Mixed representation of polynomials oriented towards fast parallel shift *

Eugene V. Zima
Department of Computational Mathematics and Cybernetics
Moscow State University, Moscow, 119899, Russia
e-mail: zima@cs.msu.su

Abstract

In this paper we consider the form of polynomial representation useful in problems connected with performing polynomial shift. We propose basic parallel algorithms suited for SIMD architecture to perform the shift in $O(1)$ time if we have $O(n^2)$ Processor Elements available, and the shift has to be performed repeatedly. Proposed algorithms are easy to generalize to multivariate polynomials shift. The possibility of applying these algorithms to polynomials with coefficients from non-commutative rings is discussed as well as the bit-wise complexity of the algorithm.

1 Introduction

Solution of many problems can be expedited, when "hardware shift" is involved in the computational process. Typical example for sequential computation is multiplication of an integer by 2^n. In parallel computation such an example gives Cannon matrix multiplication algorithm [2], which multiplies matrices in linear time using skew representation and parallel shift of matrices. In this paper we will consider representation of multivariate polynomials, in which the "shift" computer operation meets the "shift" mathematical transformation. This representation is multidimensional analog of tables of finite differences (TFD) and is suited for SIMD architecture. We will describe basic algorithms to convert polynomials into TFD-based representation and show the possibility of expediting computation of some standard problems. Throughout the paper we suppose, that we have enough Processor Elements (PEs) for representing considered objects.

2 Univariate preliminaries

In this section we consider a form of polynomial representation based on finite differences. This form is analogous to a dense form (table of coefficients) [5] and has the same features as any other form of polynomial representation. The primary feature of this form is its orientation towards performing the fast parallel shift.

2.1 Tables of finite differences as the form of polynomial representation

Consider a polynomial

$$f(x) = a_n x^n + \ldots + a_1 x + a_0 \tag{1}$$

defined by the list of coefficients $(a_0, a_1, \ldots, a_n)$. This list gives the usual dense representation of the polynomial (all operations on polynomials can be defined as operations on such lists [5]).

Consider now the table of finite differences (TFD) of $f(x)$, taken at the point $x = x_0$ with step h:

$$\Phi = [\varphi_0, \varphi_1, \varphi_2, \ldots, \varphi_n]. \tag{2}$$

Here $\varphi_0 = f(x_0)$ (0-order difference of $f(x)$), $\varphi_1 = f(x_0 + h) - f(x_0)$ (first-order difference of $f(x)$), $\varphi_2 = f(x_0 + 2h) - 2f(x_0 + h) + f(x_0)$ (second-order difference of $f(x)$), and so on. This table contains all the information about the initial polynomial (1). In fact it is just another form of polynomial representation [13]. Operations on polynomials can be formulated in terms of such tables.

Given polynomials $f(x)$ and $g(x)$ defined by tables $\Phi = [\varphi_0, \varphi_1, \ldots, \varphi_n]$ and $\Psi = [\psi_0, \psi_1, \ldots, \varphi_m]$ respectively, the table for polynomial $f(x) \pm g(x)$ looks as following: [1]

$$[\varphi_0 \pm \psi_0, \varphi_1 \pm \psi_1, \ldots, \varphi_m \pm \psi_m, \varphi_{m+1}, \ldots, \varphi_n]. \tag{3}$$

In its turn, the table for polynomial $r(x) = f(x)g(x)$ is $\Delta = [\delta_0, \delta_1, \ldots, \delta_{n+m}]$, where for $t = 0, 1, \ldots, n+m$

$$\delta_t = \sum_{u=max(0,t-m)}^{min(t,n)} \binom{t}{u} \varphi_u \sum_{v=t-u}^{min(t,m)} \binom{u}{t-v} \psi_v. \tag{4}$$

Formulae (3) and (4) give an analog of the formulae for coefficients of the sum and the product of two polynomials in usual dense representation.

*Work reported herein was supported in part by the RFBR and INTAS under Grant 95-IN-RU-412.

[1]we suppose here without loss of generality, that $n \geq m$.

2.2 Polynomial shift in TFD representation

Suppose we are provided with a table of finite differences (2) for a given polynomial (1). It is possible to compute subsequent values of $f(x)$ for $x = x_0, x_0 + h, x_0 + 2h, \ldots$ performing only n additions on each step ([7]). In the parallel case (in particular, vector or SIMD architectures), it is possible to compute these values using only two parallel operations on each step - parallel shift to the left and parallel addition.

Let A be an array $A[0], A[1], \ldots, A[n]$ and let $\mathtt{LeftShift}(A)$ denote a parallel shift of A to the left by one component, i.e., $\mathtt{LeftShift}(A)$ performs the following assignments
$A[0] := A[1]; \ldots A[n-1] := A[n]; A[n] := 0$.

If components of A are set by values $\varphi_0, \varphi_1, \ldots, \varphi_n$, then the following parallel assignment

$$A := A + \mathtt{LeftShift}(A) \tag{5}$$

updates A by the table of finite differences of $f(x)$ taken at the point $x = x_0 + h$ $(A[0] = f(x_0 + h))$.

Remark 1 *Observe, that this table of finite differences is the same as the table of finite differences for polynomial $f(x+h)$, taken at the point $x = x_0$. Another words, in TFD representation, the "shift" computer operation meets the "shift" mathematical transformation.*

Performing assignment (5) one more time, we get the table of finite differences of $f(x)$ taken at the point $x = x_0 + 2h$ $(A[0] = f(x_0 + 2h))$ and so on.

2.3 Conversion to TFD representation

From now on we will consider TFD taken at the point $x_0 = 0$ with the step $h = 1$ and then show, that all results hold for the general case. We start with a simple observation:

Remark 2 *Formulae (3) and (4) do not depend on the domain of values φ_j, ψ_k. Components of tables Φ and Ψ could be numbers, symbols, polynomials (in the usual dense representation). If we know how to add and multiply φ_j, ψ_k, we know also how to add and multiply TFDs Φ and Ψ. Another words, here the situation is the same as for polynomials in the usual dense (list of coefficients) representation: if we know how to add and multiply coefficients, we also know how to add and multiply polynomials.*

Now, let's derive from (4) a simple case of the TFD-multiplication: given polynomial $f(x)$, defined by TFD $\Phi = [\varphi_0, \varphi_1, \ldots, \varphi_k]$ and polynomial x defined by TFD $[0, 1]$, the polynomial $f(x)x$ will be defined by TFD

$$[0, (\varphi_0+\varphi_1), 2(\varphi_1+\varphi_2), \ldots, k(\varphi_{k-1}+\varphi_k), (k+1)(\varphi_k+0)]. \tag{6}$$

Let $\mathtt{RightShift}(A)$ denote parallel shift of the array A to the right by one component, i.e. performs the following assignments
$A[n] := A[n-1]; \ldots A[1] := A[0]; A[0] := 0$.
If we have prepared in advance the special array *mltr* such that $mltr[i] = i + 1, i = 0, 1, \ldots, n$, we are able to get TFD for $\Phi \cdot x$ (by given array A of components of Φ) in 4 parallel steps (two parallel shifts, one component-wise multiplication and one component-wise addition):

$$A := \mathtt{RightShift}((A + \mathtt{LeftShift}(A)) * mltr). \tag{7}$$

The last means, that for an n-degree polynomial $f(x)$ given by coefficients $s[0], s[1], \ldots, s[n]$, we are able to construct TFD in $O(n)$ parallel steps using the Horner scheme and (7):

Algorithm 1. Conversion of univariate polynomial to TFD representation.
`Input:` array $s[0..n]$ of coefficients of polynomial $f(x)$
`Output:` array $A[0..n]$, which is the TFD representation of polynomial $f(x)$

```
mltr[j]:=j+1 for all j=0,1,...,n;
A[j]:=0 for all j=0,1,...,n;  A[0]:=s[n];
for k:=n-1 downto 0 do
 A:= RightShift((A + LeftShift(A)) * mltr);
 A[0]:= s[k]
od;
```

Consider example. Let $f(x) = 6x^2 - 12x - 5$, i.e. $s[0] = -5$, $s[1] = -12$, $s[2] = 6$. Before the loop, array $mltr = [1, 2, 3]$, array $A = [6, 0, 0]$. After the first iteration of the loop $A = [-12, 6, 0]$, and after the second step, $A = [-5, -6, 12]$, which is the TFD of $f(x)$ taken at the point 0 with the step 1.

2.4 Simple application example

One of the typical computational problems connected with finding polynomial solutions of linear operator equations [1] is: given polynomials $u_1(x), \ldots, u_l(x)$, evaluate polynomials $u_i(x)$ at successive integer values of x.

Denote $n = \max \deg u_i$ and compose an l by $(n+1)$ matrix U, putting coefficients of $u_i(x)$ in the i-th row and filling empty elements by 0. Let $MLTR$ be composed of l rows equal to *mltr* above and **LeftShift** and **RightShift** act on all rows of matrices simultaneously (which is usual shift-operation for SIMD machines). Then, after preliminary work of complexity $O(n)$

```
MLTR[i,j]:=j+1 for all i=1,...,l;j=0,1,...,n;
A[i,j]:=0 for all i=1,...,l;j=0,1,...,n;
A[i,0]:=U[i,n] for all i=0,1,...,l;
for k:=n-1 downto 0 do
 A:= RightShift((A + LeftShift(A)) * MLTR);
 A[i,0]:= U[i,k] for all i=0,1,...,l;
od;
```

we get TFDs for all u_i in the array A. The first column of the matrix A contains values $u_i(0)$. Now, we are able to get successive values of all u_i at the points $x = 1, 2, \ldots$, spending $O(1)$ ring operations on each step:

```
 A := A + LeftShift(A).
```

Recall, that values needed can be found in the first column of A.

3 Basic SIMD operations

An usual dense representation of d-variate polynomial $f(x_1, \ldots, x_d)$ is d-dimensional array of coefficients. Considering basic SIMD-like operations on such arrays we will suppose that each entry of such an array is located in separate PE and neighbors entries are located in neighbors PEs. Given a d-dimensional array $s[0..n_1, \ldots, 0..n_d]$ and $i \in \{i_1, \ldots, i_d\}$, we will use the following operations as basic:

- $\mathtt{LeftShift}_i(s)$ shifts an array s one component to the left in the i direction (here "left" means towards decreasing i);

- $\texttt{RightShift}_i(s)$ shifts an array s one component to the right in the i direction (here "right" means towards increasing i);
- $s|_{i=j}$ denotes the $(d-1)$-dimensional sub-array of the array s obtained by fixing the value of the index $i=j$, where $0 \le j \le n_i$;
- $F(j)|_{i=j}$ denotes the d-dimensional array of the same shape as s, whose elements for $i=j$ and for any values of other indexes $i_1,\dots,i_d$ are equal to $F(j)$.

Observe, that every operation like shifting, computing $s|_{i=j}$ or e.g., $(j+1)|_{i=j}$, corresponds to a single parallel instruction on a SIMD machine and takes constant time [2] (of course under assumption that we do have enough PEs).

Additionally we consider binary parallel operations as basic. Let s and u be d-dimensional arrays of the same shape. Further we will use

- $s+u$ – component-wise addition of s and u;
- $s*u$ – component-wise multiplication of s and u;
- $u:=s$ – component-wise assignment.

As usually for SIMD computations we assume that arrays of the same shape are mapped to the same set of PEs, i.e. entries $u[i_1,\dots,i_d]$ and $s[i_1,\dots,i_d]$ for fixed $i_1,\dots,i_d$ are located in the same PE. That is why binary operations on these arrays take constant time.

Using operations above we can compose more complex expressions. For example,

$$u := (2^j)|_{i_2=j} * s \tag{8}$$

can be rewritten for explanation sequentially as

```
for all $i \in \{i_1,\dots,i_d\} \& i \neq i_2$ do
  for $j := 0,1,\dots,n_2$ do
     $u[i_1,j,i_3,\dots,i_d] := 2^j * s[i_1,j,i_3,\dots,i_d]$
  od
od
```

This explanations contains loops, but parallel complexity of assignment (8) is constant and consists of the following three steps:

1. temporary parallel variable of the same shape as s is assigned by values $(2^j)|_{i_2=j}$,
2. parallel multiplication of this variable and s,
3. parallel assignment of the result to u.

Let s be as earlier a d-dimensional array, u be a $(d-1)$-dimensional array, and $i \in \{i_1,\dots,i_d\}$. A bit more complex operations needed further are

- $\texttt{ReduceAdd}_i(s)$, which returns $(d-1)$-dimensional array
$$\sum_{j=0}^{n_i} s|_{i=j};$$
- $\texttt{CopySpread}_i(u)$, which returns d-dimensional array obtained by creating and spreading n_i+1 copies of u along axis i.

Both these operations needs $O(\log n_i)$ parallel steps ([3, 8]).

[2]We assume here and further until section 5.4, that the complexity is counted in the number of ring operations.

4 Mixed TFD-based representation of multivariate polynomials

Given d-variate polynomial $f(x_1,\dots,x_d)$ represented by d-dimensional array $s[0..n_1,\dots,0..n_d]$ and fixed $p \in \{1,\dots,d\}$. We can consider this polynomial as an univariate polynomial in x_p with coefficients

$$s|_{i_p=0}, s|_{i_p=1},\dots, s|_{i_p=n_p},$$

which are representations of $(d-1)$-variate polynomials.

Define arrays $MLTR$ and A of the same dimension and shape as s. Substitute in Algorithm 1: $s|_{i_p=k}$ instead of $s[k]$, $\texttt{LeftShift}_{i_p}, \texttt{RightShift}_{i_p}$ instead of $\texttt{LeftShift}, \texttt{RightShift}$ and $MLTR := (j+1)|_{i_p=j}$ instead of `mltr[j]:=j+1`. We get an algorithm to construct TFD A in variable x_p for this unvariate polynomial. Each element of this TFD is a $(d-1)$-dimensional array, representing some polynomial in $x_1,\dots,x_{p-1},x_{p+1},\dots,x_d$.

Denote by I_f a subset of the set of indexes $\{1,\dots,d\}$. A mixed representation of f connected with I_f is such an array A, that

- for each $p \in I_f$
$$A|_{i_p=0}, A|_{i_p=1},\dots, A|_{i_p=n_p},$$
is a TFD for f viewed as an univariate polynomial in x_p;
- for each $p \notin I_f$
$$A|_{i_p=0}, A|_{i_p=1},\dots, A|_{i_p=n_p},$$
is a list of coefficients for f viewed as an univariate polynomial in x_p.

Given an array s as above and I_f. Algorithm to construct mixed representation connected with I_f looks as following.

Algorithm 2. Conversion of multivariate polynomial to mixed TDF-based representation.

`Input:` array $s[0..n_1,\dots,0..n_d]$ of coefficients of polynomial $f(x_1,\dots,x_d)$ and I_f

`Output:` array $A[0..n_1,\dots,0..n_d]$, which is the mixed TFD-based representation of polynomial $f(x)$

```
for each $p \in I_f$ do
   $MLTR := (j+1)|_{i_p=j}$;
   $A[i_1,...,i_d] := 0$ for all $i_1,...,i_d$;
   $A|_{i_p=0} := s|_{i_p=n_p}$;
   for $k := n_p - 1$ downto 0 do
      $A := \texttt{RightShift}_{i_p}((A + \texttt{LeftShift}_{i_p}(A)) * MLTR)$;
      $A|_{i_p=0} := s|_{i_p=k}$
   od;
   $s := A$
od
```

If $I_f = \{1,\dots,d\}$ then this algorithm constructs "pure" TFD representation, and it will take $n_1+n_2+\dots+n_d$ parallel steps to get it. We will show the use of a mixed representation in the following section. Now we define two parallel operations on polynomials in mixed representation.

Given two polynomials of the same dimension f and g represented in mixed form by arrays u and v of the same shape. Let $I_f = I_g$ and $p \in \{1,\dots,d\}$. Then

1. the representation of $f+g$ can be got by component-wise parallel addition of arrays u and v, and $I_{f+g} = I_f = I_g$;

2. the representation of $f \cdot x_p$ is an array obtained by the operation
$\texttt{RightShift}_{i_p}((u + \texttt{LeftShift}_{i_p}(u)) * (j+1)|_{i_p=j})$, if $p \in I_f$, or
$\texttt{RightShift}_{i_p}(u)$, if $p \notin I_f$.

5 Fast parallel computation of the polynomial shift

Given a polynomial (1) with coefficients from an arbitrary ring, the shift of $f(x)$ by a constant c is the operation which computes coefficients $b_0, b_1, \dots, b_n$ of the polynomial

$$g(x) = f(x+c) = b_n x^n + \dots + b_1 x + b_0.$$

This operation is used in many applications such as, for example, polynomial root isolation ([4]), changing polynomial basis, etc. The straightforward formula desired for computation of coefficients is

$$b_j = \sum_{k=j}^{n} a_k \binom{k}{j} c^{k-j}, \quad j = 0, 1, \dots, n. \qquad (9)$$

The complexity of the shift (the number of ring operations performing the shift) computed sequentially by this formula is $O(n^3)$. Known methods (such as the Horner scheme or "synthetic division" [9]) reduce the complexity to $O(n^2)$. An advanced sequential algorithm from [10] performs this operation in $O(n \log n)$ steps via FFT polynomial multiplication. Parallelization of the Horner scheme [9] gives $O(n)$ complexity of the procedure.

We consider $c = 1$ in this section and concentrate on the case in which the shift by a given constant c has to be performed several times (repeatedly). We propose a parallel algorithm suited for SIMD architectures to perform the shift in $O(1)$ time. To achieve this speed of the algorithm, $O(n^2)$ Processor Elements (PEs) have to be available. The algorithm is based on converting a given polynomial to a mixed form of representation. The complexity of the conversion is $O(n)$.

5.1 Outline of an approach to fast parallel shift

Given polynomial (1) defined by the list of coefficients, we are interested in values of coefficients for polynomials $f(x+1), f(x+2)$ and so on. The main idea is to construct mixed representation for $f(x+y)$: TFD-based in y and usual in x. Conversion to the mixed representation consists of two steps:

1. Substitute $x+y$ for x in (1) and collect coefficients near every degree of x. We will get a polynomial in x whose coefficients are polynomials in y:

$$f(x+y) = u_n(y)x^n + \dots + u_1(y)x + u_0(y). \qquad (10)$$

Obviously, $u_j(0) = a_j, j = 0, 1, \dots, n$,
$u_j(1), j = 0, 1, \dots, n$ are coefficients of $f(x+1)$,
$u_j(2), j = 0, 1, \dots, n$ are coefficients of $f(x+2)$
and so on.

2. Construct the table of finite differences for each $u_j(y)$ at the point 0 and compose the matrix with these tables as rows (the first column of this matrix is nothing more than list of coefficients a_j).

After such preparation we are able to perform all assignments (5) simultaneously for every row of the matrix. This means, that the list of coefficients of $f(x+1)$ can be obtained in two parallel operations (parallel left shift of the matrix and parallel addition). The same steps can be used to get coefficients of $f(x+2)$ and so far.

Consider the following example. Let $f(x) = 2x^3 - 6x^2 - 5x + 1$. After first step of conversion (substitution and collecting coefficients) we get the polynomial $2x^3 + (6y-6)x^2 + (6y^2 - 12y - 5)x + 2y^3 - 6y^2 - 5y + 1$. After constructing tables of finite differences we have

$$\begin{array}{lll} u_0(y) & = 2y^3 - 6y^2 - 5y + 1 & = [1, -9, 0, 12] \\ u_1(y) & = 6y^2 - 12y - 5 & = [-5, -6, 12] \\ u_2(y) & = 6y - 6 & = [-6, 6] \\ u_3(y) & = 2 & = [2] \end{array}$$

Using the fact that for any TFD $[\varphi_0, \varphi_1, \dots, \varphi_n] = [\varphi_0, \varphi_1, \dots, \varphi_n, 0]$, we compose the matrix

$$U = \begin{pmatrix} 1 & -9 & 0 & 12 \\ -5 & -6 & 12 & 0 \\ -6 & 6 & 0 & 0 \\ 2 & 0 & 0 & 0 \end{pmatrix}.$$

Now we need to perform one parallel operation to get $\texttt{LeftShift}(U)$:

$$\texttt{LeftShift}(U) = \begin{pmatrix} -9 & 0 & 12 & 0 \\ -6 & 12 & 0 & 0 \\ 6 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

and one more parallel operation (addition of the last two matrices) to get whole result of the assignment $U := U + \texttt{LeftShift}(U)$. After this assignment

$$U = \begin{pmatrix} -8 & -9 & 12 & 12 \\ -11 & 6 & 12 & 0 \\ 0 & 6 & 0 & 0 \\ 2 & 0 & 0 & 0 \end{pmatrix}.$$

The first column of U contains coefficients of the polynomial $f(x+1) = 2x^3 - 11x - 8$.

Repeating these two steps again we get

$$\texttt{LeftShift}(U) = \begin{pmatrix} -9 & 12 & 12 & 0 \\ 6 & 12 & 0 & 0 \\ 6 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

and after performing $U := U + \texttt{LeftShift}(U)$

$$U = \begin{pmatrix} -17 & 3 & 24 & 12 \\ -5 & 18 & 12 & 0 \\ 6 & 6 & 0 & 0 \\ 2 & 0 & 0 & 0 \end{pmatrix}.$$

The first column of U now contains coefficients of the polynomial $f(x+2) = 2x^3 + 6x^2 - 5x - 17$.

Clearly, with this representation we are able to perform the parallel shift of the given polynomial in $O(1)$ (precisely two) parallel steps.

5.2 Conversion of $f(x)$ to the mixed form

After substitution $x = x + y$ into given polynomial (1) we can rewrite it in Horner-like manner:

$$((\dots(a_n(x+y) + a_{n-1})(x+y) + \dots)(x+y) + a_1)(x+y) + a_0,$$

and exploit operations 1,2, defined at the end of Section 4. These gives the following

Algorithm 3: Conversion to mixed representation.

```
Input: list $a[0], a[1], \ldots, a[n]$ of coefficients of polynomial $f(x)$
Output: two-dimensional array $U[0..n, 0..n]$ - TFD-based mixed representation of polynomial $f(x)$
Complexity: $O(n)$
  $MLTR[i,j] := j+1$ for all i=0,...,n,j=0,...,n;
  $U[i,j] := 0$ for all i=0,1,...n, j=0,1,...,n;
  $U[0,0] := a[n]$;
  for k:=n-1 downto 0 do
      $w := \mathtt{RightShift}_i(U)$;
      $U := \mathtt{RightShift}_j((U + \mathtt{LeftShift}_j(U)) * MLTR) + w$;
      $U[0,0] := a[k]$
  od;
```

It is easy to see that this algorithm is the concretization of Algorithm 2.

5.3 Performing the shift

After all preliminary work it's now rather easy to formulate the following

Algorithm 4: Polynomial shift in the mixed TFD-based representation.

```
Input: two-dimensional array $U[0..n, 0..n]$ - the TFD-based mixed representation of polynomial $f(x)$
Output: the same array, which is the TFD-based representation of polynomial $f(x+1)$
Complexity: $O(1)$
        $U := U + \mathtt{LeftShift}_j(U)$
```

Recall, the coefficients of $f(x+1)$ can be found in the first column of U after performing this operation.

5.4 Polynomial shift in the root isolation context

The polynomial shift in the root isolation context uses, as the rule, an arbitrary precision arithmetic. That is why the bit-wise complexity of this operation is important. Let $z = \max|a_i|$ in (1) and $L(u)$ stands for the bit-length of an integer u ($L(nm) \cong L(n) + L(m)$, $L(n^m) \cong mL(n)$). The parallel algorithm to perform polynomial shift ([9]) uses $O(n)$ ring operations. The bit-wise cost of one ring operation is $O(nL(z))$. Therefore the bit-wise complexity of the shift is

$$O(n^2 L(z)). \tag{11}$$

When we convert (1) into TDF (2) the size of integers involved grows essentially. The reasonable question here, how does this fact affect the bit-wise complexity of TDF-based polynomial shift. Let $Z = \max|\varphi_i|$ in (2). It can be easily derived from Algorithm 1 that $Z = o(zn^n)$, $zn! = o(Z)$ (see also [6]) . The bit-wise complexity of the addition $\varphi_i + \varphi_{i+1}$ is bounded by $L(Z) = L(z) + n\log n$. The same holds for the bit-wise complexity of the `LeftShift` instruction. Therefore the bit-wise complexity of TDF-based polynomial shift is bounded by $2L(Z) = O(L(z) + n\log n)$, which is still better then (11).

Real root isolation algorithms ([4, 9]) use (together with shift) another transformations of polynomials:

- $H_{1/2^k} : f(x) \to f(x/2^k)$, and
- $R : f(x) \to x^n f(1/x)$.

For mixed representation considered above the first transformation does not give a troubles, because it can be implemented by the operation

$$U * (\frac{1}{2^k})|_{i=k}.$$

However the second transformation (which reverse the list of coefficients of f) makes prepared in advance mixed representation U useless.

We would like to avoid reconstruction of U from afresh after every reversion. For this purposes we split data involved in the problem and consider the following representation of needed objects:

- 3-dimensional array $U[0..n, 0..n, 0..n]$ with entries $U[k,i,j]$ such that 2-dimensional array $U|_{k=m}$ is mixed representation of $(x+y)^m$ for $m = 0, 1, \ldots, n$. Observe that for given $f(x)$ this array can be constructed by a bit modified Algorithm 3 in $O(n)$ parallel steps.
- 2-dimensional array $A[0..n, 0..n]$ with entries $A[k,i]$ such that $A|_{k=m} = a_m$ for $m = 0, 1, \ldots, n$.
- 1-dimensional array $a[0..n]$ of coefficients of $f(x)$.

Transformations needed for root isolation can be defined by the following parallel operations:

1. $E : f(x) \to f(x+1)$:
   a) $U := U + \mathtt{LeftShift}_i(U)$,
   b) $A := U|_{j=0} * A$,
   c) $a := \mathtt{ReduceAdd}_k(A)$.
2. $H_{1/2^k}(f(x))$:
   $$A := A * (1/2^k)|_{i=k},$$
   which involves 2-dimensional arrays, or alternatively
   $$U := U * (1/2^k)|_{i=k},$$
   which involves 3-dimensional arrays.
3. $R(f(x))$:
   $\mathtt{Reverse}(a)$ and $A := \mathtt{CopySpread}_k(a)$.

Let's proceed with brief bit-wise complexity analysis of described transformations. The bit-length of entries of U is bounded by $n\log n$ and cost of the assignment 1.a) is $O(n\log n)$. But values of $U|_{j=0}$ are not so large as $n\log n$. It was shown in [9] that coefficients of $f(x+1)$ are bounded by $2^n z$ and the bit-length of coefficients of $f(x+1)$ is bounded by $L(z) + n$. Thus the cost of multiplication in 1.b) is bounded by $L(z)n$, and cost of the assignment in 1.c) is bounded by $\log n(n + L(z))$. The cost of operation in 2 is at most $O(n)$. Finally the cost of $\mathtt{CopySpread}_k(a)$ (which should be performed after shifting) is bounded by $\log n(n + L(z))$. Therefore all the operations involved are a bit faster then (11).

In order to avoid reversions of the array a it makes sense to consider together with U an array V of the same dimension and shape, such that $V[k,i,j] = U[n-k,i,j]$, $k = 0, 1, \ldots, n$. In this case every time when U is shifted, V has to be shifted as well, and we can alterate multiplication of A by $U|_{j=0}$ and $V|_{j=0}$.

6 Generalizations

Remark 3 *The fast parallel polynomial shift by* 1 *is based on formula (6). All the reasoning for the shift by a constant* c *stay the same. The only difference is that the polynomial* x *has in this case the TFD representation* $[0, c]$ *instead of* $[0, 1]$. *Therefore, (6) should be rewritten as*
$[\varphi_0, \varphi_1, \ldots, \varphi_k] \cdot [0, c] =$
$[0, c(\varphi_0 + \varphi_1), 2c(\varphi_1 + \varphi_2), , \ldots, (k+1)c(\varphi_k + 0)]$,
which changes only one preliminary assignment in Algorithm 3:
`MLTR[i,j]:=c*(j+1) for all i,j=0,...,n.`
Algorithm 4 remains the same as earlier.

Given d-variate polynomial $F(x_1, \ldots, x_d)$, defined by the d-dimensional array of coefficients, and a list of constants $c_1, \ldots, c_d$, the shift of $F(x_1, \ldots, x_d)$ by $c_1, \ldots, c_d$ is the operation which computes the d-dimensional array of coefficients of the polynomial

$$F(x_1 + c_1, \ldots, x_d + c_d).$$

Combination of Algorithms 2 and 3 with $I_F = \{1, \ldots, d\}$ allows us to get the multivariate TFD-based representation of F. If $n = \max_i(\deg x_i)$ in F, then the complexity of such a conversion is $O(dn)$ of parallel ring operations. When the TFD-based representation of F is constructed, the polynomial shift can be computed by consequent computation of univariate shifts along $x_1, x_2, \ldots, x_d$ [3]. Therefore, the complexity of the multivariate shift in a multivariate TFD-based representation is $O(d)$.

Neither Algorithm 1 nor Algorithms 2, 3, 4 assume commutativity of the multiplication in the domain of polynomial coefficients. It means, that above approach can be applied, for example, to polynomials of the form (1), where coefficients a_j are square matrices of the same size. All the reasoning remain the same. However, assignments such as `U[0,0]:= a[j]` will be assignments of matrices in this case. The complexity of algorithms remains the same, if it is counted in terms of the number of matrix operations.

7 Conclusion

In this paper we considered a special form of polynomial representation oriented towards fast parallel computation of polynomial shifts. This form is the specialization of a more general approach [11, 12, 14] of symbolic conversion of numerical computational schemes to chains of recurrences, which can be evaluated in "shift-and-operate" style. The specialization enables parallel computation on the preparation stage (reducing the time of conversion). It is interesting to observe here, that the conversion to the TFD representation and parallel polynomial shift itself use very similar parallel tools: parallel shifts and parallel additions (multiplications).

Of course we are able to perform the polynomial shift fast, because we pay memory (PEs). In order to perform algorithms 3, 4 with the speed announced, we need $(n+1)^2$ PEs for proceeding n-degree polynomials. At the same time straightforward approach [9] allows to do the same work in $O(n)$ time on $n+1$ PEs. In d-variate case the situation is even harder: for $F(x_1, \ldots, x_d)$ which occupies an array of the size $O((n+1)^d)$ we need $O(((n+1)^d)^2)$ PEs to perform shift in $O(d)$ time. However, all this is usual situation in programming theory and practice. The possibility to choose competitive algorithm looks quite attractive. Especially if we take into account, that the complexity of performing parallel polynomial shift in the mixed TFD representation, counted in the number of ring operations (2 parallel operations), seems to be unimprovable.

[3] It is interesting to observe here, that the result of all computations does not depend of the order of univariate shifts; it will be the same, as e.g. for consequent shifts along $x_d, x_{d-1}, \ldots, x_1$. The only important thing here is to proceed shift along every x_i.

Acknowledgment

I would like to thank Thierry Gautier (ETH, Zurich) who provided me with useful comments on earlier draft.

References

[1] S. Abramov, M. Bronstein, M. Petkovšek. *On polynomial solutions of linear operator equations.* In *ISSAC'95*, pages 290–296, Montreal, Canada, July 1995. ACM Press.

[2] S.G. Akl. *The Design and Analysis of Parallel Algorithms.* Prentice Hall, Englewood Cliffs, New Jersey 07632, 1989.

[3] *C* users guide.* Thinking Machine Co., 1992.

[4] G. Collins, J. Johnson, W. Küchlin. Parallel real root isolation using sign variation method. In R.Zippel, editor, *Computer Algebra and Parallelism*, pages 71–78. Springer Verlag, LNCS 584, 1992.

[5] J. Davenport, Y. Siret, E. Tournier. *Calcul formel.* Masson, 1987.

[6] R. Graham, D. Knuth, O. Patashnik. *Concrete Mathematics.* Addison-Wesley, 1993.

[7] D. Knuth. *The art of computer programming. Vol.2.* Addison-Wesley, 1969.

[8] W. Koch. Efficient Reduce and Scan Functions for Mesh-Connected SIMD Computers. In *PARCELLA'96*, pages 174–183, Berlin, Germany, 1996. Akademie Verlag.

[9] W. Krandick. Isolierung reeller nullstellen von polynomen. In J.Herzberger, editor, *Wissenschaftliches Rechnen*, pages 105–154. Akademie Verlag, Berlin, 1995.

[10] A. Schönhage, A.F.W. Grotefeld & E. Vetter. *Fast Algorithms – A multitape Turing machine implementation.* BI Wissenschaftsverlag, Mannhaim, 1994.

[11] E. Zima. Automatic construction of system of recurrence relations. *USSR Comput. Maths. Math. Phys.*, vol. 24(6):193–197, 1984.

[12] E. Zima. Recurrent relations technique to vectorize function evaluation in loops. In *PARCELLA'94*, pages 161-168, Potsdam, Germany, 1994. Akademie Verlag.

[13] E. Zima. Simplification and optimization transformations of chains of recurrences. In *ISSAC'95*, pages 42–50, Montreal, Canada, July 1995. ACM Press.

[14] E. Zima, T. Casavant, K. Vadivelu. Mapping techniques for parallel evaluation of chains of recurrences. In *IPPS'96*, pages 620–624, 1996.