Processing*, Arlington, December, 1992. IEEE Press.
10. R. G. K. Loos. Computing in Algebraic Extensions. In B. Buchberger, G. E. Collins, and R. G. K. Loos, editors, *Computer Algebra, Symbolic and Algebraic Computation*, pages 173–187. Springer, 1982.
11. E. Mohr, D. A. Kranz, and R. H. Halstead Jr. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. In *1990 ACM Symposium on Lisp and Functional Programming*, pages 185–197, Nice, June 27–29, 1990.
12. J. L. Roch. An Environment for Parallel Algebraic Computation. In R. E. Zippel, editor, *Computer Algebra and Parallelism — Second Int. Workshop on Parallel Algebraic Computation*, volume 584 of *Lecture Notes in Computer Science*, pages 33–50, Ithaca, May 1990. Springer.
13. W. Schreiner. Virtual Tasks for the PACLIB Kernel. In *Joint Int. Conference on Vector and Parallel Processing (CONPAR 94 - VAPP VI)*, Lecture Notes in Computer Science, Linz, September 6 – 8, 1994. Springer. Also: Technical Report 94-02, RISC-Linz.
14. S. Seitz. Parallel Algorithm Development. In J. Della Dora and J. Fitch, editors, *Computer Algebra and Parallelism*, pages 223 – 232. Academic Press, June 1988.
15. K. Siegl. ||MAPLE|| — A System for Parallel Symbolic Computation. In H. M. Alnuweiri, editor, *Parallel Systems Fair at the Seventh Int. Parallel Processing Symposium*, pages 62–67, Newport Beach, April 14, 1993.
16. P. Wang. Parallel Univariate Polynomial Factorization on Shared-Memory Multiprocessors. In S. Watanabe and M. Nagata, editors, *Int. Symposium on Symbolic and Algebraic Computation (ISSAC 90)*, pages 145 – 151. Addison-Wesley, August 1990.

This article was processed using the LaTeX macro package with LLNCS style