PARALLEL MULTIPLICATION OF A VECTOR BY A KRONECKER TENSOR PRODUCT OF MATRICES

CLAUDE TADONKI * AND BERNARD PHILIPPE †

Abstract. Different parallel algorithms are designed and evaluated for computing the multiplication of a vector by a Kronecker tensor product of elementary matrices. The algorithms are based on an analytic computation model together with some algebraic properties of the Kronecker multiplication. From that theoretical study, two algorithms are proposed which differ on the volume of floating-point operations and communication they involve. A special study of the data and computing distribution is proposed depending on the dimensions of the elementary matrices. Experimental results show the efficiency of the approach.

Key words. Kronecker product, matrix-vector product, parallel algorithm, recurrent equation, scheduling, data parallel algorithm, communication

1. Introduction. The use of *Stochastic Automata Networks* (SAN) is becoming increasingly important in performance modelling issues related to parallel and distributed computer systems [15]. As such models become increasingly complex, so also does the complexity of the modelling process. Various methods have been developed for solving *Markov models* [19, 14, 13, 17, 18].

A particular aspect of these models is that, their transition matrix called *descriptor* is not explicitly given. In fact, it is represented by a number of much smaller matrices, one for each of the stochastic automata that constitute the system, and from these, all relevant information may be determined without explicitly forming the global matrix. Although the fact that particular algorithms have to be developed for this context, a considerable saving in memory is obtained by storing the matrix in this fashion. Specifically, we consider the problem of performing a matrix-vector multiplication when the matrix is stored as a compact SAN descriptor, since this is a fundamental step for most of the iterative methods and is by far, the major cost per iteration [4, 20].

For this problem which is the bulk of this paper, the basic operation on matrices is the *tensor product* that is a well-known algebraic operation [1, 14]. For the applications of this operation and how to proceed with it, the reader may refer to [5, 6, 7, 10]. Particularly, we shall focus on the parallel issues for this computation in the context of distributed memory parallel machines. An effort has to be done to develop efficient and portable algorithms for this type of architectures [23, 24, 16]. Now, there are available languages and compilers for building such programs and making them effective [25, 11].

It is well-known that when the elementary matrices are given, the cost of performing a vector-descriptor multiply is given by

$$\rho_N = \prod_{i=1}^{N} n_i \times \sum_{i=1}^{N} n_i$$

where n_i is the number of states in the i^{th} automaton and N is the number of automata in the network. Some effort has already been done for the parallelization of this

*University of Yaoundé I, Yaoundé, Cameroon (cmtado@uycdc.uninet.cm), supported by the French agency "Aire Developpement" and by an Inria/NSF agreement.

†INRIA/IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France (philippe@irisa.fr).

computation and can be found in [22]. The reader is also advised to refer to [8, 21]. At this level, let us point out that, as the matrices may be sparse, an algorithm which takes into account the presence of zero entries would be preferable.

Our paper is organized as follows. In the second section, we provide a number of basic properties of the Kronecker product. In section 3, we present the position of the problem and its computational complexity. A polyhedral model of our computational space is proposed in section 4 followed by a formal model based on recurrence equations. An algebraic interpretation is proposed in section 5. From that analysis, various scheduling are derived for sequential computation in section 6 and for parallel computations in the following sections. On conclusion, numerical tests illustrate the behaviour of the algorithms.

2. The Kronecker multiplication. DEFINITION 2.1. *(Kronecker product). If $A \in R^{n_A \times m_A}$ and $B \in R^{n_B \times m_B}$ then the Kronecker product $C = A \otimes B$ that belongs to $R^{n_A n_B \times m_A m_B}$ is defined by the following block structure :*

$$C = (c_{ij}) \quad where \quad c_{ij} = a_{ij}B \in R^{n_B \times m_B}$$

This multiplication is also called tensor product.

EXAMPLE 2.1. *Let us consider the following matrices ;*

$$\mathcal{A} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \quad \mathcal{B} = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$

The *Kronecker product* $C = A \otimes B$ is given by

$$\mathcal{C} = \left(\begin{array}{cc|cc|cc} a_{11}b_{11} & a_{11}b_{12} & a_{12}b_{11} & a_{12}b_{12} & a_{13}b_{11} & a_{13}b_{12} \\ a_{11}b_{21} & a_{11}b_{22} & a_{12}b_{21} & a_{12}b_{22} & a_{13}b_{21} & a_{13}b_{22} \\ \hline a_{21}b_{11} & a_{21}b_{12} & a_{22}b_{11} & a_{22}b_{12} & a_{23}b_{11} & a_{23}b_{12} \\ a_{21}b_{21} & a_{21}b_{22} & a_{22}b_{21} & a_{22}b_{22} & a_{23}b_{21} & a_{23}b_{22} \\ \hline a_{31}b_{11} & a_{31}b_{12} & a_{32}b_{11} & a_{32}b_{12} & a_{33}b_{11} & a_{33}b_{12} \\ a_{31}b_{21} & a_{31}b_{22} & a_{32}b_{21} & a_{32}b_{22} & a_{33}b_{21} & a_{33}b_{22} \end{array}\right)$$

If I_n denotes the identity matrix of order n then for the four square matrices A, B, C, D of compatible orders n_A, n_B, n_C, n_D respectively, we have the following properties.

Basic properties

- Associativity

$A \otimes (B \otimes C) = (A \otimes B) \otimes C$.

- Distributivity over (ordinary matrix) addition

$(A + B) \otimes (C + D) = A \otimes C + B \otimes C + A \otimes D + B \otimes D$.

- Compatibility with (ordinary matrix) multiplication (*Grouping factor*)

$(A \times B) \otimes (C \times D) = (A \otimes C) \times (B \otimes D)$

- Compatibility with (ordinary matrix) inversion

$(A \otimes B)^{-1} = A^{-1} \otimes B^{-1}$

• Compatibility with (ordinary matrix) transposition
$(A \otimes B)^T = A^T \otimes B^T$
• *Factorization* (direct consequence of the *Grouping factor* property)
$(A \otimes B) = (A \otimes I_{n_B}) \times (I_{n_A} \otimes B) = (I_{n_A} \otimes B) \times (A \otimes I_{n_B})$

The associativity of the *Kronecker product* implies that one can naturally define the Kronecker product of N matrices $A^{(i)}$ of size $n_i \times m_i$, $i = 1, ..., N$, denoted by $\otimes_{i=1}^{N} A^{(i)}$ which is a matrix of size $\prod_{i=1}^{N} n_i \times \prod_{i=1}^{N} m_i$.

From the property of *factorization*, it appears that for square matrices $A^{(i)}$ of order n_i, $i = 1, ..., N$, we have $\otimes_{i=1}^{N} A^{(i)} = \prod_{s=1}^{N} (I_{n_1} \otimes ... \otimes I_{n_{s-1}} \otimes A^{(s)} \otimes I_{n_{s+1}} \otimes ... \otimes I_{n_N})$ in which the regular matrix multiplication denoted by $\prod$ is commutative for this special form of the factors.

The operator $\otimes$ is not commutative. However, under the use of permutation matrices, a pseudo-commutativity could be obtained as we will soon describe. For more details, the reader can refer to [1]. Let us recall that for a given *permutation* σ of the set of integer $\{1, 2, ..., n\}$, the corresponding permutation matrix P_σ is defined by $P_\sigma = (e_{\sigma(1)}, ..., e_{\sigma(n)})$ where e_i is the i^{th} column vector of the canonical basis.

In our context, we shall consider a particular *permutation* defined as follows.

DEFINITION 2.2. *(Perfect Shuffle) Let L be an integer and p, q two of its divisors such that $L = pq$. We define the map $\sigma^{(p,q)}$ called Perfect Shuffle by*

$$\sigma^{(p,q)} : \{1, ..., L\} \longrightarrow \{1, ..., L\}$$
$$a = (i-1) \times p + j \longmapsto b = (j-1) \times q + i \text{ with } 1 \leq j \leq p$$

It is obvious that we have $\sigma^{(p,q)} \circ \sigma^{(q,p)} = \varepsilon$ where ε is the identity permutation which means that $\sigma^{(q,p)}$ is the inverse permutation of $\sigma^{(p,q)}$. For a given *Perfect Shuffle* $\sigma^{(p,q)}$, the corresponding permutation matrix will be denoted by $S^{(q,p)}$ and it is clear that for $x \in R^{1 \times L}$, the vector y defined by $y = xS^{(q,p)}$ is such that $y_i = x_{\sigma^{(p,q)}(i)}$, $1 \leq i \leq L$. Because a permutation matrix is an orthogonal matrix, it is also clear that $(S^{(p,q)})^{-1} = S^{(q,p)} = (S^{(p,q)})^T$. Hence, the following result holds :

LEMMA 2.3. *If A_p is a square matrix of order p and I_q the identity matrix of order q, then we have*

$$S^{(p,q)}(A_p \otimes I_q)S^{(q,p)} = I_q \otimes A_p$$

Proof. Let $i, j \in \{1, ..., pq\}$ with $i = (s-1) \times p + \alpha$ and $j = (t-1) \times p + \beta$, $1 \leq \alpha, \beta \leq p$ ($i.e$ $1 \leq s, t \leq q$). We have

$$\begin{aligned}
[S^{(p,q)}(A_p \otimes I_q)S^{(q,p)}]_{ij} &= e_i^T [S^{(p,q)}(A_p \otimes I_q)S^{(q,p)}] e_j \\
&= (S^{(q,p)} e_i)^T (A_p \otimes I_q)(S^{(q,p)} e_j) \\
&= e_{\sigma^{(p,q)}(i)}^T (A_p \otimes I_q) e_{\sigma^{(p,q)}(j)} \\
&= e_{(\alpha-1)\times q+s}^T (A_p \otimes I_q) e_{(\beta-1)\times q+t} \\
&= a_{\alpha\beta}\delta_{st} \\
&= \delta_{st} a_{\alpha\beta} \\
&= [I_q \otimes A_p]_{ij}
\end{aligned}$$

□

From that result, we obtain the following proposition :

PROPOSITION 2.4. *If A_p and B_q are square matrices of order p and q respectively, then we have*

$$A_p \otimes B_q = S^{(q,p)}(B_q \otimes A_p)S^{(p,q)}$$

Proof. From Lemma 2.3 and the *grouping factor* property we have

$$\begin{aligned} A_p \otimes B_q &= (A_p \otimes I_q)(I_p \otimes B_q) \\ &= [S^{(q,p)}(I_q \otimes A_p)S^{(p,q)}][S^{(q,p)}(B_q \otimes I_p)S^{(p,q)}] \\ &= S^{(q,p)}(I_q \otimes A_p)[S^{(p,q)}S^{(q,p)}](B_q \otimes I_p)S^{(p,q)} \\ &= S^{(q,p)}(I_q \otimes A_p)(B_q \otimes I_p)S^{(p,q)} \text{ since } S^{(p,q)}S^{(q,p)} = I_{pq} \\ &= S^{(q,p)}(B_q \otimes A_p)S^{(p,q)} \end{aligned}$$

□

Let us now analyse the main problem and its complexity .

3. Complexity of the vector matrix multiplication. Given N square matrices $A^{(1)}, A^{(2)}, ..., A^{(N)}$ of order $n_1, n_2, ..., n_N$ respectively and a vector $x \in R^{1\times L}$ where $L = \prod_{i=1}^{N} n_i$, our concern is the computation of

$$z = x \otimes_{i=1}^{N} A^{(i)}$$

(We consider the left-sided multiplication as it is done in the field of Markov chains. However, this obviously does not reduce the generality of the problem since the opposite case can be obtained by transposing the matrices).

If we intend to first build the matrix $\otimes_{i=1}^{N} A^{(i)}$ and then perform the corresponding vector-matrix product, we will face the problem of its unfeasible size $(\prod_{i=1}^{N} n_i)^2$ that usually requires an unrealistic space-memory. Moreover, this approach leads to a computation containing too many redundant operations although it is highly parallel.

Some recurrence will therefore be considered to avoid computational redundancy and, an efficient data and task partitioning will be determined to lower data communication.

Let us now present the already known sequential complexity of the problem.

THEOREM 3.1. *[Complexity of the vector-matrix product] The multiplication*

$$x \otimes_{i=1}^{N} A^{(i)},$$

where $A^{(i)}$ is a square matrix of order n_i and x is a vector of length $L = \prod_{i=1}^{N} n_i$, can be computed using ρ_N multiplications where

$$\rho_N = n_N \times (\rho_{N-1} + \prod_{i=1}^{N} n_i) = (\prod_{i=1}^{N} n_i)(\sum_{i=1}^{N} n_i).$$

The theorem implies that the multiplication must not be performed through the assembly of the matrix $\otimes_{i=1}^{N} A^{(i)}$ since it would end up with $\tilde{\rho}_n = \left(\prod_{i=1}^{N} n_i\right)^2$ multiplications. It is a direct consequence of the sequential computation scheme induced by the following lemma :

LEMMA 3.2. *By assuming that the vector x and the matrices $A^{(i)}$ are correctly defined, the computation*

$$z = x \otimes_{i=1}^{N} A^{(i)}$$

is equivalent to

$$z = \left[\begin{array}{cccc} v_1 \otimes_{i=1}^{N-1} A^{(i)} & v_2 \otimes_{i=1}^{N-1} A^{(i)} & \cdots & v_{n_N} \otimes_{i=1}^{N-1} A^{(i)} \end{array} \right] S^{(m,n_N)}$$

where

$$v_j = \left[\begin{array}{cccc} u_1 A^{(N)}(:,j) & u_2 A^{(N)}(:,j) & \cdots & u_m A^{(N)}(:,j) \end{array} \right] \quad j = 1, ..., n_N$$

with

$$u_k = \left[\begin{array}{ccc} x_{(k-1)n_N+1} & \cdots & x_{kn_N} \end{array} \right] \quad k = 1, ..., m$$

with $m = \prod_{i=1}^{N-1} n_i$.

Proof. We have

$$\begin{aligned}
z &= x \otimes_{i=1}^{N} A^{(i)} \\
&= x((\otimes_{i=1}^{N-1} A^{(i)}) \otimes A^{(N)}) \\
&= x[(I_m \otimes A^{(N)}) \times ((\otimes_{i=1}^{N-1} A^{(i)}) \otimes I_{n_N})] \text{ (grouping factor)} \\
&= [x \times (I_m \otimes A^{(N)})] \times ((\otimes_{i=1}^{N-1} A^{(i)}) \otimes I_{n_N}) \text{ (associativity)} \\
&= [u_1 \times A^{(N)} \; u_2 \times A^{(N)} \; \cdots \; u_m \times A^{(N)}] \times ((\otimes_{i=1}^{N-1} A^{(i)}) \otimes I_{n_N}) \\
&= [u_1 \times A^{(N)} \; u_2 \times A^{(N)} \; \cdots \; u_m \times A^{(N)}] \times S^{(n_N,m)}(I_{n_N} \otimes (\otimes_{i=1}^{N-1} A^{(i)}))S^{(m,n_N)} \\
&= [v_1 \; v_2 \; \cdots \; v_{n_N}] \times (I_{n_N} \otimes (\otimes_{i=1}^{N-1} A^{(i)}))S^{(m,n_N)} \\
&= [v_1 \otimes_{i=1}^{N-1} A^{(i)} \; v_2 \otimes_{i=1}^{N-1} A^{(i)} \; \cdots \; v_m \otimes_{i=1}^{N-1} A^{(i)}]S^{(m,n_N)}
\end{aligned}$$

□

This result shows that the computation of $x \otimes_{i=1}^{N} A^{(i)}$ can be viewed as n_N computations $v \otimes_{i=1}^{N-1} A^{(i)}$ which are similar to the original problem with a smaller number of operands. We are now going to present a formal model that will help us to express our multiplication as a succession of regular computation steps.

4. Recurrent equations for the computation. In this section, we focus on an analytic model expressed with the formalism of recurrence equations. Although it is a model widely used in the *systolic* context, it is helpful in any other context as it generally provides finer information about the data dependencies of the given computation. For a more general analysis, this model will be followed in the next section by an algebraic point of view.

Let us begin by modelling our computational space. From the result presented in Lemma 3.2, it appears that for the computation of a given entry of the result vector, we use a well determined column in each matrix $A^{(s)}$. By considering the sequence of columns used during this computation, we obtain an *index sequence* $(i_1, i_2, ..., i_N)$ which means that for the matrix $A^{(s)}$, we have to use the column i_s, for $s = N, N-1, ..., 2, 1$. This decreasing progression is due to the fact that the computation recursively requires the matrices $A^{(N)}, A^{(N-1)}, ..., A^{(1)}$. It can be observed that

this *index sequence*, on its lexicographic ordering, represents the index of the corresponding entry of the resulting vector. We can therefore consider this representation to enumerate the components of the result.

Because matrices are used sequentially, we consider a variable s to index the current matrix. For the computation of one entry of the result, the corresponding column of matrix $A^{(s)}$ is used $\prod_{i=1}^{s-1} n_i$ times, due to the the implicit presence of the partial product $\otimes_{i=1}^{s-1} A^{(i)}$ on its left side, is. Therefore, we consider the variable k to contain the current processing step.

On the end, since the operation performed with each column is the classical *inner product*, we consider a variable t to manage the current accumulation.

At last, we obtain that each of our operands can be expressed by a term of the form $v(i_1, i_2, ..., i_N, s, k, t)$ which represent the corresponding partial result of the computation of the entry $(i_1, i_2, ..., i_N)$ of the result $z = \otimes_{i=1}^{N} A^{(i)}$.

From above, it logically appears that

$$v(i_1, ..., i_{s-1}, i_s, ..., i_N, s, k, t) = v(1, ..., 1, i_s, ..., i_N, s, k, t)$$

whatever the value of $(i_1, ..., i_{s-1})$ is. Consequently, we focus our attention on the terms $v(1, ..., 1, i_s, ..., i_N, s, k, t)$ that can be simply denoted as $v(i_s, ..., i_N, s, k, t)$.

Under this representation of the computational space, the following result completely describes the data and task dependencies in the considered computation.

THEOREM 4.1. *The following system of recurrent relations (4.1-4.4) achieves the computation of* $z = x \otimes_{i=1}^{N} A^{(i)}$ *:*

$$v(N+1, k, 1) = x_k \tag{4.1}$$

$$v(i_s, ..., i_N, s, k, t) = v(i_s, ..., i_N, s, k, t-1) + A^{(s)}(t, i_s) \times v(i_{s+1}, ..., i_N, s+1, (k-1)n_s + t, n_{s+1}) \tag{4.2}$$

$$v(i_s, ..., i_N, s, k, 0) = 0 \tag{4.3}$$

$$z(i_1, ..., i_N) = v(i_1, ..., i_N, 1, 1, n_1) \tag{4.4}$$

where $1 \le i_p \le n_p$ $p = 1, ..., N$, $1 \le s \le N$, $1 \le k \le \prod_{\ell=1}^{s-1} n_\ell$, $1 \le t \le n_s$, $n_{N+1} = 1$.

Proof. By induction on N.

Case $N = 1$ **:** The proposed system implies

- the rule 4.1 gives $v(2, k, 1) = x_k$ where $1 \le k \le n_1$.
- the rule 4.2 gives $v(i_1, 1, 1, t) = v(i_1, 1, 1, t-1) + A^{(1)}(t, i_1) \times v(2, t, 1)$ where $1 \le i_1, t \le n_1$ and $k = 1$.
- from these two observations together with the initializing relation 4.3, we obtain that $v(i_1, 1, 1, n_1) = \sum_{t=1}^{n_1} A^{(1)}(t, i_1) x_t$.

Then, by considering the relation 4.4 which appears as $z(i_1) = v(i_1, 1, 1, n_1)$, we obtain a complete specification of the standard vector-matrix product $z = x \times A^{(1)}$.

Induction : Let us assume that the result is true for $N-1$ and consider the system (4.1-4.4) for N matrices. In order to inherit the required relations from the induction hypothesis, we consider the quantities $w^{(i_N)}(i_s, ..., i_{N-1}, s, k, t)$, $i_N = 1, ..., n_N$ defined in the index domain $1 \le s \le N-1$, $1 \le i_\ell \le n_\ell$, $s \le \ell \le N-1$, $1 \le k \le \prod_{p=1}^{s-1} n_p$, $1 \le t \le n_s$, $n_N = 1$ by the following relations :

$$(i)\ w^{(i_N)}(N, k, 1) = v(i_N, N, k, n_N)$$

$$(ii)\ w^{(i_N)}(i_s, ..., i_{N-1}, s, k, t) = v(i_s, ..., i_N, s, k, t)$$

$$(iii)\ z(i_1, ..., i_N) = w^{(i_N)}(i_1, ..., i_{N-1}).$$

By applying the recursion, we obtain

$$\begin{aligned} w^{(i_N)}(N,k,1) &= v(i_N,N,k,n_N) \\ &= \sum_{t=1}^{n_N} A^{(N)}(t,i_N)v(N{+}1,(k{-}1)n_N{+}t,1) \\ &= \sum_{t=1}^{n_N} A^{(N)}(t,i_N)x_{(k-1)n_N+t} \\ &= u_k A^{(N)}(:,i_N) \end{aligned}$$

where u_k is the *k-th* bloc vector of length n_N in x. Moreover, by applying the rule 4.2 in (ii), we obtain the following relation as well :

$$\begin{aligned} w^{(i_N)}(i_s,...,i_{N-1},s,k,t) = {} & w^{(i_N)}(i_s,...,i_{N-1},s,k,t{-}1) \\ & {+}A^{(s)}(t,i_s) \times w^{(i_N)}(i_{s+1},...,i_{N-1},s{+}1,(k{-}1)n_s{+}t,n_{s+1}) \end{aligned}$$

which is the system for $N{-}1$ matrices. Then, while observing that for any j in $\{1,...,n_N\}$ we have $w^{(j)} = v_j \otimes_{i=1}^{N-1} A^{(i)}$, we obtain from the relation (iii) under lexicographic consideration and from the induction hypothesis that the system (4.1-4.4) computes

$$z = \left[\begin{array}{cccc} v_1 \otimes_{i=1}^{N-1} A^{(i)} & v_2 \otimes_{i=1}^{N-1} A^{(i)} & \cdots & v_{n_N} \otimes_{i=1}^{N-1} A^{(i)} \end{array} \right] S^{(m,n_N)}$$

where

$$v_j = \left[\begin{array}{cccc} u_1 \times A^{(N)}(:,j) & u_2 \times A^{(N)}(:,j) & \cdots & u_m \times A^{(N)}(:,j) \end{array} \right]$$

and

$$u_k = \left[\begin{array}{ccc} x_{(k-1)n_N+1} & \cdots & x_{kn_N} \end{array} \right]$$

is the k-th block-vector of length n_N in the vector x and $m = \prod_{i=1}^{N-1} n_i$. From Lemma 3.2, this is equivalent to $z = x \otimes_{i=1}^{N} A^{(i)}$. □

We now present an algebraic interpretation of the computation.

5. General expression of the multiplication. This section provides an analysis of our problem using *tensor algebra* operations.

Let us begin with the following definition.

DEFINITION 5.1. *(Lexicographical index). For $(i_1,...,i_N)$ a sequence of N integers where $1 \le i_p \le n_p$ for $p = 1,...,N$, we define $pos(i_1,...,i_N)$ as the corresponding rang in the natural lexicographical order.*

Since it is a coding mapping, the corresponding inverse will be denoted by lex. Actually, the notation pos stands for a set of mappings in which each of them is defined by the number of its arguments. The following relations hold :

1. $pos(i_1,...,i_N) = (i_1-1) \times \prod_{p=2}^{N} n_p + ... + (i_s-1) \times \prod_{p=s+1}^{N} n_p + ... + (i_N-1) + 1$

2. $i_s = [(i-1)\mathbf{mod}(\prod_{p=s}^{N} n_p)]\mathbf{div}(\prod_{p=s+1}^{N} n_p)+1$, $1 \leq s \leq N$ *while* $i = pos(i_1, ..., i_N)$

We are now going to express how the *lexicographical index* is related to the canonical basis. Let $(e_i^{n_s})_{i=1,n_s}$ denote the canonical basis of $R^{n_s \times 1}$ and $(e_i^L)_{i=1,L}$ the corresponding one for the whole space $R^{L \times 1}$ where $L = \prod_{i=1}^{N} n_i$.

PROPOSITION 5.2. *For any* $i \in \{1, ..., L\}$, $i = pos(i_1, ..., i_N) \Longleftrightarrow e_i^L = \otimes_{s=1}^{N} e_{i_s}^{n_s}$

Proof. Obvious. □

Because

$$\otimes_{i=1}^{N} A^{(i)} = \prod_{s=1}^{N} (I_{n_1} \otimes ... \otimes I_{n_{s-1}} \otimes A^{(s)} \otimes I_{n_{s+1}} \otimes ... \otimes I_{n_N}),$$

the recursion

$$\begin{cases} V^{(N+1)} = x \\ V^{(s)} = V^{(s+1)}(I_{n_1} \otimes ... \otimes I_{n_{s-1}} \otimes A^{(s)} \otimes I_{n_{s+1}} \otimes ... \otimes I_{n_N}) \quad 1 \leq s \leq N \end{cases}$$

obviously leads in its last step to $V^{(1)} = z$ where $z = \otimes_{i=1}^{N} A^{(i)}$. From the commutativity of the considered multiplication, it can be noticed that any order of the set $\{1, ..., N\}$ can be considered for s since each corresponding recursion provides on its last step the result z. We have chosen the right to left order because it will lead to vary more often the right most index consistently with the lexicographical ordering. This also impacts the data locality in the loops as we shall see.

Let now express the computation of one component of $V^{(s)}$ for a given $s \in \{1, ..., N\}$.

PROPOSITION 5.3. *Let* $i \in \{1, ..., L\}$ *with* $lex(i) = (i_1, ..., i_N)$. *The* i^{th} *component of* $V^{(s)}$ *is then given by*

$$V^{(s)}(i) = \sum_{t=1}^{n_s} A^{(s)}(t, i_s) V^{(s+1)}(j)$$

where $j = pos(i_1, ..., i_{s-1}, t, i_s, ..., i_N)$.

Proof. We may express $I_L = \sum_{j=1}^{L} e_j^L e_j^{L\,T}$ from $e_j^L = \otimes_{s=1}^{N} e_{j_s}^{n_s}$ and $e_j^{L\,T} = \otimes_{s=1}^{N} e_{j_s}^{n_s\,T}$ when $j = pos(j_1, ..., j_N)$. Therefore :

$$\begin{aligned} V^{(s)}(i) &= (V^{(s+1)}[\sum_{j=1}^{L} e_j^L e_j^{L\,T}][I_{n_1} \otimes ... \otimes I_{n_{s-1}} \otimes A^{(s)} \otimes I_{n_{s+1}} \otimes ... \otimes I_{n_N}])e_i^L \\ &= \sum_{j=1}^{L} (V^{(s+1)} e_j^L)[(\otimes_{p=1}^{s-1}(e_{j_p}^{n_p\,T} I_{n_p} e_{i_p}^{n_p})) \otimes (e_t^{n_s\,T} A^{(s)} e_{i_s}^{n_s}) \otimes (\otimes_{p=s+1}^{N}(e_{j_p}^{n_p\,T} I_{n_p} e_{i_p}^{n_p}))] \\ &= \sum_{j=1}^{L} (V^{(s+1)}(j))[(\prod_{p=1}^{s-1} \delta_{i_p j_p})(A^{(s)}(t, i_s))(\prod_{p=s+1}^{N} \delta_{i_p j_p}) \\ &= \sum_{t=1}^{n_s} V^{(s+1)}(j) A^{(s)}(t, i_s) \end{aligned}$$

where $j = pos(i_1, ..., i_{s-1}, t, i_s, ..., i_N)$. □

We are now going to focus on the derivation of some algorithms from the above analysis. But before that, let us check the following statement which unifies the two expressions of the computation. In fact, we have :

$$\begin{aligned} z(i_1, ..., i_N) &= [x \otimes_{s=1}^{N} A^{(s)}] e_i^L \\ &= x \otimes_{s=1}^{N} (A^{(s)} e_{i_s}^{n_s}) \end{aligned}$$

where $i = pos(i_1, ..., i_N)$.
Then it appears that the multiplication $x \otimes_{s=1}^{N} A^{(s)}$ can be performed step by step from $s = N$ to $s = 1$ by computing at step s the partial result

$$\begin{aligned} v(i_s, ..., i_N, s, k, n_s) &= V^{(s)}(i_1, ..., i_{s-1}, i_s, i_{s+1}, ...i_N) \\ &= x[(\otimes_{p=1}^{s-1} e_{i_p}^{n_p})) \otimes (\otimes_{p=s}^{N} A^{(s)} e_{i_p}^{n_p})] \end{aligned}$$

where $k = pos(i_1, ..., i_{s-1})$.
Hence, the two previous expressions of the multiplication provide appropriate recursion laying on this idear.

6. Deriving the algorithm. We begin by focalizing our attention on the operations performed in one step of the recursions provided by the two previous analyses.

6.1. Loop scheduling in the recursion step.

6.1.1. Derivation from recurrence equation. Let us store each value of $v(i_s, ..., i_N, s, k, t)$ at position $pos(k, i_s, ..., i_N)$ in a vector $V^{(s)}$. If $i = pos(i_1, ..., i_s, ..., i_N)$ then we have

$$i_s = [(i-1)\mathbf{div}(\prod_{p=s+1}^{N} n_p)]\mathbf{div}\ (n_s) + 1$$

We also have

$$pos((k-1)n_s + t, i_{s+1}, ..., i_N) = pos(k, i_s, ..., i_N) + (t - i_s) \times \prod_{p=s+1}^{N} n_p$$

Therefore, the recursion step s is obtained from the expressions 4.1-4.4 and is expressed by :

$V^{(N+1)} = x$
For $s \leftarrow N$ **downto** 1 **do**
  $V^{(s)} := 0$
  $r \leftarrow \prod_{p=s+1}^{N} n_p$
  For $i \leftarrow 1$ **to** $\prod_{p=1}^{N} n_p$ **do**
    $j \leftarrow [(i-1)\mathbf{div}(r)]\mathbf{mod}(n_s) + 1$
    For $t \leftarrow 1$ **to** n_s **do**
      $V^{(s)}[i] \leftarrow V^{(s)}[i] + A^{(s)}(t, j) \times V^{(s+1)}[i + (t - j) \times r]$
    End do
  End do
End do
$z = V^{(1)}$

Algo 1 : Optimal sequential algorithm from recurrence equation

6.1.2. Derivation from algebraic expression. From Proposition 5.3, the computation of $V^{(s)}$ from $V^{(s+1)}$ can be naturally expressed as follows:

$V^{(s)} := 0$
For i_1 **in** $\{1, ..., n_1\}$
For i_2 **in** $\{1, ..., n_2\}$
.
.
.
For i_N **in** $\{1, ..., n_N\}$
For t **in** $\{1, ..., n_s\}$
$V^{(s)}(i) \leftarrow V^{(s)}(i) + A^{(s)}(t, j) \times V^{(s+1)}(j)$

Optimal sequential algorithm from algebraic expression

with $i = pos(i_1, i_2, ..., i_N)$ and $j = pos(i_1, ..., i_{s-1}, t, i_{s+1}, ..., i_N)$.

It is important to note that the set of all the embedded loops can be expressed in any order.

6.2. Global algorithm. As we have previously said in Section 5, we consider the ordering $s = N, N-1, ..., 1$ for our recursions. In addition, because the set of the embedded loops of any step can be considered in any order, we obtain the following global algorithm by chosing an appropriate ordering which allows a maximum use of each entry of the current matrix.

$r \leftarrow 1$
$\ell \leftarrow \prod_{i=1}^{N} n_i$
above **For s**$\leftarrow N$ **downto** 1 **do**
$U \leftarrow x$
$V \leftarrow 0$
$m \leftarrow m/n_s$
For $t \leftarrow 1$ **to** n_s **do**
  For $j \leftarrow 1$ **to** n_s **do**
    If $(A^{(s)}(t, j) \neq 0)$ **then**
      For $k \leftarrow 1$ **to** m **do**
        For $\ell \leftarrow 1$ **to** r **do**
          $i = \ell + (j-1) \times r + (k-1) \times r \times n_s$
          $V[i] \leftarrow V[i] + A^{(s)}(t, j) \times U[i + (t - j) \times r]$
        end do
      end do
    end if
  end do
end do
$U \leftarrow V$
$r \leftarrow r * n_s$
end do
$z \leftarrow V$

Algo 2 : Global algorithm for the multiplication

A direct implementation of this algorithm requires two vectors of length $L = \prod_{i=1}^{N} n_i$. In some cases, the memory requirement will be considerable. Let us show how this systematic copy of the whole working vector can be avoided.

6.3. Computation in one vector. For this purpose, let us analyse the previously described allocating memory strategy towards the operations to be performed on it in a given step. Since $pos((k-1)n_s+t, i_{s+1}, ..., i_N) = pos(k, t, i_{s+1}, ..., i_N)$, it appears that a storage conflict will arise when t will be equal to i_s. To solve this conflict, we need a vector U of length n_s to save the values of $v(i_s, i_{s+1}, ..., i_N, s, k, n_s)$ for $i_s = 1, ..., n_s$. As this vector will be used for all the steps, it should be of length $max\{n_i\}_{i=1}^{N}$ and will be the only additionnal working space needed by the algorithm. By the end, because

$$pos(k, i_s, ..., i_N) = [(k-1) \times n_s + (i_s-1)] \times \prod_{\ell=s+1}^{N} n_\ell + pos(i_{s+1}, ..., i_N)$$

the announced algorithm can be expressed as follows :

```
V ← x
ℓ ← ∏_{p=1}^{N} n_p
r ← 1
For s ← N downto 1 do
  ℓ ← ℓ/n_s
  For k ← 1 to ℓ do
    For i ← 1 to r do
      For t ← 1 to n_s do U[t] ← V[((k−1) × n_s+t−1) × r+i]
      For j ← 1 to n_s do
        scal ← 0
        For t ← 1 to n_s do scal ← scal+A^(s)(t, j) × U[t]
        V[((k−1) × n_s+j−1) × r+i] ← scal
      end do
    end do
  end
  r ← r × n_s
end do
z ← V
```

Algo 2 : Computation in one vector

This algorithm computes the multiplication in an efficient time cost and reasonable space memory. We are now ready to study the parallel issue of our problem. Because, in the context of distributed memory machines, communication cost may often cause severe inefficiency, we shall first consider a parallel approach in which there will be no need of communication between processors. This is also justified by the increasing speed of actual processors.

7. Parallel algorithm without data communication. Our purpose here is to study the parallel computation of the multiplication under the constrain of avoiding data communication. The basic principle is that any processor which needs a data should compute it. Therefore, it obviously appears that some redundant computations will occur when some entries must be used by different processors. Let us begin by

presenting an important concept to which we shall refer for the selection of the number of processors to be used.

7.1. The concept of perfect-division. Let us define recursively a perfect-divisor.

DEFINITION 7.1. *An integer $n > 1$ is said to be a perfect-divisor of a sequence of N integers $(n_1, n_2, ..., n_N)$, $n_i > 1$ for all $i \in \{1, 2, ..., N\}$, if one of the following two statements is true.*

1. *n is a divisor of n_1*
2. *$N > 1$ and n is a strict multiple of n_1 and $\frac{n}{n_1}$ is a perfect-divisor of $(n_2, ..., n_N)$*

Obviously, when an integer n is a *perfect-divisor* of $(n_1, n_2, ..., n_N)$, there is a unique s such that n is a strict multiple of $\prod_{\ell=1}^{s-1} n_\ell$ and $\frac{n}{\prod_{\ell=1}^{s-1} n_\ell}$ is a divisor of n_s. It is then said that n is a *perfect-divisor* of $(n_1, n_2, ..., n_N)$ of *rank* s and the value of $q = \frac{\prod_{\ell=1}^{s} n_\ell}{n}$ will be called the *quotient* of the perfect-division of $(n_1, n_2, ..., n_N)$ by n.

EXAMPLE 7.1.

- *$\{2, 4, 8\}$ is the set of perfect-divisors of rank 1 of the sequence $(8, 6, 4, 7)$*
- *$\{16, 24, 48\}$ is the set of perfect-divisors of rank 2 of the sequence $(8, 6, 4, 7)$*
- *6 and 12 are not perfect-divisors of the sequence $(8, 6, 4, 7)$*
- *the quotient of the perfect-division of $(8, 6, 4, 7)$ by 24 is 2.*

LEMMA 7.2. *If $C_s(n_1, n_2, ..., n_N)$ denotes the set of all integers that are perfect-divisors of rank s of $(n_1, n_2, ..., n_N)$, then the set of all perfect-divisors of $(n_1, n_2, ..., n_N)$ is obtained as follows*

$$C(n_1, n_2, ..., n_N) = \bigcup_{s=1}^{N} C_s(n_1, n_2, ..., n_N) = \bigcup_{s=1}^{N} (\prod_{\ell=1}^{s-1} n_\ell) D(n_s)$$

where $D(n_s)$ is the set of the divisors of n_s greater than 1.

Proof. When considering a partition of the set of *perfect-divisors* of $(n_1, n_2, ..., n_N)$ under the *rank* of the *perfect-division*, the result follows from the definition of the *rank* of the *perfect-division* from which we have $C_s(n_1, n_2, ..., n_N) = (\prod_{\ell=1}^{s-1} n_\ell) D(n_s)$. □

EXAMPLE 7.2. *Let us consider the sequence (8, 6, 4, 7) of Example 7.1. By applying the above result, we obtain $D(8) = \{2, 4, 8\}$, $D(6) = \{2, 3, 6\}$, $D(4) = \{2, 4\}$, $D(7) = \{7\}$, and therefore*

$$C(8, 6, 4, 7) = D(8) \cup 8D(6) \cup 48D(4) \cup 192D(7) = \{2, 4, 8, 16, 24, 48, 96, 192, 1344\}.$$

From the above explanations, the following property holds for the perfect-division:

$$C_s(n_1, .., n_s, n_{s+1}, ..., n_N) \subset C_1(\prod_{\ell=1}^{s} n_\ell, n_{s+1}, ..., n_N) \tag{7.1}$$

where $1 \leq s \leq N$.

Let us now turn to the presentation of the main idea of this section.

7.2. Basic scheduling. As previously described, the computation of a given entry $(i_1, ..., i_N)$ of the result vector is performed from step $s = N$ to step $s = 1$ by using for each of them, the corresponding suffix of this index sequence. When looking carefully the formal scheme (4.1-4.4) proposed in Theorem 2, particularly the Relation (4.2), we conclude that at a given step s, the computation of a value referenced

by $(i_s, i_{s+1}, ..., i_N)$ requires the values of step $s+1$ which are referenced by the suffix $(i_{s+1}, ..., i_N)$. Consequently, if a processor owns all the values $v(i_{s+1}, .., i_N, s+1, k, n_{s+1})$ for a given subsequence $(i_{s+1}, .., i_N)$, it can therefore compute all the values $v(i_s, i_{s+1}, .., i_N, s, k, n_s)$ with a complete data autonomy. Starting from this observation, we obtain the following result which also illustrates the importance of the concept of *perfect-division.*

THEOREM 7.3. *The computation of* $z = x \otimes_{i=1}^{N} A^{(i)}$ *can be done in parallel without any communication between processors using any number of processors* $p \in C(n_N, n_{N-1}, ..., n_1)$*, according to the formal scheme (4.1-4.4), with increasing speedup w.r.t. p.*

Proof. Let us first consider the simplest case when p belongs to $C_1(n_N, n_{N-1}, ..., n_1)$ which means that p is a divisor of n_N. We define an *allocating function* which maps the data onto the processors. For the index of a given entry, its provides the index of the owning processor, which consequently is supposed to perform all required operations for the computation of this entry. Hence, the result for this case is achieved by considering the following allocating function :

$$a(i_s, .., i_N, s, k, t) = (i_N - 1)\mathbf{mod}\ (p]\ + 1$$

Let us now consider the case when p belongs to $C_\eta(n_N, n_{N-1}, ..., n_1)$ where $\eta > 1$. Because some processors will require the same entries for their computations, redundant computations will be performed to avoid communications. Therefore, we consider an extension of the allocating function that now addresses a set of indices instead of a single index as previously. Since we have $p = q \prod_{\ell=s_0}^{N} n_p$ where $s_0 = N - \eta + 2$ and $1 < q \leq n_{s_0}$, the set of our processors can be indexed by the sequences $(j, j_{s_0}, .., j_N)$ where $1 \leq j \leq q$ and $1 \leq j_\ell \leq n_\ell$, $\ell = s_0, ..., N$. Considering a given sequence $(i_s, .., i_N)$ where $s_0 \leq s \leq N$, we define the set of sequences $E(i_s, .., i_N)$ as follows :

$$E(i_s, .., i_N) = \{(j, j_{s_0}, ...j_{s-1}, i_s, ..., i_N) \quad \text{where} \quad 1 \leq j \leq q \\ \text{and} \quad 1 \leq j_\ell \leq n_\ell \quad \text{for} \quad \ell = s_0, ..., s-1.\}$$

In other words, $E(i_s, .., i_N)$ denotes the subset of the processors whose lexicographical index contains the sequence $(i_s, .., i_N)$ as a suffix. Then, the appropriate scheduling for this case is obtained by considering the following allocating function

$$a(i_s, .., i_N, s, k, t) = \begin{cases} E(i_s, .., i_N) & \text{if} \quad s_0 \leq s \leq N \\ ((i_{s_0-1} - 1)\mathbf{mod} \quad (q) \ + 1, i_{s_0}, .., i_N) & \text{if} \quad 1 \leq s < s_0 \end{cases}$$

Let us emphasize the need of redundant computations when $s_0 \leq s \leq N$ since more than one processor is concerned by the same entries. We show now that the execution time is a decreasing function w.r.t. p belonging to $C(n_N, n_{N-1}, ..., n_1)$.

It is assumed that in the present context, each processor works with its computational space. The time complexity of our computation on p processors where p is a *perfect-divisor* of *rank* η of $(n_N, n_{N-1}, ..., n_1)$, is given by

$$T(p) = (\sum_{s=s_0}^{N} \prod_{\ell=1}^{s} n_\ell) + (\prod_{\ell=1}^{N} n_\ell \times \sum_{s=1}^{s_0-1} n_s)/p$$

where $s_0 = N - \eta + 2$. The first term of this expression is obtained by reminding that, at a given step s such that $s_0 \leq s \leq N$, a processor referenced by $(j, j_{s_0}, ..., j_{s-1}, i_s, ..., i_N)$ computes all the values $v(i_s, ..., i_N, s, k, t)$ for $k = 1, ..., \prod_{\ell=1}^{s-1}$ and $t = 1, ..., n_s$. For the second term, since there is no redundant computation when $1 \leq s \leq s_0 - 1$, the

result is obtained by considering an equal task distribution of each recursion step. This expression shows that $T(p)$ is a harmonic function of p for a constant value of the *rank* η (*i.e* constant value of s_0). Consequently, T is a decreasing function when restricted to $C_\eta(n_N, n_{N-1}, ..., n_1)$ for a given η. Therefore, to complete this part of the proof, we just have to show that $T(p_1) < T(p_0)$ for $p_0 = \mathbf{max} C_\eta(n_N, n_{N-1}, ..., n_1) = \prod_{\ell=s_0-1}^{N} n_\ell$ and for any p_1 in $C_{\eta+1}(n_N, n_{N-1}, ..., n_1)$ (*i.e* $p_1 > p_0$) . If q is the *quotient* of the *perfect-division* of $(n_N, n_{N-1}, ..., n_1)$ by p_1, it is clear that we have $p_1 = qp_0$. Considering the expression of $T(p)$ and the values of p_0 and p_1, we have

$$T(p_0) = (\sum_{s=s_0}^{N} \prod_{\ell=1}^{s} n_\ell) + (\prod_{\ell=1}^{s_0-2} n_\ell \times \sum_{s=1}^{s_0-1} n_s)$$

and

$$T(p_1) = (\sum_{s=s_0-1}^{N} \prod_{\ell=1}^{s} n_\ell) + (\prod_{\ell=1}^{s_0-2} n_\ell \times \sum_{s=1}^{s_0-2} n_s)/q$$

From the fact that $T(p_1) = T(p_0)$ when consider $q = 1$ in $T(p_1)$, the result is obtained by reminding that $q > 1$.
This ends the proof of the theorem. □

Description of the algorithm when p is a divisor of n_N. Let V_j, $1 \leq j \leq n_N$, denotes the set of all values $v(i_s, .., i_{N-1}, j, s, k, t)$ where $1 \leq i_\ell \leq n_\ell$, $\ell = 1, ..., N-1$, $1 \leq s \leq N$, $1 \leq k \leq \prod_{p=1}^{s-1} n_p$, $1 \leq t \leq n_s$. As the computation of the subsets V_j can be done with complete data independency between themselves, computation without communication can then be done by applying the following distribution. The computation of all values in V_1 is allocated to Processor 1, that of V_2 to processor 2, . . ., that of V_p to processor p, that of V_{p+1} to processor 1, and so on. Formally, a processor k will be in charge of the computation of the subsets V_j such that $k = (j-1)\mathbf{mod}(p) + 1$. Practically, when considering the derived global algorithm, we just have to apply a cyclic distribution of the vectors U and V. For the loops, we distinguish two cases. When $s = N$ we just distribute the loop under j onto the processors in a *cyclic* manner and perform the total execution of the other loops in each processor. By doing that, we ensure that each processor performs the corresponding computation with its owned column(s) of the last matrix. For the other loops ($1 \leq s < N$), a total execution of the loops under t, k and j will be done in each processor and only the loop under ℓ will be distributed in a *cyclic* manner.

Let us emphasize the fact that the case when p is a divisor of n_N can be directly implemented without a particular programming effort and leads to an optimal computation since there is no redundant computation. For the other case, we show now how to proceed to turn back to the previous case.

7.3. Simplified algorithm. Let us consider the multiplication on p processors, where p is a *perfect-divisor* of $(n_N, n_{N-1}, ..., n_1)$ of *rank* η ($\eta > 1$). From the property of the *perfect-division*, p also belongs to $C_1(\prod_{\ell=s_0}^{N} n_\ell, n_{s_0-1}, ..., n_1)$ where $s_0 = N - \eta + 1$. We then turn to the expected case by noticing that $z = x[(\otimes_{i=1}^{s_0-1} A^{(i)}) \otimes (\otimes_{i=s_0}^{N} A^{(i)})]$. In other words, we first compute the partial product $\otimes_{i=s_0}^{N} A^{(i)}$ to obtain a matrix B

$p = 2^\eta$	1	2	4	8	16	32	64	128	256
$\sigma_1(p)$	1	2	4	7.6	13.3	20.6	27.8	33.2	36.5

TABLE 7.1
Grouping factor effect on the speedup

and then consider the problem $z = x[(\otimes_{i=1}^{s_0-1} A^{(i)}) \otimes B]$ which can therefore be computed with the p processors without any need of communication as described above. With this adapted form of our problem, the number of performed multiplications becomes equal to

$$\prod_{\ell=1}^{N} n_\ell \times (\sum_{s=1}^{s_0-1} n_s + \prod_{\ell=s_0}^{N} n_\ell)$$

and, since there is no data communication during this computation in parallel with p processors, the time complexity is

$$\frac{\prod_{\ell=1}^{N} n_\ell \times (\sum_{s=1}^{s_0-1} n_s + \prod_{\ell=s_0}^{N} n_\ell)}{p}$$

The *speedup*, ratio between the sequential time and the time of the corresponding parallel version, is given in this case by

$$\sigma_1 = p\frac{1}{1+\rho} \quad \text{where} \quad \rho = \frac{\prod_{\ell=s_0}^{N} n_\ell - \sum_{\ell=s_0}^{N} n_\ell}{\sum_{\ell=1}^{N} n_\ell}$$

Table 7.1 illustrates the quality of this approach for $N = 20$ and $n_\ell = 2, l = 1, ..., 20$ and $p = 2^\eta$. In this particular case,we will have $\sigma_1 = 2^\eta(\frac{N}{N-\eta+2^{\eta-1}})$ and when assuming that p is constant, it appears that $\lim_{N\to+\infty} \sigma_1 = p$. This result is more general and shows that, on matrices of constant size, the algorithm becomes of optimal works for larger number of matrices. Suitable case also occur for matrices of bigger size, more precisely the lasts.

Anyway, as redundant computations are intentionally performed, care must be taken while using this approach for large values of η as the speedup grows very slowly w.r.t. the number of used processors as it can be seen from Table 7.1. In the particular case when all the matrices are of size 2×2 and because $2 \times 2 = 2 + 2 = 4$, the last two matrices can be replaced by their Kronecker product without introducing any redundancy in the whole computation which therefore allows the use of four processors for an optimal time computation. On another hand, if the number of processors P is not a *perfect-divisor* of $(n_N, n_{N-1}, ..., n_1)$, one could select for the number of processors, the largest value $p \in C(n_N, n_{N-1}, ..., n_1)$ such that $p < P$. In the simplified version of the algorithm, the admissible values of p is extended to the set of all divisors of $\prod_{\ell=s_0}^{N} s_\ell$.

In the next section, we present another way of mapping the formal scheme (4.1-4.4) on a given number of processors $p \in C(n_N, n_{N-1}, ..., n_1)$ without redundant computation but at a price of communication cost.

8. Solution involving communications between processors. Let us consider $p \in C_\eta(n_N, n_{N-1}, ..., n_1)$ where $1 < \eta \leq N$. Our purpose is to map the computational space on the p processors so that the required volume of communication is minimal, assuming that no redundant computation is allowed. For making easier the situation, we assume that $p = \prod_{\ell=s_0}^{N}$ where $s_0 = N-\eta+1$). Let

us address our processors with sequences $(i_{s_0}, i_{s_0+1}, ..., i_N)$ where $1 \leq i_\ell \leq n_\ell$ for $\ell = s_0, ..., N$. Then, at a step s $(s_0 \leq s < N)$, if we consider a cyclic distribution of the values $v(i_s, ..., i_N, k, n_s)$ coded by $(k, i_s, ..., i_N)$, we obtain that each processor $(i_{s_0}, ..., i_{s-1}, i_s, ..., i_N)$ will be in charge of the computational points $(k, i_s, ..., i_N)$ such that $k = pos(j_1, ..., j_{s_0-1}, i_{s_0}, ..., i_{s-1})$ with $1 \leq j_p \leq n_p$, $p = 1, ..., s_0-1$. From the definition of the mapping *pos*, we can observe that, if $k = pos(j_1, ..., j_{s_0-1}, i_{s_0}, ..., i_{s-1})$ then $(k-1)n_s + t = pos(j_1, ..., j_{s_0-1}, i_{s_0}, ..., i_{s-1}, t)$. Hence, from the rule 4.2 of our recurrence equations, it follows that at step s, the computation of the value $v(i_s, ...i_N, s, k, t)$ $(k = pos(j_1, ..., j_{s_0-1}, i_{s_0}, ..., i_{s-1}))$ in Processor
$I = pos(i_{s_0}, ..., i_{s-1}, i_s, i_{s+1}..., i_N)$ requires the value $v(i_{s+1}, ...i_N, s+1, (k-1)n_s + t, n_{s+1})$ owned by Processor $J = pos(i_{s_0}, ..., i_{s-1}, t, i_{s+1}, ..., i_N) = I + (t - i_s) \prod_{\ell=s+1}^{N} n_\ell$. We obtain on the end that, at step s, a processor I sends and receives values from its neighbourhood defined by

$$\Gamma(I) = \{I + (t - T) \times r, \quad t = 1, ..., n_s\}$$

where $r = \prod_{p=s+1}^{N} n_p$ and $T = [(I-1)\mathbf{div}(r)]\mathbf{mod}(n_s) + 1$.
and performs the following operation with collected data (u_t is the vector which is received from $I + (t - T) \times r$)

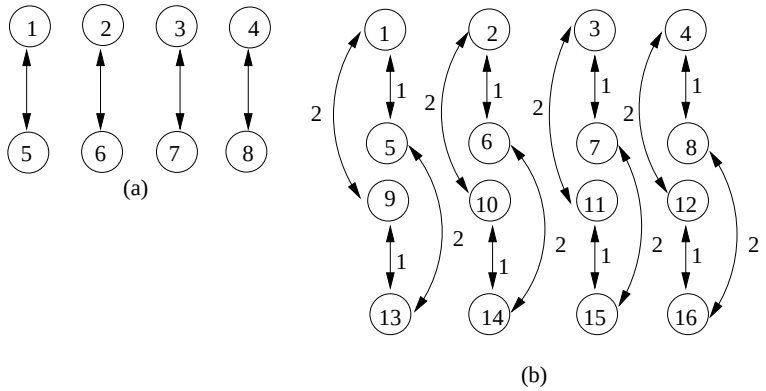
$$v = \sum_{t=1}^{n_{s-1}} A^{(s-1)}(t, T).u_t.$$

Let us notice at this point that before any communication, the vector u_T is already available in Processor I. This last aspect, together with the associativity and the commutativity of the operation $\sum$, introduces the possibility of doing floating point operations while performing communications. On another hand, as computation to be done is composed of a succession of regular operations of linear algebra, an opportunity is offered for using routines from the set BLAS [3].
Moreover, it appears from the above description that the communication graph is *symmetric* and *reflexive*. The reflexivity means that any processor uses values previously computed by itself. The *symmetry* implies a *send/receive* instruction can be used. Figure 8.1 displays this graph for a 2×2 matrices when using respectively 8 and 16 processors. In the case when there is more than one communication time, the edges will be valuated by the rank of the corresponding communication step.

Let us now describe the entire process. During the steps including communications between processors $(s_0 \leq s < N)$, computation is performed as already described. For the other steps $(1 \leq s < N-\eta)$, because the behavior is the same as in the case where $1 \leq s \leq N-1$ in the first parallel issue, we apply the same organization, means that a total execution of the loops under t, k and j will be performed in each processor and the loop under ℓ will be distributed in a *cyclic* manner. We end by noticing that for the case when the quotient q of the $perfect - division$ is lower than the size of the corresponding matrix, a mixed recursion should be considered.

Further details will be given in the implementation subsection. Let us now turn to the efficiency of this approach.

8.1. Timing model. Let us consider our computation when using p processors where $p \in C_\eta(n_N, n_{N-1}, ..., n_1)$ with $1 < \eta \leq N$. During the first $(\eta-1)^{th}$ steps, communications will occur while going from a given step to another. As previously



(a) Case for 8 processors. (b) Case for 16 processors.

FIG. 8.1. *Graph of communication for* 2×2 *matrices.*

$p = 2^\eta$	1	2	4	8	16	32	64	128	256
$\sigma_1(p)$	1	2	4	7.8	15.5	30.6	60.3	119	234.6

TABLE 8.1
Theoretical speedups for the algorithm involving communications

proved, at step s each processor I sends a vector of length $\frac{\prod_{\ell=1}^{N} n_\ell}{p}$ to each of the $n_{s-1}-1$ processors in $\Gamma(I)$. By considering the ideal case for physical communication network, it appears that the communication cost is given by

$$\sum_{s=s_0}^{N} (n_{s-1}-1)(\beta + \frac{\prod_{\ell=1}^{N} n_\ell}{p}\tau_c)$$

where $s_0 = N-\eta+2$ and the tranmission time for a vector of length w is modelized by $\beta + \tau_c w$. Therefore, the cost of the whole computation is equal to

$$\frac{\prod_{\ell=1}^{N} n_\ell \times \sum_{\ell=1}^{N} n_\ell}{p}\tau_a + (\beta + \frac{\prod_{\ell=1}^{N} n_\ell}{p}\tau_c) \times \sum_{s=s_0}^{N} (n_{s-1}-1)$$

where τ_a is the elementary time for a floating point operation. The corresponding *speedup* is then given by

$$\sigma_2 = p\frac{1}{1 + (\frac{\beta p}{L\tau_a} + \frac{\tau_c}{\tau_a})\frac{A}{B}}$$

where $L = \prod_{\ell=1}^{N} n_\ell$, $A = \sum_{s=s_0}^{N} (n_{s-1}-1)$ and $B = \sum_{\ell=1}^{N} n_\ell$. Table 8.1 illustrates the quality of this approach for $N = 20$ and $n_\ell = 2, l = 1, ..., 20$ and $p = 2^\eta$. In this particular case, we consider $L = 2^N$, $A = \eta-2$ and $B = 2N$. The values of the parameter are those of the machine PARAGON for which $\beta = 40 \times 10^{-6}s$, $\tau_c = 0.03 \times 10^{-6}s$ and $\tau_a = \frac{1}{75} \times 10^{-6}s$.

Before turning to the implementation, we must point out that, since the operations to be performed during the steps including communications are regular, the measured speedup could be better than expected, and it would be even better when using specialized BLAS routines.

8.2. Implementation. Let us define some routines that are used in the next algorithm.

- **send**$(tag, u, length, idest)$ for sending a vector u of lentgh $length$ to the processor $idest$ with the associated tag tag.
- **recv**$(tag, v, length)$ for receiving in the vector v, a vector of length $length$ with the corresponding tag tag.
- **gather**$(z, v, mode, length)$ for gathering communications and rebuilding the vector z, according to the kind of its distribution to the processors that is specified by $mode$ which can be $block$ or $cyclic$. The vector v, of size $length$, is the part owned by each processor.
- **perfect**(p, n, N) returns the rank of the $perfect{-}division$ of $(n(N), ..., n(1))$ by p where n is an integer vector of length N. When it is not a perfect-divisor, the rank is set to 0.
- **numproc**() and **mynode**() return respectively the number of processors to be used and the rank of a processor between 1 and P=**numproc**().

Every processor needs three vectors U, V and W of length $\frac{\prod_{\ell=1}^{N} n_\ell}{P}$ where P is the number of processors. Since a cyclic distribution of the whole vectors U and V and of the loops is done, the following relation, between the absolute index ia and its relative value ir in the processor $rank$, holds

$$ia = rank + (ir{-}1) \times P.$$

Because our algorithm follows a SPMD model, the instructions are run by every processor. The code for our computation is displayed in Table 8.2.

Let us now state some remarks about this algorithm. The communications are performed selectively in order to avoid a deadlock due to the symmetry of the communication graph and also to avoid an accumulation of messages sent to one processor by its neighbours. In addition, also due to the symmetric aspect of the communications, the use of a simultaneous *send/receive* is possible. In the particular case of 2×2 matrices, the use of the additional vector W can be omitted as well as the communicating loops, due to the binary value of T. On another hand, during the steps including communication, performed operations provide a regular access to the right hand side vectors which therefore increases the speed.

Let us now turn to the presentation of the performance results obtained in our tests.

9. Performance results and comments.

9.1. Hardware.

9.1.1. NEC CENJU3. CENJU-3 [12] is a distributed memory parallel machine. Up to 256 processing elements (16 in our case), each having a MIPS R4400 running at 75 MHZ clock and local memory of 32 MB, are connected by multistage network based on 4×4 switches. The point-to-point throughput of the network is 40 Mbytes/sec. A host workstation is connected to the above network, and provides interfaces to external storage and LAN to tasks running on processor elements. Each processor is accompanied by a network interface hardware (NIF), which is capable of DMA data transfer to and from the network.

DenEn is a parallel operating system for Cenju-3, composed of CMU's Mach microkernel and a set of small servers, and supports multi-user, multitask, and multithread execution. Programming interfaces provided for users programs include:

P =**numproc**()
η = **perfect**(P, n, N)
I =**mynode**()
$L = \prod_{\ell=1}^{N} n_\ell$
$m = L$
$\ell = \frac{L}{P}$
$n = n(N)$
$j \leftarrow (I{-}1)$**mod**$(n) + 1$
For $t \leftarrow 1$ **to** n **do**
For $ir \leftarrow 1$ **to** ℓ **do**
$V[ir] \leftarrow V[ir] + A^{(N)}(t, j) \times X[(ir-1) \times p + I + (t-j)]$
end do
endo do
$m = \frac{m}{n}$
$r = n$
For $s \leftarrow N{-}1$ **down to** $N{-}\eta{+}1$ **do**
$n = n(s)$
$m = \frac{m}{n}$
$W = V$
$V = 0$
$K =$ **div**$((I{-}1), r) + 1$
$T =$ **mod**$(K{-}1, n) + 1$
For $t \leftarrow 1$ **to** n **do**
$J = I + (j - T) \times r$
if $(t \neq T)$ **then**
If $(T < t)$ **then**
send(t, W, ℓ, J)
recv(t, U, ℓ)
else
recv(t, U, ℓ)
send(t, W, ℓ, J)
end if
If $(A^{(s)}(t, T) \neq 0)$ **then** $V \leftarrow V + A^{(s)}(t, T).U$
else
If $(A^{(s)}(t, T) \neq 0)$ **then** $V \leftarrow V + A^{(s)}(t, T).W$
end if
end do
$r = r \times n$
end do
$U = V$
For $s \leftarrow N{-}\eta$ **down to** 1 **do**
$n = n(s)$
$m = \frac{m}{n}$
$V = 0$
For $t \leftarrow 1$ **to** n_s **do**
For $j \leftarrow 1$ **to** n_s **do**
If $(A^{(s)}(t, j) \neq 0)$ **then**
For $k \leftarrow 1$ **to** m **do**
For $\ell \leftarrow 1$ **to** $\frac{r}{p}$ **do**
$i = \ell + (j{-}1) \times \frac{r}{p} + (k{-}1) \times \frac{r}{p} \times n_s$
$V[i] \leftarrow V[i] + A^{(s)}(t, j) \times U[i + (t-j) \times \frac{r}{p}]$
end do
end do
end if
end do
end do

$U = V$
$r = r \times n$
endo do
gather$(Z, V, cyclic, \ell)$
END

TABLE 8.2
Parallel algorithm involving communications

Mach kernel calls, C threads, most UNIX file I/O system calls, and MPI. MPI implementation on CENJU-3 is called MPI/DE and the version 0.9 has all MPI functions except error handlings ; it reaches 40 microseconds for the minimum latency and 20 Mbytes/sec for the maximum throughput. UNIX system calls are redirected to and processed by the host workstation.

9.1.2. INTEL PARAGON. PARAGON is a distributed memory parallel machine. It consists of a set of *nodes* connected by a high-speed internal network. Each node contains two processors $i860^{TM}$: one processor is specialized for the computation and it runs at 75 MHZ clock and is equipped of a local memory of 16 MB, and one processor specialized for communication. Care must be taken for the local memory of each processor as 8 to 9 MB is used by the system OSF. The nodes of the machine are divided into a *service partition* (8 nodes) for the system and a *compute partition* (56 nodes) for running parallel programs.

9.2. Software.

9.2.1. HPF. High Performance Fortran (HPF) is a data parallel language that includes the following features:

- It mainly targets parallel processing specifications on distributed-memory parallel processing computers.
- It is based on data parallel considerations. That is, the user usually needs only to specify in the program how to distribute the data between the local memories, and loops and statements are automatically partitioned concurrently according to that data distribution. The user may also specify a particular distribution of loops and statements to the processors.
- The user specifies distribution of data that is performed in two steps using the ALIGN and DISTRIBUTE directives :
  - ALIGN directive : Maps arrays accordingly to a reference array.
  - DISTRIBUTE directive : Distributes each element of the reference array in the memory of an abstract set of processors.
  - PROCESSORS directive : Specifies the topology of the abstract set of processors.

HPF becomes a simple way to develop distributed parallel programs when they involve regularly structured data.

9.2.2. MPI and NX communication libraries. NX is a message passing library available on the Intel PARAGON machine which we used. It provides routines for sending and receiving messages from a given node to another. Each node is referenced by a number from 0 to $p-1$ where p is the number of processors to be used. NX can be viewed as a library of the similar to MPI but the later makes program portable since the MPI library is now implemented on most of the architectures.

9.3. Performance results. We present here the performance results of our programs. We have considered the case of twenty 2×2 matrices. In the first case (see Table 9.1), the program has been developped using HPF compiler on CENJU-3 and in the second case (see Table 9.2 and Figure 9.1), we have used the Fortran 77 compiler with the NX library on PARAGON.

Number of processors	1	2	4	8	16
Measured time (s)	20	11	5.4	2.8	1.4
Measured speedup	1	1.8	3.7	7.1	14.2

TABLE 9.1
Timings (HPF - CENJU-3 - no communication)

Number of processors	1	2	4	8	16	32
Measured time (s) (1)	18.75	10.25	5.8	3.0	1.6	1.1
Measured speedup (1)	1	1.8	3.2	6.2	11.2	16.7
Measured time (s) (2)	18.75	10.25	5.8	2.8	1.4	0.75
Measured speedup (2)	1	1.8	3.2	6.7	13.4	25

(1) refers to the algorithm without communication
(2) refers to the algorithm with communications
TABLE 9.2
Timings (NX - PARAGON)

$N = 20$
$n_i = 2, i = 1, ..., 20.$
Vector length = 1,048,576
Number of multiplications = 41,943,040 (when no redundancy)

Characteristics of the test problem.

9.4. Comments. On CENJU-3, the obtained speedups are nearly those expected since the program do not involve communications ; they are even slightly better, possibly because the speed per processor increases with the redundancy. On Paragon, the efficiency of the version without communication decreases when the number of processors increases. This happens when redundant computations occur. In that situation, the second version becomes more efficient since the communications are of low cost. If the program was run on a network of workstations, the conclusion would be different since the cost of the communications would be much higher.

For the two programs, the user only specifies the number of processors to be used without any other change inside the considered program.

10. Conclusion. Our purpose in this paper was to provide some parallel efficient algorithms for computing the multiplication of vector by a Kronecker product of matrices. A formal model computation has first been proposed in order to obtain a deep exploration of the parallel computation issues. Based on this model, two mains directions were explored. In the first direction, a computation without any communication between processors, it has been shown that it may be necessary to group some small matrices to form a larger one according to the number of available processors. It is clear that care must be taken for the set of matrices to be grouped not to be too large because it introduces redundant computations. However, acceptable speedup can be achieved and the implementation of this approach is quite simple.

The second approach requires communications instead of redundant computations. A connecting graph with nice properties as reflexivity, symmetry and low number of neighbours, has been obtained from the proposed scheduling and nearly optimal behaviour has been proved and practically observed. A special version has been proposed when the elementary matrices are sparse so that each zero entry is

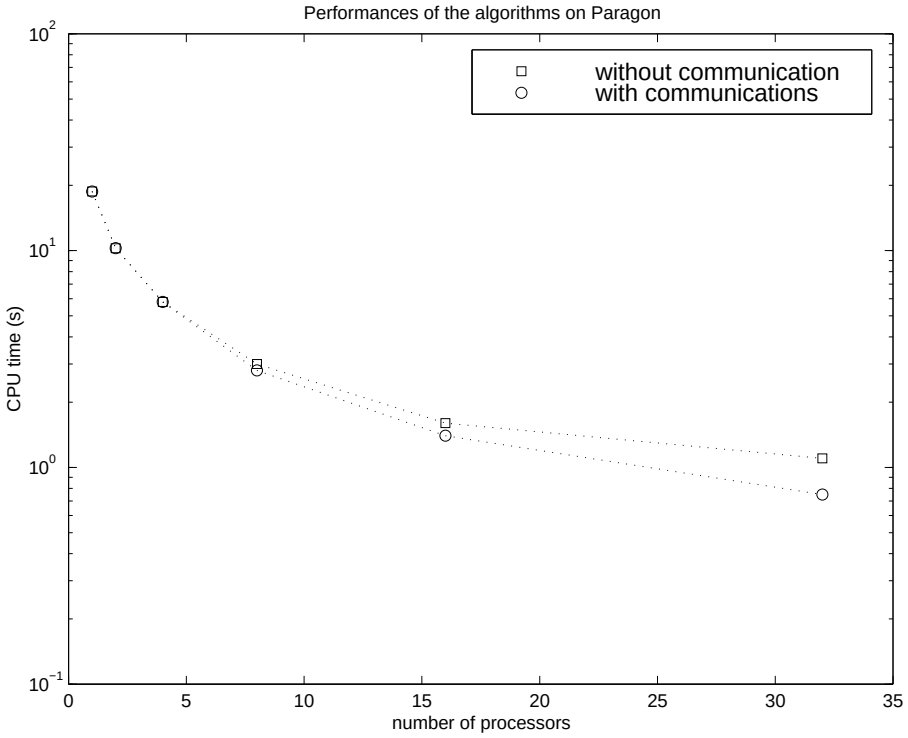


FIG. 9.1. *Graph of the performance results of the two algorithms on* PARAGON.

detected in such a way that non-necessary computation is avoided.

These two approaches rely on the fact that the number of processors to be considered must belong to a particular set which depends on the sequence of the dimensions of the matrices. Although that restriction is clearly justified by our scheduling, it should be possible to consider other numbers of processors. This aspect has to be explored.

REFERENCES

[1] M. Davio, *Kronecker Products and Shuffle Algebra.*, IEEE Trans. Comput., Vol. C-30, No. 2, pp. 1099-1109, 1981.

[2] S. Donatelli, *A Class of Stochastic Petri Nets with Parallel Solution and Distributed State Space.*, Performance Evaluation, Vol. 18, pp. 21-36, 1993.

[3] J. J. Dongarra, J. DuCroz, S. Hammarling, and R. Hanson, *An extended set of Fortran Basic Linear Algebra Subprograms*, ACM Trans. Math. Software 14, 1, pp. 1-17, 1988.

[4] P. Fernandes, B. Plateau, and W. J. Stewart, *Efficient Descriptor-Vector Multiplications in Stochastic Automata Networks*, INRIA Rapport de recherche interne No. 2935 , July 1996.

[5] J. Granta, M. Conner, and R. Tolimieri, *Recursive fast algorithms and the role of the tensor product*, IEEE Transaction on Signal Processing, 40(12):2921-2930, December 1992.

[6] S. K. S. Gupta, Z. Li, and J. H. Reif, *Generating efficient programs for two-level memories from tensor products*, In Prof. of the Seventh IASTED/ISMM Int. Conf. on Parallel and Distributed Computing and Systems, pp. 510-513, Washington D.C (USA), October 1995.

[7] C. H. Huang, J. R. Johnson, and R. W. Johnson, *Generating parallel programs from tensor product formulas: A case study of Strassen's matrix multiplication algorithm*, In Proc. Int. Conf. on Parallel Processing, pp. 104-108, August 1992.

[8] R. W. Johnson, C. H. Huang, and J. R. Johnson, *Multilinear algebra and parallel programming*,

Journal of Supercomputing, 5:189-218, 1991.

[9] R. M. Karp, R. E. Miller, and S. Winograd, *The organization of computations for uniform recurrence equations*, Journal of the ACM, 14(3):563-590, July 1967.

[10] S. D. Kaushik, C. H. Huang, R. W. Johnson, and P. Sadayappan, *A methodology for generating efficient disk-based algorithms from tensor product formulas*, In Proc. Sixth Annual Workshop on Languages and Compilers for Parallel Computing, pp. 338-358, August 1993.

[11] C. H. Koelbel, D. B. Loveman, R. S. Schreider, G. L. Steele Jr, M. E. Zose *The High Performance Fortran Handbook*, The MIT Press.

[12] N. Koike, *NEC Cenju-3: A Microprocessor-Based Parallel Computer*, Proceedings of IPPS, pp. 396-401, 1974.

[13] B. Philippe, W. J. Stewart, and Y. Saad, *Numerical Methods in Markov Chain Modelling.* Operations Research, Vol. 40, No. 6, 1992.

[14] B. Plateau, *A methodology for solving markov models of parallel systems.*, Journal of Parallel and Distributed Computing, 12(370-387), 1991.

[15] B. Plateau, *On the stochastic structure of parallelism and synchronization models for distributed algorithms.*, In ACM Sigmetrics Conference on Measurement and Modelling of Computer systems, Austin, August 1985.

[16] Y. Saad, *Data communication in parallel architectures.*, Parallel Computing, (11), 1989.

[17] P. J. Schweitze, *Methods for large markov chains.*, International workshop on applied mathematics and performance reliability models of computer connection systems, University of Pisa, Italy, pp. 225-234, 1983.

[18] M. Siegle, *On efficient Markov Modelling.*, In Proc. QMIPS Workshop on Stochastic Petri Nets, pp. 213-225, Sophia-Antipolis, France, November 1992.

[19] W. J. Stewart, *An introduction to the Numerical Solution of Markov Chains.*, Princeton University Press, New Jersey, 1994.

[20] W. J. Stewart, K. Atif, and B. Plateau, *The Numerical Solution of Stochastic Automata Networks*, European Journal of Operations Research, vol. 86, No. 3, pp. 503-525 1995.

[21] H. S. Stone, *Parallel processing with the perfect shuffle*, IEEE Trans. on Computers, 20(2):153-161, 1971.

[22] A. Touzene, *Resolution des Modeles Markoviens sur Machine a Memoires Distribuees.*, These de l'INPG de Grenoble, September 1992.

[23] J. N. Tsitsikis, and D. P. Bertsekas, *High Parallel and Distributed Computation.*, Printice-Hall. International Edition, 1989.

[24] J. N. Tsitsikis, and D. P. Bertsekas, *A survey of some aspect of parallel and distributed iterative algorithm.*, CICS Publication Office M.I.T, June 1990.

[25] M. Wolfe, *High Performance Compilers for Parallel Computing.*, Addison-Wesley Publishing Compagny, 1996.