Parallel Sparse Multivariate Polynomial Division

Mickaël Gastineau
IMCCE-CNRS UMR8028, Observatoire de Paris, UPMC
Astronomie et Systèmes Dynamiques
77 Avenue Denfert-Rochereau
75014 Paris, France
gastineau@imcce.fr

Jacques Laskar
IMCCE-CNRS UMR8028, Observatoire de Paris, UPMC
Astronomie et Systèmes Dynamiques
77 Avenue Denfert-Rochereau
75014 Paris, France
laskar@imcce.fr

ABSTRACT

We present a scalable algorithm for dividing two sparse multivariate polynomials represented in a distributed format on shared memory multicore computers. The scalability on the large number of cores is ensured by the lack of synchronizations during the main parallel step. The merge and sorting operations are based on binary heap or tree data structures.

Categories and Subject Descriptors

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

Keywords

Parallel, Sparse, Polynomial, Division

1. INTRODUCTION

Basic multivariate polynomials arithmetic may take advantage of the availability of multiple cores present in modern computers to accelerate computations in many scientific areas. Sparse multivariate polynomials are involved in many of these problems. The representation of the polynomials in a distributed format, such as a collection of tuples of coefficients and exponents stored in arrays, allows to benefit from the caches present inside the processors. Parallel multiplication and division algorithms, targeting a single processor with multiple cores, have been designed by Monagan and Pearce [12, 13]. Both algorithms merge and sort the terms using a chained binary heap [14], based on a sequential heap designed by Johnson [10]. These algorithms could not be applied to a large number of cores due to the presence of a global heap shared by all the threads. Other sequential division algorithms have been designed using a linear algebra approach [2] for dense polynomials. In [5], we have proposed a multiplication algorithm targeting a large number of cores on different kinds of hardware. This algorithm allows to compute independent terms by the threads at the same time and merges the terms of the product using any available comparison-based sorting algorithm. Using a dense approach, Fateman [4] and Ponder [15] suggest to split recursively the polynomials in order to perform the multiplication but many terms with the same exponents are generated before the merging terms step. These temporary duplicated terms increase largely the pressure on the memory allocator, which is a bottleneck in a parallel context. The FFT or block based methods are mostly suitable for the multiplication of multivariate dense polynomials and not much for very sparse polynomials, even if the barrier between the sparse and dense algorithms becomes weaker [17, 16, 19].

The division with a remainder of a sparse multivariate polynomial A by a polynomial B, using a chosen monomial order, may be defined as $A = Q \times B + R$ such that any term of the remainder R is not divisible by the leading term of B, noted $lt(B)$. With the initial values $Q = R = 0$, the sequential division algorithm [10] consists in canceling successively the leading term of the expression $A - (Q \times B + R)$. If $lt(B)$ does not divide $lt(A - (Q \times B + R))$, the leading term of this expression is added to the current value of the remainder. As the division with a remainder is very similar to the exact division (case $R = 0$), we will focus in parallelizing the algorithm of the exact division.

The sequential exact division of two sparse multivariate polynomials represented in a sparse distributed format over the ring of integers or rationals could be done in the following manner. Let us consider two sparse polynomials in m variables $x_1, \ldots, x_m$,

$$A(\mathbf{x}) = \sum_{j=1}^{n_a} A_j = \sum_{i=1}^{n_a} a_i \mathbf{x}^{\alpha_i} \text{ and } B(\mathbf{x}) = \sum_{j=1}^{n_b} B_j = \sum_{j=1}^{n_b} b_j \mathbf{x}^{\beta_j}$$

where $\mathbf{x}$ corresponds to the variables $x_1, \ldots, x_m$, the a_i and b_j are non-zero numerical coefficients, and the m-dimensional integer vectors α_i and β_j are the exponents. Their terms are sorted with a monomial order $\prec$ [6] such that $\alpha_{n_a} \prec \alpha_{n_a-1} \prec \cdots \prec \alpha_1$ and $\beta_{n_b} \prec \beta_{n_b-1} \prec \cdots \prec \beta_1$. So A_1 and B_1 are the leading term of A and B. Our algorithm computes the polynomial $Q \in \mathbb{Z}[x_1, \ldots, x_m]$ or $\mathbb{Q}[x_1, \ldots, x_m]$ such that $Q = A \div B = Q_1 + \cdots + Q_{n_q} = \sum_{j=1}^{n_q} q_j \mathbf{x}^{\sigma_j}$ where $Q_1 = q_1 \mathbf{x}^{\sigma_1}$ is the leading term of Q. In addition, the algorithm allows to check that B divides exactly A.

The sequential division algorithm computes iteratively the terms Q_k of Q. The first step consists in dividing the leading term of A by the leading term of B in order to obtain the first term Q_1 of Q and to compute $A - Q_1 B$. The leading term of A is canceled in the expression $A - Q_1 B$. The computation of the next term Q_{k+1} is given by the division of $lt(A - \sum_{j=1}^{k} Q_j B)$ by $lt(B)$. This operation is repeated until $A -$

PASCO '15, July 10 - 12, 2015, Bath, United Kingdom

DOI: http://dx.doi.org/10.1145/2790282.2790285

$\sum_{j=1}^{n_q} Q_j B = 0$. We define the polynomial P_{n_q} such that

$$P_{n_q} = \begin{cases} A - \sum_{j=1}^{n_q} Q_j B & \text{if } n_q \geq 1 \\ A & \text{otherwise.} \end{cases}$$

A way to parallelize this algorithm is to perform the product $Q_j B$ and the subtraction by the threads and synchronize them before the generation of the next term Q_k. But this algorithm requires $\mathcal{O}(n_q)$ synchronizations between the threads [18] and the cost of each synchronization is proportional to the logarithm of the number of threads.

In the section 2, we consider to parallelize the exact division of sparse multivariate polynomials targeting shared memory computers, such as multiple processors with multiple cores sharing some memory. The merge and sort operations on the terms are presented in the section 3. Our implementation is compared to other computer algebra systems in section 6. The optimization for the division with a remainder are given in the last section.

2. PARALLEL EXACT DIVISION ALGORITHM

The main part of the computation of P_{n_q} consists in performing the multiplication of Q and B. As the polynomial B is sparse, the naive school-book multiplication, with a complexity $\mathcal{O}(n_q n_b)$, is used to multiply them. This multiplication of two sparse multivariate polynomials could be efficiently parallelized on a multi-core computer by reducing the number of required communications between the threads [5]. Like this multiplication algorithm, a possible efficient parallel division algorithm consists in the threads compute independent terms of P_{n_q}. Like for the multiplication, we use a similar approach for the division to split the work in independent parts, except the application of the grid.

The critical step splits the polynomials involved during the division in independent parts. We note $lt(G)$, respectively $lm(G)$, the leading term, respectively monomial, of a polynomial G. Two exponents γ and δ verifying the relation $\gamma \preceq \delta$ means that $\gamma \prec \delta$ or $\gamma = \delta$. Using the monomial order, any ordered polynomial $G(\mathbf{x}) = \sum_{i=1}^{n_g} g_i(\mathbf{x})$ could be simply broken in several parts using an ordered set (S) of intermediary splitting exponents. We choose the set S of n_s distinct exponents such as it includes the exponents 0_m and ρ_{end} where 0_m is the m-dimensional integer vector $(0, 0, \ldots, 0)$ and ρ_{end} is any exponent verifying the relation $lm(G) \prec \mathbf{x}^{\rho_{end}}$. We defined this set S as

$$S = \{S_r \mid 1 \leq r \leq n_s \text{ and } S_r \in \mathbb{N}^m\}$$
$$\text{with } S_1 = \rho_{end}, S_{n_s} = 0_m \text{ and } S_{r+1} \prec S_r$$

This polynomial G is decomposed as $G(\mathbf{x}) = \sum_{r=1}^{n_s-1} \mathcal{G}_r(\mathbf{x})$ where any monomial t of $\mathcal{G}_r(\mathbf{x})$ verifies the relation $\mathbf{x}^{S_{r+1}} \preceq t \prec \mathbf{x}^{S_r}$. ρ_{end} is only introduced to have the same relation for all $\mathcal{G}_r$, even for the first one. Using this definition, any term t_u of $\mathcal{G}_i(\mathbf{x})$ and t_v of $\mathcal{G}_j(\mathbf{x})$ verifies the relation $t_u \prec t_v$ for any i and j verifying $j < i$. We write $\mathcal{G}_r(\mathbf{x})$ as $\mathcal{G}_r(\mathbf{x}; [S_{r+1}; S_r[)$ to display the possible terms present in the polynomial $\mathcal{G}_r$.

For example, the univariate polynomial $G(x) = x^{10} + x^8 + x^6 + x^5 + x^3 + x^2 + x + 1$ with the set $S = \{11, 7, 3, 0\}$ is decomposed in a unique manner as $G(x) = G_1(x) + G_2(x) + G_3(x)$ with $G_1(x) = x^{10} + x^8, G_2(x) = x^6 + x^5 + x^3$ and $G_3(x) = x^2 + x + 1$.

We apply this decomposition on the expanded form of $P_{n_q}(\mathbf{x}) = \sum_i p_i \mathbf{x}^{\rho_i}$, where p_i are integer coefficients and the m-dimensional integer vectors ρ_i are the exponents. As the possible values of all the ρ_i verify the relation $0_m \preceq \rho_i \preceq \alpha_1$, we choose ρ_{end} such that $\alpha_1 \prec \rho_{end}$. The value of n_s and the way to choose the set S are discussed later. So the polynomial P_{n_q} is splitted as a sum of grouped terms

$$P_{n_q}(\mathbf{x}) = \sum_{r=1}^{n_s-1} \mathcal{P}_r(\mathbf{x}; n_q; [S_{r+1}; S_r[) \tag{1}$$

Using the same set, the product of a single term Q_k by the polynomial B could be decomposed in the same manner. We define the product $\times_{[S_{r+1};S_r[}$ of the monomial $Q_k = q_k \mathbf{x}^{\sigma_k}$ and B as the following product

$$Q_k \times_{[S_{r+1};S_r[} B = \sum_{\substack{1 \leqslant j \leqslant n_b \\ S_{r+1} \preceq \sigma_k + \beta_j \prec S_r}} q_k b_j \mathbf{x}^{\sigma_k + \beta_j}$$

We would like to process in parallel the set of polynomials $\mathcal{P}_r$ but the threads will not merge and sort at the same speed the terms of $A - \sum_{k=1}^{n_q} Q_k B$ occured in these $\mathcal{P}_r$. We need to consider the parts of A and $\sum_{k=1}^{n_q} Q_k B$ that have been processed or not in each $\mathcal{P}_r$. We write it as two parts which correspond to the processed part, noted $\mathcal{M}_r$, and the pending part, noted $\mathcal{N}_r$.

$$\begin{aligned} \mathcal{P}_r(\mathbf{x}; n_q; [S_{r+1}; S_r[) &= \mathcal{M}_r(\mathbf{x}; m_a, m_q, n_q; [S_{r+1}; S_r[) \\ &+ \mathcal{N}_r(\mathbf{x}; m_a, m_q, n_q; [S_{r+1}; S_r[) \end{aligned} \tag{2}$$

with

$$\mathcal{M}_r = \sum_{\substack{i=1 \\ S_{r+1} \preceq \alpha_i \prec S_r}}^{m_a - 1} a_i \mathbf{x}^{\alpha_i} - \sum_{k=1}^{m_q} Q_k \times_{[S_{r+1};S_r[} B$$

$$\mathcal{N}_r = \sum_{\substack{i=m_a \\ S_{r+1} \preceq \alpha_i \prec S_r}}^{n_a} a_i \mathbf{x}^{\alpha_i} - \sum_{k=m_q+1}^{n_q} Q_k \times_{[S_{r+1};S_r[} B$$

$\mathcal{M}_r$, is the part of $A - \sum_{j=1}^{n_q} q_j B$, whose terms were processed, that is to say already merged and sorted. $\mathcal{N}_r$ is the other part of $A - \sum_{j=1}^{n_q} q_j B$, whose terms were not yet processed. It is the pending work to be performed. So every $\mathcal{P}_r$ will depend additionally from m_a and m_q.

2.1 Single Producer - Multiple Consumers

From the equation 1, it is easy to understand that the term Q_1 will be computed by using $\mathcal{P}_1$ only. Indeed, the division algorithm consists in canceling the leading term of $\mathcal{P}_1$. The next terms $Q_2, Q_3, \ldots$ may be still generated by the same polynomial $\mathcal{P}_1$ until this polynomial is equal to 0. Only the content of the first non-zero polynomial $\mathcal{P}_r$ is used to compute new terms of Q at the same time. When the first non-zero polynomial $\mathcal{P}_r$ is becoming equal to 0, the content of the next polynomial $\mathcal{P}_{r+1}$ is then used to generate the next term of Q.

Our algorithm requires that a polynomial $\mathcal{P}_r$ is processed by a single thread at the same time but multiple threads are allowed to process it during the execution of the algorithm. The thread, which processes the first non-zero $\mathcal{P}_r$,

is called *producer*. Indeed, only this thread adds new terms to the quotient Q. The other threads, which process the remaining polynomials $\mathcal{P}_r$, are called the *consumers*. When $\mathcal{P}_r$ becomes equal to 0, the $\mathcal{P}_{r+1}$ becomes the new *producer*.

When a new term is added to Q by the producer, n_q is incremented and the part N_r of all polynomials $\mathcal{P}_r$ implicitly grows. Indeed, $Q_{n_q} \times_{[S_{r+1};S_r[} B$ must be processed for each other polynomials $\mathcal{P}_r$ but it can be done asynchronously. The producer must merge and sort the terms of $Q_{n_q} \times_{[S_{r+1};S_r[} B$ before generating the next term of Q. This approach allows us to adopt a single producer and multiple consumers based on the list defined by the polynomials $\mathcal{P}_r$.

The main task of a thread consists in merging and sorting the new terms generated by $\mathcal{N}_r$ in order to have a canonical representation of this polynomial $\mathcal{M}_r$. Multiple threads cannot process different parts of the same polynomial $\mathcal{P}_r$ because an expensive mutual exclusion mechanism is required to update safely the memory representation of $\mathcal{P}_r$. So we need to attach some additional information to each polynomial $\mathcal{P}_r$ in order to process them in parallel. Our algorithm processes, using $\mathcal{T}$ threads, numbered from 0 to $\mathcal{T}-1$, the following shared order list $\mathcal{L}$, which contains n_s-1 elements at the beginning and corresponds in the same order to the sequence $\mathcal{P}_1, \ldots, \mathcal{P}_{n_s-1}$.

$$\mathcal{L} = (\mathcal{L}_1, \ldots, \mathcal{L}_r, \ldots, \mathcal{L}_{n_s-1}) \tag{3}$$

$$\text{where } \mathcal{L}_r = \begin{cases} lock & \text{integer} \\ producer & \text{logical} \\ m_a & \text{integer} \\ m_q & \text{integer} \\ E_{inf}, E_{sup} & \in S \\ \mathcal{C} & \mathcal{M}_r(\mathbf{x}; m_q; [E_{inf}; E_{sup}[) \end{cases} \tag{4}$$

The elements at the head and tail of the sequence verify these properties

1. E_{sup} of $L_1 = S_1, E_{inf}$ of $L_{n_s-1} = S_{n_s}$
2. *producer* of L_1 = true at the beginning

If an element of this list has some pending work, a thread should acquire an exclusive access to the element in order to process the pending work. A global lock to protect the access to the list will harm the scalability of the algorithm on a large number of cores. This explains why the integer *lock* is present in each element of $\mathcal{L}$. The behaviour of the *lock* field of each element is presented later in 2.4. Each element $\mathcal{L}_r$ allows to have a thread-safe representation of each polynomial $\mathcal{P}_r$. The fields of an element, at the index r in the sequence $\mathcal{L}$, always verify the following properties

1. *lock* may have the following values
   - -2 if $\mathcal{L}_r$ was never processed
   - -1 if $\mathcal{L}_r$ is currently updated by a thread
   - $0 \ldots \mathcal{T}-1$ last thread to update $\mathcal{L}_r$ but it does not longer access to it
2. $producer = \begin{cases} \text{true if } \mathcal{L}_r \text{ is the producer} \\ \text{false if } \mathcal{L}_r \text{ is a consumer} \end{cases}$
3. $0 \le m_q \le n_q$
4. E_{sup} of $\mathcal{L}_r = E_{inf}$ of $\mathcal{L}_{r+1}$

The global value n_q and $\mathcal{L}$ are obviously shared between the threads. Moreover, the elements located before the current producer in the list $\mathcal{L}$ should no longer be considered,

Algorithm 1: div_exact(A, B, n_s). Return $A \div B$ using at most n_s intervals.

Input: $A = \sum_{i=1}^{n_a} a_i \mathbf{x}^{\alpha_i}$
Input: $B = \sum_{j=1}^{n_b} b_j \mathbf{x}^{\beta_j}$
Input: n_s : integer number of intervals
Output: $Q = Q_1 + \cdots + Q_{n_q}$
Output: $failure$: logical, true if the algorithm fails

```
   // First step
1  S ←Choose n_s exponents from the list of α_i and β_j
2  sort S using the monomial order
3  remove duplicate values from S
   // S has now at most n_s sorted elements
   // Second step
4  Initialize L using S
5  L_1.producer ← true
6  H ← reference to L_1
   // Third step : compute leading terms of Q
7  (Q, failure) ← bootstrap(A, B)
   /* The step Four and Five are processed in
   parallel with T threads, numbered 0...T-1
   The threads share H, Q, failure and final        */
8  Each thread initializes an empty private Cache
   // Fourth step:  0...T-1 threads process it
9  nqprevious ← 0
10 countnowork ← 0
11 while H is not empty and failure = false and final
   = false do
12    workdone ← false
13    nqprivate ← load from memory n_q
14    if nqprevious < nqprivate then
15       foreach L_r ∈ Cache do
16          if tryproc (L_r, failure, final,
            H, Q, A, B)=true then
17             workdone ← true
18          else
19             remove L_r from the Cache
20          end
21       end
22    end
23    if workdone= false then
24       foreach L_r ∈ H do
25          if tryproc (L_r, failure, ..., B)=true then
26             add L_r to the Cache
27             workdone ← true
28             exit this loop
29          end
30       end
31    end
32    nqprevious ← nqprivate
33    if workdone = false then
34       increment countnowork
35       if countnowork > thresoldyield then yield()
36    else
37       countnowork ← 0
38    end
39 end
   // Fifth step
40 foreach L_r ∈ Cache do tryprocfinal (L_r, ..., B)
41 foreach L_r ∈ H do tryprocfinal (L_r, ..., B)
```

as their terms have been already canceled. So, only the tail of the list, starting at the current producer element, should be processed. In addition to these two previous shared variables, a reference to the current producer or its successor in the list $\mathcal{L}$, named $\mathcal{H}$, is shared between the threads.

2.2 Global Workflow

At the beginning of the algorithm 1, the list $\mathcal{L}$ is generated from the set of n_s exponents and the simple leading terms of Q are generated. By default, all elements of the list $\mathcal{L}$ have the field *producer* with the value false, except for the head $\mathcal{L}_1$ which is always the first producer. $\mathcal{H}$ references the head of the list $\mathcal{L}$ as all elements may have some work to do, even the first element, which is the first producer and is not assigned to any thread. Between the line 11 and 39, the pool of threads loops on the tail of the list referenced by $\mathcal{H}$ to process these elements. Each thread maintains a bounded cache of the references to the last items of the sequence updated by itself. The thread tries to first update the elements from its cache. This technique reduces the number of remote memory access on the computer with a Non-Uniformed Memory Architecture. If no update from elements of its cache occurs, the thread tries to steal elements from the list $\mathcal{H}$.

The merge and sort operations are performed in the algorithm 2. An element needs to be updated only if its m_q is less than the value n_q or its field *producer* is equal to true. If a thread acquires the lock on the element whose the *producer* is true, this thread will become the producer and it is the single thread to update the container of Q until it finished on this item. In the other case, the thread knows that it updates this item as a consumer. When a thread which is finishing to run as a consumer on an item, but before changing the value of *lock*, this thread checks that this item does not become the new producer. If the element becomes the new producer, it will continue to process this item but as the new producer.

When few elements remain after $\mathcal{H}$, few threads have work to do and other threads yield if they iterate many times without work (lines 33-38). Between the fourth and fifth step, no barrier synchronization is required. So these two steps should be considered as a single parallel section.

The goal of the consumer threads is to compute the different $\mathcal{N}_r$ in order to merge it into $\mathcal{M}_r$. After this merge, m_q is set to the value of n_q. But the used value of n_q is the value at the beginning of the merge and not at the end as it may evolve between these two moments. So N_r will become implicitly equal to 0. When an element becomes a new producer, the operations to generate the new terms of Q will be accelerated.

When the thread acquires a lock to the producer element, the thread updates immediately $\mathcal{H}$ to the next element after the producer element. Then the thread merges and sorts its terms, computes the next term of Q and adds this new term to the data structure of Q before the value n_q is incremented. Thus the other threads have always a coherent view of Q. The other threads read only the data from the container Q to process the elements $\mathcal{H}, \ldots, \mathcal{L}_{n_s-1}$. The container of Q does not require any lock to be accessed.

When the producer finishes to process it, the field *producer* of its element is set to false. Moreover, the thread left the element in a locked state as threads should no longer access this element. The field *producer* of the next element, which is the same as $\mathcal{H}$, is changed to true in order to notify all other threads that this item becomes the new producer.

2.3 Bootstrap

The term Q_1 is generated before the list $\mathcal{L}$ is started to be processed in parallel. That implies that the consumer threads will not wait for the first term of Q when they will start to iterate on the loop. If A and B have more than two terms, some additional terms of Q could be generated with a simple comparison operations. Let the following example $A(x) = x^{12} + x^{11} + x^{10} + x^9 + x^7 + x^5$ and $B = x^3 + x$ then $Q = A \div B = x^9 + x^7 + x^4$.

$Q_2 \left(= x^7\right)$ could be generated immediately after $Q_1 \left(= x^9\right)$ as $\alpha_2 = 11 > \sigma_1 + \beta_2 = 9 + 1 = 10$. On the contrary, $Q_3 \left(= x^4\right)$ could not be generated immediately. Indeed, as

Algorithm 2: tryproc($\mathcal{L}_r, \mathcal{H}, Q, failure, final, A, B,$)

lockprev ← load from memory $L_r.lock$
if *($L_r.producer$=true* **or** $L_r.m_q < n_q$*)*
and $lockprev \neq -1$
and `compare-and-swap` *($L_r.lock$,lockprev,-1)=true*
then
  if $L_r.producer$=*true* **then** $\mathcal{T} \leftarrow$ next element of $\mathcal{T}$
  `// fill` $\mathcal{M}_r$ `with the pending terms of` $\mathcal{N}_r$
1 $nqprev \leftarrow n_q$
2 **for** $k \leftarrow m_q$ **to** $nqprev$ **do**
3 compute $j \mid Q_k B_j = \mathtt{lt}(Q_k \times_{\lfloor S_{r+1}; S_r \rfloor} B)$
4 **while** $S_r \preceq \rho_k + \beta_j$ **do**
5 insert $-Q_k B_j$ in the tree $\mathcal{M}_r$
6 decrement j
7 **end**
8 **end**
9 $m_q \leftarrow nqprev$
10 **if** $L_r.producer$=*true* **then**
  `// produce new terms`
11 **while** $\mathcal{M}_r \neq 0$ **or** $S_{r+1} \preceq \alpha_{m_a} \prec S_r$ **do**
12 **while** $S_{r+1} \preceq \alpha_{m_a} \prec S_r$
13 **and** *(*$\mathcal{M}_r = 0$ **or** $\mathtt{lm}(\mathcal{M}_r) \preceq \mathbf{x}^{\alpha_{m_a}}$*)* **do**
14 add $a_{m_a}\mathbf{x}^{\alpha_{m_a}}$ to $\mathcal{M}_r$
15 increment m_a
16 **end**
17 **if** $\mathcal{M}_r \neq 0$ **then**
18 $G \leftarrow \mathtt{lt}(\mathcal{M}_r)$
19 **if** $B_1 \nmid G$ **then** $failure \leftarrow$true **and exit**
20 $Q_{n_q+1} \leftarrow G \div B_1$
21 increment n_q `// memory flush`
22 **if** $\sigma_{n_q} + \beta_1 \preceq \beta_1$ **then** $final \leftarrow$true
23 execute from line 3 to 7 with $k = n_q$
24 **end**
25 **end**
26 $\mathcal{L}_r.producer \leftarrow$ false, $\mathcal{H}.producer \leftarrow$ true
27 **else**
28 insert $\sum_{S_{r+1} \preceq \alpha_i \prec S_r} a_i \mathbf{x}^{\alpha_i}$ in $\mathcal{M}_r$, $m_a \leftarrow n_a$
29 **if** $\mathcal{H}.producer = true$ **then**
30 tryproc($\mathcal{L}_r, failure, final, \mathcal{H}, Q, A, B$)
31 **else**
32 $L_r.lock \leftarrow$ id of the thread
33 **end**
34 **end**
end

Algorithm 3: bootstrap(A, B). Return the leading terms of $A \div B$.

Input: $A = \sum_{i=1}^{n_a} A_i = \sum_{i=1}^{n_a} a_i \mathbf{x}^{\alpha_i}$
Input: $B = \sum_{j=1}^{n_b} B_j = \sum_{j=1}^{n_b} b_j \mathbf{x}^{\beta_j}$
Output: $Q = Q_1 + \ldots$
Output: $failure$: logical, true if B does not divide A

$failure \leftarrow$false
if $B_1 \nmid A_1$ **then** $failure \leftarrow$true **and exit**
$Q \leftarrow a_1/b_1 \mathbf{x}^{\alpha_1 - \beta_1}$, $k \leftarrow 2$
while $k \leq n_a$ **and** $\alpha_k \succ \alpha_1 - \beta_1 + \beta_2$ **do**
  if $b_1 \nmid a_k$ **then** $failure \leftarrow$true **and exit**
  $Q \leftarrow Q + a_k/b_1 \mathbf{x}^{\alpha_k - \beta_1}$
  increment k
end

$\alpha_3 = 10 = \sigma_1 + \beta_2 = 9 + 1 = 10$, at least one merge operation is required to compute the exponent of Q_3.

The leading terms Q_k could be generated immediately if the leading terms of A verify $\alpha_k \succ \sigma_1 + \beta_2 = \alpha_1 - \beta_1 + \beta_2$ for $k \geq 2$. The algorithm 3 describes this generation.

These additional terms are generated before the main loop as their execution time is very low and it increases the amount of available work for the consumer threads when they start to iterate on the main loop.

2.4 Lock Behavior

Each thread, producer or consumers, works on a different $\mathcal{L}_r$. The integer *lock* indicates that the element is currently or not accessed by a thread. We assume that the processor offers some atomic operations to modify safely this value in a lock-free manner. Indeed, a *compare-and-swap (CAS)* atomic operation [7] must be available in order to be sure that a thread has the correct previous value of this integer when it modifies the value of this one. If an element has its field *lock* to a value different from -1, then a thread can perform a *CAS* operation to set it to -1. If this operation fails, this thread is not allowed to update the element and knows that another thread has acquired this lock to update the element. If the *CAS* operation is successful, then the thread can safely update the element. When the thread finishes to update this element, it sets *lock* to the identification number of the thread, a value between 0 and $t-1$, only if the field *producer* is equal to false. If the field *producer* is equal to true, *lock* is left to the value -1 as no further update will occur on this element. *lock* is never reverted to the value -2 for the cache management. The value of the thread is useful for a NUMA-aware work-stealing strategy.

3. MERGE AND SORT TERMS

We have explored two merging and sorting methods for the computation of $\mathcal{M}_r$ and $\mathcal{N}_r$. The first one uses a chained binary heap and the second one uses a tree. The chained binary heap is suitable for division of very sparse polynomials although the tree data structure should be used with more dense polynomials.

To perform the product $\sum_{k=1}^{n_q} Q_k \times_{[S_{r+1};S_r[} B$, it is not necessary to perform $n_q n_b$ comparisons on the exponents to determine if a resulting term is inside the range defined by S_r and S_{r+1}. Indeed, the order of the product of two terms follows the property of the *pp-matrix* [9] : $q_k b_j \mathbf{x}^{\sigma_k + \beta_j} \prec q_{k+1} b_j \mathbf{x}^{\sigma_{k+1} + \beta_j}$ and $\sigma_k + \beta_j \prec q_k b_{j+1} \mathbf{x}^{\sigma_k + \beta_{j+1}}$. For one element $\mathcal{L}_r$ and $k = 1 \ldots n_q$, all leading terms of $Q_k \times_{[S_{r+1};S_r[} B$ could be determined with a complexity $\mathcal{O}(n_q + n_b)$, as shown in [5].

3.1 Tree Method

The terms of the temporary polynomial $\mathcal{M}_r$ are stored in a tree data structure where each internal node has exactly the same number 2^c of children. At each level of this tree, c bits of the exponent are used to index the next children. If the exponents are encoded on 2^d bits, the tree has an height of 2^{d-c}. The leaves of the tree store the coefficients of $\mathcal{M}_r$. The insertion of the terms has a complexity $\mathcal{O}(n_q n_b \log(2^{d-c}))$. The efficiency of this method requires to have a packed representation of the exponents [11]. The extraction of the leading term has a complexity $\mathcal{O}(\log(2^{d-c}))$ for the worst case and could be done in $\mathcal{O}(1)$ for dense polynomials.

The consumer threads compute the terms of $\mathcal{N}_r$ and insert them in these trees according to the value n_q. Only once, when a consumer thread finishes the processing of an element, it inserts all the corresponding part of A inside the tree and updates the value m_a, such that $m_a \prec S_{r+1}$, to signal that A has been merged inside $\mathcal{M}_r$.

When a thread starts to process the producer element, it first computes the terms of $\mathcal{N}_r$ and inserts them in the tree. Then the thread iterates on the following processing until $\mathcal{M}_r$ becomes empty and $\alpha_{m_a} \prec S_{r+1}$. If $S_{r+1} \preceq \alpha_{m_a} \prec S_r$, it takes the leading term of $a_{m_a} \mathbf{x}^{\alpha_{m_a}} + \mathcal{M}_r$ by extracting the term from A or from the tree $\mathcal{M}_r$ to compute the new term of the quotient. If α_{m_a} corresponds to the exponent of the extracted term, the term $a_{m_a} \mathbf{x}^{\alpha_{m_a}}$ is inserted in the tree. In the other case, i.e. $\alpha_{m_a} \prec S_{r+1}$, only the leading term of $\mathcal{M}_r$ is extracted from the tree to produce the new term of Q. The producer adds this term to the data structure of Q before the value n_q is incremented. Thus the other threads have always a coherent view of Q. After that the thread inserts the term of $Q_{n_q} \times_{[S_{r+1};S_r[} B$ to the tree.

3.2 Heap Method

Using the heap method, $\mathcal{M}_r$ is represented using a sorted singly linked list. When a thread attaches to an element $\mathcal{L}_r$ for the first time, this list is filled with the corresponding terms of A. An alternative data structure to this sorted list could be a B-tree which has logarithmic complexity for insertions and deletions.The terms $\mathcal{N}_r$ are merged and sorted using a chained binary heap with a complexity $\mathcal{O}(n_q n_b \log(n_q))$. This complexity is worse than $\mathcal{O}(n_q n_b \log(\min(n_q, n_b))$ of Monagan's division algorithm. The binary heap contains only the terms $Q_k B_j$.

The consumer threads only merge and sort the terms of the second part of $\mathcal{N}_r$. The generated terms are added to the previous list. In the worst case, as an element could be attached $n_q - 1$ times by the threads, the insertion of the terms to the list has a complexity of $\mathcal{O}(n_q n_a)$. In the best case, the insertion to the list is $\mathcal{O}(n_a)$ when the element is attached a single time by a consumer. When the thread detaches itself from an element, the binary heap is always empty, only the list remains non empty.

When a thread processes the producer element, the thread iterates on the following processing until the sorted list $\mathcal{M}_r$ becomes empty and the binary heap representing $\mathcal{N}_r$ is empty.

The thread removes the leading term from the binary heap and/or from the sorted linked list. This operation is done in $\mathcal{O}(1)$. If the extracted term, e.g. $Q_k B_j$, comes from the binary heap, the next term $Q_k + B_{j+1}$ is inserted to the heap if and only if $S_{r+1} \preceq \sigma_k + \beta_{j+1} \prec S_r$. Using this extracted term, it computes the next term of Q and adds the leading term of $Q_{n_q} \times_{[S_{r+1};S_r[} B$ to the binary heap. This producer never adds new terms in the sorted list.

4. CHOICE OF THE SET S

As the terms of Q are unknown at the beginning of the algorithm, the grid applied for the multiplication [5] could not be used for the division. All terms of A should be canceled by the division, some exponents of the array generated by the exponents of A could be selected using regular spacing. Moreover, the exponent of the leading term of Q allows to add an additional regular split over the exponents generated by the product $Q_1 B$. As an exact division is performed and if we assume that the division does not fail, the exponent of the tailing term of Q is $\alpha_{n_a} - \beta_{n_b}$. This exponent is used to applied another regular split over the exponents generated by the product $Q_{n_q} B$. Using these three rules, the selection of n_s exponents is done as following

$$S = \begin{cases} \alpha_i & \text{for } i = 1 \text{ to } n_a \text{ step } \lfloor \frac{2n_a}{n_s} \rfloor \\ \alpha_1 - \beta_1 + \beta_j & \text{for } j = 1 \text{ to } n_b \text{ step } \lfloor \frac{4n_b}{n_s} \rfloor \\ \alpha_{n_a} - \beta_{n_b} + \beta_j & \text{for } j = 1 \text{ to } n_b \text{ step } \lfloor \frac{4n_b}{n_s} \rfloor \\ 0_m & \\ \rho_{end} & \end{cases} \quad (5)$$

According to the monomial order, if $\alpha_1 \prec \beta_1$ or $\alpha_{n_a} \prec \beta_{n_b}$, then the exact division fails immediately. That selection may give duplicate exponents. These duplicated values are removed in order to avoid empty elements inside the list $\mathcal{L}$.

The chosen value for n_s should be much smaller in front of n_a and n_b but it must be larger than the number of threads in order to have the best possible load-balancing between the threads. This value could be tuned for the underlining hardware, such as the number of available cores, using random polynomials or using the tuned value of the multiplication. This tuned value may be different for the tree and heap method.

5. OPTIMIZATION

5.1 No Future Update of Q

If the exponent of $lt(B)$ equals to the exponent of $lt(\mathcal{P}_{n_q})$ of the producer's element then B cannot divide the future P_{n_q+1}. So we are sure that no more terms will be added to the quotient in the future when the remaining elements $\mathcal{H}$ will be processed. We use a shared variable $final$ which signals to the other threads that Q will not change in the future when it set to true. In that case, the thread processes the elements in its cache and iterates only once on the list $\mathcal{H}$ to process the pending work. This is done only to check that the remainder of the division is zero. During this last operation, if the thread finishes to process the element but $\mathcal{M}_r$ is not zero, then it signals that the division is not exact using the shared variable $failure$. This work will be done in the function $tryprocfinal$ which is a simplified version of the algorithm 2.

After that $final$ is set to true, any processing could be discarded if the caller of the algorithm knows that the division never fails, such as the divisions inside the Bareiss algorithm to calculate the determinant of a symbolic matrix. This can reduce the execution time by a large factor.

5.2 Dynamic Split

A perfect choice of S, to have the best load-balancing between the threads, could not be done. So we introduce that a polynomial $\mathcal{P}_r$, or its corresponding element L_r, could be decomposed in two parts if too much work must be done by the thread. Indeed, if a thread processes more than $2n_b/\mathcal{T}$ products for the single product $Q_k \times_{[S_{r+1};S_r[} B$, the corresponding element will be split in two parts. Of course, for too small values $2n_b/\mathcal{T}$, e.g. less than 64, this split is never done. We choose the exponent computed at the middle of $Q_k \times_{[S_{r+1};S_r[} B$ as the pivot for the decomposition. This decomposition does not break the properties of the list $\mathcal{L}_r$. Indeed, if the exponent γ is chosen as the pivot, the element $\mathcal{P}_r(\mathbf{x}; n_q; [S_{r+1}; S_r[)$ becomes $\mathcal{P}_r(\mathbf{x}; n_q; [\gamma; S_r[) + \mathcal{P}'_r(\mathbf{x}; n_q; [S_{r+1}; \gamma[)$. The element $\mathcal{P}'_r(\mathbf{x}; n_q; [S_{r+1}; \gamma[)$ could be inserted without any lock by the owner thread of the original element because a simple sorted singly linked list is used for the representation of $\mathcal{L}$. The split could be done in $\mathcal{O}(2^{d-c})$ for the tree method and in $\mathcal{O}(n_a)$ for the heap method. In practice, the split operation occurs with very low frequency.

6. BENCHMARKS

Three examples are selected to test the implementation of our algorithm. The two first examples are due to Fateman in [3] and Monagan and Pearce in [13].

- Example 1 : $A = p_1 = f_1 \times g_1$ and $B = f_1$ with $f_1 = (1 + x + y + z + t)^{35}$ and $g_1 = f_1 + 1$.
- Example 2 : $A = p_2 = f_2 \times g_2$ and $B = f_2$ with $f_2 = (1 + x + y + 2z^2 + 3t^3 + 5u^5)^{22}$ and $g_2 = (1 + u + t + 2z^2 + 3y^3 + 5x^5)^{22}$.
- Example 3 : $A = p_3 = f_3 \times g_3$ and $B = f_3$ with $f_3 = (1 + u^2 + v + w^2 + x - y^2)^{24}$ and $g_3 = (1 + u + v^2 + w + x^2 + y^3)^{24} + 1$.

For each example, we compute $A = p_i = f_i g_i$ and divide $A/B = p_i/f_i$ using the lexicographic order and graded lexicographic order. These three examples are executed on a computer with 8 Intel Xeon processors X7560 running at 2.27Ghz under the Linux operating system. Each processor has 8 physical cores sharing 24 Mbytes of cache. This computer has a total of 512Gbytes of RAM shared by its 64 cores. The parallel dynamic scheduling of the fourth and fifth step of the algorithm is performed by the OpenMP API. As the memory management could be a bottleneck in a multi-threading multiplication of sparse polynomials, the memory management is performed by the scalable allocator of the Intel Threading Building Blocks library.

Our algorithm, noted *DMPMC*, is compared to the computer algebra systems Maple 18 and Singular 4.0.1. The implementation of our algorithm requires that the memory operations, such as read, write, compare-and-swap on the shared variables, must be respected to be sure that all threads have a correct view of the values. The chosen value

Table 1: Time in seconds and speedup of the three examples on the shared memory computer

	order	software	1 core	4 cores		16 cores		64 cores	
$g_1 = p_1/f_1$	lex	DMPMC tree	230.4	57.7	4.0x	14.5	15.7x	4.4	54.6x
		DMPMC heap	480.0	119.5	4.0x	30.6	15.6x	8.8	54.2x
		Singular 4.0.1	911.6						
	grlex	DMPMC tree	229.4	57.6	4.0x	14.5	15.8x	4.3	53.4x
		DMPMC heap	475.0	59.6	4.0x	30.4	15.6x	8.4	55.4x
		Maple 18	262.7	61.5	4.3x	22.6	11.6x	22.9	11.5x
		Singular 4.0.1	3247.9						
$g_2 = p_2/f_2$	lex	DMPMC tree	359.9	92.9	3.9x	24.7	14.6x	7.5	48.1x
		DMPMC heap	393.1	107.3	3.6x	28.2	13.9x	8.5	46.3x
		Singular 4.0.1	911.6						
	grlex	DMPMC tree	458.6	119.8	3.8x	31.0	14.7x	9.6	47.7x
		DMPMC heap	829.4	211.3	3.9x	54.8	15.1x	16.3	50.8x
		Maple 18	306.2	211.9	1.5x	314.3	1.0x	377.5	0.8x
		Singular 4.0.1	2305.8						
$g_3 = p_3/f_3$	lex	DMPMC tree	400.0	100.7	4.0x	25.7	15.6x	7.4	53.7x
		DMPMC heap	668.2	171.9	3.9x	45.0	14.9x	13.0	51.3x
		Singular 4.0.1	4322.5						
	grlex	DMPMC tree	473.1	119.4	3.9x	30.0	15.7x	8.7	54.4x
		DMPMC heap	835.1	214.3	3.9x	56.4	14.8x	16.1	51.9x
		Maple 18	605.7	174.2	3.5x	174.4	3.5x	243.2	2.5x
		Singular 4.0.1	3850.5						

for n_s is the value 8192 obtained after a tuning step on random polynomials. In Maple 18 and DMPMC, the coefficients of the polynomials are represented with integers using a mixed representation. For the integers smaller than $2^{62}-1$ for Maple, respectively $2^{63}-1$ for DMPMC, on 64-bit computers, hardware integers are used instead of integers' type of the GMP library. The multiplication and additions of the terms use a three word-sized integers accumulator (a total of 192 bits) for the small integers. The tree nodes for DMPMC have exactly 16 children. Table 1 shows the execution times of the three examples. Only Maple and DMPMC provide a parallel algorithm for the exact division. The command 'divide' of Maple implements the Monagan's parallel division algorithm. We define the efficiency as $E_p = T_1/(pT_p)$ where T_1 is the execution time on a single core and T_p is the execution time on p cores. Figure 1 confirms that the division algorithm of Maple is designed for a single processor with multiple cores. Our algorithm maintains an efficiency of 0.7 on 64 cores. The change of the efficiency of our algorithm after 32 cores is due to the NUMA topology of our computer. Our implementation could be improved if a NUMA-aware work-stealing method is adopted to process the elements. Indeed, the *lock* integer contains the information about the last updater thread and could be reused for the work-stealing at line 25 of the algorithm 1 if the NUMA memory distance is known between the cores where the threads are executing.

The performance of the heap and tree method depends on the sparsity of the polynomials. The sparsity W is measured with the number of terms of the polynomials : $W(Q,A,B) = \frac{n_q n_b}{n_a}$. The third previous examples have the following sparsity: $W_1 = 5879.6, W_2 = 41.2, W_3 = 212$. W tends to 1 when the sparsity increases. We have generated 20 random polynomials in three variables between 35000 and 50000 terms. The maximal degree in each variable is up to 1000 and the coefficients are in the range $-99 \ldots 99$. We perform the products in pairs and them divide the result by both of them. Figure 2, which reports the efficiency of the division, shows that the tree method is largely affected by the sparsity. Indeed, the tree contains few terms per leaf for very sparse polynomials. Our heap method is much less affected by the sparsity of the polynomials. Even with 32 threads, we maintain an efficiency of 60% when W tends to 1. For 64 threads, about 85% divisions using the heaps have an efficiency greater than 0.6. The remaining divisions require many more memory operations because our algorithm lacks of a dynamic switch between quotient heap and remainder heap [14].

7. DIVISION WITH A REMAINDER

At the beginning of the algorithm, R equals 0. When the producer fails to divide P_{n_q} by the leading term of B, the producer adds the leading term of P_{n_q} to the remainder R. A term of R cancels only a term of M_r which has generated it. As a result, other elements of $\mathcal{L}$ don't need to check the remainder. The optimization provided in section 5.1 requires an attention because several threads may produce a part of the remainder at the same time. An efficient implementation is possible without any additional mutual-exclusion statement. During this step, each thread stores its part of remainder to an own data structure. After the parallel section, the reconstruction of the remainder is just the concatenation of these data structures in the same order as the list $\mathcal{L}$.

8. CONCLUSIONS

The presented algorithm for the sparse multivariate polynomials division achieves a large speedup on a large number of cores even with very sparse polynomials. Other data structures than tree or sparse data structures may be used for the merging and sorting terms' operations. Recent theoretical progress based on the sparse interpolation have been published for the multiplication [8, 1]. When an efficient

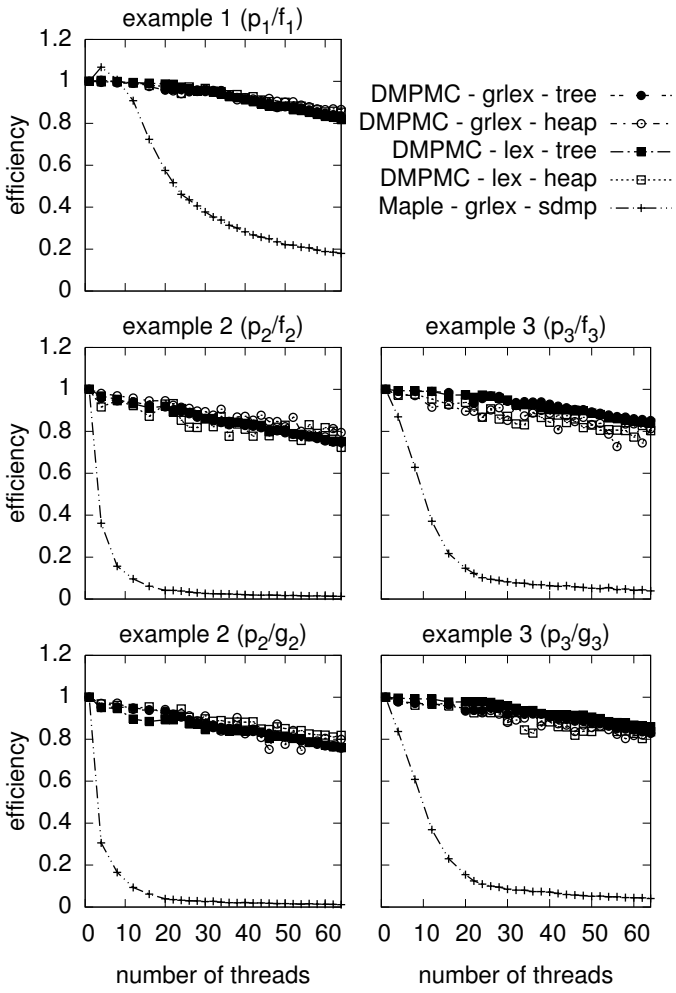

Figure 1: Efficiency to perform the division of the three examples on 64-cores computer using different parallel algorithms and monomial orderings.

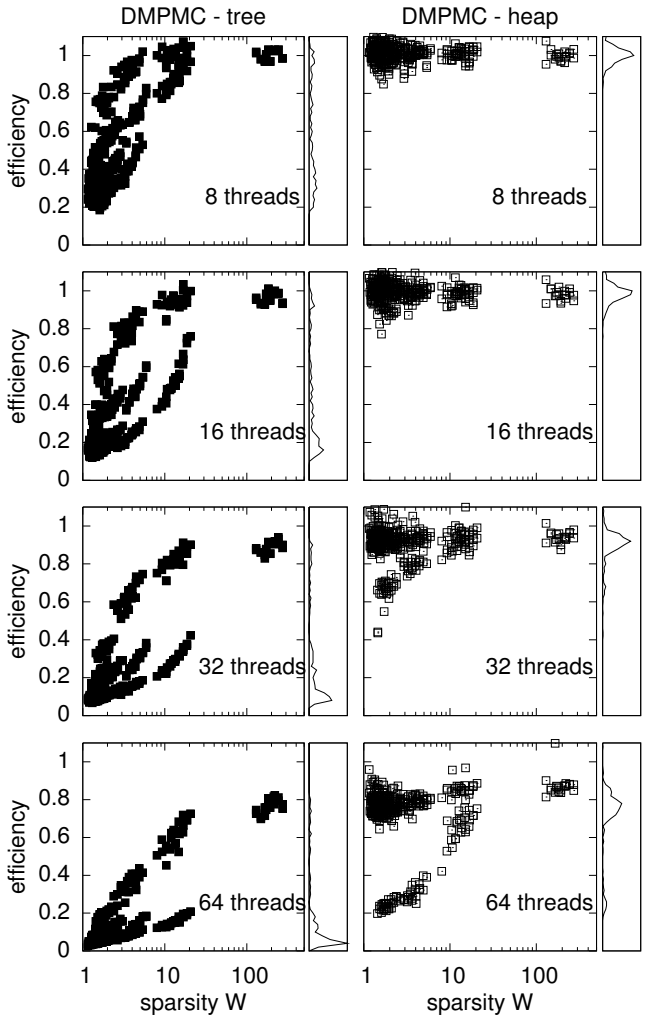

Figure 2: Efficiency of the DMPMC algorithm with different kind of containers to perform 380 divisions on sparse random trivariate polynomials with different sparsities. Its histogram on the right side.

parallel implementation of the division based on these results will be available, it could be interesting to compare them to our results.

9. ACKNOWLEDGMENTS

This work was granted access to the HPC resources of ICS, financed by Region Île de France and Equip@Meso (ANR-10-EQPX-29-01).

10. REFERENCES

[1] A. Arnold and D. S. Roche. Output-sensitive algorithms for sumset and sparse polynomial multiplication. *CoRR*, abs/1501.05296, 2015.

[2] K. Batselier, P. Dreesen, and B. Moor. The geometry of multivariate polynomial division and elimination. *SIAM Journal on Matrix Analysis and Applications*, 34(1):102–125, 2013.

[3] R. Fateman. Comparing the speed of programs for sparse polynomial multiplication. *SIGSAM Bull.*, 37(1):4–15, 2003.

[4] R. J. Fateman. Polynomial multiplication, powers and asymptotic analysis: Some comments. *SIAM Journal on Computing*, 3(3):196–213, 1974.

[5] M. Gastineau and J. Laskar. Highly scalable multiplication for distributed sparse multivariate polynomials on many-core systems. In V. Gerdt, W. Koepf, E. Mayr, and E. Vorozhtsov, editors, *Computer Algebra in Scientific Computing*, volume 8136 of *Lecture Notes in Computer Science*, pages 100–115. Springer International Publishing, 2013.

[6] J. V. Z. Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, New York, NY, USA, 2 edition, 2003.

[7] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[8] J. v. d. Hoeven and G. Lecerf. Sparse polynomial interpolation in practice. Technical report, HAL, 2014.

http://hal.archives-ouvertes.fr/hal-00980366.

[9] E. Horowitz. A sorting algorithm for polynomial multiplication. *J. ACM*, 22(4):450–462, Oct. 1975.

[10] S. C. Johnson. Sparse polynomial arithmetic. *SIGSAM Bull.*, 8(3):63–71, Aug. 1974.

[11] M. Monagan and R. Pearce. Polynomial division using dynamic arrays, heaps, and packed exponent vectors. In V. Ganzha, E. Mayr, and E. Vorozhtsov, editors, *Computer Algebra in Scientific Computing*, volume 4770 of *Lecture Notes in Computer Science*, pages 295–315. Springer Berlin Heidelberg, 2007.

[12] M. Monagan and R. Pearce. Parallel sparse polynomial multiplication using heaps. In *Proceedings of the 2009 International Symposium on Symbolic and Algebraic Computation*, ISSAC '09, pages 263–270, New York, NY, USA, 2009. ACM.

[13] M. Monagan and R. Pearce. Parallel sparse polynomial division using heaps. In *Proceedings of the 4th International Workshop on Parallel and Symbolic Computation*, PASCO '10, pages 105–111, New York, NY, USA, 2010. ACM.

[14] M. Monagan and R. Pearce. Sparse polynomial division using a heap. *Journal of Symbolic Computation*, 46(7):807 – 822, 2011. Special Issue in Honour of Keith Geddes on his 60th Birthday.

[15] C. G. Ponder. Parallel multiplication and powering of polynomials. *Journal of Symbolic Computation*, 11(4):307 – 320, 1991.

[16] D. S. Roche. Chunky and equal-spaced polynomial multiplication. *Journal of Symbolic Computation*, 46(7):791 – 806, 2011. Special Issue in Honour of Keith Geddes on his 60th Birthday.

[17] J. van der Hoeven and G. Lecerf. On the complexity of multivariate blockwise polynomial multiplication. In *Proceedings of the 37th International Symposium on Symbolic and Algebraic Computation*, ISSAC '12, pages 211–218, New York, NY, USA, 2012. ACM.

[18] P. S. Wang. Parallel polynomial operations on smps: An overview. *J. Symb. Comput.*, 21(4-6):397–410, June 1996.

[19] S. K. ÌĹhler and M. Ziegler. On the stability of fast polynomial arithmetic. In *Proceedings of the 8th Conference on Real Numbers and Computers*, pages 147–156, Santiago de Compostela, Spain, 2008.