International Conference on Automation, Control, Engineering and Computer Science (ACECS'14)
Proceedings - Copyright IPCO-2014, pp.28-36
ISSN 2356-5608

On the Parallelisation of a Dynamic Programming Algorithm for Solving the 1D Array Partitioning Problem

Hajer Salhi[#1], Zaher Mahjoub[#2]

[#] *Faculty of Sciences of Tunis, University of Tunis El Manar,*
University Campus - 2092 Manar II, Tunis, Tunisia

[1]hajer.salhy@gmail.com
[2]zaher.mahjoub@fst.rnu.tn

Abstract— **We address in this paper the 1D array partitioning problem (1D-APP), a combinatorial optimisation problem (COP) for which exact polynomial time algorithms are known in the literature. We particularly consider the parallelisation of the dynamic programming algorithm of Skiena (DPA-S) which is structured in a DO loop nest of depth 3. Our approach is based on a three-phase procedure. The first consists in transforming the DPA-S into a perfect loop nest (PLN) from which five other versions are derived by applying the loop interchange technique (LIT). The second applies a dependency analysis on the initial PLN permitting the determination of the type of each loop (serial or parallel). As to the third phase, it applies on the initial PLN the LIT which leads to versions exhibiting a higher degree of parallelism. Finally, an experimental study achieved on a multicore parallel computer permits to validate our theoretical contribution.**

Keywords— ***Array partitioning, COP, dependency analysis, dynamic programming, loop interchange, loop nest, multicore machine, parallelisation.***

I. Introduction

Dynamic programming (DP) is an efficient paradigm for the design of algorithms solving a large class of combinatorial optimisation problems (COP). DP algorithms (DPA) have the particular structure of DO loop nests and are, in most cases, of polynomial complexity. Such algorithms are also polyhedral algorithms. Given an input COP, DP adopts a bottom-up approach leading to first solving subproblems whose solutions are used to solve subproblems of larger size. The procedure is then iterated until determining the solution of the input problem. The key idea is to express, through a recurrence formula, the solution of the initial problem in terms of the solutions of its son subproblems [1].

The 1D array partitioning problem (1D-APP) in which we are interested is in fact an easy COP for which several solving algorithms are known in the literature, among which we find the DPA of Skiena [2], denoted henceforth DPA-S. Our aim is first to design several versions of the original DPA-S, then study their parallelization by using the standard methodology for polyhedral algorithms.

The remainder of the paper is organised as follows. In section II, we first present the 1D-APP then a brief survey on solving algorithms known in the literature. Section III is devoted to the study of the (sequential) dynamic programming algorithm of Skiena (DPA-S). We develop in section IV our approach for the parallelisation of DPA-S. An experimental study is described in section V. Finally we conclude our work in section VI and propose some perspectives.

II. The 1D Array partitioning problem

The array partitioning problem (APP) with its diverse 1D and 2D variants is a COP having several real world applications such as in data base fragmentation and distribution, image processing, task scheduling, sparse scientific computing … [3], [4].

In its standard 1D variant, denoted 1D-APP, the problem we address may be formulated as follows. Given a list of, say, n items of costs c1 … cn, and an integer m (<n), the 1D-APP consists in fragmenting (splitting) the list into m successive i.e. contiguous fragments (sublists) such that the weight of the fragment of maximal weight is minimised, the weight of a fragment being the sum of the costs of the items it involves. This formulation may also be seen as the problem of scheduling n non preemptive independent tasks T1 … Tn of costs c1 … cn, onto m (identical) processors with the constraint that each processor must receive a subset of contiguous tasks. The objective is clearly to minimise the Makespan [2]. It has to be underlined that the same scheduling problem without the contiguity constraint is a classical hard COP [5]. But, with this constraint, the problem becomes an easy COP.

Several algorithms for solving the 1D-APP are known in the literature [6]. We are interested here on those based on dynamic programming, namely DPA's. Four are the mostly known. The following table recapitulates their complexities.

TABLE 1
DPA'S FOR SOLVING THE 1D-APP

Reference	Complexity
Anily & Fergruen, 1991 [7]	*O(mn²)*
Hansen & Lih., 1992 [8]	*O(mn²)*
Olstad & Manne 1995 [9]	*O(m(n-m))*
(Skiena, 1998) [2]	*O(mn²)*

It has to be underlined that the algorithm of Skiena (DPA-S) has some attracting properties as will be detailed below. We may particularly cite its regular structure i.e. a DO loop nest and the fact that it solves not only the instance of the original problem, denoted *P(n,m)*, but also all the subproblem instances *P(i,j) for i=2…n-1 and j=2…m-1* thus *O(nm)* subproblems. Furthermore it is the mostly cited and used.

Concerning the parallelisation of algorithms for solving the 1D-APP, to our knowledge, no works have been achieved so far.

III. STUDY OF THE DYNAMIC PROGRAMMING ALGORITHM OF SKIENA

A. *Original algorithm*

We first describe the original algorithm of Skiena (DPA-S). We adopt the task scheduling formulation of the 1D-APP and use the following notations:

- c_i : cost of task i, denoted T_i ; f(i): sum of the costs of the first i tasks i.e. $c_1+\dots+c_i$
- M(i,j) : i=1..n, j=1..m is the makespan obtained by scheduling the first i tasks onto j processors.
- D(i,j) corresponds to the position where begins the j-th fragment (i.e. the task subset assigned to processor j) when scheduling the first i tasks.

DPA-S practically consists in determining the m-1 optimal splitting indices $i_1 \dots i_{m-1}$ such that fragment 1 corresponds to the interval $[1\ i_1]$, fragment 2 to $[i_1+1\ \ i_2]$, …, fragment r to $[i_{r-1}+1\ \ i_r]$ … and fragment m to $[i_{m-1}+1\ \ n]$.

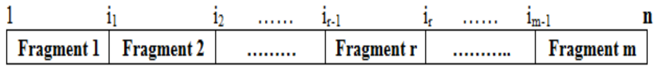

Fig. 1 Fragmentation in the 1D-APP

DPA-S is in fact based on the following dynamic programming recurrence formula:

M(i,j)=min(max(M(k,j-1), f(i)-f(k)), k=1…i-1) : i=2…n, j=2…m.

We detail DPA-S below.

```
DPA-S
f(0)=0                                          }
DO i=1,n                                        }  Initialization
    f(i)=f(i-1)+ c_i                            }  phase
    M(i,1)= f(i)                                }  Complexity :
DO i=1,m                                        }  O(n)
    M(1,i)= c_1                                 }
ENDDO                                           }
DO i=2,n                                        }
    DO j=2,m                                    }
        M(i,j)=∞        / statement 1 /         }
        DO k=1,i-1                              }  Processing
            s=max(M(k,j-1), f(i)-f(k))          }  phase
            IF (M(i,j) > s) THEN                }  Complexity :
                M(i,j)=s ; D(i,j)=k             }  O(mn²)
            ENDIF                               }
        ENDDO                                   }
    ENDDO                                       }
ENDDO                                           }
```

Fig. 2 DPA-S. Original version

We remark that DPA-S original version involves two parts. The first, corresponding to an initialization phase, is a simple DO loop and the second, corresponding to the processing phase, is a loop nest of depth 3 i.e. involving three loops. We'll restrict our study to this second part.

Since DPA-S is a non perfect loop nest i.e. the three loops are not tightly nested, we'll proceed to its transformation into a perfect loop nest where the loops are tightly nested.

Let us recall that DPA-S solves, in addition to the original problem whose instance is denoted *P(n,m)*, all instances *P(i,j) for i=2…n-1 and j=2…m-1.*

B. *From a non perfect loop nest to a perfect loop nest*

The analysis of perfect loop nests (PLN) being easier than that of non perfect loop nests (NPLN), we often proceed to transforming a NPLN into a PLN. Several techniques are known for this purpose [10]. As far as DPA-S is concerned, the transformation is immediate and consists in removing statement 1 i.e. "M(i,j)=∞" from the NPLN and including it in a outer perfect two-loop nest placed just before the NPLN which thus becomes a PLN. We hence get two successive PLN's : a perfect two-loop nest and a perfect three-loop nest as detailed below.

```
DO i=2,n
    DO j=2,m
        M(i,j)=∞
        DO k=1,i-1
            s=max(M(k,j-1), f(i)-f(k))
            IF (M(i,j) > s) THEN
                M(i,j)=s ; D(i,j)=k
            ENDIF
        ENDDO
    ENDDO
ENDDO
```

Non perfect 3-loop nest

```
DO i=2,n
    DO j=2,m
        M(i,j)=∞
    ENDDO
ENDDO
-----------------------------------
DO i=2,n    / loop ℒ1/
    DO j=2,m    / Loop ℒ2/
        DO k=1,i-1   / Loop ℒ3/
            s=max(M(k,j-1), f(i)-f(k))
            IF (M(i,j) > s) THEN
                M(i,j)=s ; D(i,j)=k
            ENDIF
        ENDDO
    ENDDO
ENDDO
```

Perfect 2-loop and 3-loop nests

Fig. 3 Transformation: from non perfect loop nest to perfect loop nest

Now, consider the new perfect three-loop nest. Its body involves logical and arithmetic operations. For sake of simplicity and without lost of generality, let *c (=O(1))* be its cost. The complexity of the nest, denoted *C(n,m)*, may be easily computed.

Indeed, we have *C(n,m)=c(m-1)n(n-1)/2=O(mn2)*. Since we obviously have $2 \leq m \leq n-1$, the complexity order of *C(n,m)* is $O(n^2)$ in the best case and $O(n^3)$ in the worst case.

The new DPA-S version will be called henceforth IJK version. It has to be underlined that a well known optimisation technique used for nested loops, also called polyhedral algorithms, consists in deriving, from an initial version, other semantically equivalent versions by applying the loop interchange technique (LIT) [11]. Hence, from the initial IJK version, we can derive five others i.e. versions JIK, IKJ, JKI, KIJ and KJI. However, we must check the validity of the LIT before applying it [11], [12]. This important fact is studied in section IV. However, since the five versions are valid and for sake of clearness, we anticipate their presentation.

C. *Derived versions by loop interchange*

It has to be firstly noticed that, since DPA-S uses two 2D arrays i.e. M and D, an important point that has a direct impact on the practical efficiency of any among the 6 versions is the array access mode i.e. either row-wise (R) or column-wise (C).

The following table recapitulates the characteristics of the 6 versions. Remark that the nest body remains the same in the six versions. Only the loop bounds are different.

TABLE 2
CHARACTERISTICS OF THE 6 DPA-S VERSIONS

Version		**IJK**	**IKJ**	**JIK**	**JKI**	**KIJ**	**KJI**
Loop bounds	$\mathscr{L}1$	*2, n*	*2, n*	*2, m*	*2, m*	*1, n-1*	*1, n-1*
	$\mathscr{L}2$	*2, m*	*1, i-1*	*2, n*	*1, n-1*	*k+1,n*	*2, m*
	$\mathscr{L}3$	*1, i-1*	*2, m*	*1, i-1*	*k+1, n*	*2, m*	*k+1, n*
Access mode	*M(k,j-1)*	*C*	***R***	*C*	*C*	***R***	*R*
	M(i,j)	*R*	***R***	*C*	*C*	***R***	*C*
	D(i,j)	*R*	***R***	*C*	*C*	***R***	*C*

Since we have for versions IKJ and KIJ three row-wise access modes (R), they will be the best in a C programming environment where arrays are stored row-wise. The four other versions, particularly the standard IJK one, will be less efficient since we have (one to three) conflicts between storing mode and access modes. The experimental study will in fact confirm these propositions.

IV. DEPENDENCY ANALYSIS

A. *Fine grain case*

Dependency analysis (DA) is the most important phase in algorithm parallelisation [11], [12]. Given a sequential algorithm, we first have to choose the granularity. For perfect loop nests, the immediate choice is the fine grain (FG) corresponding to the body of the innermost loop. Medium or coarse grains (MG, CG) may also be used. However, the fine grain leads in general to a higher parallelism degree. We'll begin by choosing the fine grain and see further the medium grain where a grain corresponds to the body of the second loop.

Now, Let T(i,j,k) be the loop nest body of version IJK. The dependency analysis consists in the determination of eventual read-write conflicts between two instances $T(i_1,j_1,k_1)$ and $T(i_2,j_2,k_2)$ if they are simultaneously executed. This analysis may be done according to the so called Bernstein conditions [11], [12]. The application of this standard procedure leads to the following (3,2) dependency distance matrix DDM involving two dependency distance vectors (DDV) :

$$\text{DDM} = \begin{pmatrix} d_{21} & 0 \\ d_{22} & 0 \\ d_{23} & d_{13} \end{pmatrix} \text{ with } \begin{cases} d_{21} = i_2 - i_1 \\ d_{22} = j_2 - j_1 \\ d_{23} = k_2 - k_1 \\ d_{13} = k_2 - k_1 \end{cases} ;$$

$$\text{such that } \begin{cases} 1 \leq i_2 - i_1 \leq n-2 \\ 1 \leq j_2 - j_1 \leq m-2 \\ 1 \leq k_2 - k_1 \leq n-2 \\ 1 \leq k_2 - k_1 \leq n-2 \end{cases}$$

We deduce the sign dependency distance matrix SDDM where a positive (resp. negative) component in the DDM is replaced by 1 (resp. -1).

$$\text{SDDM} = \begin{pmatrix} 1 & 0 \\ 1 & 0 \\ 1 & 1 \end{pmatrix}$$

Remark that a DDV must be lexicographically positive (LP) i.e. its first nonzero element must be positive. Since the (3,2) SDDM has non negative elements, any row permutation leads to LP columns. We hence conclude the following:

The five versions derived by loop interchange i.e. JIK, IKJ, JKI, KIJ and KJI are both valid since the SDDM of each of them which corresponds to a row permutation of the initial SDDM (associated to version IJK) is LP.

The SDDM induces two dependency levels : level 1 (position of the first nonzero element of the first column vector) and level 3 (position of the first nonzero element of the second column vector). Therefore the I and K loops in version IKJ are serial (S) whereas the J loop is parallel (P). We hence associate to version IJK the tuple SPS (IJK $\rightarrow$ SPS).

As to the other versions, we easily deduce the following: IKJ $\rightarrow$ SSP, JIK $\rightarrow$ SPS, JKI $\rightarrow$ SSP, KIJ $\rightarrow$ SPP, KJI $\rightarrow$ SPP. Therefore the versions exhibiting the highest parallelism are KIJ and KJI.

As an illustrative example, we present below in Fig.4 the dependency graph of version IJK for n=5 and m=4. This permits to better see the inherent parallelism that is expressed the best in versions KIJ and KJI.

International Conference on Automation, Control, Engineering and Computer Science (ACECS'14)
Proceedings - Copyright IPCO-2014, pp.28-36
ISSN 2356-5608

TABLE 3
CHARACTERISTICS OF THE SIX VERSIONS – FINE GRAIN CASE

Version	IJK	IKJ	JIK	JKI	KIJ	KJI
Loop types	SPS	SSP	SPS	SSP	SPP	SPP
p_{max}	$m-1$	$m-1$	$n-1$	$n-1$	$(n-1)(m-1)$	$(n-1)(m-1)$
T_{opt}	$cn(n-1)/2$	$cn(n-1)/2$	$c(n-1)(m-1)$	$c(n-1)(m-1)$	$c(n-1)$	$c(n-1)$
$\mathcal{C}$	$\frac{cn(n-1)(m-1)}{2}$	$\frac{cn(n-1)(m-1)}{2}$	$c(n-1)^2(m-1)$	$c(n-1)^2(m-1)$	$c(n-1)^2(m-1)$	$c(n-1)^2(m-1)$
S	$m-1$	$m-1$	$n/2$	$n/2$	$n(m-1)/2$	$n(m-1)/2$
E	1	1	$n/2(n-1)$	$n/2(n-1)$	$n/2(n-1)$	$n/2(n-1)$
E_∞	1	1	$1/2$	$1/2$	$1/2$	$1/2$
T_p	$\frac{(cn(n-1)/2)}{\lceil (m-1)/p \rceil}$	$\frac{(cn(n-1)/2)}{\lceil (m-1)/p \rceil}$	$\frac{c(n-1)(m-1)}{\lceil (n-1)/p \rceil}$	$\frac{c(n-1)(m-1)}{\lceil (n-1)/p \rceil}$	$\frac{c(n-1)}{\lceil (n-1)(m-1)/p \rceil}$	$\frac{c(n-1)}{\lceil (n-1)(m-1)/p \rceil}$
S_p	$(m-1)\left\lceil \frac{m-1}{p} \right\rceil$	$(m-1)\left\lceil \frac{m-1}{p} \right\rceil$	$\left(\frac{n}{2}\right)\left\lceil \frac{n-1}{p} \right\rceil$	$\left(\frac{n}{2}\right)\left\lceil \frac{n-1}{p} \right\rceil$	$\frac{n(m-1)}{2}\left\lceil \frac{(n-1)(m-1)}{p} \right\rceil$	$\frac{n(m-1)}{2}\left\lceil \frac{(n-1)(m-1)}{p} \right\rceil$
E_p	$\frac{(m-1)}{p}\left\lceil \frac{m-1}{p} \right\rceil$	$\frac{(m-1)}{p}\left\lceil \frac{m-1}{p} \right\rceil$	$\frac{n}{2p}\left\lceil \frac{n-1}{p} \right\rceil$	$\frac{n}{2p}\left\lceil \frac{n-1}{p} \right\rceil$	$\frac{n(m-1)}{2p}\left\lceil \frac{(n-1)(m-1)}{p} \right\rceil$	$\frac{n(m-1)}{2p}\left\lceil \frac{(n-1)(m-1)}{p} \right\rceil$

We recapitulate after in Table 2 the characteristics of the 6 versions where the following notations are adopted:

p_{max} : maximal degree of parallelism i.e. maximal number of parallel iterations in the loop nest

T_{opt}: parallel execution time (complexity) when p_{max} processors are used

C: cost of the parallel algorithm i.e. $p_{max}*T_{opt}$

S: speed-up i.e. T_{seq}/T_{opt} where T_{seq} is the sequential execution time *$(=c(n(n-1)(m-1)/2)$*

E: efficiency i.e. S/p_{max} ; E_∞: limit of E when $n \to \infty$

T_p: parallel execution time when $p < p_{max}$ processors are used

S_p, E_p: speed-up and efficiency when p processors are used

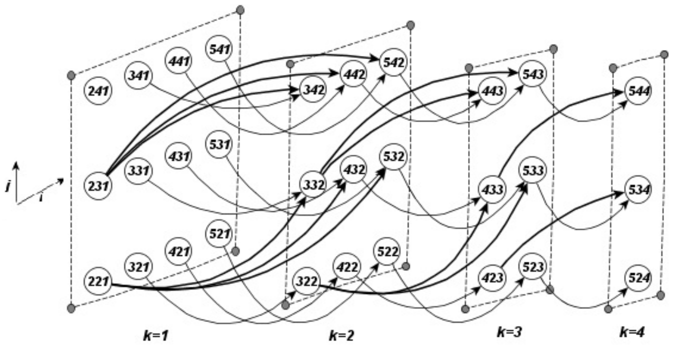

Fig. 4 IJK Reduced Dependency graph for n=5, m=4 - Fine grain case
(Transitive arcs are omitted)

Remarks.

- The six parallel algorithms corresponding to the 6 versions are both cost-optimal since we have for each ***$C=O(T_{seq})=O(mn^2)$*** i.e. E=O(1) when $n \to \infty$.

- Although involving only one parallel loop, versions IJK and IKJ are asymptotically more efficient since E_∞=1 for both, whereas E_∞ =1/2 for the four others.

- Versions KIJ and KJI considered the best in sequential (in a C environment) exhibit the highest parallelism degree i.e. *(n-1)(m-1).*

B. Medium grain case

We now proceed to the study of the medium grain (MG) case. Here the grain corresponds to the body of the second loop. Considering the 6 versions seen above, we easily deduce the following properties for the six induced two-loop nests IJ, JI, KI, KJ, JK and IK :

IJ → SP, JI → SP, KI → SP, KJ → SP, JK → SS, IK→ SS

We therefore restrict to the first four versions since the others involve no parallel loop. We present below in Fig.5 the dependency graphs for n=5, m=4 and recapitulate their characteristics in Table 3 where we included the grain size (gs).

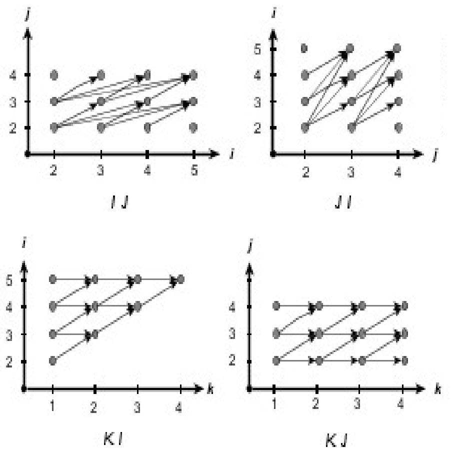

Fig. 5 Reduced Dependency graphs for n=5, m=4 - Medium grain case
(Transitive arcs are omitted)

TABLE 4
CHARACTERISTICS OF THE FOUR VERSIONS - MEDIUM GRAIN CASE

Version	IJ	JI	KI	KJ
Loop bounds	$2, n$	$2, m$	$1, n-1$	$1, n-1$
	$2, m$	$2, n$	$k+1, n$	$2, m$
gs	$c(i-1)$	$c(i-1)$	$c(m-1)$	$c(n-k)$
p_{max}	$m-1$	$n-1$	$n-1$	$m-1$
T_{opt}	$cn(n-1)/2$	$c(n-1)(m-1)$	$cn(m-1)$	$cn(n-1)/2$
$\mathcal{C}$	$cn(n-1)(m-1)/2$	$c(n-1)^2(m-1)$	$cn(n-1)(m-1)$	$cn(n-1)(m-1)/2$
S	$m-1$	$n/2$	$(n-1)/2$	$m-1$
E	1	$n/2(n-1)$	$1/2$	1
E_∞	1	$1/2$	$1/2$	1
T_p	$\frac{(cn(n-1)/2)}{\lceil (m-1)/p \rceil}$	$\frac{c(n-1)(m-1)}{\lceil (n-1)/p \rceil}$	$\frac{(cn(m-1))}{\lceil (n-1)/p \rceil}$	$\frac{(cn(n-1)/2)}{\lceil (m-1)/p \rceil}$
S_p	$(m-1)\left\lceil \frac{m-1}{p} \right\rceil$	$\left(\frac{n}{2}\right)\left\lceil \frac{n-1}{p} \right\rceil$	$\frac{(n-1)}{2}\left\lceil \frac{n-1}{p} \right\rceil$	$(m-1)\left\lceil \frac{m-1}{p} \right\rceil$
E_p	$\frac{(m-1)}{p}\left\lceil \frac{m-1}{p} \right\rceil$	$\frac{n}{2p}\left\lceil \frac{n-1}{p} \right\rceil$	$\frac{(n-1)}{2p}\left\lceil \frac{n-1}{p} \right\rceil$	$\frac{(m-1)}{p}\left\lceil \frac{m-1}{p} \right\rceil$

Remarks.

- The grain size (gs) is constant only in version KI (equal to *c(m-1))* and varies in the other versions. In versions IJ and JI, *gs=c(i-1)* and increases from c to *c(n-1)*. In version KJ, *gs=c(n-k)* and decreases from *c(n-1)* to c.
- The four parallel algorithms corresponding to the 4 versions are both cost-optimal since we have for each *$\mathcal{C}$= O(Tseq)=O(mn2)* i.e. E=O(1).
- Versions IJ, JI and KJ are more efficient since E=1 for both, whereas E =1/2 for version KI.
- As to the coarse grain case where the grain corresponds to the body of the first loop, it represents no interest since the unique loop of each version is serial.

V. EXPERIMENTAL STUDY

A. *Introduction*

We achieved a series of experiments involving two parts: a sequential and a parallel. The sequential part covers the 6 sequential DPA-S versions. As to the parallel part, we choosed version KIJ since it was globally the best in sequential and exhibits the highest parallelism degree (two parallel loops). The target machine we used is a Dell T5400 quad-core biprocessor (see configuration below).

Our algorithms were coded in C under Linux. For the parallel experiments, we used the shared memory OpenMP environment. Concerning the execution times, they are the means of several runs.

TABLE 5
TARGET MACHINE CONFIGURATION

Model		Dell T5400
# Proc.		2
Processor	***Model***	Intel® Xeon® E5420
	Clock	2.50 GHz
	Bus	1.33 GHz
	# Cores	4 per proc
Cache L1		128 Ko (inst) 128 Ko (data)
Cache L2		12 Mo
HD		250 Go
RAM		4 Go
OS		Ubuntu 11.04 64bits

B. *Sequential Part*

We precise that we chose 14 values for n in the range [100 5000] and for each n, 5 to 13 values for m such that *$5 \leq m \leq n/2$*. In fact, we achieved 132 tests for each of the 6 versions, thus 792 tests.

Excerpts of the results we obtained are depicted in Table 6 were we give the execution times, denoted ext (in seconds), and the following three ratios:

r1=ext(IJK)/ext(IKJ),
r2=ext(IJK)/ext(KJI),
r3=ext(IKJ)/ext(KIJ).

TABLE 6
EXECUTION TIMES (S) OF THE 6 VERSIONS

n	**m**	**IJK**	***IKJ***	**JIK**	**JKI**	**KIJ**	**KJI**	**r1**	**r2**	**r3**
100	**10**	0.000066	***0.000100***	0.000066	0.000072	**0.000068**	0.000086	0.66	0.97	1.47
	25	0.000174	***0.000100***	0.000172	0.000186	**0.000180**	0.000200	1.74	0.97	0.56
	50	0.000362	***0.000400***	0.000366	0.000392	**0.000366**	0.000400	0.91	0.99	1.09
500	**50**	0.011000	***0.009200***	0.010800	0.020700	**0.009300**	0.020800	1.20	1.18	0.99
	100	0.022300	***0.018800***	0.022100	0.042500	**0.018700**	0.042400	1.19	1.19	1.01
	250	0.056000	***0.046800***	0.056400	0.107900	**0.046600**	0.108000	1.20	1.20	1.00
1000	**50**	0.047000	***0.037000***	0.047000	0.087000	**0.037000**	0.086500	1.27	1.27	1.00
	250	0.241500	***0.188500***	0.241000	0.452000	**0.186500**	0.453500	1.28	1.29	1.01
	500	0.486000	***0.384000***	0.485000	0.910000	**0.374500**	0.929500	1.27	1.30	1.03
3000	**50**	0.433500	***0.345000***	0.433500	0.789000	**0.334500**	0.789000	1.26	1.30	1.03
	500	4.827000	***3.534500***	4.457500	8.318500	**3.498500**	9.207000	1.37	1.38	1.01
	1000	10.254000	***7.028500***	8.933000	16.694000	**7.049000**	18.837999	1.46	1.45	1.00
5000	**50**	1.207500	***1.031500***	1.206500	2.194000	**0.943500**	2.217000	1.17	1.28	1.09
	1000	28.959501	***19.848000***	24.853500	46.445000	**19.639999**	53.338501	1.46	1.47	1.01
	2000	58.375999	***39.496998***	49.752499	93.063004	**39.102001**	107.197006	1.48	1.49	1.01

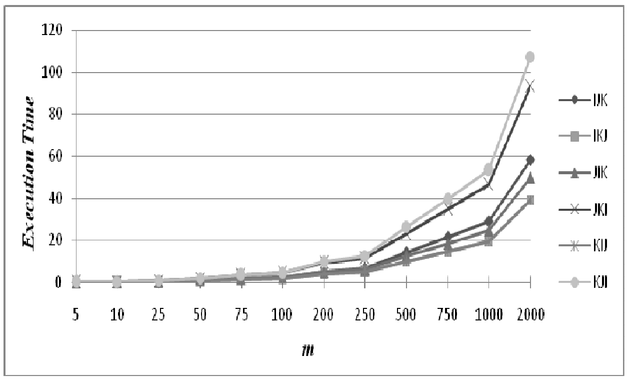

Fig. 6 Execution time (s) for n=5000

We remark from Table 6 and Fig.6 that, for n ≥ 500, versions IKJ and KJI are better than the standard version IJK, thus confirming our previous comments (see section III.C). This is due to the 3 row-wise accesses and the row-wise storing. On the other hand, versions JKI and KJI are the worst since we have 3 column-wise accesses in the first and two in the second. We have to add that version KIJ is up to almost 1.5 time faster than the standard version IJK (see ratio r2).

To better clarify the performances of each version, we give in Table 7 the rank ratio for each i.e. the number of cases (%) for which a given version is first ranked, second ranked…. For instance, version KIJ (resp. IKJ) was ranked first in 81.06% (resp. 18.94%) of the 132 tested cases and ranked second in 7.58% (resp. 67.42%) of the 132 cases.

TABLE 7
RANKING OF THE SIX VERSIONS

Rank	**IJK**	**IKJ**	**JIK**	**JKI**	**KIJ**	**KJI**
1	9.85	***18.94***	6.82	0	**81.06**	1.52
2	6.06	***67.42***	12.88	0	**7.58**	0
3	48.48	2.27	57.58	0.76	6.06	0.00
4	35.61	9.09	22.73	1.52	4.55	0.00
5	0	1.52	0	65.15	0	53.79
6	0	0.76	0	32.58	0.76	44.70

We remark that KIJ is the best in most cases, followed by IKJ, and KJI was the last in more cases (44.70%) than the others.

We conclude this section by underlying that the standard version IJK is outperformed by versions KIJ then IKJ. Thus we have better using version KIJ.

C. *Parallel Part*

Since version KIJ is the best in sequential and exhibits the highest parallelism degree (2 parallel loops), we chose it in this parallel experimentations part. We precise that, in addition to the choice of the values of n and m (see section V.B above), we chose 4 values for p (number of processors) i.e. 2, 4, 6, 8. In addition to the execution times, we give the speed-ups and efficiencies (%). Excerpts of the results we obtained are given below. Remark that 132*4 = 8 tests have been achieved.

TABLE 8
EXECUTION TIME (S)

		p				
n	**m**	**1**	**2**	**4**	**6**	**8**
		Execution Times (s)				
100	**10**	0.0000680	*0.0001000*	0.0000240	0.0000183	0.0000125
	25	0.0001800	0.0001750	0.0000600	0.0000383	0.0000313
	50	0.0003660	0.0003500	0.0001720	0.0000783	0.0000644
500	**50**	0.0093000	0.0064500	0.0040720	0.0026100	0.0023889
	100	0.0187000	0.0122500	0.0086160	0.0049533	*0.0050015*
	250	0.0465999	0.0302500	0.0220240	0.0128967	0.0104342
1000	**50**	0.0370000	0.0241250	0.0169000	0.0098000	0.0094825
	250	0.1864999	0.1160000	0.0882800	0.0547917	0.0415964
	500	0.3745000	0.2305000	0.1776600	0.1022750	0.0894732
3000	**50**	0.3345000	0.2252500	0.1474800	0.0987083	0.0764814
	500	3.4985001	2.3602500	1.4798001	1.0406334	0.8041461
	1000	7.0489997	4.8397498	2.9900601	2.4920998	1.6201014
5000	**50**	0.9434999	0.5453750	0.4385400	0.2799917	0.2376551
	1000	19.6399999	12.6201248	8.7922400	6.5241333	4.3135986
	2000	39.1020011	25.2596264	17.5527197	13.1734416	9.8327878

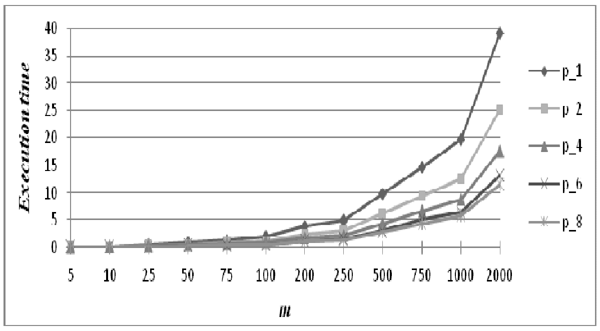

Fig. 7 Execution time (s) for n=5000

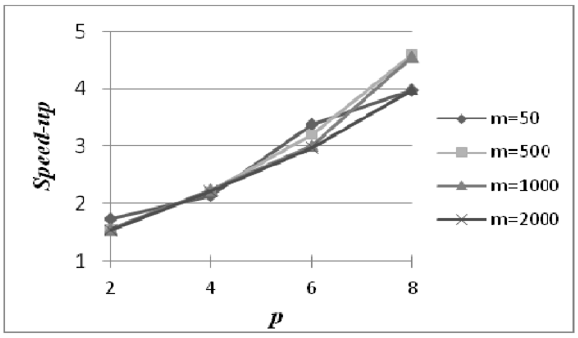

Fig. 8 Speed-up for n=5000

TABLE 9
SPEED-UP AND EFFICIENCY

		p							
n	**m**	**2**	**4**	**6**	**8**	**2**	**4**	**6**	**8**
		Speed-up				**Efficiency (%)**			
100	**10**	0.68	2.83	3.71	5.30	34.00	70.83	61.82	66.30
	25	***1.03***	***3.00***	***4.70***	***5.09***	51.43	75.00	78.26	63.68
	50	***1.05***	***2.13***	***4.67***	***5.55***	52.29	53.20	77.87	69.32
500	**50**	***1.44***	***2.28***	***3.56***	***3.83***	72.09	57.10	59.39	47.91
	100	1.53	2.17	3.78	3.47	76.33	54.26	62.92	43.32
	250	1.54	2.12	3.61	3.29	77.02	52.90	60.22	41.14
1000	**50**	1.53	2.19	3.78	3.74	76.68	54.73	62.93	46.70
	250	1.61	2.11	3.40	3.49	80.39	52.81	56.73	43.57
	500	1.62	2.11	3.66	3.49	81.24	52.70	61.03	43.63
3000	**50**	1.49	2.27	3.39	3.35	***74.25***	***56.70***	***56.48***	***41.82***
	500	***1.48***	***2.36***	***3.36***	***3.41***	***74.11***	***59.10***	***56.03***	***42.60***
	1000	***1.46***	***2.36***	***2.83***	***3.44***	***72.82***	***58.94***	***47.14***	***42.99***
5000	**50**	1.73	2.15	3.37	3.37	86.50	53.79	56.16	42.17
	1000	***1.55***	***2.23***	***2.97***	***3.41***	***77.81***	***55.84***	***50.17***	***42.81***
	2000	***1.55***	***2.23***	***2.92***	***3.41***	***77.40***	***55.69***	***49.47***	***42.60***

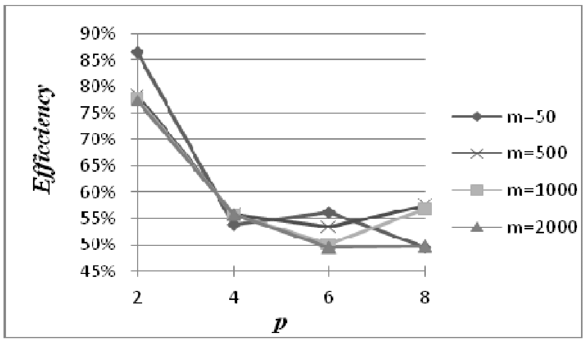

Fig. 9 Efficiency for n=5000

Remarks.

- The parallelization is beneficial since for fixed n and m, the execution time decreases for increasing p (Table 8 and Fig.7),(except for very few and non significant cases not exceeding 2.73% of the cases, 2 may be seen in Table 8).

- The best value for the speed-up (resp. efficiency) is 5.55 (resp. 86.50%) and is reached for the tuple n=100, m=50, p=8 (resp. n=5000, m=50, p=2).

- The speed-up as well as the efficiency do not always follow uniform and classical behaviours i.e. an increasing one for the speed-up and a decreasing one for the efficiency when n, m are fixed and p increases. This may be seen in Table 9 where the results written in bold italic correspond to uniform behaviours. As to the non uniform case, we notice that the speed-up often increases from p=2 to p=6, then decreases for p=8 (see underlined results in Table 9 and Fig.8). As to the efficiency, it follows an alternative behaviour (decrease-increase-decrease… or increase-decrease-increase…, see underlined results in Table 9 and Fig.9). Notice in addition that these uniform/non uniform behaviours do not occur for the same tuples (m,n,p) as far as speed-up and efficiency are concerned.

- The non uniform behaviours seen for the speed-up and the efficiency, though not exceptional in practice, may be due to combined reasons related to both the target machine architecture and the parallel program. We may particularly cite the eventual increase of cache misses and inter-core communication amounts induced by changes in the program parameters particularly p.

VI. Conclusion

In the study developed in this paper and addressing a particular dynamic programming algorithm (DPA) for solving the 1D array partitioning problem, we presented first a series of different versions of the DPA. We then described a parallelization procedure of the previous algorithms. A set of experimentations could validate the contribution and precise its practical interest. However, several interesting points remain to be seen, particularly:

- Study the problem of the determination of several optimal solutions for the 1D-APP and establish comparison criteria between them. Interesting preliminary results based on the use of the different versions of the DPA have been obtained so far.

- Achieve a series of experiments targeting a massively parallel computer in order to evaluate the scalability of the parallel algorithm and their behaviour when a large number of processors are used.

- Extend the parallelization approach to a heterogeneous environment i.e. where the available processors have different speeds.

References

[1] T. Cormen, C. Leiserson, R. Rivest. & C. Stein. *Introduction à l'algorithmique*, Paris, Dunod, 2002.

[2] S. Skiena, *The Algorithm design manual*, New York, Springer, 2008.

[3] O. Hamdi, "Etude de la distribution sur système à grand échelle de calcul numérique traitant des matrices creuses compressées", Doctoral thesis, Univ. of Tunis El Manar, Tunis, Tunisia, 2010.

[4] T.F. Zurek, "Optimization of partitioned temporal joins", Doctoral thesis, Univ. of Edinburgh. Scotland, 1997.

[5] T. N'takpé, "Ordonnancement de tâches parallèles sur plates-formes hétérogènes partagées", Doctoral thesis, University of Henri Poincaré - Nancy1, France, 2009.

[6] S. Khanna, S. Muthukrishnan & S. Skiena, "Efficient array partitioning", in *Proc. of the 24th International Colloquium on Automata, Languages and Programming,* Bologna, Italy, 1997, pp. 616-626.

[7] S. Anily & A. Fergruen, "Structured partitioning problems", *Operations Research, Vol.* 39(1), pp. 130-149, 1991.

[8] P. Hansen, & K.W. Lih, "Improved algorithms for partitioning problems in parallel, pipelined and distributed computing", *IEEE Transactions on Computers, Vol.* 41(6), pp. 769-771, 1992.

[9] B. Olstad & F. Manne, "Efficient partitioning of sequences". *IEEE Transactions on Computers. Vol.* 44(11), pp.1322-1326, 1995.

[10] A. Legrand & Y. Robert, *Algorithmique Parallèle,* Paris. Dunod, 2003.

[11] M.Cosnard, & D. Trystram, *Algorithmes et architectures parallèles*, Paris, InterEditions, 1993.

[12] B. Ben Mabrouk, H. Hasni & Z. Mahjoub, "Parallélisation de l'algorithme de la programmation dynamique pour la résolution du problème de produit d'une chaîne de matrices", in *Proc RenPar'09*, Toulouse, France, 2009.