Cost-Efficient Parallel Programs Based on Set-Distributions for Polynomial Interpolation

Virginia Niculescu

Department of Computer Science
Faculty of Mathematics and Computer Science
Babeş-Bolyai University
Cluj-Napoca, Romania

Correspondence address:
Department of Computer Science
Faculty of Mathematics and Computer Science
Babeş-Bolyai University
1 M. Kogalniceanu, 400084
Cluj-Napoca, Romania

fax: +40-264-591906
telephone:+40-264-418655 (int 5807)
e-mail: vniculescu@cs.ubbcluj.ro

Preprint submitted to Elsevier Science *27 March 2007*

Abstract

The paper presents parallel algorithms for Lagrange and Hermite interpolation methods formally derived from specifications, and using set-distributions. Set-distributions are based on set-valued mappings, and they assign a data object to more than one process. The derivation from specifications assures the correctness, and the set-distributions assure the efficiency of the programs. The obtained parallel algorithms have very good time complexities and speeds-up, and they are also cost-efficient. We consider the number of processes p to be a parameter of the algorithms, so, bounded parallelism is considered. The derivation of the algorithms is not ruled by any particular interconnection network. The possible mappings on different networks could be evaluated. The performance analysis is done considering a full-connected network, and other two interconnection networks: hypercube and multi-mesh hypercube, which preserve the cost-efficiency of the algorithms.

Key words: parallel computation, polynomial interpolation, Lagrange, Hermite, data-distribution, complexity, cost, efficiency, formal derivation

1 Introduction

Lagrange and Hermite interpolations are well known methods of polynomial interpolation, and at the same time they are of great importance since they are widely used. Their parallelization has been discussed before, and parallel algorithms with good time complexities have been obtained [1,7,8,2,11,20,21]. But for these parallel algorithms the case of bounded parallelism (the number of processors is limitated) was not explicitely analyzed, and also they are not formally derived. For simple interpolation (the information is the values of the function in n points) with Lagrange or Newton representation, very good complexities ($O(\log n)$) have been obtained, provided that large number of processors are used - polynomial in the size of the problem (n) [1,14,9]. If the number of processor was limited to n, and special interconnection networks used (star graph, k-ary cube, tree augmented with ring connections, systolic), then complexities equal to $O(n)$ have been obtained [11,20,21]. Also, there is an algorithm with a complexity equal to $O(\sqrt{n})$ using a multi-mesh of $k^2(k = n^2)$ processors presented by M. De [4]. For the more general problem of Hermite interpolation, where the input is a set of distinct points (m), and corresponding to each point, prescribed values for the function and all its derivatives up to some arbitrary order (r), Egecioglu et al. [7,8] have described a parallel algorithm with a $O(\log m + \log^2 r)$ complexity, provided that more than $O(m^2 r)$ processors are used.

This paper presents a formal derivation of parallel algorithms for polynomial interpolation, considering *bounded parallelism*, thus a more practical approach. It means that we will consider the number of the processes p to be a parameter of the parallel programs derivation. Parallel algorithms for polynomial interpolation based on Lagrange and Hermite methods are derived starting from formal specifications and using set-distributions. The obtained algorithms are cost-efficient and correct by construction. If the number of processor is large enough (m^2), then the logaritmic complexities are achieved. The algorithms are not derived for a particular interconnection network; we start by considering a full-connected network, and then we analyze the mappings onto two networks: hypercube and multi-mesh hypercube.

A process aiming at building software that is correct by construction is essential for parallel programming, since the verification of parallel programs correctness after construction seems too difficult for practical use [22]. Similar techniques with those used for sequential programming may be used for parallel programming, too.

Set-distributions [19], based on set-valued mappings, can be used with good results in parallel processing. These distributions assign a data object to more than one process; very good results are obtained especially when a datum is

used in more than one computation.

The derivation of the algorithms is done in a formal way using a method for parallel programs design, based on derivation from specification [18]. The method is based on that presented in [17]. The model considers a parallel program as a number of cooperating parameterized processes with similar structures. The first part of the paper briefly explains this method.

1.1 Notations

A notation for quantifications is used:

$$(\odot k : \ Q : \ E)$$

where $\odot$ is a quantifier ($\sum, \forall, \max, \ldots$), k is the list of bound variables, Q is the predicate describing the domain of the bound variables, and E is an expression. Function application is denoted by an infix, left associative dot '.' operator. The set $\{\forall i : \ 0 \leq i < n : \ i\}$ is denoted by $\overline{n}$. For the integer remainder we use the symbol %, and $p|n$ means that n is divisible by p ($\forall n, p \in \mathbb{N}$).

Also, the following proof notation is used:

$$\begin{array}{ll} & expression_1 \\ = & \{\text{explanations}\} \\ & expression_2 \\ \ldots & \end{array}$$

Program notation is given in Appendix A.

1.2 Program Derivation

A parallel program is considered to be formed by many parameterized processes $S.q(0 \leq q < p)$, which are running in parallel. There is no shared memory. A parameterized process is much like a procedure in sequential programming. The difference is that, instead of having only one instantiation in a sequential program, we have many instantiations in a parallel program.

In sequential programming, the Hoare-triple [6]

$$\{Q\}S\{R\}$$

is commonly used to denote a formal specification of a program S. This notation means that if the program S starts in a state described by the predicate Q, and if the program terminates, then, upon completion, the predicate R is satisfied.

For specifying a parallel program, both pre- and post-conditions, Q and R, are split up as conjunctions of $p (p > 0)$ local pre- and post-conditions, and a process is associated with each such pair:

$$\{Q.q\}S.q\{R.q\}, \forall q : 0 \leq q < p.$$

So, we may specify a parallel program as follows:

$|[\ \{Q\}$

$\quad par\ q : 0 \leq q < p :$

$\quad \{Q.q\}$

$\quad\quad S.q$

$\quad \{R.q\}$

$\quad rap$

$\{R\}]|$

A parameterized specification usually refers to some local variables representing a part of a distributed object. Therefore, the parameters of a specification are q, p and a data distribution D. Many choices are possible for D, each of them having an impact on the complexity of the parallel program.

Such a specification forms the starting point for a parallel program derivation, a formal construction of parameterized processes constituting a parallel program.

Our approach for obtaining a parameterized process S from a functional specification is similar to the methods used in sequential programming. It is based on the axioms and inference rules specified in Appendix B. These methods obtain an invariant from a specification, in a calculational style. The programs are derived by calculating the necessary conditions to maintain the invariant. Because we have parameterized specifications, we will derive parameterized invariants.

A global specification requires some form of coordination between processes, so, in order to satisfy the specification, several processes have to interact with one another, via message exchanging.

A parameterized process is refined into a sequence of ordinary sequential programs and communication processes; communication is based on message-passing. A parallel program is, thus, decomposed into layers of process instances. All the communications occur between instances of the same parameterized communication process, and such a process is called *communication closed*.

1.3 Performance Analysis

Time-Complexity
For a parallel program with the input size n and the number of processes p, we consider that the time complexity is expressed by:

$$T.p.n = T_f.p.n + \alpha * T_c.p.n \tag{1}$$

where

- $T_f.p.n =$ the computation complexity (measured in t_f);
- $T_c.p.n =$ the communication complexity (measured in t_c);
- $t_c =$ the time required to communicate a single value;
- $t_f =$ the time required to execute a single basic computational operation;
- $\alpha = \dfrac{t_c}{t_f}$.

The communication complexity depends on the communication network. Different kinds of networks may be considered: ideal(full-connected) intercommunication network, chain, mesh, toroid, hypercube,...; the communication processes have to be evaluated depending on their structure. In what it follows, we present two interconnection networks for which we will analyze the mapping of our algorithms.

Hypercube-Mesh Network
The first network on which we analyse the mapping of our algorithms is the hypercube. We consider an equivalent variant of it, in which each node is labeled with a pair of numbers $(i, j), 0 \leq i < 2^m, 0 \leq j < 2^n$, so we have $2^m * 2^n$ nodes . We call this variant *hypercube-mesh*, and it is characterized by the pair $(m, n),\ m, n \in \mathbb{N}^*$ (we denote it by $HM_{(m,n)}$).

In $HM_{(m,n)}$, between two nodes (i_x, j_x) and (i_y, j_y)

(1) there is a hypercube link iff
  (a) $i_x = i_y$, and
  (b) j_x and j_y differ by one bit position in their binary representation;

(2) there is a hypercube link iff
  (a) $j_x = j_y$, and
  (b) i_x and i_y differ by one bit position in their binary representation.

We may consider that we have 2^m connected hypercubes of order n, or 2^n connected hypercubes of order m. The total number of nodes is equal to $2^m * 2^n = 2^{m+n}$, and the diameter is $m+n$. An example is shown in Appendix C, Figure .1.

Theorem 1 *The interconnection network hypercube-mesh $HM_{(m,n)}$ is equivalent to the network hypercube H_{m+n}.*

Proof: Each node (i, j) from $HM_{(m,n)}$ corresponds to the node $i * 2^n + j$ in H_{m+n}. (Vice versa, a node x of the hypercube coresponds to a node (i_x, j_x) in $HM_{(m,n)}$, where i_x has its binary representation equal to the first m bits from the binary representation of x, and j_x has its binary representation equal to the last n bits from the binary representation of x). For example, if we have the hypercube H_5, and we consider $m = 2$ and $n = 3$, then the node $01100_2 = 12$ in H_5 corresponds to the node $(01_2, 100_2) = (1, 4)$ in $HM_{(2,3)}$. Also, if x and y are two nodes in H_{m+n}, and they differ by one bit position in their binary representation (there is a link between them), then between their correspondents (i_x, j_x) and (i_y, j_y) we have the relation (1) or (2) – so there is a link between them, too.

Multi-Mesh Hypercube Network
A Multi-Mesh Hypercube Network [16] is characterized by a triplet (r, c, n), where r represents the row dimension of a torus, c the column dimension of the torus, and n the dimension of a hypercube.

Between two nodes (i_1, j_1, k_1) and (i_2, j_2, k_2), where $0 \leq i_1 < r, 0 \leq i_2 < r, 0 \leq j_1 < c, 0 \leq j_2 < c, 0 \leq k_1 < 2^n$, and $0 \leq k_2 < 2^n$:

(1) there is a torus link iff:
  (a) $k_1 = k_2$, and
  (b) $(i_1 = (i_2 + 1)\%r \vee i_2 = (i_1 + 1)\%r) \wedge j_1 = j_2 \vee$
  $(j_1 = (j_2 + 1)\%c \vee j_2 = (j_1 + 1)\%c) \wedge i_1 = i_2$;
(2) there is a hypercube link iff:
  (a) $i_1 = i_2$, and (b) $j_1 = j_2$, and
  (c) k_1 and k_2 differ by one bit position in their binary representation.

The total number of nodes is equal to $r * c * 2^n$, and the diameter is $r * c * n$. An example for OMMH$(2, 2, 3)$ is shown in Appendix C, Figure .2.

For this network, which is scalable, there are practical realizations with optical interconnections OMMH [16].

Cost
In the analysis of the parallel programs, we also need to evaluate the total amount of work, and for that we use the *cost* measure [13].
The *cost* $C.p$ of a parallel algorithm using p processes is defined by:

$$C.p.n = T.p.n * p, \tag{2}$$

where $T.p.n$ is the time complexity of the parallel algorithm, and n is the size of the problem. A parallel algorithm is considered *cost-optimal* if $C.p.n = O(T_s.n)$, where $T_s.n$ is the complexity of the best known sequential algorithm of the problem. And, a parallel algorithm is *cost-efficient* if $C.p.n = O(T_s.n * (\log_2 n)^k), k \in \mathbb{N}$.

1.4 Distributions

Two kinds of data distributions can be used:

- *simple* distributions, defined by single-valued mappings;
- *set*-distributions, defined by set-valued mappings.

Definition 1 *$D = (\delta, A, B)$ is called a* simple distribution *if A and B are finite sets, and δ is a mapping from A to B. Set A specifies the set of data objects (an array with n elements), and set B specifies the set of processes, which is usually $\overline{p}$. The function δ assigns each index $i (0 \leq i < n)$, and its corresponding element to a process number [17].*

Definition 2 *A* set-distribution *for n input objects on p processes is defined by a set–valued mapping $\theta : \overline{n} \multimap \overline{p}$; $\theta.i$ represents the set of processes containing the data object i [19].*

Examples of set-distributions, when $p > n$ are:

$$\widetilde{\theta^{linear}}.i = \{\forall k : 0 \leq k < p/n : i(p/n) + k\}, \text{ if } n \mid p$$
$$\widetilde{\theta^{cyclic}}.i = \{\forall k : 0 \leq k < p/n : kn + i\}, \text{ if } n \mid p$$

When the processes' number is greater than the data input size, it is desirable to assign a datum to more than one process. Also, when a data object is used in more than one computation, this kind of distribution may lead to efficient algorithms.

There are cases when, in order to express more easily the sets of the processes that contain a datum, a set-distribution may be defined using a simple distri-

bution, which is, for example, replicated. More formally, such a set-distribution is defined by considering the processes organized as a $M \times N$ matrix of processes ($p = M * N$), and by using simple distributions: $\delta_0 : \overline{n} \to \overline{M}$ and $\delta^t : \overline{M} \to \overline{M}$ (the expression of δ^t depends on $t, 0 \leq t < N$), such that x_i belongs to (s,t) iff $\delta_0.i = \delta^t.s$. If $\delta^t = 1_{\overline{M}}$, then δ_0 is simply replicated on columns.

2 Lagrange Polynomial

We consider the problem of evaluating the Lagrange polynomial on a given point, and we formally derive a parallel algorithm for it.

2.1 The Problem

Let $[a,b] \subset \mathbb{R}, x_i \in [a,b], 0 \leq i \leq m$, such that $x_i \neq x_j$ for $i \neq j$ and $f : [a,b] \to \mathbb{R}$.

Lagrange interpolation polynomial is defined as [3]:

$$(L_m f).x = \left(\sum i : 0 \leq i \leq m : l_i.x * f.x_i\right) \tag{3}$$

where $l_i, 0 \leq i \leq m$ are the fundamental Lagrange interpolation polynomials:

$$\begin{aligned} l_i.x &= \frac{(x-x_0)\ldots(x-x_{i-1})(x-x_{i+1})\ldots(x-x_m)}{(x_i-x_0)\ldots(x_i-x_{i-1})(x_i-x_{i+1})\ldots(x_i-x_m} \\ &= \frac{u.x}{(x-x_i)} \cdot \frac{1}{(x_i-x_0)\ldots(x_i-x_{i-1})(x_i-x_{i+1})\ldots(x_i-x_m)} \end{aligned} \tag{4}$$

where $u.x = (\prod i : 0 \leq i \leq m : (x - x_i))$.

For a given value x, the global postcondition for the computation of $(L_m f).x$ is

$$R : \ lx = (L_m f).x$$

2.2 Derivation

Let $p = M * M$ and all the processes be identified by pairs $(s,t), 0 \leq s,t < M$.

s\t	0	1	2
0	x_0, x_3, x_6	x_1, x_4, x_7	x_2, x_5, x_8
1	x_1, x_4, x_7	x_2, x_5, x_8	x_0, x_3, x_6
2	x_2, x_5, x_8	x_0, x_3, x_6	x_1, x_4, x_7

Fig. 1. The data distribution for $m = 8$ and $M = 3$.

We consider a cyclic distribution $\delta : \overline{m+1} \rightarrow \overline{M}, \delta.i = i\%M$, and M permutations $\pi_t : \overline{M} \rightarrow \overline{M}, 0 \leq t < M$, defined by $\pi_t.i = (i + t)\%M$.

The set-distribution is defined by:

$$x_i \in O.s.t \Leftrightarrow \delta.i = \pi_t.s \Leftrightarrow i\%M = (s + t)\%M \tag{5}$$

where $O.s.t$ is the set of data elements that are assigned to the process (s, t).

For $m = 8$ and $M = 3$ the data distribution is shown in Figure 1.

There are three stages defined by the following local postconditions:

$$R0.s.t : \ (ux = u.x \wedge (\forall i : \ i \in O.s.t : xx[i] = x - x_i))$$
$$R1.s.t : (\forall i : i \in O.s.t : l[i] = l_i.x)$$
$$R2.s.t : \ (lx = (L_m f).x)$$

The first and the last stages represent a product and a summation, and they may be computed using tree-like computations, with slight modification over the classic one. Two stages are used: a computation on columns and one on rows.

In what follows, we discuss about the second stage, in more details.

We will consider two sub-stages: one for partial computations, and one for combining the partial computations.

Each row $(s, .)$ computes the values $l(i), \forall i : 0 \leq i \leq m \wedge \delta.i = s$.

The postcondition $R1.s.t$ may be rewritten as:

$$R1.s.t : (\forall i : i \in O.s.t : \quad l(i) = ux/xx[i] * 1/prod.i.m)$$

where $prod.i.m = (\prod j : 0 \leq j \leq m \wedge i \neq j : (x_i - x_j))$.

To compute the products $prod.i.m$, we split them in M products. Each of

these subproducts corresponds to the set of elements assigned to a process. The elements (x_i) of $O.s.0$ are sent by broadcast onto the row s.

Hence, we rewrite the products $prod.i.m$ as:

$$
\begin{aligned}
& prod.i.m \\
= & \{\text{range spliting}\} \\
& (\prod t : 0 \leq t < M : (\prod j : j \in O.s.t \wedge i \neq j : (x_i - x_j))) \\
= & \{(j \in O.s.t \Leftrightarrow 0 \leq j \leq m \wedge j\%M = (s+t)\%M) \wedge s = \delta.i\} \\
& (\prod t : 0 \leq t < M : (\prod j : 0 \leq j \leq m \wedge \delta.j = \pi_t.(\delta.i) \\
& \qquad \wedge i \neq j : (x_i - x_j))) \\
= & \{parprod.i.t \stackrel{not}{=} (\prod j : 0 \leq j \leq m \wedge \delta.j = \pi_t.(\delta.i) \\
& \qquad \wedge i \neq j : (x_i - x_j))\} \\
& (\prod t : 0 \leq t < M : parprod.i.t)
\end{aligned}
$$

The values $prod.i.m$ are obtained from $parprod.i.t$ $(0 \leq t < M)$, using a tree-like computation on each row.

The invariants for partial products computations are defined by introducing the variable k, which is incremented by M:

$$
\begin{aligned}
& P1.s.t : P1.0.s.t \wedge P1.1.s.t \\
& P1.0.s.t : \ 0 \leq k \leq m \wedge k\%M = 0 \\
& P1.1.s.t : \ (\forall i : 0 \leq i < k \wedge \delta.i = s : ppr[i] = parprod.i.t).
\end{aligned}
$$

The parameterized processes $S.s.t, 0 \leq s, t < M$ for partial products computation are presented in Appendix B.

2.3 Time Complexity and Cost

First we consider that we have an ideal interconnection network, and then we analyze the cases of hypercube-mesh and multi-mesh hypercube networks.

Lemma 1 *The time complexity for the second stage is:*

$$T^{R1}.p.m = 2\frac{(m+1)^2}{M^2} + \frac{m+1}{M}(\log_2 M) + \alpha\frac{(1+\log_2 M)}{M}(m+1). \qquad (6)$$

And the parallel algorithm for the second stage is cost-efficient.

Proof: The values *prod.i.m* are obtained using a tree-like computation on each row, which means that the time complexity for these computations is:

$$\left(\frac{m+1}{M}\log_2 M\right)(\alpha+1)$$

For the broadcast communications on rows, and for the computation of the partial products *parproduct.i.t*, the time complexity is:

$$\frac{m+1}{M}\left[\alpha+2\frac{m+1}{M}\right]$$

If we add these complexities, we obtain the time complexity $T^{R1}.p.m$.

The time complexity in the sequential case is $O(m^2)$ and the cost of the parallel algorithm (the number of processes is $p = M^2$) is

$$C^{R1}.p.m = T^{R1}.p.m * M^2 = O(m^2 + m * M * \log_2 M)$$

The maximal value for M is m, so, the algorithm is cost-efficient.
If $M < m/\log_2 M$, then the algorithm is *cost-optimal*.

The values *prod.i.m* are independent of x (the current point where the function is approximated), so their computation could be separated and executed only once, if more than one point x is considered. Sequentially, their computation is the most costing one, and the corresponding parallel computation is cost-efficient.

Theorem 2 *The parallel algorithm for Lagrange interpolation is cost-efficient.*

Proof: The first and the third stages execute tree-like computations, and these kinds of computations are cost-efficient [13]. The second stage of computation is cost-efficient (Lemma 1), and thus the global parallel algoritm is cost-efficient.

If $p = m^2$, the time complexity of the parallel program is $O(\log_2 m)$, which is equal to the previous results [1,8], where more than m^2 processors are used. Here bounded parallelism has been considered, and the formally derived program is cost-efficient.

2.3.1 Mappings

Tree-like computations are executed on rows and columns. Generally, the tree-like computations are efficiently implemented on hypercube interconnection

networks. On a hypercube with N nodes, a tree-like computation is executed with $(\alpha+1)\log_2 N$ time-complexity, and a broadcast is executed with $\alpha(\log_2 N)$ time-complexity.

If we consider a hypercube-mesh interconnection network $HM_{(n,n)}$, where n is such that $M = 2^n$, then for our algorithm, the time complexity and also the cost are preserved. (We have hypercubes on each row and also on each column.)

We may also consider a multi-mesh hypercube interconnection network characterized by the triple $(k*2^n, k, n)$ $(M = k*2^n, k \geq 1, n \geq 0)$, and the mapping $(s,t) \rightarrow (s, t/2^n, t\%2^n)$. In this case, a tree-like computation on columns could not be executed with $(\alpha+1)(\log_2 M)$ time-complexity, but with $(\alpha+1)(M/2)$ time-complexity, since on columns there are only ring connections. On rows, the tree-like computations and the broadcast could be executed with $O(k+n)$ time-complexity. There are k hypercubes of order n on each row; so first, partial computations are executed on each hypercube (time complexity is $(\alpha+1)n$), and then, they are combined using the ring connections (time-complexity $k/2(\alpha+1)$). If $k \in O(\log_2^t m), t \geq 0$, then the algorithm remains cost-efficient.

3 Hermite Interpolation

3.1 The Problem

Let $[a,b] \subset \mathbb{R}, x_i \in [a,b], 0 \leq i < m$, such that $x_i \neq x_j$ for $i \neq j$, $r_k \in N, k \in \overline{m+1}$ and $f : [a,b] \rightarrow \mathbb{R}$ such that $\exists f^{(j)}.x_k, \forall k \in \overline{m+1}, 0 \leq j \leq r_k$.

Hermite interpolation polynomial is defined as [3]:

$$(H_{(n)}f).x = \Big(\sum k : 0 \leq k \leq m : \Big(\sum j : 0 \leq j \leq r_k : h_{kj}.x * f^{(j)}.x_k\Big)\Big) \quad (7)$$

where $m + r_0 + \ldots + r_m = n$, and $h_{kj}, 0 \leq k \leq m, 0 \leq j \leq r_k$ are the fundamental Hermite interpolation polynomials:

$$h_{kj}.x = \frac{(x-x_j)^j}{j\ !} * u_k.x* \left(\sum \nu : 0 \leq \nu \leq r_k - j : \frac{(x-x_k)^\nu}{\nu\ !} \left[\frac{1}{u_k.x}\right]^{(\nu)}\Bigg|_{x=x_k}\right) \quad (8)$$

where $u_k.x = \Big(\prod i : 0 \leq i \leq m \wedge i \neq k : (x-x_i)^{r_i+1}\Big)$.

3.2 Derivation

The global postcondition for the computation of the value $H_{(n)}f.x$ in the given point x is

$$R: \ Hf = (H_{(n)}f).x$$

We consider $p = M * N$ processes, where $M \leq m+1$, $N \leq (\min \ k : 0 \leq k \leq m : r_k)$, and the processes are identified by the pairs $(s,t), 0 \leq s < M, 0 \leq t < N$.

The input data: $x, x_k, r_k (k \in \overline{m+1}), f[k,j] = f^{(j)}.x_k \ (k \in \overline{m+1}, j \in \overline{r_k+1}))$ are distributed as folllows:

- x is assigned to all the processes,
- $x_k, r_k (k \in \overline{m+1})$ are local to the process (s,t) iff $\delta_0.k = s$ ($\delta_0 : \overline{m+1} \to \overline{M}$ is a simple distribution, so we have a set-distribution with $\delta^t = 1_{\overline{M}}$),
- $f[k,j] (k \in \overline{m+1}, j \in \overline{r_k+1})$ are assigned to the process (s,t) iff $\delta_0.k = s$ and $\delta_1^k.j = t$ (where $\delta_1^k : \overline{r_k+1} \to \overline{N}, \forall k \in \overline{m+1}$ are simple distributions).

From the global postcondition we can obtain the local postconditions, for $0 \leq s < M, 0 \leq t < N$:

$$R.s.t: \ s = t = 0 \Rightarrow Hf = (H_{(n)}f).x$$

The local postconditions are split into three partial postconditions:

$$R0.s.t \wedge R1.s.t \wedge R2.s.t \Rightarrow R.s.t$$

$$R0.s.t : (\forall (k,j) : (k,j) \in local.s.t : h[k,j] = h_{kj}.x)$$

$$R1.s.t : lHf.s.t = (\sum k, j : (k,j) \in local.s.t : h[k,j] * f[k,j])$$

$$R2.s.t : s = t = 0 \Rightarrow Hf = (\sum s, t : 0 \leq s < M \wedge 0 \leq t < N : lHf.s.t)$$

where $local.s.t = \{\forall k, j : 0 \leq k \leq m \wedge 0 \leq j \leq r_k \wedge \delta_0.k = s \wedge \delta_1^k.j = t : (k,j)\}$.

Computation for $R1.s.t$ and $R2.s.t$

The postconditions $R1.s.t$ and $R2.s.t$ are easily satisfied because they only imply simple summations: $R1.s.t$ implies partial sums, and $R2.s.t$ leads to the global sum. The global sum may be obtained by computing the partial sums of each row, and finally the global sum may be computed using a tree-like computation on the first column.

The time complexity of the computations for $R1.s.t$ and $R2.s.t$ may be evaluated as:

$$T^{R1+R2}.p.m.r = \frac{(m+1)(r+1)}{M*N} + (\log_2 M + \log_2 N)(1+\alpha), \tag{9}$$

where $r = (\max\, i : 0 \leq i \leq m : r_i)$.

Computation for $R0.s.t$

We also split this postcondition into three partial postconditions:

$$R01.s.t \wedge R02.s.t \wedge R03.s.t \Rightarrow R0.s.t$$
$$R01.s.t : (\forall k :\ \delta_0.k = s : ux[k] = u_k.x)$$
$$R02.s.t : (\forall k :\ \delta_0.k = s : u[k] = u_k.x_k)$$
$$R03.s.t : (\forall (k,j) :\ (k,j) \in local.s.t : h[k,j] = h_{kj}.x)$$

Each of them will be analyzed in what follows.

Computation for $R01.s.t$

The postcondition $R01.s.t$ may be rewritten as:

$$R01.s.t : ux[k] = ux/px.k.(r_k+1)\ \forall k : \delta_0.k = s,$$

where

$$ux = \left(\prod i : 0 \leq i \leq m : (x - x_i)^{r_i+1}\right), \text{ and } px.k.j = (x - x_k)^j$$

For computing the powers $px.k.j = (x - x_k)^j$ a parallel-prefix algorithm [13] on the row $(\delta_0.k, \cdot)$ may be used. We impose that $\delta_1^k.(r_k) = 0$, and so the value $px.k.(r_k+1)$ will be available in the process $(\delta_0.k, 0)$. Then, the product ux may be computed using local computations followed by a tree-like computation on the first column. So, the time complexity for this stage is:

$$T.p.m.r = \log_2 N \frac{(m+1)(r+1)}{M*N}(\alpha+1) + \frac{m+1}{M} + \log_2 M(\alpha+1) \tag{10}$$

Computation for $R02.s.t$

The postcondition $R02.s.t$ implies more complex communications, so it has to be more carefully analyzed. The values x_k are distributed using a set-

Fig. 2. Generalized diagonals for $M = 9$ and $N = 3$.(The matrix is 90° rotated.)

distribution defined by the simple distributions δ_0 and $\delta^t = 1_{\overline{M}}$. We are going to use this, in order to obtain an efficient algorithm for $u_k.x_k$ computation. We try to split the postconditions such that all the processes to be used:

$$
\begin{aligned}
& (\prod i : 0 \leq i \leq m \wedge i \neq k : (x_k - x_i)^{r_i+1}) \\
= \; & \{\text{range splitting }\} \\
& (\prod t : 0 \leq t \leq N : \\
& \quad (\prod i : 0 \leq i \leq m \wedge i \neq k \wedge \mathit{condition.s.t.i} : (x_k - x_i)^{r_i+1}))
\end{aligned}
$$

The condition introduced by the predicate *condition.s.t.i* has to specify which values x_i are used by the process (s, t). For the sake of simplicity, we assume that $N|M$. So, we may think that the rows are grouped in subgroups of N rows. We would like to group together the processes which are on the same "generalized diagonal". Two processes (s_1, t_1) and (s_2, t_2) are on the same generalized diagonal iff $(s_1 - t_1)\%N = (s_2 - t_2)\%N$. (If we do not make the assumption $N|M$, the formula for defining a "generalized diagonal" is more complicated, but we are free to choose the values M and N under the assumption $M * N = p$.) Figure 2 shows the generalized diagonals for the case $M = 9$ and $N = 3$. (For saving space, the matrix is 90° rotated, so the rows are shown vertically.)

Based on this, we choose $\mathit{condition.s.t.i} \equiv (s - t)\%N = \delta_0.i\%N$.

$$
\begin{aligned}
& (\prod i : 0 \leq i \leq m \wedge i \neq k : (x_k - x_i)^{r_i+1}) \\
= \; & \{ \text{ for given } s \text{ and } i \; \exists!\; t : (s - t)\%N = \delta_0.i\%N\} \\
& (\prod t : 0 \leq t \leq N : \\
& \quad (\prod i : 0 \leq i \leq m \wedge i \neq k \wedge (s - t)\%N = \delta_0.i\%N : (x_k - x_i)^{r_i+1}))
\end{aligned}
$$

So, the values x_k, r_k $(\forall k : \delta_0.k = 0)$ local to the processes $(0, t)$ are received by the processes: $(1, 1)$, $(2, 2)$, $(3, 0)$, $(4, 1)$, $(5, 2)$,$(6, 0)$,$(7, 1)$, $(8, 2)$. The values $x_k, r_k (\forall k : \delta_0.k = 1)$ local to the processes $(1, t)$ are received by the processes: $(2, 1)$, $(3, 2)$, $(4, 0)$, ..., $(8, 1)$, $(0, 2)$; and so on...

In the case of an ideal interconnection network, the communication process is described in Figure 3; we have denoted by X_s a vector that contains all the points x_i for which $\delta_0.i = s$ (and similar for R_s).

$C0.s.t$ ::

|[

{local values are sent to the processes on the same generalized diagonal}

par $u : 0 < u < M \wedge u \neq s \wedge (u - t)\%N = s\%N$:

(u, t) ! (X_s, R_s)

rap

par $u : 0 < u < M \wedge u \neq s \wedge u\%N = (s - t)\%N$:

(u, t) ? (X_u, R_u)

rap

]|

Fig. 3. The communication process for $u[k]$ computation.

Finally, the values $u_k.x_k$ are obtained by using tree-like computations on rows with a time complexity equal to $T_{row} = \dfrac{m+1}{M} \log_2 N(1+\alpha)$.

For an ideal interconnection network, the time complexity for computing $u_k.x_k$ may be evaluated as:

$$\begin{aligned} T^{R02}.p.m.r &= (T_f + \alpha T_c) + T_{row} \\ &\cong \frac{(r+1)(m+1)^2}{M*N} + 2\alpha\frac{m+1}{M}\frac{M}{N} + \frac{m+1}{M}\log_2 N(1+\alpha) \quad (11) \\ &= \frac{(r+1)(m+1)^2}{M*N} + 2\alpha\frac{m+1}{N} + \frac{m+1}{M}\log_2 N(1+\alpha) \end{aligned}$$

Sequentially, this computation can be done with a time complexity $T_s \cong (r+1)(m+1)^2$. So, the speed-up is

$$S^{R02}.p.m.r \cong p/(1+K)) \quad (12)$$

where K is a number less than $2\alpha/(r+1) + (1+\alpha)\log_2 N/(m+1)$.

If we want to hide the communication cost by overlapping communications and computation, we may use an algorithm similar to a systolic one, in which a communication step is followed by a computation step.

Due to the fact that each datum is used here in more than one computation, the advantage of using set-distributions is very important.

Computation for $R03.s.t$

First, we have to analyze the expression $\left[\frac{1}{u_k.x}\right]^{(\nu)}\Big|_{x=x_k}$.
We consider the following notations:

$$D_k^\nu = \begin{cases} \frac{1}{u_k.x_k}, \text{ if } \nu = 0 \\ \frac{1}{(\nu-1)!}\left[\frac{1}{u_k.x}\right]^{(\nu)}_{|x=x_k}, \text{ if } \nu > 0 \end{cases} \tag{13}$$

$$S_k^\nu = (-1)^\nu \left(\sum i : 0 \leq i \leq m \wedge i \neq k : \frac{r_i + 1}{(x_k - x_i)^\nu} \right), \tag{14}$$

$$SD_k^j.x = \begin{cases} D_k^0, \text{ if } j = r_k, \\ D_k^0 + \left(\sum \nu : 1 \leq \nu \leq r_k - j : \frac{(x-x_k)^\nu}{\nu} {D_k}^\nu \right), \text{ if } j < r_k. \end{cases} \tag{15}$$

Then the Hermite interpolation polynomials may be computed by the following formula:

$$h_{kj}.x = \frac{(x - x_k)^j}{j\ !} * u_k.x * SD_k^j.x \tag{16}$$

The products $px.k.j = (x - x_k)^j$ are computed at the stage of computation for $R01.s.t$. The fractions $pxf.k.j = (x - x_k)^j/j!$ may be computed simultaneously with $px.k.j = (x - x_k)^j$ at the same stage ($j!$ could be computed with a parallel prefix computation, too).

Using the differentiation rules, the following equality can be proven:

$$D_k^\nu = D_k^0 * S_k^\nu + \left(\sum j : 1 \leq j < \nu : \frac{D_k^j}{j} * S_k^{\nu - j} \right) \tag{17}$$

The sums S_k^ν may be computed by using an algorithm similar to the algorithm used for $u[k]$ computations. In fact, these computations have to be done simultaneously, because the same communications are necessary.

To compute the values $h_{kj}.x$ we need the values D_k^ν and $SD_k^j.x$. We initially know the values D_k^0 and $SD_k^{r_k}.x$. The process (s, t), which is responsible for computing $h_{kj}.x$, is also responsible for computing $SD_k^j.x$ ($\delta_0.k = s \wedge \delta_1^k.j = t$).

The computation of the values $D_k^\nu, j \leq r_k$ is similar to a triangular system solving. The processes on the row $(\delta_0.k, \cdot)$ are used.

In order to balance the work-loading on processes, we choose for distributions δ_1^k to be cyclic distributions, defined by the following relations: $\delta_1^k.j = (r_k - j)\%N$.

The time complexity for computation of the values $h_{kj}.x$ may be evaluated as:

$$T^{R03}.p.m.r = O\left(\frac{m+1}{M}(r+1)(1+\alpha) + \frac{(m+1)(r+1)}{MN}\right) \tag{18}$$

3.3 Time Complexity and Cost

We start again by considering a full-connected interconnection network, and then we analyze the mapping on hypecube-mesh(hypercube) and multi-mesh hypercube networks.

Theorem 3 *The global time complexity of the parallel algorithm for Hermite interpolation, described before is*

$$T.p.m.r = O((r+1)(m+1)^2/(M*N)) \tag{19}$$

Proof: The time complexity results by summing the time complexities for the computations described before: $R1, R2, R01, R02, R03$.

$$\begin{aligned}
& T.p.m.r \\
= & \{\text{summing the partial time complexities}\} \\
& T^{R1+R2}.p.m.r + T^{R01}.p.m.r + T^{R01}.p.m.r + T^{R03}.p.m.r \\
= & \{\text{equations 9, 10, 11, 18}\} \\
& O((r+1)(m+1)^2/(M*N))
\end{aligned}$$

Theorem 4 *The parallel algorithm for Hermite interpolation is efficient from the cost point of view.*

Proof: The computation for $R1$ and $R2$ are tree-like computations and they are cost-efficient. For the computation of $R0$, the sequential algorithm has the

time complexity $T_s.n = O(r(m+1)^2)$.

$$
\begin{aligned}
& C^{R0}.p.m.r \\
= & \ \{\text{equation 2}\} \\
& T^{R0}.p.m.r * p \\
= & \ \{\text{calculus}\} \\
& O((m+1)^2(r+1) + (m+1)(r+1)\log N + (m+1)(r+1)N)
\end{aligned}
$$

So, this computation is cost-optimal.
Together, these computations form a cost-efficient parallel algorithm.

The computations of the values $u_k.x_k$ and S_k^j are the most costing ones, and for them we have used the advantages brought by set-distributions. And, these parallel computations are also cost-optimal. They are independent of the given value x (D_k^ν are also independent of x), and their computation could be separated, when more than one given point has to be considered.

We have chosen cyclic distributions (δ_1^k) on each row, but we didn't impose any conditions on the distribution $\delta_0 : \overline{m+1} \to \overline{M}$. So, the distribution δ_0 can be chosen to assure a balanced distribution not only for the x_k values, but also a balanced loading of the rows: the sums $r.s = (\sum k : \delta_0.k = s : r_k)$ have to be balanced. In this way, the execution time is improved.

The number of processors $p = M * N$ is bounded by the following relations: $M \leq m+1$ and $N \leq r+1$; if $M = m+1$ and $N = r+1$ the time complexity becomes $O(m)$. The time complexity of the parallel algorithm presented in [8] is $O(\log^2 r + \log m)$ but it is obtained by using a very large number of processors – much more than $m^2 * r$; it is not cost-efficient.

We may consider $p = M * M, M \leq m+1$ processors, in which case the computation for $R0.s.t$ may use a distribution and an algorithm similar to those used for Lagrange interpolation. In that case, the obtained time complexity is $O(r + \log m)$, but the processors are not used very efficiently.

3.3.1 Mappings

If we consider a mapping on a hypercube-mesh network $HM_{(m,n)}$, such that $M = 2^m$ and $N = 2^n$, the complexities for tree-like and parallel prefix computations on rows and columns remain the same (they are executed on hypercubes).

For $u[k]$ computations, the communication process described in Figure 3 could be implemented in the following way:

There are two communication stages; first is executed on simple diagonals (with N elements), and the second is formed by communications on columns.

We will use for these stages the fact that each column forms a hypercube H_m, and that can be also seen as a hypercube-mesh $HM_{(q,n)}$ ($M = NQ, Q = 2^q, m = n+q$). This means that we may consider a three dimensional hypercube-mesh $HM_{(q,n,n)}$, and the mapping $(s,t) \rightarrow ((s/2^n, s\%2^n), t)$.

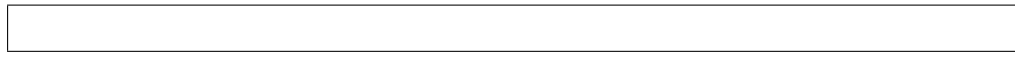

Fig. 4. The first stage of the communication process for $u[k]$ computations when $M = 8$ and $N = 4$. The simple diagonals are connected by lines, and the communications of the values that are local to the processes $(0,t)$ are marked.(For saving space, the matrix is rotated, so the rows are shown vertically.)

In the first stage (Figure 4), the values $(x_k, r_k) : \delta_0.k = s$, (local to the processes $(s,t), 0 \leq t < N$), have to be sent to the processes: $((s+1)\%M, 1)$, ..., $((s+N-1)\%M, N-1)$ (they form a simple diagonal). Each node $((s/2^n, x), t)$ sends its local values to the node $((s/2^n, (x+t)\%2^n), t)$ using the links of the hypercube H_n formed by the nodes $((s/2^n, \cdot), t)$. The time complexity of these comunications is equal to $n = \log_2 N$. On different columns the communications is executed in parallel, and on one column there are M/N communications which can be done in parallel. So, the time complexity of this stage is $N \log_2 N \left(\frac{m+1}{M} 2\alpha\right)$.

In the second stage, we consider communications on columns – Figure 5. On each column t, there are $M/N = Q$ nodes which belongs to the same generalized diagonal $d, 0 \leq d < M$. A broadcast from one of these nodes to the others could be executed with a time complexity equal to $\log_2 Q = q$, since it is a broadcast in the hypercube H_q with the nodes $((\cdot, d\%2^n), t)$. There are N broadcasts like these, which could be executed in parallel on each column, and globally, there are M broadcasts to be done on each column. Hence, the time complexity of the second stage is $\frac{M}{N} \log_2 \frac{M}{N} \left(\frac{m+1}{M} 2\alpha\right)$.

Globally, the time complexity of the communication process is

$$T_c.p.m.r = \left(N \log_2 N + \frac{M}{N} \log_2 \frac{M}{N}\right) \frac{m+1}{M}$$

Based on these, we conclude that the mapping of the program on a hypercube

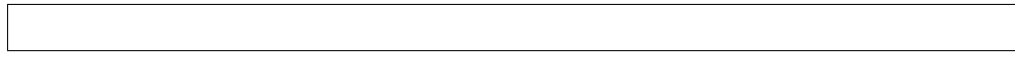

Fig. 5. Column communication in the second stage of the communication process for $u[k]$ computations; $M = 8$ and $N = 4$.

preserves the cost-efficiency.

A multi-mesh hypercube network (N, N, q) (where $M = N * 2^q$) could also be successfully used, if we consider the mapping $(s, t) \rightarrow (s\%N, t, s/N)$. Similar to the case of the hypercube mapping, the communication process for $u[k]$ computations is formed by two stages. In the first stage, the values local to the node $(x, t, s/N)$ are shifted on the column t of the corresponding torus, such that after that operation they arrive to the node $((x+t)\%N, t, s/N)$. The size of each torus is $N \times N$ and the shifting operation is executed in parallel on each column of each torus. The global time complexity of this stage is equal to $N\left(\frac{m+1}{M}2\alpha\right)$. The second stage is similar to that of the hypercube-mesh case, and the time complexity is the same. Hypercube links are used.
For the communications of the first stage, this mapping is better than the hypercube mapping (by a factor equal to $\log_2 N$). But, for tree-like and parallel prefix computations on rows, the term $(\log_2 N)$ in the time complexity would be replaced with $(N/2)$. Also, for tree-like computations on columns the term $(\log_2 M)$ in the time complexity would be replaced with $(N/2 + q)$.

4 Conclusions

The parallel algorithms for Lagrange and Hermite interpolation methods are developed by starting from formal specifications – which assure the correctness, and using set-distributions – which assure the efficiency. The obtained algorithms consider bounded parallelism, and they are both cost-efficient parallel algorithms.

Generally, if each datum appears more than once in the local postconditions, by using set-distributions we may get better results than by using simple distributions. This is the case of polynomial interpolation, and the results emphasize the advantages. Using set-distributions does not always mean adding a new dimension into the organization of the processes – for example, Hermite interpolation algorithm may consider rows with different number of processes (not all equal to N) depending on the values of $r_k, 0 \leq k \leq m$.

The derivation of the algorithms is not ruled by a particular interconnection network. The possible mappings on different networks could be evaluated. The two networks that we have analyzed, hypercube-mesh and multi-mesh hypercube, preserve the cost-efficiency of the algorithms.

It is known that by the time the size of the problem becomes large enough to justify the use of parallelism, polynomial interpolation could break down due to ill-conditioning. Still, as it is argued in [8], some things can be done to avoid this. The improvement in Lagrange interpolation using Chebyshev

rather than equidistant points is well known. Computing a good set of points is a challenging problem, and some work has been already done (e.g. [12,15]).

Appendix A

The program notation is based on Dijkstra's guarded command language [5].
The programs use the following constructs:

- $x := e$ – assignment;
- $S0; S1$ – sequential composition;
- $if\ B0\ \rightarrow\ S0\ []B1 \rightarrow\ S1\ fi$ – alternative construct;
- $do\ B0 \rightarrow S0\ []B1 \rightarrow S1\ od$ – repetition.

An extension to Dijkstra's notation is introduced for the $for\ all$ statement:
$for\ all\ i : i \in Set : S.i\ lla\ rof$ – arbitrary order.

Parallel composition is denoted by:
$par\ q : 0 \leq q < p : S.q\ rap$

And communication is expressed by the following statements
(they are included in parameterized process $C.q$):

- $r\ !\ e$ – the process q sends the value of the expression e to the process r;
- $s\ ?\ x$ – the process q receives from the process s a value, which is assigned to x.

Appendix B

Beside the classic axioms and the inference rules of sequential programming [6], some new specific rules are introduced for parallel composition and for communication.

For the parallel composition we have the following *par*-rule:
Par Rule:

$$\frac{(\forall i : 0 \leq i < p : \{Q.i\}\ S.i\ \{R.i\})}{\{(\forall i : 0 \leq i < p : Q.i)\}\ par\ q : 0 \leq q < p : S.q\ rap\ \{(\forall i : 0 \leq i < p : R.i)\}}$$

For communication, we have:

Input Axiom:

$$r :: \{true\}\ s\ ?\ x\ \{M.x\}$$

?! Rule:

$$\frac{r ::\ \{true\}\ s\ ?\ x\ \{M.x\}}{s :: \{M.x(x := e)\}\ r\ !\ e\ \{M.x(x := e)\}}$$

where $M.x$ is a predicate in terms of local variable x of the process r and its process number, and e is a local expression of process s.

Appendix C

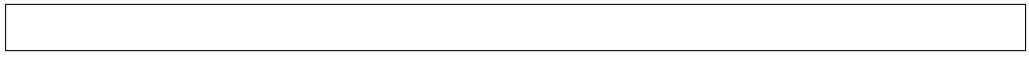

Fig. .1. Hypercube-mesh $HM_{(2,2)}$ equivalent to hypercube H_4.

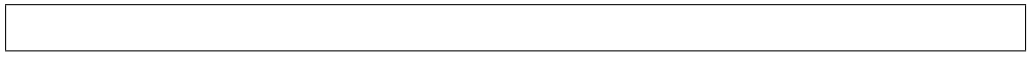

Fig. .2. Multi-mesh Hypercube $(2, 2, 3)$.

Appendix D

$S.s.t ::$
$|[\ ppr[i : 0 \leq i < m] : array\ of\ real;$
$\quad a[i : 0 \leq i < m] : array\ of\ real;$
$\quad for\ all\ i : i \in O.s.t :$
$\quad\quad a[i] := x[i];$
$\quad\quad ppr[i] := 1$
$\quad lla\ rof$
$\quad \{(\forall i : i \in O.s.t : a[i] = x[i])\}$
$\quad k := 0; \quad \{P1.q(k := 0)\}$
$\quad do\ (k \neq m) \rightarrow$
$\quad\ RestoreP1.1.s.t\ \{P1.1.s.t(k := k + M)\}$
$\quad\ ; k := k + M\ \{P1.s.t(k := k + M)\}$
$\quad od$
$]|$

$RestoreP1.1.s.t ::$
$|[$
$\quad C0.s.t\ \{a[s + k] = x[s + k]\}$
$\quad ; S0.s.t$
$\quad \{ppr[s + k] = (\prod i :$
$\quad\quad\quad i \in O.s.t \wedge i \neq s + k :$
$\quad\quad\quad\quad a[s + k] - a[i])\}$
$]|$

$C0.s.t ::$
$|[$
$\quad if(t = 0) \rightarrow$
$\quad\quad par\ v : 0 < v < M :$
$\quad\quad\quad (s, v)!a[k + s]$
$\quad\quad rap$
$\quad [](t \neq 0) \rightarrow$
$\quad\quad (s, 0)?a[k + s]$
$\quad fi$
$]|$

$S0.s.t ::$
$|[$
$\quad ppr[s + k] := 1;$
$\quad\ for\ all\ j : j \in O.s.t :$
$\quad\ if\ (j \neq s + k) \rightarrow$
$\quad\quad ppr[s + k] := ppr[s + k] * (a[s + k] - a[j])$
$\quad\ fi$
$\quad lla\ rof$
$]|$

References

[1] G.H. Atwood, Parallel Lagrangian Interpolation, in: Proc. 1988 Intl. Conf. Parallel Processing (Aug 1988), D. H. Bailey Ed., pp. 120-123.

[2] P. Capello, E. Gallopoulos, C.K. Koc, Systolic computation of interpolation polynomials, Computing, 45(2), (1990) pp.95-117.

[3] G. Coman, Numerical Analysis, Editura Libris, 1995 (in Romanian).

[4] D. Das, M. De, B.P. Sinha, A New Network Topology with Multiple Meshes, IEEE Transactions on Computers, vol. 48(5), (1999), pp. 536-551.

[5] E.W. Dijkstra, A Discipline of Programming, Prentice-Hall, Englewood Cliffs, NJ, 1976.

[6] C.A.R. Hoare, An Axiomatic Basis for Computer Programming, Communications of the ACM, 12(10):576-580, (1969).

[7] O. Egecioglu, E. Gallopoulos, C. K. Koc, Parallel Hermite Interpolation: An algebraic approach, Computing, 42 (1989), pp. 291-307.

[8] O. Egecioglu, E. Gallopoulos, C. K. Koc, Fast Computation of Divided Differences and Parallel Hermite Interpolation, Journal of Complexity, 5 (1989), pp. 417-437.

[9] O. Egecioglu, E. Gallopoulos, C. K. Koc, A parallel method for fast and practical high-order Newton interpolation. BIT, 30(2):268-288, 1990.

[10] O. Egecioglu, C. K. Koc, Parallel Prefix Computation with Few Processors, Computers & Mathematics with Applications, vol. 24, 1992, pp. 77-84.

[11] B. Goertzel, Lagrange Interpolation on a Processor Tree with Ring Connections, J. Parallel Distrib. Comput. 22(2): 321-323, (1994).

[12] B. Fischer, L. Reichel, Newton interpolation in Fejer and Chebyshev points, Mathematics of Computation, Vol. 53, No. 187 (Jul., 1989), pp. 265-278.

[13] J. JaJa, An Introduction to Parallel Algorithms, Addison Wesley Reading, MA, 1992.

[14] P.K. Jana, Multi-mesh of trees with its parallel algorithms, Journal of Systems Architecture, 50(4) (2004), pp. 193-206.

[15] T.A. Kilgore, A characterization of the Lagrange interpolation projection with minimal Chebyshev norm, J. Approx. Theory 24 (1978), 273-288.

[16] A. Louri, H. Sung, An optical multi-mesh hypercube: a scalable optical interconnection network for massively parallel computing, J. Lightwave. Technol., vol. 12, pp. 704-716, Apr. 1994.

[17] L.D. Loyens, A Design Method for Parallel Programs, Ph.D. Thesis, Technische Universiteit Eindhoven, 1992.

[18] C. Morgan, Programming from Specifications, Prentice Hall, 1990.

[19] V. Niculescu, On Data Distributions in the Construction of Parallel Programs, The Journal of Supercomputing, Kluwer Academic Publishers, 29(1): 5-25, 2004.

[20] H. Sarbaz-Azad, M. Ould-Khaoua, L.M. Mackenzie, A Parallel Algorithm for Lagrange Interpolation on k-ary n-Cubes, ACPC (1999), 85-95.

[21] H. Sarbaz-Azad, M. Ould-Khaoua, L.M. Mackenzie, S.G Akl, Parallel Lagrange Interpolation on the Star Graph, in: Proc. 14th IPDPS Mexico, 2000, pp. 777-782.

[22] D.B. Skillicorn, D. Talia, Models and Languages for Parallel Computation, ACM Computer Surveys, 30(2) (1998) pg.123-136.