RAIRO-Oper. Res. 45 (2011) 1–16
DOI: 10.1051/ro/2011100

RAIRO Operations Research
www.rairo-ro.org

GREEDY ALGORITHMS FOR OPTIMAL COMPUTING OF MATRIX CHAIN PRODUCTS INVOLVING SQUARE DENSE AND TRIANGULAR MATRICES

Faouzi Ben Charrada[1], Sana Ezouaoui[1] and Zaher Mahjoub[1]

Abstract. This paper addresses a combinatorial optimization problem (COP), namely a variant of the (standard) matrix chain product (MCP) problem where the matrices are square and either dense (*i.e.* full) or lower/upper triangular. Given a matrix chain of length n, we first present a dynamic programming algorithm (DPA) adapted from the well known standard algorithm and having the same $O(n^3)$ complexity. We then design and analyse two optimal $O(n)$ greedy algorithms leading in general to different optimal solutions *i.e.* chain parenthesizations. Afterwards, we establish a comparison between these two algorithms based on the parallel computing of the matrix chain product through intra and inter-subchains coarse grain parallelism. Finally, an experimental study illustrates the theoretical parallel performances of the designed algorithms.

Keywords. Combinatorial optimization, dynamic programming, greedy algorithm, matrix chain product, parallel computing.

Mathematics Subject Classification. 65K05, 90C27, 90C39, 90C59.

Received November 16, 2009. Accepted December 14, 2010.

[1] University of Tunis El Manar, Faculty of Sciences of Tunis, Campus Universitaire 2092 Manar II, Tunis, Tunisia; f.charrada@gnet.tn; zwawi_sana@yahoo.fr; zaher.mahjoub@fst.rnu.tn

Article published by EDP Sciences
© EDP Sciences, ROADEF, SMAI 2011

1. Introduction

Let A and B be two sparse square matrices of size p. It is well known that the number of operations (including additions and multiplications), denoted NOP, required to compute the product matrix C = AB and the structure of this latter (*i.e.* distribution of its non zero elements) depend on both the densities (*i.e.* ratios of non zero elements) and the structures of A and/or B [12,15,16]. If we consider square dense (*i.e.* full) and lower/upper triangular matrices in which we are interested in this paper, we can summarize as given in Table 1, the trivial results as far as the structure of matrix C and the required NOP for computing it are concerned. We precise that in the remainder, D will denote a dense matrix and L (resp. U) will denote a lower (resp. an upper) triangular matrix.

Notice that when p is large, the above NOP formulae may be simplified by keeping only cubic terms. From Table 1, we can make the following remarks.

- The product matrix C is an L (resp. a U) matrix only when both A and B are L (resp. U) matrices, otherwise it is a D matrix. Furthermore, matrices A and B play symmetric roles *i.e.* given the structures of A and B, the structures of C = AB and C′ = BA are identical as well as the required NOP's for computing C or C'.
- The NOP required by a DD (resp. an LL or a UU) product is the highest (resp. lowest). More precisely, for large p, we have the following relations:

NOP_DD = 2*NOP_DU = 2*NOP_DL = 3*NOP_LU = 6*NOP_LL = 6*NOP_UU

Now, given n square matrices $A_1, \dots, A_n$ of size p, the variant of the matrix chain product (MCP) problem we address here is a combinatorial optimization problem (COP) consisting in optimally computing the product matrix $A = A_1 \dots A_i \dots A_n$, where A_i may be either dense or upper/lower triangular. Indeed, the total number of operations required to compute A greatly depends on the product sequence *i.e.* the order in which the matrices are multiplied. Assume for instance that A_1 is dense and both $A_2, \dots, A_n$ are upper (or lower) triangular. The Left-Right Parenthesization (LRP) *i.e.* $(A_1A_2)A_3)\dots)A_n$ requires $(n-1)p^3 + O(np^2)$ operations whereas the (optimal) Right-Left Parenthesization (RLP) *i.e.* $A_1(\dots(A_{n-2}(A_{n-1}A_n)$ only requires $((n+1)/3)p^3+O(np^2)$ operations *i.e.* about 3 times less. Therefore, the point we address is to determine an optimal parenthesization (OP) corresponding to the minimum number of operations to compute A = $A_1A_2 \dots A_n$. The combinatorial property of our problem is due to the fact that, for a chain of n matrices, we may exhibit an exponential number of parenthesizations equal to the Catalan number which increases in $\Omega(4^n/n^{3/2})$ [3].

We recall that the (standard) well known MCP problem considers a chain involving dense rectangular matrices. As far as this problem is concerned, an optimal O(n^3) dynamic programming algorithm (DPA) due to Godbole, is known since 1973 [7]. In 1984, Hu and Shing [9] proposed an optimal O($n \log n$) algorithm based on polygon partitioning, a problem proved to be equivalent to the MCP one. Particular instances that may be solved in O(n) time are discussed in [6]. O(n)

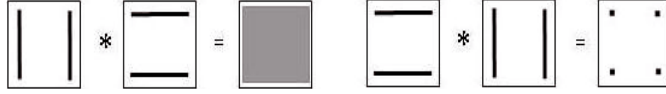

FIGURE 1. Product of sparse square matrices.

TABLE 1. Structure of product matrix C = AB and required NOP.

B\A	D	L	U
D	D $2p^3$	D $p^3 + p^2$	D $p^3 + p^2$
L	D $p^3 + p^2$	L $p^3/3 + (3p^2 + 2p)/3$	D $2p^3/3 + (3p^2 + p)/3$
U	D $p^3 + p^2$	D $2p^3/3 + (3p^2 + p)/3$	U $p^3/3 + (3p^2 + 2p)/3$

time algorithms for finding sub-OP's are also known in the literature [2,8,14]. So far, the case of sparse matrices has received, to our knowledge, little attention. We may particularly cite [12] where the extension of the MCP problem to sparse square matrices is briefly mentioned. For this purpose, the authors first introduced a formula for the number of multiplications required to compute the product of two sparse square matrices, namely $d_1 d_2 p^3$ [15], where d_1 and d_2 are the two matrices densities and p their size, the density being the ratio of non zero elements. Before going further, let us analyse the above formula which is basic for the remainder. Consider for instance a chain of three sparse rectangular matrices A_1, A_2, A_3 whose dimensions are denoted (p_{i-1}, p_i): $i = 1, 2, 3$. Assume that A_i has n_i non zeros, thus $d_i = n_i/(p_{i-1}p_i)$. If we want to determine the minimum number of operations (including additions and multiplications) and an OP for computing A $= A_1A_2A_3$, we first have to study the product $A_{12} = A_1A_2$. Here, two main points arise, namely the determination of (i) the (minimum) number of operations to compute A_{12} and (ii) the structure of A_{12} namely the distribution and number of its non zeros, denoted n_{12}, and its density $d_{12} = n_{12}/(p_0p_2)$. In fact, (i) as well as (ii) requires knowing the structures of both A_1 and A_2. Indeed, particular distributions of non zeros in a couple of very sparse matrices may lead to a dense product matrix. To be convinced, assume for sake of simplicity, that both A_1 and A_2 are square *i.e.* $p_i = p$ ($i = 0, 1, 2$), and that the whole elements of A_1 (resp. A_2) are zeros except those of columns (resp. rows) 1 and p (see Fig. 1). Thus we have $n_1 = n_2 = 2p$, $d_1 = d_2 = 2/\mathrm{p}$ and $d_1 d_2 p^3 = 4p$. It is easy to remark that A_{12} is dense and $2p^2$ multiplications (or additions) are required to compute it. Notice, however, that $A_{21} = A_2A_1$ has four non zero elements and only 4p multiplications (or additions) are required to compute it.

Hence, stating according to [12,15] that $d_1 d_2 p^3 = 4p$ multiplications (and the same number of additions) are required here for computing A_{12} is wrong (though

true for computing A_{21}). Obviously, the formula is not always true and may lead to a non OP when solving the considered MCP problem.

Now, going back to the case we address, consider for example a chain of three square matrices A_1, A_2, A_3 of size p where A_1 (resp. A_2) is an L matrix (resp. a U matrix) and A_3 is a D matrix. According to the above formula, the Left-Right Parenthesization *i.e.* $(A_1A_2)A_3$ would be wrongly considered as an OP whereas the Right-Left one *i.e.* $A_1(A_2A_3)$ is the true OP. This is due to the fact that A_1A_2 is a dense matrix, thus its density is equal to 1 and not 1/4 ($=d_1d_2$).

To conclude, we think that the sparse MCP problem cannot be correctly addressed without solving the two-point problem mentioned above. For this reason, we restrict our study here to the case of square dense and triangular matrices for which the associated two-point problem is trivial as previously seen (see Tab. 1). Studying this MCP variant, denoted SDT-MCP, is obviously easier than the general sparse MCP problem where randomly structured sparse matrices are considered.

The remainder of the paper is organized as follows. In Section 2 we present an optimal dynamic programming algorithm (DPA) of $O(n^3)$ complexity. Section 3 is devoted first to the description of two optimal $O(n)$ greedy algorithms, leading in general to different optimal solutions namely chain parenthesizations. We then present a comparative study between the two algorithms whose aim is to choose the more suitable solution for computing in parallel the matrix chain product through intra and inter-subchains parallelism. Then follows an experimental study illustrating the theoretical coarse grain parallel performances of the two greedy algorithms. We finally conclude our work in Section 4 and detail some perspectives.

2. The Dynamic Programming Algorithm (DPA)

We first recall the well known optimization formula for the standard MCP problem. Consider a chain of n matrices $A_1, A_2, \ldots, A_n$ where (p_{i-1}, p_i), $i = 1 \ldots n$, are the dimensions of A_i. Let $A_{ij} = A_iA_{i+1} \ldots A_j$ and $m(i, j)$ the minimum number of operations, denoted MNO, to compute A_{ij}. We have the following [3,7,9]:

$$\min(i, j) = \min_{i \le k \le j-1} (m(i, k) + m(k+1, j) + f(p_{i-1}, p_k, p_j)) \tag{2.1}$$

where $m(i, k)$ (resp. $m(k+1, j)$) is the minimum number of operations to compute $A_{ik} = A_iA_{i+1} \ldots A_k$ (resp. $A_{k+1,j} = A_{k+1}A_{k+2} \ldots A_j$) and $f(p_{i-1}, p_k, p_j)$ is the number of operations required to compute $A_{ij} = A_{ik}A_{k+1,j}$. The classical optimization formula restricts operations to only multiplications *i.e.* $f(p_{i-1}, p_k, p_j) = p_{i-1}p_kp_j$ whereas the total number involving both multiplications and additions is $2p_{i-1}p_kp_j$. We prefer using, for sake of consistency, the latter expression as given for square matrices in Table 1.

From (2.1), it is easy to derive the optimization formula when the n matrices are square of size p and either D, L or U. For this purpose, let s_{ik} be the structure

of A_{ik} (*i.e.* either D, L or U), $s_{k+1,j}$ the structure of $A_{k+1,j}$ and $s_{i,j}$ the structure of A_{ij}. Hence, we get the following:

$$\min(i,j) = \min_{i \leq k \leq j-1} (m(i,k) + m(k+1,j) + c(s_{ik}, s_{k+1,j})). \tag{2.2}$$

Here, $c(s_{ik}, s_{k+1,j})$ is the number of operations required for computing $A_{ij} = A_{ik}A_{k+1,j}$ (see Tab. 1).

Notice that the structure of matrix A_{ij}, denoted $s_{i,j}$ which depends on both s_{ik} and $s_{k+1,j}$ (see Tab. 1), has to be saved for the remainder. Thus, the derived DPA may be formally written as follows.

DO $l = 2, n$ / 1st *loop, l is the subchain length/*
DO $i = 1, n-l+1$ / 2nd *loop, i is the number of subchains of length l/*
$j = i + l - 1$
/ 3rd *loop:* $k = i, j-1$ /

$$\min(i,j) = \min_{i \leq k \leq j-1} (m(i,k) + m(k+1,j) + c(s_{ik}, s_{k+1,j}))$$

. save the "cut" index k where the minimum is reached
save the structure $s_{i,j}$ of the product matrix A_{ij}
ENDDO
ENDDO

As the DPA involves three nested loops, it obviously has an $O(n^3)$ complexity. We will see below that better algorithms may de designed.

Remark. Is it possible to derive, as done by Hu and Shing for the standard MCP problem [8,9], an $O(n \log n)$ algorithm based on polygon partitioning? The answer is in fact negative as proven below. As a matter of fact, Hu and Shing proved the following lemma resulting from the equivalence between the MCP and the polygon partitioning (PP) problems.

Lemma_HS. The minimum numbers of operations to evaluate the (n+1) following matrix chain products are identical: $A_1A_2 \ldots A_{n-2}A_{n-1}A_n, A_{n+1}A_1 \ldots A_{n-2}$ $A_{n-1}, \ldots, A_2A_3 \ldots A_nA_{n+1}$ where A_i has dimensions (p_{i-1}, p_i): $i = 1 \ldots n+1$ and $p_{n+1} = p_0$.

In other words, let $S_0 = p_0, p_1 \ldots, p_n$ be a sequence of n+1 positive integers and $S_1, \ldots, S_n$ be the n sequences deduced from S_0 par cyclic permutations. The minimum numbers of operations to compute the matrix chain products whose dimensions correspond to any sequence S_i $(i = 0 \ldots n)$ are identical.

Applied to our SDT-MCP problem of a chain involving square dense and triangular matrices, the above lemma reduces to the following. "The minimum numbers of operations to evaluate the n following matrix chain products are identical: $A_1A_2 \ldots A_{n-2}A_{n-1}A_n, A_nA_1 \ldots A_{n-2}A_{n-1}, \ldots, A_2A_3 \ldots A_nA_1$ where A_i is a square den-se or lower/upper triangular matrix".

It is easy to remark that this lemma is in general no longer true. Indeed, consider for instance a chain of three matrices A_1, A_2, A_3 of size p where A_1 and A_3 are L

matrices and A_2 is a U one. An OP for computing $A_1A_2A_3$ = LUL is $(A_1A_2)A_3$ (or $A_1(A_2A_3)$) and the MNO is $(5/3)p^3 + O(p^2)$, whereas the (unique) OP for computing $A_3A_1A_2$ = LLU is $(A_3A_1)A_2$ and the MNO is $p^3 + O(p^2)$. Notice, however, that the (unique) OP for computing $A_2A_3A_1$ = ULL is $A_2(A_3A_1)$ and the MNO is $p^3 + O(p^2)$ *i.e.* the same as for $A_3A_1A_2$ = LLU.

As lemma_HS does no longer hold, a direct consequence is that the SDT-MCP and the Polygon Partitioning problems are not equivalent. Hence, an O($n \log n$) algorithm based on polygon partitioning cannot be derived for the SDT-MCP problem.

3. Optimal greedy algorithms

3.1. First algorithm description

A two-fold idea is behind the (first) greedy algorithm (GA) we designed, the rationale being the choice of the more adequate couple of matrices to first process. It consists in first combining similar matrices (D with D, L with L, U with U). This permits to obtain a so called *compressed chain*. This latter is then processed in such a way that avoids, whenever it is possible, increasing the number of D matrices (a new D matrix is created when combining an L one with a U one, see Tab. 1). Consider for instance a DLU chain. It is obvious that an OP is the *Left-Right Parenthesization* (LRP) *i.e.* (DL)U and the MNO is $2p^3+O(p^2)$, whereas the *Right-Left Parenthesization (RLP) i.e.* D(LU) creates a new D matrix and costs $(8/3)p^3+O(p^2)$. However, if we consider an LUL chain, the (LU)L parenthesization as well as the L(UL) one will unavoidably create a new D matrix. We detail below the greedy algorithm (GA) wich involves two phases. An alternative algorithm will be presented afterwards.

- **i) Phase 1:** Compression phase Compute the product of every subchain involving matrices of same structure (D with D, L with L, U with U), according to either the LRP ($\longrightarrow$) or the RLP ($\longleftarrow$). Let $\mathcal{C}_c$ be the obtained compressed chain. Remark that in $\mathcal{C}_c$ any two successive matrices are necessarily of different structures.
- **ii) Phase 2:** Two cases have to be considered.
  - Case 1. $\mathcal{C}_c$ involves no D matrix: compute $\mathcal{C}_c$ according to either the LRP ($\longrightarrow$) or the RLP ($\longleftarrow$). Remark here that only one new D matrix will be created.
  - Case 2. $\mathcal{C}_c$ involves at least one D matrix. Clearly $\mathcal{C}_c$ may be written as follows:
  $$\mathcal{C}_c = \mathcal{C}_1 D \mathcal{C}_2 \tag{3.1}$$
  where $\mathcal{C}_1$ involves no D matrix but $\mathcal{C}_2$ may involve. Three subcases may be considered here.
    - $\mathcal{C}_1 = \phi$: compute D$\mathcal{C}_2$ according to the LRP ($\longrightarrow$).
    - $\mathcal{C}_2 = \phi$: compute $\mathcal{C}_1$D according to the RLP ($\longleftarrow$).

– $\mathcal{C}_1 \neq \phi$ and $\mathcal{C}_2 \neq \phi$: compute $\mathcal{C}_1$D according to the RLP ($\longleftarrow$). Then, the result being a D matrix, compute D$\mathcal{C}_2$ according to the LRP ($\longrightarrow$). Another way consists in first computing D$\mathcal{C}_2$ according to the LRP. Then, the result being a D matrix, compute $\mathcal{C}_1$D according to the RLP.

Notice that in Phase 1, combining successive D matrices is not necessary and may be postponed to Phase 2 (see remark on Ex. 3.1 given below). Notice in addition that Phase 1 vanishes when the original chain is already compressed *i.e.* is such that any two successive matrices are of different structures.

3.2. Optimality of the Greedy Algorithm (GA)

As already precised, the main ideas that are behind the GA are (i) performing products of matrices of similar structures, thus requiring the lowest costs if we restrict to L and U matrices (see remarks on Tab. 1); and (ii) avoiding the creation of a new D matrix (if possible). We precise that for sake of notation simplicity, we will use below the simplified NOP formulae taken from Table 1 (*i.e.* restricted to cubic terms). The optimality proof is based on the following two lemmas.

Lemma 3.1. *In an OP, successive L matrices (resp. U matrices) are necessarily combined together.*

Proof. Assume, without loss of generality, that the chain to process may be written as follows: $\mathcal{C} = \mathcal{C}_1\text{LL}\mathcal{C}_2$ where either $\mathcal{C}_1$ or $\mathcal{C}_2$ may be empty. The negation of Lemma 3.1 states that an OP for $\mathcal{C}$ is necessarily one among the three following: $((\mathcal{C}_1\text{L})\text{L})\mathcal{C}_2$, $\mathcal{C}_1(\text{L}(\text{L}\mathcal{C}_2))$ and $(\mathcal{C}_1\text{L})(\text{L}\mathcal{C}_2)$. We may assume now again, without loss of generality, that $\mathcal{C}_2$ is empty and $\mathcal{C}_1$ reduces to one matrix. Hence, two cases may be considered.

- Case 1. $\mathcal{C}_1$ is a D matrix thus $\mathcal{C}$ = DLL and (DL)L costs $2p^3$ whereas D(LL) only costs $(4/3)p^3$.
- Case 2. $\mathcal{C}_1$ is a U matrix thus $\mathcal{C}$ = ULL and (UL)L costs $(5/3)p^3$ whereas U(LL) only costs p^3.

Therefore the negation of Lemma 3.1 leads to a non OP. It is easy to notice that assuming $\mathcal{C}_1$ empty and $\mathcal{C}_2$ non empty leads to the same result. □

Lemma 3.2. *If the chain $\mathcal{C}$ to process is such that any two successive matrices are of different structures, then an OP consists in proceeding as follows.*

- *(i) If $\mathcal{C}$ involves no D matrix, then the RLP ($\longleftarrow$) as well as the LRP ($\longrightarrow$) are optimal.*
- *(ii) If $\mathcal{C}$ involves at least one D matrix, then an OP consists in never combining an L matrix with a U one but in combining a D matrix with either an L, a U or a D matrix.*

Proof. Let us first notice that cases (i) and (ii) are closely related. Indeed, in (i), any OP will necessarily begin by combining a U matrix with an L one, thus creating a D matrix. The new chain is then processed according to (ii). Let us

now assume, without loss of generality, that $\mathcal{C} = \mathcal{C}_1\text{LU}\mathcal{C}_2$ where either $\mathcal{C}_1$ or $\mathcal{C}_2$ may be empty. The negation of (ii) states that an OP for $\mathcal{C}$ is necessarily either $(\mathcal{C}_1(\text{LU}))\mathcal{C}_2$ or $\mathcal{C}_1((\text{LU})\mathcal{C}_2)$. Here, we may also assume, without loss of generality, that $\mathcal{C}_2$ is empty and $\mathcal{C}_1$ reduces to one matrix, namely a D one. Thus only one case has to be considered *i.e.* $\mathcal{C} = \text{D(LU)}$ which costs $(8/3)p^3$ whereas (DL)U only costs $2p^3$. Hence, the negation of Lemma 3.2 leads to a non OP. Clearly, assuming $\mathcal{C}_1 = \phi$ and $\mathcal{C}_2 \neq \phi$ leads to the same result. □

3.3. Complexity analysis

- Phase 1 may be achieved in one or two steps where each step requires scanning the original chain. In the first step, the subchains involving matrices of same structure are detected. The second step consists in extracting the subchains and constructing the compressed chain. Remark that the second step is needless if the original chain is already compressed, thus may be merged with the first step if we choose to process the subchains as soon as they are detected. In all cases, obviously a linear time *i.e.* O(n) is required.
- Phase 2 consists, once the first occurence of a D matrix is detected, in scanning and computing the left subchain, then processing the right one. Clearly, this may be done in linear time *i.e.* O(n). Therefore GA requires an O(n) time.

Remark. The high O(n^3) complexity of the Dynamic Programming algorithm seen in Section 2 is due to the fact that it constructs an OP not only for the input n-chain but for any sub-chain of length $2 \dots n-1$ as well (namely $n-k+1$ chains of length k, for $k = 2 \dots n$) *i.e.* O(n^2) OP's. In the opposite side, the Greedy algorithm constructs only one OP *i.e.* for only the n-chain. Constructing as many OP's as does the DPA would need in this case an O(n^3) time.

Illustrative examples are given below (for sake of simplicity, the MNO is restricted to cubic terms).

Example 3.1. $n = 10$, $\mathcal{C} = \text{LLDDDLUULD}$

- Phase 1: LLDDDLUULD ⇒ (LL)((DD)D)L(UU)LD ⇒ $\mathcal{C}_c = \text{LDLULD}$ ⇒ $\mathcal{C}_c = \mathcal{C}_1\text{D}\mathcal{C}_2$.
- Phase 2: LDLULD ⇒ ((((LD)L)U)L)D or L((((DL)U)L)D).
- OP's: (((((LL)((DD)D))L)(UU))L)D, (LL)((((((DD)D)L)(UU))L)D); MNO $= (10 + 2/3)p^3$.
- We have to add that GA generates the same OP when using either the complete or the simplified NOP formulae (*i.e.* restricted to cubic terms) as given in Table 1 (see Sect. 1).
- As to the Dynamic Programming algorithm (DPA, see Sect. 2), the generated OP in general depends on the formulae used for NOP. However, the interesting feature is that the OP generated with one is also an OP for the

other. For the above chain, the OP generated by the complete (resp. simplified) formulae is "(((((((LL)D)D)D)L)(UU))L)D" (resp. "(((((LL)(DD)) D)(UU))L)D"). Notice that in the first OP, the DPA first combines L matrices together as well as U matrices, but not D matrices as their combination is not necessary for optimality (see remark in Sect. 3.1 on Phase 1 of algorithm GA).

Example 3.2. $n = 10$, $\mathcal{C}$ = DLLUDDULLD

- Phase 1: DLLUDDULLD $\Rightarrow$ D(LL)U(DD)U(LL)D $\Rightarrow$ $\mathcal{C}_c$ = DLUDULD $\Rightarrow$ $\mathcal{C}_c$ = D$\mathcal{C}_2$: $\mathcal{C}_1 = \phi$.
- Phase 2: DLUDULD $\Rightarrow$ (((((DL)U)D)U)L)D or D(L(U(D(U(LD))))).
- OP's: (((((D(LL))U)(DD))U)(LL))D, D((LL)(U((DD)(U((LL)D))))); MNO = $(10+2/3)p^3$.
- Alternative OP's: ((((((D(LL))U)D)D)U)(LL))D, D((LL)(U(D(D(U((LL)D)))))).
- OP generated by the DPA (complete or simplified NOP formulae): ((((((D(LL))U)D)D)U)(LL))D.

3.4. Alternative Greedy Algorithm (AGA)

We now present an alternative greedy algorithm (AGA) which also involves two phases, the first being a compression one too. It may be described as follows.

(i) Phase 1. Compression phase (as previously decribed in Sect. 3.1) The compressed chain $\mathcal{C}_c$ may be written as follows:

$$\mathcal{C}_c = \mathcal{C}_1 D \mathcal{C}_2 D \dots \mathcal{C}_i D \dots \mathcal{C}_{r-1} D \mathcal{C}_r \tag{3.2}$$

where each subchain $\mathcal{C}_i$, $i = 1 \dots r$, involves no D matrix *i.e.* only L and/or U matrices.

(ii) Phase 2. Two cases have to be considered

- $\mathcal{C}_c$ involves no D matrix (*i.e.* $r = 1$): the LRP ($\longrightarrow$) as well as well as the RLP ($\longleftarrow$) are optimal.
- $\mathcal{C}_c$ involves at least one D matrix. Here four subcases may be considered.

  (a) $\mathcal{C}_1 = \mathcal{C}_r = \phi$ *i.e.* $\mathcal{C}_c$ = D$\mathcal{C}_2$D$\mathcal{C}_3$ ... D$\mathcal{C}_{r-1}$D: compute (D$\mathcal{C}_i$), $i = 2 \dots r-1$, according to the LRP ($\longrightarrow$). Then compute the resulting chain involving $(r-1)$ D matrices according to either the LRP or the RLP ($\longleftarrow$). Another symmetric alternative consists in first computing ($\mathcal{C}_i$D), $i = 2 \dots r-1$, according to the RLP, then computing the resulting chain of $(r-1)$ D matrices according to either the LRP or the RLP.

  (b) $\mathcal{C}_1 = \phi$ and $\mathcal{C}_r \neq \phi$ *i.e.* $\mathcal{C}_c$ = D$\mathcal{C}_2$D$\mathcal{C}_3$ ... D$\mathcal{C}_r$: compute (D$\mathcal{C}_i$), $i = 2 \dots r$, according to the LRP. Then compute the resulting chain involving $(r-1)$ D matrices according to either the LRP or the RLP.

  (c) $\mathcal{C}_1 \neq \phi$ and $\mathcal{C}_r = \phi$ *i.e.* $\mathcal{C}_c = \mathcal{C}_1D\mathcal{C}_2D\mathcal{C}_3$ ... D$\mathcal{C}_{r-1}$D: compute ($\mathcal{C}_i$D), $i = 1 \dots r-1$, according to the RLP. Then compute the resulting chain involving (r) D matrices according to either the LRP or the RLP.

(d) $\mathcal{C}_1 \neq \phi$ and $\mathcal{C}_r \neq \phi$ *i.e.* $\mathcal{C}_c = \mathcal{C}_1\mathrm{D}\mathcal{C}_2\mathrm{D}\ldots\mathcal{C}_{r-1}\mathrm{D}\mathcal{C}_r$: compute $(\mathcal{C}_i\mathrm{D})$, $i = 1\ldots r-1$, according to the RLP. Then compute the resulting chain constituted by $(r-1)$ D matrices followed by $\mathcal{C}_r$ according to the LRP. Another alternative consists in computing $(\mathrm{D}\mathcal{C}_i)$, $i = 2\ldots r$, according to the LRP, then computing the resulting chain involving $\mathcal{C}_1$ followed by $(r-1)$ D matrices according to the RLP.

Remark.

- Optimality: the optimality proof of AGA may be easily deduced from the proof already seen for GA. Just remark that from expression (4): we get $\mathcal{C}_c = \mathcal{C}_1\mathrm{D}\ldots\mathcal{C}_{r-1}\mathrm{D}\mathcal{C}_r \Rightarrow \mathcal{C}_c = \mathcal{C}_1\mathrm{D}(\mathcal{C}_2\mathrm{D}\ldots\mathcal{C}_{r-1}\mathrm{D}\mathcal{C}_r) \equiv \mathcal{C}_1\mathrm{D}\mathcal{C}_{21}$ where $\mathcal{C}_{21} = (\mathcal{C}_2\mathrm{D}\ldots\mathcal{C}_{r-1}\mathrm{D}\mathcal{C}_r)$; $\mathcal{C}_{21} = \mathcal{C}_2\mathrm{D}(\mathcal{C}_3\mathrm{D}\ldots\mathcal{C}_{r-1}\mathrm{D}\mathcal{C}_r) \equiv \mathcal{C}_2\mathrm{D}\mathcal{C}_{22}$ where $\mathcal{C}_{22} = (\mathcal{C}_3\mathrm{D}\ldots\mathcal{C}_{r-1}\mathrm{D}\mathcal{C}_r)$; etc. These successive transformations leading to a series of expressions similar to (3.1) permit to see that we can follow a similar proof argument as done for GA. We think it is useless to detail it again.
- Complexity: Phase 1 requires as in AG an $\mathrm{O}(n)$ time. As to Phase 2, it may be designed in two steps. The first one consists in detecting the D matrices and the corresponding $\mathcal{C}_i$ subchains. The second consists in processing each subchain then combining the whole. This obviously may be done in $\mathrm{O}(n)$ time. Therefore AGA requires as GA an $\mathrm{O}(n)$ time.

Illustrative examples are presented below (the MNO is restricted to cubic terms).

Example 3.3 (Example 3.1 revisited)**.** $n = 10$, $\mathcal{C}$ = LLDDDLUULD

- Phase 1: LLDDDLUULD ⇒ (LL)((DD)D)L(UU)LD ⇒ $\mathcal{C}_c$ = LDLULD ⇒ $\mathcal{C}_c = \mathcal{C}_1\mathrm{D}\mathcal{C}_2\mathrm{D}$: $\mathcal{C}_3 = \phi$.
- Phase 2: LDLULD ⇒ (LD)(L(U(LD))).
- OP: ((LL)((DD)D))(L((UU)(LD))); MNO = $(10{+}2/3)p^3$.
- Alternative OP's: (((LL)(DD))(DL))((UU)(LD)), ((LL)(DD))((DL)((UU)(LD))).

Example 3.4 (Example 3.2 revisited)**.** $n = 10$, $\mathcal{C}$ = DLLUDDULLD

- Phase 1: DLLUDDULLD ⇒ D(LL)U(DD)U(LL)D ⇒ $\mathcal{C}_c$ = DLUDULD ⇒ $\mathcal{C}_c = \mathrm{D}\mathcal{C}_2\mathrm{D}\mathcal{C}_3\mathrm{D}$: $\mathcal{C}_1 = \phi$.
- Phase 2: DLUDULD ⇒ (((DL)U)((DU)L))D or D((L(UD))(U(LD))).
- OP's: ((D(LL))U)((((DD)U)(LL))D), (D(LL))((U(DD))(U((LL)D))); MNO = $(10{+}2/3)p^3$.
- Alternative OP's: ((D(LL))(UD))((DU)((LL)D)), ((D(LL))U)((DD)(U(LL)D)).

3.5. Comparing the two greedy algorithms and parallelism

Although the two greedy algorithms GA and AGA are both optimal, we can however notice from the examples seen above, that the constructed OP's are in general different. We may ask about the interest of the second algorithm AGA which seems more elaborate and less direct (in its second phase). In fact, it is easy

to remark that this latter is more adequate for computing the product matrix in parallel as detailed in the following.

- Phase 1 (common to the two algorithms). Obviously, the subchains involving matrices of same structure may be processed independently *i.e.* in parallel. In addition to this ***inter-subchains parallelism*** (provided that there are at least two subchains), each subchain involving at least four matrices may be processed in parallel by using the so-called *associative fan-in algorithm* [11]. This algorithm adopts a divide-and-conquer strategy consisting in processing the matrices of the subchain couple by couple. Thus ***intra-subchain parallelism*** may also be achieved. We precise that computing in parallel a (sub)chain constituted by m matrices of same structure) may be achieved in $\lceil \log_2 m \rceil$ steps instead of $m-1$ when done serially [4].
- Phase 2. In the first algorithm GA, the compressed chain written under the form $\mathcal{C}_c = \mathcal{C}_1\mathrm{D}\mathcal{C}_2$ is processed serially and cannot be processed in parallel *i.e.* $\mathcal{C}_1\mathrm{D}$ is first computed according to the RLP ($\longleftarrow$), then the resulting D matrix is combined with $\mathcal{C}_2$ according to the LRP ($\longrightarrow$). However, in the second algorithm AGA, $\mathcal{C}_c$ is written in a more suitable form for parallel computing *i.e.* $\mathcal{C}_c = \mathcal{C}_1\mathrm{D}\mathcal{C}_2\mathrm{D}\ldots\mathcal{C}_{r-1}\mathrm{D}\mathcal{C}_r$. If we analyse the last case (d) *i.e.* where $\mathcal{C}_1 \neq \phi$ and $\mathcal{C}_r \neq \phi$ (see Sect. 3.4), obviously the $(r-1)$ subchains $\mathcal{C}_1\mathrm{D},\ldots,\mathcal{C}_{r-1}\mathrm{D}$ may be processed independently *i.e.* in parallel when $r \geq 3$. Thus we have inter-subchains parallelism. Notice that here each subchain is and must be processed serially according to the RLP ($\longleftarrow$). On the other hand, processing these $(r-1)$ subchains leads to a chain of $(r-1)$ D matrices that may be processed in parallel when $r \geq 3$, by using the above mentioned associative fan-in algorithm. Thus, we have intra-subchain parallelism. Remark that as far as the three other cases (a), (b) and (c) of Phase 2 in AGA are concerned, a similar argumentation permits to easily exhibit intra-subchain parallelism as well as inter-subchains parallelism.

Let us add that an OP may be represented by a binary tree (BT) where each node corresponds to the product of two matrices and is weighted by the number of operations (NOP) required by this product [12]. If we adopt a level representation of this BT [4] such that the root (corresponding to the final matrix product) is at the bottom, we may define (i) the height (or depth) h which corresponds to the number of levels, and (ii) the width w which corresponds to the maximum number of nodes per level. It is known that, in general, *the larger the width and the smaller the height, the higher the parallelism* [4,13] as the nodes of each level may be processed in parallel. Therefore, analysing the structures of the BT's representing the different optimal parenthesizations is important and permits to extract the inherent parallelism.

On the other hand, given a BT, it is also known [4,13] that if we dispose of an unlimited number of identical processors, the optimal (*i.e.* minimal) parallel time to execute the BT tasks (corresponding to the nodes), *i.e.* compute the associated

product matrix, is equal to the *cost of a critical path* (CCP) of the BT. We precise that a critical path in a weighted graph is a path whose cost is maximal, the cost being the sum of the weights of its nodes. It is also known [4] that w processors (w being the width of the BT) are sufficient to execute the BT tasks in optimal time.

An illustrative example is presented below (MNO is given according to the simplified formulae).

Example 3.5. $n = 16$, $\mathcal{C}$ = ULLDDDLUDUUUUDLD

- Phase 1: ULLDDDLUDUUUUDLD
  $\Rightarrow$ U(LL)((DD)D)LUD((UU)(UU))DLD $\Rightarrow \mathcal{C}_c$ = ULDLUDUDLD
  Both intra-chain parallelism and inter-chains parallelism may be achieved *i.e.* LL // DD // UU // UU.
- Phase 2 of GA: $\mathcal{C}_c = \mathcal{C}_1 \mathrm{D} \mathcal{C}_2$ = ULDLUDUDLD
  $\Rightarrow$ (((((((U(LD))L)U)D)U)D)L)D
  No intra-chain parallelism nor inter-chains parallelism.
- OP: (((((((U((LL)((DD)D)))L)((UU)(UU)))D)U)D)L)D;
  MNO $= (17 + 1/3)p^3$
  For the associated binary tree, we have $h = 11$ and $w = 4$. The cost of a critical path (CCP) is equal to $(16{+}1/3)p^3$.
- Phase 2 of AGA: $\mathcal{C}_c = \mathcal{C}_1 \mathrm{D} \mathcal{C}_2 \mathrm{D} \mathcal{C}_3 \mathrm{D} \mathcal{C}_4 \mathrm{D}$ = ULDLUDUDLD
  $\Rightarrow$ ((U(LD))(L(UD)))((UD)(LD))
  Inter-chains parallelism may be achieved *i.e.* LD // UD // UD // LD...
- OP: ((U((LL)((DD)D)))(L(UD)))((((UU)(UU))D)(LD));
  MNO $= (17 + 1/3)p^3$.

For the associated binary tree, we have $h = w = 6$ and CCP $= 9p^3$ *i.e.* parallel AGA is $(16{+}1/3)/9 = 1.81$ faster than parallel GA. Thus AGA leads to a higher parallelism. The two binary trees are depicted in Figure 2 (dotted arcs correspond to node entries and do not belong to the graph).

We have to mention that the above described parallelism may be called coarse-grain parallelism (CGP) [10] where the grain size of computation corresponds to the cost (amount of computation) of one matrix product *i.e.* the weight of a node of the BT.

A higher parallelism may be obtained by using ***fine-grain parallelism*** (FGP) [10] where the grain size may be the cost of computing either a column block, a row block or a submatrix of each product matrix. Such procedure consists in first splitting each node of the BT into a set of independent sub-nodes (*i.e.* arc-free set) where the sub-nodes are of equal cost (in order to achieve load balancing) and may be processed independently *i.e.* in parallel. Notice that the number of sub-nodes per level, induced by the grain size, may be fitted to the number of available processors. However, the main drawback of FGP is that it generally induces an overhead, particularly due to inter-processor communication delays [5]. Remark that sometimes fine-grain parallelism may be the unique possible choice. Indeed, no coarse-grain parallelism can be exhibited if the width w of the binary tree (BT) is equal to 1. In this case, the height h of the latter is equal to $n - 1$ *i.e.*

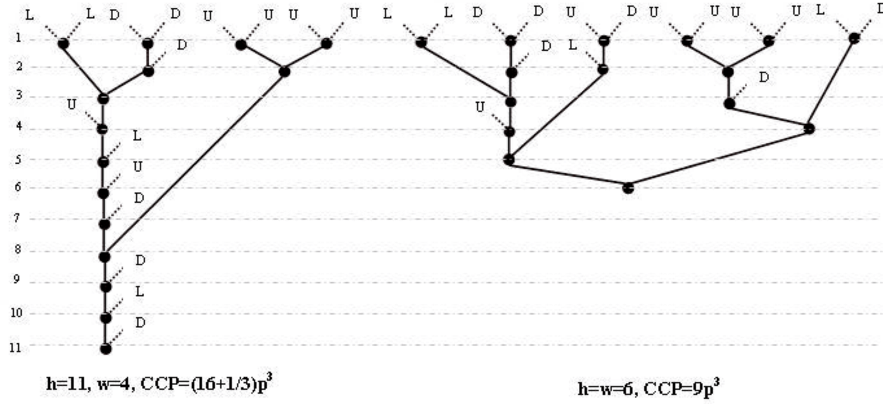

Figure 2. Binary trees and Parallelism in GA and AGA.

the BT is a 'chain' structured graph. Thus, the only alternative consists in using fine-grain parallelism.

An illustrative example is given below.

Example 3.6. $n = 10$, $\mathcal{C}$ = DLLLULULUL

- Phase 1: DLLLULULUL $\Rightarrow$ D(LLL)ULULUL $\Rightarrow$ $\mathcal{C}_c$=DLULULUL=D$\mathcal{C}_2$: $\mathcal{C}_1 = \phi$
  No intra-chain parallelism nor inter-chains parallelism.
- Phase 2: (identical in both AG and AGA):
  $\mathcal{C}_c$ = DLULULUL $\Rightarrow$ ((((((DL)U)L)U)L)U)L
  No intra-chain parallelism nor inter-chains parallelism.
- OP: ((((((D((LL)L))U)L)U)L)U)L; MNO = (7+2/3)p^3.

The associated binary tree is a chain structured graph where $h = 9$ and $w = 1$. Therefore no coarse-grain parallelism may be extracted.

3.6. Experimentations

A series of experimentations (see Tab. 2 below) were achieved and permit to illustrate the interest of AGA when compared with GA as far as the parallel computing of the chain product is concerned. For sake of simplicity, both MNO (minimal number of operations) and CCP (cost of a critical path, denoted CCP_GA for algorithm GA and CCP_AGA for algorithm AGA, are given in terms of p^3 and according to the simplified formulae (see Tab. 1).

On the other hand, in order to deepen our comparative performance analysis, we give in addition to MNO, h (height of the binary tree), w (its width), CCP_GA, CCP_AGA, the following *speed-ups* S_1 = MNO/CCP_GA, S_2 = MNO/CCP_AGA and S_{12} = CCP_GA/CCP_AGA. We precise that the processed chains were randomly generated.

Table 2. Structures of the BT's and parallel performances of GA and AGA.

GA						AGA				
n	MNO	h	w	CCP	S_1	h	w	CCP	S_2	S_{12}
10	9+2/3	7	3	9	1.07	5	4	7	1.38	1.29
15	17	12	3	15+1/3	1.11	8	5	11	1.50	1.35
20	26+2/3	13	6	18	1.48	7	8	12	2.22	1.50
25	34+2/3	16	5	24	1.44	10	6	18	1.93	1.33
30	31+2/3	19	10	23	1.36	10	12	14	2.24	1.64
35	35+1/3	21	7	24+1/3	1.45	15	9	18	1.96	1.35
40	48	28	9	36	1.33	11	13	18	2.67	2.00
45	48	29	12	38	1.26	12	16	21	2.29	1.81
50	60+1/3	36	9	47+2/3	1.27	19	18	30+2/3	1.97	1.55
60	77+2/3	43	14	59	1.32	18	21	34	2.28	1.74
65	86+2/3	44	13	60	1.44	21	21	36+2/3	2.39	1.65
70	84	43	18	57	1.47	23	23	37	2.27	1.54
75	86	42	20	55	1.56	20	24	33	2.61	1.67
80	102+2/3	55	15	73	1.41	22	23	38	2.70	1.92
85	105	60	15	85	1.24	29	27	54	1.94	1.57
90	110+1/3	59	25	77	1.43	22	30	41	2.69	1.88
95	104+2/3	62	20	79	1.32	19	26	36	2.91	2.19

The analysis of Table 2 leads to the following remarks.

- The height (resp. width) of the binary tree (BT) corresponding to the OP generated by AGA is always lower (resp. larger) than the height (resp. width) of the BT associated to the OP generated by GA. Hence, AGA exhibits a higher parallelism.
- The parallel speed-up of GA (S_1) is in the range [1.07, 1.56].
- The parallel speed-up of AGA (S_2) is in the range [1.38, 2.91].
- Parallel AGA is always faster (sometimes more than twice) than parallel GA and the inter-algorithms speed-up S_{12} is in the range [1.29, 2.19]. This confirms the first point seen above.

Hence, AGA outperforms GA *i.e.* it is more suitable for parallel computing.

4. Conclusion

In this paper we studied a specific variant of the matrix chain product problem where the chain involves square dense and lower/upper triangular matrices. We designed, after adapting a Dynamic Programming algorithm of cubic complexity, two optimal greedy algorithms of linear complexity. A comparison between the two latter who generally permit to derive different optimal chain parenthesizations (OP's), raised the interest of computing the matrix chain product in parallel for which the second greedy algorithm more efficient. This leads us to precise some

attracting perspectives we intend to study in the future. We may particularly cite the following points.

- Given an OP and adopting a coarse grain parallelism, determine (i) the minimum number of processors to compute the matrix chain product in minimum parallel time *i.e.* equal to the cost of a critical path in the binary tree (a hard problem [4,13]) and (ii) design an efficient scheduling when a given number of processors is available.
- Given an OP and adopting a fine grain parallelism (particularly when no coarse grain parallelism may be exhibited), choose an adequate grain size fitted to the number of available processors and design an efficient parallel algorithm achieving a good load balancing and where inter-processor communication delays overhead is reduced.
- Achieve an experimental study on a target parallel computer.

Acknowledgements. Special thanks are addressed to Dr. A.R. Mahjoub and Dr. H. Hasni for their valuable remarks.

References

[1] A.K. Chandra, *Computing matrix products in near-optimal time.* IBM Research Report, RC 5625 (1975).
[2] F.Y. Chin, An O(n) algorithm for determining a near-optimal computation order of matrix chain products. *Commun. ACM* **21** (1978) 544–549.
[3] T.H. Cormen, C.E. Leicerson, R.L. Rivest and C. Stein, *Introduction à l'Algorithmique.* Dunod (2002).
[4] M. Cosnard and D. Trystram, *Algorithmes et Architectures Parallèles.* InterEditions (1993).
[5] H. El-Rewini and M. Abd-El-Bar, *Advanced Computer Architecture and Parallel Processing.* Wiley (2005).
[6] S. Ezouaoui, F. Ben Charrada and Z. Mahjoub, O(n) instances of the matrix chain product problem solved in linear time, in *Proc. of ROADEF'09*, Nancy, France (2009).
[7] S.S. Godbole, An efficient computation of matrix chain products. *IEEE Trans. Comput.* **C-22** (1973) 864–866.
[8] T.C. Hu and M.T. Shing, Computation of matrix chain products. Part I. *SIAM J. Comput.* **11** (1982) 362–373.
[9] T.C. Hu and M.T. Shing, Computation of matrix chain products. Part II. *SIAM J. Comput.* **13** (1984) 229–251.
[10] V. Kumar, A. Grama, A. Gupta and G. Karypis, *Introduction to Parallel Computing – Design and Analysis of Algorithms.* The Benjamin/Cummings Pub. Co. (1994).
[11] S. Lakshmivarahan and S.K. Dhall, *Analysis and Design of Parallel Algorithms – Arithmetic and Matrix problems.* Mc Graw Hill (1990).
[12] H. Lee, J. Kim, S.J. Hong and S. Lee, Processor allocation and task scheduling of matrix chain products on parallel systems. *IEEE Trans. Parallel Distrib. Syst.* **14** (2003) 3–14.
[13] Z. Mahjoub and F. Karoui-Sahtout, Maximal and optimal degrees of parallelism for a parallel algorithm, in *Proc. of Transputers'94*, IOS Press (1994) 220–233.
[14] N. Santoro, Chain multiplication of matrices of approximately or exactly the same size. *Commun. ACM* **27** (1984) 152–156.
[15] A. Schoor, Fast algorithms for sparse matrix multiplication. *Inform. Process. Lett.* **15** (1982) 87–89.
[16] J. Takche, Complexities of special matrix multiplication problems. *Comput. Math. Appl.* **12** (1988) 977–989.