

MASSIVELY PARALLEL COMPUTATIONS ON MANY-VARIABLE POLYNOMIALS

“when seconds count. . .”

by

Bernard Beauzamy

Institut de Calcul Mathématique

Université de Paris 7

2 Place Jussieu, 75251 Paris Cedex 05

Jean-Louis Frot

Institut de Calcul Mathématique

Université de Paris 7

2 Place Jussieu, 75251 Paris Cedex 05

and

Christian Millour

Etablissement Technique Central de l'Armement

16 bis Avenue Prieur de la Côte d'Or, 94110 Arcueil

Abstract. – We show that a multivariate homogeneous polynomial can be represented on an hypercube, in such a way that sums, products, partial derivatives, can be performed by massively parallel computers. This representation is derived from the theoretical results of Beauzamy-Bombieri-Enflo-Montgomery [1]. The norm associated with it, denoted by $[\cdot]$, is itself a very efficient tool : when products of polynomials are performed, the best constant in inequalities of the form $[PQ] \geq C [P][Q]$ are provided, and the extremal pairs (that is the pairs of polynomials for which the product is as small as possible) can be identified.

Supported by the C.N.R.S. (France) and the N.S.F. (U.S.A.),
by contracts E.T.C.A./C.R.E.A. no 20351/90 and no 20357/91 (Ministry of Defense, France)
and by research contract EERP-FR 22, DIGITAL Eq. Corp.

0. Introduction.

Polynomials are basic objects in mathematics and applications. In physics, chemistry, economics. . . , the behavior of a complex system is usually represented as a function of several variables, and this function, at least in some range and within some accuracy, can be approximated by a polynomial with the same number of variables.

The present development of parallel computing turns it into a very efficient tool, but quite rigid, especially when we deal with *Single Instruction Multiple Data* machines, meaning that all processors are executing the same instruction at the same time. Then, so far, only a limited number of problems, of situations, have been handled by SIMD parallel computing. The most frequent ones are : matrix computation and partial differential equations (by discretization).

Returning to polynomials, the difficulty comes from the fact that there is no obvious order or hierarchy between the terms containing various variables, hence one does not see at first glance how to represent the whole polynomial on the computer in order to make use of the parallel structure, and one might think that this is the type of object for which sequential processing is unavoidable.

The first aim of the present paper is precisely to establish a canonical representation of the polynomial on a geometric structure, namely an hypercube, which will allow us to make efficient use of SIMD parallelism and perform efficiently basic operations : sums, products, partial derivatives.

Associated to this structure there is a very simple norm : the l_2 -norm, sum of squares of coefficients after distribution on the hypercube. It turns out, as an independent surprise, that this norm itself is a very efficient mathematical tool in order to study products of polynomials : when products of polynomials are performed under this norm, the constant involved in inequalities of the form $[PQ] \geq C [P][Q]$ is the best possible, and we will determine all the extremal pairs, that is the pairs of polynomials for which the product is as small as possible.

This norm, as well as the mathematical treatment of the product result, borrow heavily from the paper [1] by Beauzamy-Bombieri-Enflo-Montgomery, but the key notion of hypercube, central in the present work, was not present there.

The norm has other applications : as was shown by Beauzamy [2], it gives a sharp estimate for coefficients in polynomial decompositions. Recent work by X. Gourdon [3] shows that it also has applications to the problem of locating the zeros of one-variable polynomials.

The paper is organized as follows : in Section 1, we introduce the representation and show how the operations are performed on the computer. In Section 2, we give the mathematical background, the norm and its properties. All proofs are presented in great detail and illustrated by numerous examples. In Section 3 we present the implementation on the *Connection Machine* and an Appendix is devoted to timing comparisons.

We have tried to give as many details as possible, both on the mathematical tools and on the computer implementation : we hope, this way, that the paper will be of interest both to computer scientists and to mathematicians. To finish this introduction, we wish to thank the referee for his numerous advices and comments.

1. The representation of a multi-variate polynomial on an hypercube.

1.1. Definitions and notations.

If a polynomial in several variables is not homogeneous, we can associate to it a homogeneous polynomial, just adding one more variable, that is by the formula

$$P(x_1, \dots, x_N, x_{N+1}) = x_{N+1}^m P\left(\frac{x_1}{x_{N+1}}, \dots, \frac{x_N}{x_{N+1}}\right),$$

where m is the degree of P . For instance the polynomial in one variable $P(z) = 1 + 2z + 3z^2$ becomes $P(z, z') = z'^2 + 2zz' + 3z^2$. Since all our constructions and results will be independent of the number of variables, we assume in the sequel that this identification has been made, that is we restrict ourselves to homogeneous polynomials. To come back to the original polynomial, just take the last variable equal to 1.

A polynomial P , in N variables (x_1, \dots, x_N) , homogeneous with degree m , is usually written as

$$P(x_1, \dots, x_N) = \sum_{|\alpha|=m} a_\alpha x_1^{\alpha_1} \dots x_N^{\alpha_N}, \quad (1)$$

where $\alpha = (\alpha_1, \dots, \alpha_N)$, $|\alpha| = \alpha_1 + \dots + \alpha_N$. The form given by formula (1) will be called *reduced form* of the polynomial.

We now give another expression of the polynomial, obtained by replacing each power of any variable by its repetition, and considering all the permutations of the variables. Such a representation is given by Taylor's formula :

$$P(x_1, \dots, x_N) = \frac{1}{m!} \sum_{i_1, \dots, i_m=1}^N \frac{\partial^m P}{\partial x_{i_1} \dots \partial x_{i_m}} x_{i_1} \dots x_{i_m}. \quad (2)$$

We write simply $c_{i_1, \dots, i_m} = \frac{1}{m!} \frac{\partial^m P}{\partial x_{i_1} \dots \partial x_{i_m}}$.

This second form will be called *symmetric form* of the polynomial.

As an example, the polynomial $P = x_1^3 - 3x_1x_2^2$ has the symmetric form

$$P = x_1x_1x_1 - (x_1x_1x_2 + x_1x_2x_1 + x_2x_1x_1).$$

Note that c_{i_1, \dots, i_m} is invariant under any permutation of the indices. We will now show how to pass from one representation to the other.

A term

$$\frac{\partial^m (x_1^{\alpha_1} \dots x_N^{\alpha_N})}{\partial x_{i_1} \dots \partial x_{i_m}}$$

is zero unless among the indices i_1, \dots, i_m , α_1 of them are equal to 1, α_2 of them are equal to 2, \dots , α_N of them are equal to N . If this is the case, then

$$\frac{\partial^m (x_1^{\alpha_1} \dots x_N^{\alpha_N})}{\partial x_{i_1} \dots \partial x_{i_m}} = \alpha!,$$

where $\alpha!$ means $\alpha_1! \alpha_2! \dots \alpha_N!$.

So, from formula (1), we get, for every choice of i_1, \dots, i_m :

$$c_{i_1, \dots, i_m} = \frac{\alpha!}{m!} a_\alpha, \quad (3)$$

where α_1 is the number of i_j 's equal to 1, \dots , α_N is the number of i_j 's equal to N . For any m -tuple of indices i_1, \dots, i_m , we just denote by $\alpha(i_1, \dots, i_m)$ the multi-index $(\alpha_1, \dots, \alpha_N)$ obtained in this way (it is an N -index, satisfying $|\alpha| = m$).

We now construct the hypercube. For this, we consider the unit interval $[0, 1]$, and its cartesian product $[0, 1]^m$, where m as before is the degree of P ($m \geq 1$).

Let $N \geq 1$ (the number of variables in P). We divide $[0, 1]$ into N equal intervals, by considering the points

$$0, \frac{1}{N}, \frac{2}{N}, \dots, \frac{N}{N}.$$

In the hypercube $[0, 1]^m$, we consider the points

$$M_{i_1, \dots, i_m} = \left(\frac{i_1}{N}, \dots, \frac{i_m}{N} \right),$$

where i_1, \dots, i_m take any value in $\{1, \dots, N\}$. So there are N^m such points. We observe that none of the coordinates of any such point may be zero.

We now fill the hypercube the following way : at each of the points $M(i_1, \dots, i_m)$, having coordinates $(i_1/N, \dots, i_m/N)$, we put the coefficient c_{i_1, \dots, i_m} obtained in formula (3).

Another way of saying the same thing is the following : the term $a_\alpha x_1^{\alpha_1} \dots x_N^{\alpha_N}$ is mapped as $\frac{\alpha!}{m!} a_\alpha$ at all points $(\frac{i_1}{N}, \dots, \frac{i_m}{N})$ such that $\{i_1, \dots, i_m\}$ is a permutation of

$$\underbrace{1, \dots, 1}_{\alpha_1 \text{ times}}, \underbrace{2, \dots, 2}_{\alpha_2 \text{ times}}, \dots, \underbrace{N, \dots, N}_{\alpha_N \text{ times}}.$$

The set of coefficients c_{i_1, \dots, i_m} mapped on the points M_{i_1, \dots, i_m} is called the *hypercube associated to P* and will be denoted by $H(P)$. We will write

$$H(P) = \{c_{i_1, \dots, i_m}\}_N,$$

or more simply $\{c_I\}_{N, m}$, with $I = \{i_1, \dots, i_m\}$. The number m will be called the *dimension* of the hypercube (indeed, it is the euclidean dimension of the space in which we are), and the number N will be the *edge-size* of the hypercube.

Remark. – There is no standard definition of the word “hypercube”, but sometimes people call this way a product $\{0, 1\}^m$, that is with just two points on each edge. For us, here, it should be clear that we have N points on each edge.

Let’s describe several examples.

- The polynomial $P_1 = 4x_1x_2 - x_3^2$ has degree 2 and 3 variables. So it will be represented as a cube of dimension 2, that is in a plane, by the matrix

$$H(P) = \begin{pmatrix} 0 & 2 & 0 \\ 2 & 0 & 0 \\ 0 & 0 & -1 \end{pmatrix}.$$

- The polynomial $P_2 = -3x_2 + 2x_3$ has degree 1 and 3 variables, and will be represented on the segment $[0, 1]$ by the three points $(0, -3, 2)$, disposed respectively at the coordinates $1/3, 2/3, 3/3$.

In a polynomial of degree 6, with 17 variables, the term $8x_1^2x_2x_3^3$ will be mapped as $8 \cdot \frac{2!3!}{6!}$ at the point $(\frac{1}{17}, \frac{1}{17}, \frac{2}{17}, \frac{17}{17}, \frac{17}{17}, \frac{17}{17})$ and those which are deduced from this one by permutations.

This representation as an hypercube is unique and well-defined because we require the hypercube to be *symmetric* (all coefficients are invariant under permutation of the indexes). But we will meet later other types of representations : the *reduced* one, and more general ones, being neither symmetric nor reduced (see fig. 2 and 3 at the end of the paper). These representations will be used here only as intermediate tools, and a general study of all possible representations of a polynomial on a given hypercube (together with their specific properties) would be certainly be worth investigating.

1.2. Basic operations on the hypercube.

Elementary operations on polynomials can be transferred to the corresponding hypercubes, with obvious definitions. This is the case for the sum (for two polynomials of the same degree, since the result has to be homogeneous), for multiplication by a scalar and for conjugation.

We now turn to operations which do not make immediate sense for polynomials but will be used for our computer implementation of the product operation.

– Parallel Product.

This is the product, term-wise, of two hypercubes of same dimensions and edge-sizes.

$$\{c_I\}_{N,m} \otimes \{d_I\}_{N,m} = \{c_I \times d_I\}_{N,m} .$$

– Cartesian Product.

If C is an hypercube of dimension m and edge-size N , and D has dimension n , edge-size N , their cartesian product, denoted by

$$\{c_I\}_{N,m} \times \{d_J\}_{N,n}$$

is the hypercube of dimension $m + n$, edge-size N , defined by

$$e_{k_1, \dots, k_{m+n}} = c_{k_1, \dots, k_m} \times d_{k_{m+1}, \dots, k_{m+n}} .$$

We observe that this product is not commutative.

– Global Summation.

$$\text{Sum} (\{c_{i_1, \dots, i_m}\}_N) = \sum_{i_1, \dots, i_m=1}^N c_{i_1, \dots, i_m} .$$

For the polynomial, this operation just returns the sum of coefficients.

– Selection.

Given k , $1 \leq k \leq N$, this operation extracts from the cube $\{c_{i_1, \dots, i_m}\}_N$ the hypercube of coefficients defined by $i_1 = k$ (one might choose any of the i_j 's, since they are permutation-invariant). This hypercube is therefore of dimension $m - 1$ and edge-size N . Formally, its definition is :

$$\text{Sel}_k (\{c_{i_1, \dots, i_m}\}_N) = \{c'_{j_1, \dots, j_{m-1}}\}_N ,$$

where

$$c'_{j_1, \dots, j_{m-1}} = c_{k, i_2, \dots, i_m} .$$

The corresponding notion for the polynomial is a choice of partial derivatives. Recall that the hypercube is built as

$$c_{i_1, \dots, i_m} = \frac{1}{m!} \frac{\partial^m P}{\partial x_{i_1} \cdots \partial x_{i_m}} .$$

Thus selecting $i_1 = 1$ means that one (at least) of the partial derivatives will be taken with respect to x_1 . This means that

$$\text{Sel}_1 (\{c_{i_1, \dots, i_m}\}_N)$$

is the hypercube

$$\frac{1}{m!} \left\{ \frac{\partial^{m-1}}{\partial x_{i_2} \cdots \partial x_{i_m}} \left(\frac{\partial P}{\partial x_1} \right) \right\}_N .$$

This selection operation can of course be iterated : Sel_{k_1, k_2} corresponds to the second derivatives, and so on. (*See fig. 1*).

– **Extensions.**

We have seen that the cartesian product of cubes of dimensions m , n respectively was a cube of dimension $m + n$. For technical reasons, the parallel computer has to work inside a given geometry, and so, if we want to perform products, we have to deal with cubes of dimension $m + n$. This is obtained by repeating each of the original cubes, identical to itself, so as to make it $m + n$ -dimensional. This operation is called *extension*.

Let $\{1\}_{N,n}$ denote the hypercube of dimension n , edge-size N , whose coefficients are all equal to 1. The *left-extension* of $\{c_I\}_{N,m}$ is the cartesian product

$$\text{Lext}_n(\{c_I\}_{N,m}) = \{1\}_{N,n} \times \{c_I\}_{N,m} ,$$

and the *right-extension* of $\{c_I\}_{N,m}$ the cartesian product

$$\text{Rext}_n(\{c_I\}_{N,m}) = \{c_I\}_{N,m} \times \{1\}_{N,n} .$$

Both have dimension $m + n$. Their names illustrate the fact that the source hypercube is extended (or repeated) along the first n axes (for the left extension), or along the last n axes (for the right extension).

We may now work inside a geometry of dimension $m + n$ and write the cartesian product as :

$$\{c_I\}_{N,m} \times \{d_J\}_{N,n} = \text{Rext}_n(\{c_I\}_{N,m}) \otimes \text{Lext}_m(\{d_J\}_{N,n}) .$$

– **Symetrization, Reduction.**

When we perform direct products as above, the result is obviously not a symmetric hypercube. In order to *symmetrize* it (and get the hypercube corresponding to the product PQ), we have first to perform a *reduction* :

a) **Reducing the hypercube.**

We take all coefficients differing only by a permutation of their coordinates, and map their sum at a single point, namely the one with increasing coordinates (this choice is arbitrary). All other points receive coefficients 0.

More precisely, for i_1, \dots, i_m given, we denote by α_1 the number of i_j 's equal to 1, α_2 the number of i_j 's equal to 2, and so on, and finally α_N is the number of i_j 's equal to N . So $\alpha_1 + \dots + \alpha_N = m$, and $\{i_1, \dots, i_m\}$ is a permutation of

$$I_\alpha = \overbrace{1, \dots, 1}^{\alpha_1 \text{ times}}, \overbrace{2, \dots, 2}^{\alpha_2 \text{ times}}, \dots, \overbrace{N, \dots, N}^{\alpha_N \text{ times}} .$$

Let S_α be the set of all distinct permutations of I_α . The point of coordinates I_α receives $a_\alpha = \sum_{S_\alpha} c_I$. The other points, of coordinates c_I , $I \in S_\alpha$ but $I \neq I_\alpha$ receive 0.

b) **Symmetrization.**

When the reduction operation has been performed, we build a symmetric hypercube by the same operation we introduced to build the hypercube from P , in Section 1.1. Namely, we build the hypercube $\{s_I\}_{N,m}$ by

$$s_I = \frac{\alpha!}{m!} a_\alpha , \text{ if } I \in S_\alpha .$$

This means that the sole coefficient a_α is now mapped as $\frac{\alpha!}{m!} a_\alpha$ on all points which differ from it by a permutation of its coefficients. There are $\frac{m!}{\alpha!}$ such points, so the sum is preserved.

The combination of both operations : reduction, symmetrization, produces a symmetric cube, and preserves the sum of coefficients inside a given permutation of the indexes :

$$\sum_{I \in S_\alpha} c_I = \sum_{I \in S_\alpha} s_I ,$$

for any $\alpha = (\alpha_1, \dots, \alpha_N)$, with $|\alpha| = m$. (See fig. 2).

The procedure used by the computer in order to perform products can now be completely described. First, P and Q are mapped as $H(P)$ and $H(Q)$, hypercubes of dimensions m and n respectively. Then both of them are mapped inside an $(m+n)$ -geometry, as $\text{Rext}_n(\{c_I\}_{N,m})$ and $\text{Lext}_m(\{d_J\}_{N,n})$. Then the parallel product is performed :

$$\text{Rext}_n(\{c_I\}_{N,m}) \otimes \text{Lext}_m(\{d_J\}_{N,n}) ,$$

and this gives a (non-symmetric) hypercube $H'(PQ)$. This hypercube is reduced and becomes $H_r(PQ)$, which in turn is symmetrized in order to give the true hypercube associated to the usual product PQ (see fig. 3).

As a simple complexity analysis, we observe that, in order to build the hypercube, a coefficient must be sent to at most $m!$ processors. But when the representation is built (that is, when the polynomial appears in symmetric form), all operations are independent of the number of variables (all being treated at the same time) and depend exponentially on the degree. We will come back on this at the end of the paper. We now turn to the mathematical properties of this representation.

2. Mathematical properties of the representation.

In [1], Bombieri introduced a function associated to the polynomial P , by the formula :

$$F(t_1, \dots, t_m) = \frac{1}{m!} \frac{\partial^m P}{\partial x_{i_1} \cdots \partial x_{i_m}} , \quad (4)$$

where $i_1 = [Nt_1] + 1, \dots, i_m = [Nt_m] + 1$.

We see therefore that

$$F(t_1, \dots, t_m) = c_{i_1, \dots, i_m} \quad \text{when} \quad \frac{i_1 - 1}{N} \leq t_1 < \frac{i_1}{N}, \dots, \frac{i_m - 1}{N} \leq t_m < \frac{i_m}{N} ,$$

that is inside the cube which ‘‘terminates’’ at the point M_{i_1, \dots, i_m} . So we see that mapping the polynomial on the hypercube, as we did, or constructing the function, are the same thing : the function extends inside each small cube the value given at its ‘‘extreme’’ point. The theory which follows uses implicitly this representation : the notions have been transferred to the polynomial, so as to avoid changes in notation.

2.1. The norms.

Usually, one considers the norms associated to the coefficients of P . They are denoted by $|P|_1, |P|_2$, and are defined respectively by the formulas

$$|P|_1 = \sum_{|\alpha|=m} |a_\alpha| , \quad |P|_2 = \left(\sum_{|\alpha|=m} |a_\alpha|^2 \right)^{1/2} ,$$

Here, for us, the representation on the hypercube will be the canonical one, so we introduce instead the *Taylor norms* $[P]_1, [P]_2$, defined by

$$[P]_1 = \sum_{i_1, \dots, i_m=1}^N |c_{i_1, \dots, i_m}| , \quad [P]_2 = \left(\sum_{i_1, \dots, i_m=1}^N |c_{i_1, \dots, i_m}|^2 \right)^{1/2} .$$

The connections with the usual norms are easy to establish : each coefficient c_{i_1, \dots, i_m} appears in the cube a number of times equal to $\frac{m!}{\alpha!}$; therefore :

$$\sum_{i_1, \dots, i_m=1}^N |c_{i_1, \dots, i_m}| = \sum_{|\alpha|=m} |a_\alpha| ,$$

that is $|P|_1 = [P]_1$, and

$$[P]_2 = \left(\sum_{\alpha} \frac{\alpha!}{m!} |a_\alpha|^2 \right)^{1/2} .$$

So the Taylor norm $[\cdot]_2$ appears as an l_2 norm with weights. This norm, introduced by Bombieri in [1], was used by the first named author in order to produce sharp a priori estimates on the size of coefficients in polynomial decompositions (see B. Beauzamy [2]).

2.2. Products of polynomials on the hypercube.

Let P, Q be polynomials in N variables x_1, \dots, x_N , homogeneous of degrees m and n respectively. We write them in symmetric form (2) :

$$P(x_1, \dots, x_N) = \sum_{i_1, \dots, i_m=1}^N c_{i_1, \dots, i_m} x_{i_1} \cdots x_{i_m} ;$$

$$Q(x_1, \dots, x_N) = \sum_{j_1, \dots, j_n=1}^N d_{j_1, \dots, j_n} x_{j_1} \cdots x_{j_n} .$$

We are going to prove :

Theorem 1. – For all polynomials P, Q homogeneous of degrees m and n respectively,

$$[PQ]_2 \geq \sqrt{\frac{m!n!}{(m+n)!}} [P]_2 [Q]_2 . \quad (5)$$

Proof. – We have :

$$[P]_2 = \left(\sum_I |c_I|^2 \right)^{1/2}, \quad [Q]_2 = \left(\sum_J |d_J|^2 \right)^{1/2},$$

where I stands for (i_1, \dots, i_m) , J for (j_1, \dots, j_n) .

For the product PQ , we write :

$$PQ = \sum_{i_1, \dots, i_m} \sum_{j_1, \dots, j_n} c_{i_1, \dots, i_m} d_{j_1, \dots, j_n} x_{i_1} \cdots x_{i_m} x_{j_1} \cdots x_{j_n}$$

$$= \sum_{l_1, \dots, l_{m+n}} \frac{1}{(m+n)!} \left(\sum_{i_1, \dots, i_m, j_1, \dots, j_n} c_{i_1, \dots, i_m} d_{j_1, \dots, j_n} \right) x_{l_1} \cdots x_{l_{m+n}} ,$$

where in the second sum $\{i_1, \dots, i_m, j_1, \dots, j_n\}$ is a permutation of the $m+n$ -tuple $\{l_1, \dots, l_{m+n}\}$.

Instead of writing that i_1 is one of the l 's, we prefer to assign to it a rank among the l 's, that is to write $i_1 = l_{u_1}$, where $u_1 \in \{1, \dots, m+n\}$. We do the same with i_2, \dots, i_m , and introduce u_2, \dots, u_m . Then for j_1, \dots, j_n , we introduce v_1, \dots, v_n , which also belong to $\{1, \dots, m+n\}$.

This way, the previous formula becomes :

$$PQ = \sum_{l_1, \dots, l_{m+n}=1}^N \frac{1}{(m+n)!} \left(\sum_{u_1, \dots, u_m, v_1, \dots, v_n} c_{l_{u_1}, \dots, l_{u_m}} d_{l_{v_1}, \dots, l_{v_n}} \right) x_{l_1} \cdots x_{l_{m+n}} ,$$

where in the second sum the indexes $u_1, \dots, u_m, v_1, \dots, v_n$ run over all permutations of $\{1, \dots, m+n\}$.

Let $U = (u_1, \dots, u_m)$, $V = (v_1, \dots, v_n)$. We say that (U, V) form a *shuffle of type* (m, n) if

$$u_1 < \cdots < u_m, \quad v_1 < \cdots < v_n,$$

and the set $\{u_1, \dots, u_m, v_1, \dots, v_n\}$ is a permutation of $\{1, \dots, m+n\}$. We write $sh(m, n)$ for the set of shuffles of type (m, n) . Its cardinality is

$$|sh(m, n)| = \frac{(m+n)!}{m!n!} .$$

Since $c_{l_{u_1}, \dots, l_{u_m}}$ is invariant under permutation of the coordinates, we find :

$$PQ = \sum_{l_1, \dots, l_{m+n}=1}^N \frac{m!n!}{(m+n)!} \left(\sum_{(U, V) \in sh(m, n)} c_{l_U} d_{l_V} \right) x_{l_1} \cdots x_{l_{m+n}} ,$$

where

$$l_U = l_{u_1} \cdots l_{u_m}, \quad l_V = l_{v_1} \cdots l_{v_n}.$$

So we obtain :

$$[PQ]_2^2 = \left(\frac{m!n!}{(m+n)!} \right)^2 \sum_{l_1, \dots, l_{m+n}=1}^N \left| \sum_{(U,V) \in sh(m,n)} c_{l_U} d_{l_V} \right|^2.$$

In order to obtain the estimate for the product, we write, for each l_1, \dots, l_{m+n} :

$$\begin{aligned} A(l_1, \dots, l_{m+n}) &= \left| \sum_{(U,V) \in sh(m,n)} c_{l_U} d_{l_V} \right|^2 \\ &= \left(\sum_{(U,V) \in sh(m,n)} c_{l_U} d_{l_V} \right) \left(\sum_{(U',V') \in sh(m,n)} \bar{c}_{l_{U'}} \bar{d}_{l_{V'}} \right) \\ &= \sum_{(U,V), (U',V') \in sh(m,n)} c_{l_U} d_{l_V} \bar{c}_{l_{U'}} \bar{d}_{l_{V'}}. \end{aligned}$$

In this sum, we distinguish between two kinds of terms : those for which $U = U'$ (and consequently $V = V'$), and those for which $U \neq U'$ (and consequently $V \neq V'$).

We call $E(l_1, \dots, l_{m+n})$ the sum of terms of the first kind, $D(l_1, \dots, l_{m+n})$ the sum of terms of the second kind (E stands for equal, D for different). This way, we get

$$A(l_1, \dots, l_{m+n}) = E(l_1, \dots, l_{m+n}) + D(l_1, \dots, l_{m+n}),$$

with

$$E(l_1, \dots, l_{m+n}) = \sum_{(U,V) \in sh(m,n)} |c_{l_U}|^2 |d_{l_V}|^2.$$

Then

$$\begin{aligned} \sum_{l_1, \dots, l_{m+n}} E(l_1, \dots, l_{m+n}) &= \sum_{l_1, \dots, l_{m+n}} \sum_{(U,V) \in sh(m,n)} |c_{l_U}|^2 |d_{l_V}|^2 \\ &= \frac{(m+n)!}{m!n!} \left(\sum_{i_1, \dots, i_m=1}^N |c_{i_1, \dots, i_m}|^2 \right) \left(\sum_{j_1, \dots, j_n=1}^N |d_{j_1, \dots, j_n}|^2 \right) \\ &= \frac{(m+n)!}{m!n!} [P]_2^2 [Q]_2^2. \end{aligned}$$

For each term in $D(l_1, \dots, l_{m+n})$, that is for each $(U, V), (U', V')$ with $U \neq U'$, we introduce the set of indexes which are in U and U' , in U and V' , V and U' , V and V' , that is, we consider the four sets $U \cap U'$, $U \cap V'$, $V \cap U'$, $V \cap V'$ and $c_{l_U} d_{l_V} \bar{c}_{l_{U'}} \bar{d}_{l_{V'}}$ can be written as :

$$c_{l_{U \cap U'}, l_{U \cap V'}} d_{l_{V \cap U'}, l_{V \cap V'}} \bar{c}_{l_{U' \cap U}, l_{U' \cap V}} \bar{d}_{l_{V' \cap U}, l_{V' \cap V}}$$

which is the same as :

$$(c_{l_{U \cap U'}, l_{U \cap V'}} \bar{d}_{l_{U \cap V'}, l_{V \cap V'}}) (\bar{c}_{l_{U \cap U'}, l_{V \cap U'}} d_{l_{V \cap U'}, l_{V \cap V'}}).$$

We consider

$$\sum_{l_{U \cap V'}} \sum_{l_{V \cap U'}} (c_{l_{U \cap U'}, l_{U \cap V'}} \bar{d}_{l_{U \cap V'}, l_{V \cap V'}}) (\bar{c}_{l_{U \cap U'}, l_{V \cap U'}} d_{l_{V \cap U'}, l_{V \cap V'}}).$$

We observe that $U \cap V'$ and $V \cap U'$ have the same cardinality, and this double sum is therefore equal to

$$\left| \sum_{l_W} c_{l_{U \cap U'}, l_W} \bar{d}_{l_W, l_{V \cap V'}} \right|^2$$

where $W = U \cap V'$ (or $V \cap U'$). Summing upon $l_{U \cap U'}$ and $l_{V \cap V'}$, we get

$$\sum_{l_1, \dots, l_{m+n}} D(l_1, \dots, l_{m+n}) = \sum_{(U, V) \neq (U', V')} \sum_{l_{U \cap U'}} \sum_{l_{V \cap V'}} \left| \sum_{l_W} c_{l_{U \cap U'}, l_W} \bar{d}_{l_W, l_{V \cap V'}} \right|^2$$

and finally, the representation formula :

$$\begin{aligned} [PQ]_2^2 &= \frac{m!n!}{(m+n)!} [P]_2^2 [Q]_2^2 + \\ &\left(\frac{m!n!}{(m+n)!} \right)^2 \sum_{(U, V) \neq (U', V')} \sum_{l_{U \cap U'}} \sum_{l_{V \cap V'}} \left| \sum_{l_W} c_{l_{U \cap U'}, l_W} \bar{d}_{l_W, l_{V \cap V'}} \right|^2. \end{aligned} \quad (6)$$

Since the terms of the second sum are positive, the announced estimate follows. It is also clear that the extremal pairs will be those for which

$$\sum_{l_W} c_{l_{U \cap U'}, l_W} \bar{d}_{l_W, l_{V \cap V'}} = 0, \quad (7)$$

for all $(U, V), (U', V')$ with $U \neq U'$. We will come back on this in § 2.3.

We also observe that Taylor's norm has the interesting property of being submultiplicative (see B. Beauzamy [2]) :

Theorem 2. - For all polynomials P, Q ,

$$[PQ]_2 \leq [P]_2 [Q]_2.$$

Proof. - Let $P = \sum_{|\alpha|=m} a_\alpha x_1^{\alpha_1} \dots x_N^{\alpha_N}$, $Q = \sum_{|\beta|=n} b_\beta x_1^{\beta_1} \dots x_N^{\beta_N}$

$$R = PQ = \sum_{|\gamma|=m+n} \left(\sum_{\alpha+\beta=\gamma} a_\alpha b_\beta \right) x_1^{\gamma_1} \dots x_N^{\gamma_N}$$

We have to show that

$$\sum_{|\gamma|=m+n} \left| \sum_{\alpha+\beta=\gamma} a_\alpha b_\beta \right|^2 \frac{\gamma!}{(m+n)!} \leq \left(\sum_{|\alpha|=m} |a_\alpha|^2 \frac{\alpha!}{m!} \right) \left(\sum_{|\beta|=n} |b_\beta|^2 \frac{\beta!}{n!} \right).$$

But since $|\alpha| = m$, $|\beta| = n$, we have, for every γ , by Cauchy-Schwarz inequality :

$$\sum_{\alpha+\beta=\gamma} |a_\alpha b_\beta| \leq \left(\sum_{\alpha+\beta=\gamma} \frac{|a_\alpha b_\beta|^2}{\frac{m!}{\alpha!} \frac{n!}{\beta!}} \right)^{1/2} \times \left(\sum_{\alpha+\beta=\gamma} \frac{m!}{\alpha!} \frac{n!}{\beta!} \right)^{1/2}.$$

Using the identity :

$$\sum_{\alpha+\beta=\gamma} \frac{m!}{\alpha!} \frac{n!}{\beta!} = \frac{(m+n)!}{\gamma!},$$

we deduce

$$\left| \sum_{\alpha+\beta=\gamma} a_\alpha b_\beta \right|^2 \frac{\gamma!}{(m+n)!} \leq \sum_{\alpha+\beta=\gamma} |a_\alpha|^2 |b_\beta|^2 \frac{\alpha! \beta!}{m! n!},$$

and since this holds for every γ , the result follows.

Remark. - The above proof shows that the only polynomials P, Q for which $[PQ]_2 = [P]_2 [Q]_2$ are those for which a_α is proportional to $m!/\alpha!$, b_β proportional to $n!/\beta!$. These polynomials are $P = \lambda(t_1 x_1 + \dots + t_N x_N)^m$, $Q = \mu(t_1 x_1 + \dots + t_N x_N)^n$, for $\lambda, \mu, t_i \in \mathbb{C}$.

2.3. The extremal pairs.

As explained in [1], the estimate given in Theorem 1 is best possible, among all estimates *independent of the number of variables*. This is obtained precisely by letting the number of variables tend to infinity. In the case of two variables, it was shown by Beauzamy [2] that (for $m = n$), the estimate (5) is also best possible, an example of extremal polynomials being $(x_1 + x_2)^m$, $(x_1 - x_2)^m$. We now investigate this question for any number of variables and give the extremal pairs.

Theorem 3. – *The extremal pairs P , Q , that is those for which estimate (5) is an equality, are obtained the following way : if $H(P)$ and $H(Q)$ are the associated hypercubes, of dimensions m and n respectively, the pair (P, Q) is extremal if and only if the column-vectors of $H(P)$ and those of $H(Q)$ are pairwise orthogonal.*

As an example, the two polynomials

$$\begin{aligned} P &= (x_1 + \cdots + x_N)^m \\ Q &= \left(e^{2i\pi \frac{1}{N}} x_1 + \cdots + e^{2i\pi \frac{N}{N}} x_N \right)^n, \end{aligned}$$

form an extremal pair, for any m , n , $N \geq 1$.

Before turning to the proof, let us explain the meaning of this statement.

By a ‘‘column’’, we mean the vector

$$C_{i_1, \dots, i_{m-1}} = \begin{pmatrix} c_{i_1, \dots, i_{m-1}, 1} \\ \vdots \\ c_{i_1, \dots, i_{m-1}, N} \end{pmatrix}.$$

There are of course N^{m-1} such columns for $H(P)$ and N^{n-1} such columns for $H(Q)$. The required orthogonality reads :

$$\sum_{k=1}^N c_{i_1, \dots, i_{m-1}, k} \bar{d}_{j_1, \dots, j_{n-1}, k} = 0, \quad (8)$$

for all i_1, \dots, i_{m-1} , j_1, \dots, j_{n-1} .

Proof of Theorem 3. – We consider the shuffles

$$U = \{1, 2, \dots, m\}, \quad V = \{m+1, \dots, m+n\},$$

$$U' = \{1, 2, \dots, m-1, m+1\}, \quad V' = \{m, m+2, \dots, m+n\},$$

and, using (7), we find

$$\sum_{k=1}^N c_{i_1, \dots, i_{m-1}, k} \bar{d}_{j_1, \dots, j_{n-1}, k} = 0,$$

which is the same as (8), and proves the result.

We observe that, due to the invariance under permutations of coordinates, the number of orthogonality relations to be satisfied is

$$\binom{N+m-2}{m-1} \binom{N+n-2}{n-1}$$

The above example of

$$\begin{aligned} P &= (x_1 + \cdots + x_N)^m \\ Q &= \left(e^{2i\pi \frac{1}{N}} x_1 + \cdots + e^{2i\pi \frac{N}{N}} x_N \right)^n, \end{aligned}$$

is obtained by means of the following two lemmas :

Lemma 4. – The polynomial P , in N variables, with coefficients in symmetric form

$$c_{i_1, \dots, i_m} = 1$$

for all i_1, \dots, i_m , is $P(x_1, \dots, x_N) = (x_1 + \dots + x_N)^m$.

Proof of Lemma 4. – This is clear, since for every i_1, \dots, i_m ,

$$\frac{\partial^m P}{\partial x_{i_1} \dots \partial x_{i_m}} = m!$$

Lemma 5. – The polynomial Q , in N variables, with symmetric coefficients

$$d_{j_1, \dots, j_n} = e^{2i\pi(j_1 + \dots + j_n)/N},$$

is

$$Q = \left(e^{2i\pi \frac{1}{N}} x_1 + \dots + e^{2i\pi \frac{N}{N}} x_N \right)^n.$$

Proof of Lemma 5. – This follows immediately from the multinomial identity.

Since quite clearly the column-vectors of the cubes are orthogonal, the result follows.

In the case of two variables ($N = 2$), we can easily describe the extremal pairs for any m, n :

Theorem 6. – In the case $N = 2$, the extremal pairs are :

$$\begin{aligned} P &= (ax_1 + bx_2)^m \\ Q &= (\bar{b}x_1 - \bar{a}x_2)^n, \end{aligned}$$

where $a, b \in \mathbb{C}$.

A proof of this theorem can be given by explicit computation of the hypercubes. However, it can be deduced from a more general result : if (P, Q) form an extremal pair, so do (P^a, Q^b) , for any couple of integers a and b . This will be the topic of forthcoming paper, devoted to the study of extremal pairs, by the last two authors. It will also be shown that if (P, Q) form an extremal pair, so do (\tilde{P}, \tilde{Q}) , obtained after unitary change of variables. For a given P , the set of Q 's of degree n which form with P an extremal pair is a vector space, which will be described.

3. Implementation on a parallel computer.

3.1. The Connection Machine CM-2.

3.1.1. Overview.

The Connection Machine CM-2 is a massively parallel computer, composed of 4096 to 65536 processors, each with 64k to 1M bits of memory. The processors operate in parallel, each one performing at any given time the same operation as all the others. They can process the data stored in their memory or exchange data with other processors, along regular or arbitrary communication patterns.

User programs run on a front-end computer, producing macroinstructions : they are fed to a sequencer in the Connection Machine that in turn broadcasts instructions to the physical processor and associated hardware. Both assembly-level (PARIS) and high-level (C*, CM-Fortran and *LISP) programming languages are available to the user. The latter are parallel extensions to C, Fortran and Common LISP respectively.

The encouraged programming model is that of *data parallelism*, in which one processor is assigned to every elementary data element. A transparent virtual processing mechanism makes it possible to present the user with one or several *virtual processors sets* (VP sets) of size(s) adapted to the data set(s) being

considered. Several *VP* sets of various sizes may coexist in a given application, through a partition of the memory of the physical processors (*not* a partition of the set of physical processors). Only one *VP* set can be active at a given time : it is called the *current VP* set.

All virtual processors (*VP*'s) composing a *VP* set are assigned a unique identifier, called their *cube address*, which is used in the expression of some parallel inter-processor communication operations. Such communications can proceed either within a *VP* set or across *VP* sets.

Using the concept of *geometry*, the *VP*'s can also be logically organized into multi-dimensional grids (e.g. a *VP* set of size 8192 can be seen as a 128x64 2D grid, a 32x32x8 3D grid, etc). The mapping is most efficient when the grid dimensions are powers of 2, as in this case adjacent nodes of the grid can be mapped to processors that are adjacent on the physical interconnection network, thus allowing fast communications along the grid axes.

All operations can be performed only within a subset of *active* processors (the other ones remaining idle) in order to implement processorwise conditional operations. The pattern of active and idle processors is called the *context*.

The basic computational entity is the *parallel variable* (*pvar*), which is simply a set of elementary data in a given *VP* set, one per virtual processor. It is possible to access individually any element of a *pvar*, but most operations proceed on *pvar*'s as a whole.

3.1.2. Programming.

Our code has been developed in *LISP. This choice was motivated by the fact that this language inherits some invaluable features of LISP as far as development, prototyping and test are concerned (incremental compilation, built-in interpreter) without sacrificing the performance, at least for our class of applications. Also, a simulator is available which makes it possible to develop and debug most of the code before actually running it on the Connection Machine.

We shall now give some elements of LISP and *LISP for those not familiar with these languages. The execution of a LISP code proceeds basically by recursive evaluation of *forms* expressed in prefixed polish notation and delimited by parentheses. For instance the evaluation of the form `(+ 2 3)` produces the value 5, and `(setf x (+ 2 (* y 4)))` assigns to the variable `x` the sum of 2 with the result of the product of `y` and 4. Special constructions allow sequential, conditional and iterative evaluation of forms, as well as the definition of functions and macros. For instance :

`(progn form1 ... formN)` evaluates in sequence the forms `form1 ... formN` and returns the value produced by the evaluation of the last form.

`(if test then [else])` first evaluates the form `test`. If the result is not `nil` (which serves here as an indicator of "false") the `then` form is selected, otherwise the optional `else` form is selected, defaulting to `nil` (which evaluates to itself) if not present. The selected form is then evaluated and its result is taken as the value returned by `if`.

`(when test form1 ... formN)` first evaluates the `test` form. If the result is `nil` no `form` is evaluated and `nil` is returned. Otherwise `form1 ... formN` constitutes an explicit `progn` ; they are evaluated sequentially from left to right, and the value of the evaluation of the last one is returned.

`(loop clauses)`, actually an extension to Common Lisp, provides a very general iteration facility somewhat easier to read than more basic iteration constructions. We will generally use it as

`(loop for index from start to end do form1...formN)`

which iterates the evaluation of `form1 ... formN` for all values of `index` from the value of `start` to that of `end` inclusively.

(**defun** *name lambda-list [declarations and documentation] forms*) defines named functions. The *lambda-list* describes the parameters. The *declaration* part may be used to specify the types of these parameters (the compiler may use these informations to generate more efficient code – they are not mandatory though). The *forms* constitute the body of the defined function and will be executed as an implicit **progn**. For instance a function to compute the factorial of positive integers could be defined as

```
(defun factorial (n)
  "computes the factorial of a positive integer n"
  (if (= n 0) 1 (* n (fact (- n 1)))))
```

*LISP extends Common LISP by introducing new data types such as *VP* sets and *pvar*'s, as well as operations on such data. A *LISP program operates both on scalar and parallel data : scalar processing is done on the front end computer, and parallel processing on the Connection Machine itself. Hereafter is a presentation of the operations on *pvar*'s that we have used. As a notational convention, *pvar* names end with a “ ! ” as in **a!**, the names of functions returning *pvar*'s end with “ !! ” as in **pref!!** and the names of functions that take *pvar* operand but do not return a *pvar* as their value start with a “ * ”, as in ***pset**.

– Processorwise arithmetical and logical operations.

Most operations on scalars can be extended for *pvar* operands. Let **a!** and **b!** be two *pvar*'s belonging to the same *VP* set. Then **(+!! a! b!)** produces a *pvar* equal in each active *VP* to the sum of the elementary values of **a!** and **b!** held by this *VP* (its value in idle processors is undefined). Also, **(<!! a! b!)** returns a boolean *pvar*, equal to **t** in every active processor in which **a!** is strictly lower than **b!**, and to **nil** in every other active processor. Scalars appearing in parallel expressions are automatically promoted to *pvar*'s, so **(+!! a! 1)** increments the values of **a!** by 1 in all active processors.

– Context.

The pattern of active and idle processors can be altered using parallel extensions of the conditional constructions of Common LISP :

```
(*when test form1 ... formN)
```

will evaluate *form1 ... formN* in the restriction of the current context defined by the result of the evaluation of *test*. Thus **(***when** (<!! a! 0) (***setf** b! 2))** will store the value 2 in **b!** within all currently active processors in which **a!** is negative.

– General communications.

Two functions are available to perform general communications within a *VP* set or across *VP* sets :

(pref!! source address) : The *pvar address* must specify for each active processor in the current *VP* set a valid cube address in the *VP* set of *source*. All active processors retrieve in parallel the value held in *source* by the processors specified by these addresses.

(*pset combine-method source dest address) : The *pvar address* must specify for each active processor in the current *VP* set a valid cube address in the *VP* set of *dest*. All processors send in parallel their value of *source* to the specified destination. If several source processors want to access the same destination processor, the values must be combined. Examples of implemented methods are **:add**, which computes in each destination processor the sum of incoming values, **:max** and **:min** which compute the maximum or minimum of these values, **:no-collisions** which expresses a statement that no two source processors will attempt to access the same destination.

– Global reduction operations.

Global reduction operations compute a scalar value from a *pvar*. For instance **(***sum** a!)** computes the global sum of all elementary values in **a!** held by currently active processors (thus **(***sum** 1)** may be used

to compute the number of currently active processors). Other global reduction operations are for instance `*max` and `*min`.

Most functionalities of LISP and *LISP have of course been omitted from the above description (for complete descriptions of these languages, see [3] and [4]). The elements given here should however suffice in order to understand the code fragments provided in the remainder of this section.

3.2. Overview of the code.

The code manipulates homogenous polynomials of fixed number of variables N and of degree limited only by the available memory. Operations on polynomials are implemented as operations on hypercubes of coefficients. Each hypercube of coefficients is stored as a *pvar*.

The following sections describe the initialization procedures and the implementation of various operations on polynomials, from the reduction and symmetrization of their hypercube representations to linear combinations, products, computations of norms and of partial derivatives.

3.3. Initializations.

The initializations are done only once in a session. We create one *VP* set for each combination of number of variables N and degree m of interest (in practice N is fixed and only m varies). Given N and m , the size of the hypercube of coefficients is N^m . However the size of a *VP* set must be a power of 2 greater than the number of physical processors (8192 or 16384 in our case), so for each m we determine the smallest p such that $2^p \geq N^m$ and create a *VP* set *VPS- m* of size 2^p .

If N is not a power of 2, or if N^m is lower than the number of physical processors, we must limit the operations to N^m processors in the *VP* set. We define a boolean *pvar context- m !* in the *VP* set *VPS- m* which holds the value T (true) in the processors whose cube-address is strictly lower than N^m (the numerotation starts at 0) and nil (i.e. false) otherwise. It will be used as an overall context for operations within *VPS- m* .

Before operating on a *pvar* representing an hypercube of coefficients of dimension m and edge-size N , we may have to switch *vp* sets to make *VPS- m* the current one, and establish the context defined by *context- m !*. This is done by

```
(*with-vp-set-and-context-for-degree degree forms)
(*with-vp-set-and-context pvar forms)
```

The *forms* will be evaluated in the *vp* set and with the context defined above. In the first form the adequate value of m is given explicitly. In the second, it is deduced from the *pvar* argument.

During the initializations, two other *pvar*'s are computed for each N and m , holding for each processor the address of its representant and a weight. Their precise computation will be described later.

3.4. Cube addresses and coordinates on the hypercube.

We assign to the point of coordinates (i_1, \dots, i_m) the cube address c defined by

$$c = \sum_{k=1}^m (i_k - 1)N^{k-1} .$$

The function (`self-address!!`) computes a *pvar* giving for each virtual processor in the current *VP*-set its cube address. We define three translation functions :

(`my-coordinate!! axis`) : produces a *pvar* whose value in each processor is equal to the coordinate along the specified axis computed from the cube address of this processor, using the above formula.

(`*deposit-coordinate coord! axis cube!`) : will be used to build incrementally a cube address `cube!` in the current *vp* set from a sequence of coordinates.

(***deposit-vp-coordinate** *coord axis cube vp-set*) : will be used to build incrementally a cube address in the specified *VP* set, to be used in general communication operations across *VP* sets.

For computation efficiency, the internal coordinates and axis numbers are zero-based.

3.5. Reduction and symmetrization.

For every α defined in section 1, we shall call a *class* the set of virtual processors whose coordinates are defined by the elements of S_α (that is : they differ from one another by a permutation of the coordinates). The processor of coordinates I_α (the one with increasing coordinates) will be called the *representant* of the class (see sections 1.2.a and 1.2.b above).

In the reduction operation the coefficients held by all *VP*'s in a class are set to zero except for the coefficient of the representant which receives the sum of the old coefficients. In the symmetrization operation the coefficient of the representant is equally distributed among all *VP*'s in the class. These redistributions of coefficients are implemented on the Connection Machine with general communication operations.

Assuming that we have a function **representant-cube-address!!** that returns a *pvar* containing in each *VP* of the current *VP* set the cube address of the representant of its class, the reduction operation is implemented in two elementary instructions. First the coefficients in the destination are all set to zero, and then all *VP*'s send the coefficients of the source to their representant ; if collisions occur (several *VP*'s sending to the same representant) the messages are summed.

```
(defun reduce-poly!! (source-poly!)
  "returns the reduced representation of source-poly!"
  (*with-vp-set-and-context source-poly!
    (*let (dest-poly!)          ;allocates a temporary pvar
      (*setf dest-poly! 0)
      (*pset :add source-poly! dest-poly! (representant-cube-address!!))
      dest-poly!)              ;return dest-poly!
    )
  )
```

Assuming that we have a function **weights!!** that returns a *pvar* containing in each *VP* of the current *VP* set the reciprocal of the cardinal of its class, the symmetrization of a reduced hypercube representation is implemented in two elementary instructions. First the coefficients of the symmetrized representation are initialized to the quotient of the coefficients of the reduced representation by the cardinal of their class (i.e. multiplied by the weight) ; this operation affects only the representant *VP*'s since the coefficients in the other ones are zero. Then each *VP* fetches the resulting coefficient held by the representant of its class.

```
(defun symmetrize-reduced-poly (source-poly!)
  "returns a symmetric representation of source-poly!
  - which should be in reduced form"
  (*with-vp-set-and-context source-poly!
    (*let (dest-poly!) ;allocates a temporary pvar
      (*setf dest-poly! (*!! source-poly! (weights!!)))
      (*setf dest-poly! (pref!! dest-poly! (representant-cube-address!!)))
      dest-poly!) ;return dest-poly!
    )
  )
```

A general symmetrization function can be written as a combination of the two functions above :

```
(defun symmetrize-poly!! (source-poly!)
  (symmetrize-reduced-poly!! (reduce-poly!! source-poly!))
  )
```

3.6. Representant addresses and weights.

Each virtual processor (*VP*) must compute the address of the representant of its class. The *VP* of coordinates (i_1, i_2, \dots, i_m) is represented by the *VP* whose coordinates $(i'_1, i'_2, \dots, i'_m)$ are a permutation of (i_1, i_2, \dots, i_m) and are increasing from left to right. Each *VP* can thus easily determine the coordinates of its representant by sorting its own coordinates, and then translate these coordinates into a cube address to be used afterwards by the data transfer functions.

The following code fragment implements these operations. A temporary array *pvar c!* of *m* elements is allocated and initialized, its *i*-th element (**aref!! c! i**) receiving the *pvar* holding the coordinates of the virtual processors along axis *i*. The elementary datum in each virtual processor is thus an array containing its grid coordinates. These arrays are then sorted in place within all processors in parallel.

```
(*let ((c! (make-array!! m))) ;allocate a temporary array pvar of m elements
;; initialize the array of coordinates : the ith element of the array
;; receives the coordinates along axis i. Note that the internal axis
;; numerotation and the array indices vary from 0 to (- m 1)
(loop for i from 0 to (- m 1) do
(*setf (aref!! c! (!! i)) (my-coordinate!! i)))
;; bubble-sort the array of coordinates
(*let (tmp!) ;allocate a temporary pvar
(loop for j from 0 to (- m 1)
(loop for i from 0 to (- m (+ j 1))
(*setf tmp! (aref!! c! i)
(*when (<!! tmp! (aref!! c! (1+ i)))
(*setf (aref!! c! i) (aref!! c! (!! (1+ i))))
(*setf (aref!! c! (1+ i)) tmp!)
))))))
;; build the pvar holding in each VP the cube address of its
representant
...
;; build the pvar of weights
...
)
```

The sorted coordinates are then used in each *VP* to build the cube address *cube!* of its representant. This *pvar* is then stored as a permanent *pvar* for future reference by the function **representant-address!!**.

```
;; build the pvar holding in each VP the cube address of its
representant
(loop for j from 0 to (- m 1)
(*deposit-coordinate (aref!! c! (!! j)) j cube!))
(*store-representant-address cube!)
```

To compute the cardinal of its class each *VP* must count the number α_i of occurrences of the value *i* in its own grid coordinates and compute $\alpha!$, i.e. the product of the factorials of these α_i . These operations are most conveniently performed on the array *pvar* of sorted coordinates, as implemented by the function **multifact!!** below. The array of sorted coordinates is explored in ascending order and the *pvar result!* is computed in an incremental fashion. A *pvar* containing in each processor the number of occurrences of the lastly considered coordinate value is modified at each step according to the repartition of coordinates within each *VP*. At a given step, all the *VP*'s in which the current coordinate is equal to the previous one increment this *pvar*, and all other *VP*'s reset it to one. The product of this modified *pvar* and of **result!** is then computed and stored back in **result!**. This process is iterated for the remaining elements of the array.

```

(defun multifact!! (vect! length)
  "computes the multiple factorial of the sorted array pvar VECT! of
  size LENGTH, i.e. the product of the factorials of the number of
  occurrences of each value within each elementary vector."
  (*let ((result! 1)           ;allocates two temporary pvars,
        (occurrences! 1))    ;both initialized to 1
    (loop for n from 1 to length do
      (*if (=!! (aref!! vect! n) (aref!! vect! (1- n)))
          (*incf occurrences!) ;increment occurrences! by 1 in all VP's
          ;where the current coordinate is equal
          ;to the previous one
          (*setf occurrences! 1) ;reset occurrences! to 1 in all VP's
          ;where the current coordinate differs
          ;from the previous one
        )
      (*setf result! (*!! result! occurrences!))
    )
    result!)) ;return result!

```

The computation of the pvar of weights is then very simple. Again this *pvar* is stored permanently, for future reference by the function `weights!!`.

```

;; build the pvar of weights
(*setf weights! (/!! (multifact!! c! m) (factorial m)))
(*store-weights-pvar weights!)

```

3.7. Linear combinations of polynomials.

Since polynomials –or rather hypercube representations of polynomials– are stored as *pvar*'s, linear combinations of polynomials are trivially implemented using linear combinations of *pvar*'s : let **a**, **b** be two scalars and **p1!**, **p2!** be two *pvar*'s associated with two polynomials of same degree (and same number of variables). The linear combination **a.p1!+b.p2!** is implemented as

```
(+!! (*!! a p1!) (*!! b p2!))
```

3.8. Product of polynomials.

This product is implemented as the parallel product of extended hypercubes of coefficients :

```

(defun poly-product!! (poly1! poly2!)
  "computes the direct product of the two hypercube representations poly1!
  and poly2!"
  ;; set up the proper VP set and context for the result
  (*with-vp-set-and-context-for-degree (+ (degree-of poly1!)
                                         (degree-of poly2!))
    ;; compute and return the parallel product of the adequate extensions
    ;; of poly1! and poly2!
    (*!! (extend-poly!! poly1! :RIGHT) (extend-poly!! poly2! :LEFT))
  ))

```

where the extension function is defined as follows :

```

(defun extend-poly!! (poly! left-or-right)
  ;; extends poly! in the current vp set, leftwards or rightwards according
  ;; to the value of left-or-right
  (*let (cube!) ;allocate a pvar to hold addresses
        ;in the vp set of poly!
        ;; incremental construction of cube!:
        ;; when extending RIGHTWARDS, each processor retrieves the values in poly!
        ;; of coordinates equal to its (degree-of poly!) FIRST coordinates
        ;; when extending LEFTWARDS, each processor retrieves the value in poly!
        ;; of coordinates equal to its (degree-of poly!) LAST coordinates
        (let ((start (if (equal left-or-right :LEFT)
                        (- (current-degree) (degree-of poly!)) ;left start axis
                        0))) ;right start axis
          (loop for axis-in-poly from 0 to (- (degree-of poly!) 1) do
            (*deposit-vp-coordinate (my-coordinate (+ start axis-in-poly))
                                   axis-in-poly
                                   cube!
                                   (pvar-vp-set poly!))))
        ;; each active processor in the current vp set now gets the value held by
        ;; the processor in the VP set of poly! of address specified by cube!. The
        ;; resulting pvar is returned.
        (pref!! poly! cube!)
        )
  )
)

```

3.9. Computation of norms.

The *LISP name for the global summation $Sum()$ is `*sum`. Functions to compute various norms can be defined easily, as

```

(defun taylor-norm (poly!)
  "computes the ||2 norm of poly!"
  (*with-vp-set-and-context poly!
    (*let (sym-poly!) ;allocate a temporary pvar to hold the result
          ;of the symmetrization of poly!
          ;; symmetrization
          (*setf sym-poly! (symmetrize-poly!! poly!))
          ;; actual computation of the norm. The scalar result is returned.
          (sqrt (*sum (*!! sym-poly! (conjugate!! sym-poly!))))
          )
    )
  )
)

(defun L2-norm (poly!)
  "computes the L2 norm of poly!"
  (*with-vp-set-and-context poly!
    (*let (red-poly!) ;allocate a temporary pvar to hold the result
          ;of the reduction of poly!
          ;; reduction
          (*setf red-poly! (reduce-poly!! poly!))
          ;; actual computation of the norm. the scalar result is returned.
          (sqrt (*sum (*!! red-poly! (conjugate!! red-poly!))))
          )
    )
  )
)

```

3.10. Computation of partial derivatives.

As expressed in section 1.3, partial derivatives are obtained by a straightforward selection process. This selection is implemented using again general communication primitives : each active processor in the result *VP* set computes the address of the processor in the source *VP* set whose coordinates are the concatenation of its own with the indices of the variables with respect to which the derivative is taken, and then fetches the corresponding value in the symmetric representation of the source polynomial. The resulting *pvar* is multiplied by a scalar `quot-fact` equal to $m!/(m-p)!$ where m is the degree of the source polynomial and p the order of partial derivation.

```
(defun partial-derivative!! (poly! list-of-variables)
  "computes the partial derivative of poly! with respect to the variables
  specified by their index in list-of-variables : for instance to take the
  partial derivative of poly! with respect to x1, x2, x1 the calling sequence
  is (partial-derivative!! poly! '(1 2 1))"
  (let ((result-degree (- (degree-of poly!) (length list-of-variables)))
        (quot-fact 1))
    ;; computes the scalar normalization factor
    (loop for val from (+ result-degree 1) to (degree-of poly!) do
      (setf quot-fact (* quot-fact val)))
    ;; set up proper context and degree for the result
    (*with-vp-set-and-context-for-degree result-degree
      (*let (cube!) ;allocate a pvar to hold addresses
          ;in the vp set of poly!
          ;; incremental construction of cube!:
          ;; each processor retrieves the value in poly! whose first coordinates
          ;; are equal to its own, and whose last coordinates are specified by
          ;; list-of-variables (up to a translation of 1 since the internal
          ;; coordinates are zero-based)
          (loop for axis from 0 to (- result-degree 1) do
            (*deposit-vp-coordinate (my-coordinate axis)
              axis
              cube!
              (pvar-vp-set poly!)))
          (loop for index from 0 to (- (length list-of-variables) 1) do
            (*deposit-vp-coordinate (elt list-of-variables index)
              (+ result-degree index)
              cube!
              (pvar-vp-set poly!)))
          ;; each active processor in the current vp set now gets the value
          ;; in the symmetric representation of poly! held by the processor
          ;; of address given by cube!. The resulting pvar is then multiplied
          ;; by the quot-fact and returned.
          (*!! quot-fact pref!! (symmetrize-poly!! poly!) cube!)
        )
      )
    )
  )
)
```

Appendix : Practical experiments.

The aim of our work is to develop parallel processing for polynomials, and to realize the concrete implementation. It is certainly reasonable to ask the question “Do we gain any time compared to sequential processing ?”. Sequential computation was made on a *DECStation* 5000 model 200, using *MAPLE*. We computed the expansion of $(x_1 + \dots + x_N)^m$. The results, in seconds, are indicated below. The symbol “ ∞ ” means that the result was not obtained : either the machine ran out of memory, or we ran out of patience. Let’s also recall that timings with *MAPLE* are not quite reliable : they may differ from one experiment to the next, on the same machine. Also, depending on the amount of memory available at a given time, some computations can, or cannot, be performed.

- For $N = 2$:

m	8	10	15	16	17	18	19	20	21
CM	0.05	0.15	0.22	0.3	0.9	1.6	4.3	11	∞
DS	0	0.01	0.03	0.03	0.03	0.03	0.05	0.05	0.05

- For $N = 4$:

m	5	6	7	8	9	10	11
CM	0.05	0.11	0.13	0.9	6	22	∞
DS	0.03	0.08	0.08	0.11	0.18	0.21	0.30

They indicate, as one would expect, that for few variables and high degrees sequential computation is by far preferable. Conversely, for many variables and low degrees, parallel processing gains a lot, and we see a speed-up by a factor 10 to 100 :

- For $N = 8$:

m	4	5	6	7
CM	0.05	0.23	1.6	15.6
DS	0.36	0.75	2.08	4.8

- For $N = 16$:

m	2	3	4	5
CM	0.01	0.05	0.32	9.5
DS	0.08	0.88	6.38	37.2

- For $N = 32$:

m	2	3	4
CM	0.03	0.13	5.9
DS	0.45	3.48	∞

- For $N = 64$:

m	2	3
CM	0.03	0.9
DS	0.45	∞

- For $N = 128$:

m	2	3
CM	0.04	7.4
DS	1.63	∞

- For $N = 256$:

m	2
CM	0.13
DS	11.80

Finally, to expand $(x_1 + \dots + x_N)^2$, it takes $0.56s$ to the CM for $N = 512$ and $2.3s$ for $N = 1024$.

The limitations on the Connection Machine are given by the number of virtual processors. Computations are very fast in the range $N^m \leq 2^{16}$, they quickly deteriorate outside this range.

The slow part on the Connection Machine is the data transfer between processors (on average 30 times slower than operations inside processors). In order to describe these communications, one has to say a word about the geometry of the Connection Machine. Our set of N^m processors is in fact mapped onto a set of $\{0, 1\}^K$ (a dyadic hypercube), and communications are fast between two processors having all dyadic coordinates equal but one, and communications are slower in all other cases. The former are called “regular communications”, the former “general communications” (a general study has been made by Nassimi-Shami [4] of data transfers in SIMD machines, but the Connection Machine does not fall exactly into their schemes).

When the number of variables is a power of 2 (as in the examples above), we have exploited regular communications as much as we could, but we did not try to optimize the representation for other polynomials.

The length of time taken by operations on polynomials depend much more on the degree than on the number of variables. When the polynomial is represented on the hypercube, and if we perform operations with no data transfer (such as direct products, sums, partial derivatives) then all processors work at the same time, and timing is independent of the number of variables (provided, of course, there are enough processors).

For operations which require data transfer, such as products, the number of successive transfers depends mostly on the degree : they are performed inside each equivalence class of processors, and each class contains at most $m!$ processors.

For high degree and low number of variables, two factors appear : one must create a high-dimensional geometry, and fill it with redundant data.

An homogeneous polynomial of degree m , with N variables, has a number of terms at most equal to $\binom{N+m-1}{m}$, and the hypercube has N^m points. The latter is of course bigger than the former, but for fixed m they are of the same order of magnitude when N becomes large. For instance, for $m = 3$, the polynomial has $(N + 2)(N + 1)N/6$ terms, and the cube N^3 . Conversely, for fixed N , the ratio increases rapidly with m , and the number of redundant data will be very large. Since the operation of data transfer is slow, the whole procedure will not be advantageous.

But of course, for all operations which are realized on the hypercube (such as computing the $[\cdot]_2$ -norm), parallel processing will be highly efficient, far more than sequential processing.

References.

- [1] Beauzamy, B. – Bombieri, E. – Enflo, P. – Montgomery, H. : Products of polynomials in many variables. *Journal of Number Theory*, vol. 36, 2, oct. 1990, pp. 219–245.
- [2] Beauzamy, B. : Products of polynomials and a priori estimates for coefficients in polynomial decompositions : a sharp result. To appear in the *Journal of Symbolic Computation*, 1991.
- [3] Gourdon, X. : Algorithmique du théorème fondamental de l’algèbre. Rapport de recherche INRIA, juin 1992. *To appear*.
- [4] Nassimi, D. - Sahni, S. : Data broadcasting in SIMD computers. *IEEE Trans. on Computers*, vol. C-30, No 2, feb. 1981, pp. 101-107.
- [5] Programming in *LISP. *Thinking Machines Corp.*, Cambridge, Mass.