Free Scheduling of General Nested Loops For Distributed Memory Architectures

Nectarios Koziris, Alexandros Pappas, George Papakonstantinou and Panayotis Tsanakas

Computer Science Division, Electrical Engineering Department
National Technical University of Athens
Zografou Campus, 15773, Athens, Greece
E-mail: {nkoziris, papakon}@dcslab.ece.ntua.gr

Abstract: The most extensive, in terms of time execution, part of a program is the nested loops. Loop parallelization involves two steps: First the time partitioning of the index space to achieve the minimum makespan, and second the efficient assignment of the concurrent partitions into the target parallel architecture. If distributed memory multiprocessor systems are used, overall performance is declined due to communication overheads, which increase the total completion time. This paper enhances the method of free scheduling general loop computations into time presented in [9], by considerably reducing the communication overheads. In [9] we proposed a schedule optimal in terms of time, which outperformed all linear schedules. Furthermore, in terms of total number of processors, a heuristic refinement of the free schedule which "shuffles" computations into time was proposed, without loss of the optimal time performance, to augment the mean processor utilisation. In this paper the communication cost is also considered; thus we enhance this schedule by assigning chains of computations to the same processor. Interprocessor communication is thus considerably cut down, by increasing local cache references inside every processing element. Our method reduces the required number of processors and the overall communication overhead, while preserving the optimal makespan.

Keywords
Nested loops, communication delays, scheduling, distributed memory architectures.

I. Introduction

Experience has shown that any program that requires a significant amount of computer time, spends most of this time in executing one or more loops. For a large class of programs these loops are nested FOR loops. The primary task when parallelizing a nested loop, is finding a schedule of computations into time, while preserving the data dependencies of the initial lexicographic loop order. Many methods were proposed, for time schedules, especially for uniform dependence loops, originating from the hyperplane method, presented by Lamport in [11]. Other researchers such as Shang, Darte, further extended the hyperplane method, by allowing more computations to be executed in parallel through the notion of displacement [15] and affine by statement transformations [5]. The main notion behind such scheduling is the introduction of a linear transformation, which organises indexed computations into well-defined disjoint groups called hyperplanes. Mathematical conditions which guarantee validity for a linear schedule, were well defined and presented, and algorithms, which find the optimal linear schedule, were extensively elaborated. Nevertheless, due to linearity's uniformity, tackling all computations uniformly is not the best way. Loop index spaces are sometimes irregularly bounded, and boundary effects and non-uniformities cannot be well organised by a uniform transformation. Transforming an index space with e.g. affine dependencies into another with uniform ones, using methods such as [16] or [18], increases artificially the optimal makespan when using a linear schedule An optimal time schedule, has to keep up with index space's irregularities, and organise them properly, without destroying the algorithm's initial data dependencies. In addition to this, the search for linear transformations imposes serious obstacles, since it requires the solution of diophantine equations [12] or integer programming [15].

On the other hand, general scheduling approaches were never used in loop parallelization. General multiprocessor scheduling with precedence constraints was proved to be NP-complete, see [6], even when the number of available processors is unbounded [13]. However, in our case, scheduling is simpler. We model the index space with a DAG of UET, and we initially ignore communication delays between computations (tasks) which are to be executed on different processors. Restricting to constant or small communication delays (SCT) or even ignoring them at all decreases dramatically the complexity, resulting in polynomial time solutions as in [1, 3, 8, 9].

Once optimal parallel execution is found, an efficient method of mapping the concurrent groups of computations into a parallel architecture should be applied. In the past, many proposed mapping methodologies were focused upon special purpose parallel architectures such as systolic arrays. Since systolic cells are synchronously operating, there is no need for communication efficient mapping. Fewer papers dealt with the problem of mapping nested loops into MIMD architectures. In message passing multiprocessor systems, communication delays play the most important role, since they can increase dramatically the total execution time. Researchers first tried to minimise the interprocessor communication, thus increasing the intraprocessor activity, by dividing the index space into as much as possible independent groups of computations. Shang in [14] and Hollander in [7] have presented methods for dividing the index space into independent sets of computations which are assigned to different processors, thus zeroing the communication cost. Unfortunately the independent partitioning of the index space in not always feasible. Sheu and Tai, in [17], presented a systematic method of partitioning the index space into groups and assign them into different processing elements. Their method is based on hyperplane time scheduling. Space partitioning is done without taking into consideration the notion of displacement. Furthermore the grouping is done along an axis perpendicular to the hyperplane, which is not the best choice since it ignores the locality of references for chains of computations. Nevertheless, Koziris et al. in [10] have considerably improved this method by considering grouping along dependence vectors.

In [9] we have presented a methodology for finding the free schedule for a general type nested FOR loop. Although a restricted computational model of uniform dependence loops was used, this was done only for simplicity reasons. Without any extension, this proposed methodology is applicable to any nested loop structure, including any non-uniform dependence statements. Our scheduling algorithm performs the best in terms of time free scheduling, thus outperforming all other kinds of schedules. In addition to this, it uses less number of processors, whereas many mapping methodologies so far presented [5, 12] use too many processors inefficiently. We have established lower bounds of processors, thus achieving efficient processor utilisation for all nested loop algorithms. The index space is depicted to a DAG, whose edges and nodes corresponds to dependence vectors and loop instances, respectively. Once the graph is well defined, a longest path algorithm is executed and PERT problem techniques are used to identify the margins of "shuffling" for various indexed computations. Finally, a refinement algorithm organises computations by using equal to lower bound number of processors.

In this paper, since the target architecture is a distributed memory multiprocessor, the proposed enhanced schedule should be consistent with communication overhead minimisation. After establishing an optimal time schedule based on [9], we arrange neighbour computations to be executed on the same processor, if possible. Our method maps chains of computations to the same processor, thus increasing the intraprocessor communication. Since the total number of communication activity is constant, increasing the intraprocessor communication is equivalent to decreasing the interprocessor communication. This model is realistic for distributed memory systems with cache memory in every processor. Assigning chains of computations to the same processor, increases the number of references from local cache memory, thus cutting down the communication overhead.

The rest of the paper is organised as follows: Basic terminology and definitions used throughout this paper are introduced in section II. In section III the structural properties of the index space are presented. Section IV includes theoretical lower and upper bounds for the total number of processors used in an optimal, by all means, schedule. Finally, in Section V, we present the proposed methodology for finding an optimal time schedule while preserving efficient communication and processor utilisation, together with illustrative examples.

II. Basic Concepts and Definitions

A. The Model of the Algorithms

Although our method is applicable to any multidimensional loop with arbitrary loop-carried dependencies, for simplicity of presentation, we will use a simplest model. Our computational model, without lack of generality, is loops that exhibit regular computation patterns and particularly nested FOR-loop algorithms with *uniform* data dependencies. That is, our algorithms are of the form shown in fig. 1:

```
FOR I1=l1 TO u1 DO
      ...
      FOR In=ln TO un DO
            AS1(I)
            ...
            ASk(I)
      ENDFOR
    ....
ENDFOR
```

Fig. 1: The Algorithmic model

where: (1) l_i and u_i are integer-valued constants, (2) $\mathbf{I}=(I_1, ..., I_n)$ and (3) $AS_1, ..., AS_k$ are assignment statements of the form $V_0=E(V_1, ..., V_l)$, where V_0 is an output variable indexed by **I** and produced by expression E operating on input variables $V_1, ..., V_l$, also indexed by **I**.

B. Notation

Throughout this paper the following notation is used: N is the set of naturals, Z is the set of integers, n is the number of nested FOR-loops of the algorithm and m is the number of dependence vectors of the algorithm. $J^n \subset Z^n$ is the set of indices: $J^n = \{(I_1, ..., I_n) \mid I_i \in Z \wedge l_i \leq I_i \leq u_i, 1 \leq i \leq n\}$. Each point in this n-dimensional integer space is a distinct instantiation of the loop body. Notice that the points of J^n are ordered *lexicographically*; the usual symbol < is used to denote this (linear) ordering. We shall discuss further the significance of this ordering in Section III. A *dependence vector* is denoted $\mathbf{d}_i=(d_{i1}, ..., d_{in})$, $1 \leq i \leq m$. The following two things should be emphasised: (1) By definition, dependence vectors are always > than **0**, where $\mathbf{0}=(0, ..., 0)$ and > is the *lexicographic* ordering and (2) In our model all dependence vectors are *uniform*, i.e. their elements are constants and not functions defined over index sets. The *dependence set* D of an algorithm A is the set of all dependence vectors of this algorithm: $D=\{\mathbf{d}_1, \mathbf{d}_2, ..., \mathbf{d}_m\}$.

C. The Directed Acyclic Graph Model for the Index Space

Since the proposed algorithmic model defines J^n as the vector space of computations, and the various dependence vectors join different indexed points, we can transform J^n to a DAG. In [7] we have presented a 1-1 function which corresponds points of J^n to vertices of a V set. Since there exists 1-1 correspondence between any $j \in J^n$ and $v_i \in V$, we will consider for simplicity of presentation $V \equiv J^n$, so the G(V, E) is defined as:

$V = \{\mathbf{j}(j_1, ..., j_n) \mid j_i \in Z \wedge l_i \leq j_i \leq u_i, 1 \leq i \leq n\}$, $E = \{(\mathbf{j_1}, \mathbf{j_2}) \mid \exists d \in D, \mathbf{j_1} + \mathbf{d} = \mathbf{j_2}\}$

In the next sections we will interchangeably use V and J^n.

III. STRUCTURAL PROPERTIES OF THE INDEX SPACE

In this section we shall present the partial ordering introduced by the dependence vectors. Dependencies impose precedence constraints, and partition the index space into sets, chains and antichains through the establishment of a partial ordering among computations.

Definition 3.1: Given an algorithm $A(J^n,D)$ we define the following two sets:

- the basis B of J^n, $B = \{\mathbf{j} \in J^n \mid \forall \mathbf{d} \in D, \mathbf{j}-\mathbf{d} \notin J^n\}$
- the top T of J^n, $T = \{\mathbf{j} \in J^n \mid \forall \mathbf{d} \in D, \mathbf{j}+\mathbf{d} \notin J^n\}$ ■

Clearly, B, T are unique. Intuitively, B is the set of points of J^n from which computations can be initiated at t_o in parallel, because they do not depend on any previous computations. Obviously $|B| \geq 1$, otherwise no computation would ever be performed by the algorithm. In addition to this $T \neq \emptyset$, otherwise the algorithm would never terminate.

We now proceed to define the following sequences of sets and binary relations over J^n :

Definition 3.2:

(i) $B_0 = B$ and $F_0 = \emptyset$

(ii) $B_{i+1} = \{\mathbf{j} \in J^n \mid \exists \mathbf{d} \in D\ \exists \mathbf{i} \in B_i$ such that $\mathbf{i}+\mathbf{d}=\mathbf{j}\}$ and

$F_{i+1} = F_i \cup \{(\mathbf{i}, \mathbf{j}) \mid \mathbf{j} \in J^n$ and $\exists \mathbf{d} \in D\ \exists \mathbf{i} \in B_i$ such that $\mathbf{i}+\mathbf{d}=\mathbf{j}\}$ ■

Definition 3.3: Let us define the following sequence of sets :

$GB_i = \{\mathbf{j} \in B_i \mid$ i is the greatest possible natural$\}$, $i \geq 0$.

Definition 3.4: The binary dependence relation $D^* \subseteq (J^n)^2$ is the reflexive transitive closure of the binary relation F_r :

$D^* = \cup_0^{r-1} D_i$, where

$D_0(\mathbf{j}_1, \mathbf{j}_2) \Leftrightarrow \mathbf{j}_1 = \mathbf{j}_2$

$D_1(\mathbf{j}_1, \mathbf{j}_2) \Leftrightarrow F_r(\mathbf{j}_1, \mathbf{j}_2)$

$D_{i+1}(\mathbf{j}_1, \mathbf{j}_2) \Leftrightarrow \exists \mathbf{j}_3\ (D_i(\mathbf{j}_1, \mathbf{j}_3) \wedge F_r(\mathbf{j}_3, \mathbf{j}_2))$ ■

In the rest of the paper we will simply write $\mathbf{j}_1 <_D \mathbf{j}_2$ instead of the more awkward $D^*(\mathbf{j}_1, \mathbf{j}_2)$.

The J^n set equipped with the binary relation $<_D$ is a partially ordered set, since it can be easily proved that $<_D$ is reflexive, antisymmetric and transitive. Abbreviating, $(J^n, <_D)$ is a poset, and this term will be used next.

The main property of a poset is its partitioning in chains and antichains. These terms are defined as follows:

Definition 3.5: Consider the poset $(J^n, <_D)$.

- A subset $C = \{\mathbf{j_1}, ..., \mathbf{j_r}\}$ of J^n is called a chain iff every two elements of this set are related with $<_D$. Formally, there exists an order $\mathbf{j_1} <_D .... <_D \mathbf{j_i} <_D .... <_D \mathbf{j_r}$
- A subset $A = \{\mathbf{j_1}, .., \mathbf{j_r}\}$ of J^n is called antichain iff no two distinct elements are related with $<_D$.

Definition 3.6 : For every $\mathbf{j} \in J^n$, the sets *ANCESTOR*($\mathbf{j}$) and *DESCENDANT*($\mathbf{j}$) are defined as follows:

ANCESTOR($\mathbf{j}$)=$\{\mathbf{i} \in J^n: \mathbf{i} <_D \mathbf{j}, \mathbf{i} \neq \mathbf{j}\}$

DESCENDANT($\mathbf{j}$)=$\{\mathbf{i} \in J^n: \mathbf{j} <_D \mathbf{i}, \mathbf{i} \neq \mathbf{j}\}$ ■

The following definition introduces the sets of *immediate* ancestors and descendants of $\mathbf{j} \in J^n$, with respect to the partial ordering:

Definition 3.7: For every $\mathbf{j} \in J^n$, the sets *IN*($\mathbf{j}$) and *OUT*($\mathbf{j}$) are defined as follows:

IN($\mathbf{j}$)=$\{\mathbf{i} \in J^n: \mathbf{i}+d=\mathbf{j}, \forall d \in D\}$

OUT($\mathbf{j}$)=$\{\mathbf{i} \in J^n: \mathbf{j}+d=\mathbf{i}, \forall d \in D\}$ ■

Corollary 3.1: For every $\mathbf{j} \in B$ it holds: $\mathbf{j} \in B \Leftrightarrow IN(\mathbf{j}) = \emptyset$

For every $\mathbf{j} \in T$ it holds: $\mathbf{j} \in T \Leftrightarrow OUT(\mathbf{j}) = \emptyset$ ■

Proof: Easily derived by definition 3.7 and 3.1

IV. LOWER AND UPPER BOUNDS FOR TIME AND PROCESSORS

In this section, we present upper and lower bounds concerning total time execution and processor utilisation, for the class of uniform dependence algorithms which were published in [9]. The properties of the index space, discussed previously, enable the definition of optimal execution time and optimal number of processors.

In [9] we have defined *T(A)* as the total time in which A is executed, whereas *OT(A)*=min *T(A)*, thus *OT(A)* as the least possible time in which A can be correctly executed. In addition to this, *TP(A)* was denoted as the total number of processors required by A, so as to be executed in *OT(A)*.

In [9] we proved: $\max\{MA_{max}, \left\lceil \frac{|J^n|}{|C_{\max}|} \right\rceil\} \leq TP(A) \leq A_{max}$,

where $MA_{max} = \max\{MA_i$: $MA_i = \{j_k \mid j_k \in A_i \wedge j_k \in C$ with $|C| = |C_{max}|\}$ and A_{max} is the maximal antichain of computations.

The above formula gives the lower and upper bounds for *OTP(A).* We should remark that these bounds are too tight and for a great number of problems, these bounds collide, thus producing the optimal *OTP(A).* In addition to this, in cases where all index points belong to maximal chains, thus every **j** belongs to a C_{max}, the exact lowest number of processors needed to execute algorithm A(J^n, D) in *OT(A)* is obviously given by MA_{max}.

What remains is a straightforward method to define the properties of J^n needed to calculate *OT(A)*, and present an analytical time and space schedule which implements the above metrics. Our algorithm, which finds for each index point the exact time and processor in which should be executed, is based on graph theory approaches, and outperforms all other presented methods in terms of total processor utilisation and total time execution. Before presenting the enhanced scheduling algorithm, some time scheduling properties are given.

V. Optimal Free Scheduling with Reduced Communication Costs

In this section we will present the properties of the free schedule of computations onto a clique of infinite number of processors. We will establish the properties of both free time and space scheduling. Furthermore a metric to calculate the total communication requirements of a specific free schedule is given. A free schedule contains time and space assignment of computations and is defined below as a couple of functions:

Definition 5.1: Given a task graph G(V, E) a schedule S is an ordered couple (S_{TIME}, S_{PROC}), where S_{TIME} and S_{PROC} are the time and processor schedules, respectively, defined as follows:
(1) S_{PROC}: $V \to \{0, \ldots, m\}$, $m \in N$, such that task $\mathbf{j} \in V$ is assigned to processor $S_{PROC}(\mathbf{j})$, and
(2) S_{TIME}: $V \to N$ such that for every task $\mathbf{i}, \mathbf{j} \in V$ we have $\mathbf{i} <_D \mathbf{j}$ then $S_{TIME}(\mathbf{j}) > S_{TIME}(\mathbf{i})$.

- The *makespan* of a time schedule S_{TIME} for the graph V, denoted M(S_{TIME}), is $\max\{S_{TIME}(\mathbf{j})+w \mid \mathbf{j} \in V\}$, where *w* is the processing time.
- Given the schedule $S(G_{P_n})=(S_{TIME}, S_{PROC})$, we define the following sets and functions:
  - $TIME(t)=\{\mathbf{i}: S_{TIME}(\mathbf{i})=t\}$
  - $TASK(p, t) =\{\mathbf{j}: S_{PROC}(\mathbf{j})=p \wedge S_{TIME}(\mathbf{j})=t\}$ ■

The makespan gives the completion time of the last task and, therefore, determines the time required for the completion of the whole J^n. *TIME*(t) is the set of nodes which are scheduled to be executed at t time instance. Finally, *TASK*(p, t) is the task **j** which is executed by p-th processor at t time instance.

Let us define E_{comm} the subset of E which contain the external edges i.e the edges between computations belonging to different processing elements.

$E_{comm}=\{(\mathbf{j_1}, \mathbf{j_2}) \in E: S_{PROC}(\mathbf{j_1}) \neq S_{PROC}(\mathbf{j_2})\}$

We define the Communication Cost Function for a scheduling S(S_{TIME}, S_{PROC}):

$$CCF\{S(S_{TIME}, S_{PROC})\} = |E_{comm}|$$

In this paper, our objectives are:

(1) To find an *optimal time schedule* $S_{TIME_{OPT}}$, i.e., a schedule whose makespan is *minimum.*
(2) To reduce the CCF{S(S_{TIME}, S_{PROC})} from an arbitrary initial schedule
(3) To find a *space schedule* S_{PROC} that realizes $S_{TIME_{OPT}}$ while having low $|E_{comm}|$.

A. Free time scheduling:

Our goal is to find the sets GB_i. If, for any $\mathbf{j} \in J^n$, we know the i: $\mathbf{j} \in GB_i$ then we can schedule computation **j** to be executed in i instance.

In [9], theorem 6.1, we have proved that function f: f(**j**) = i iff $\mathbf{j} \in GB_i$ performs the valid optimal free time scheduling of computations indexed by $\mathbf{j} \in J^n$. Clearly f(**j**)=0 for every $\mathbf{j} \in B$. Therefore for an arbitrary $\mathbf{j} \in GB_i$, i is the length of the longest chain $\mathbf{j_0} <_D \ldots <_D \mathbf{j}$ from $\mathbf{j_o} \in B$ to **j**. Referring to graph G(V, E) the equivalent property of such a length, is the longest path from $\mathbf{j_o}$ to **j.**

Definition 5.2: We define the following properties for a $\mathbf{j} \in V$:
long (**i**, **j**): is the longest path from $\mathbf{i} \in V$ to $\mathbf{j} \in V$.
early(**i**): the earliest time at which a computation i can start
late(**i**): the latest time at which a task i can start without increasing the earliest possible completion time of the overall system tasks. ■
By the above definitions: *early* ($\mathbf{j_i}$)=*long*($\mathbf{j_0}$, $\mathbf{j_i}$), *late*($\mathbf{j_i}$) = *long*($\mathbf{j_0}$, $\mathbf{j_n}$) - *long*($\mathbf{j_i}$, $\mathbf{j_n}$), where $\mathbf{j_n} \in T$.
In [9] we have presented that: $S_f(\mathbf{j})$ = **i** iff $\mathbf{j} \in GB_i$ *early*($\mathbf{j_i}$) = *long*($\mathbf{j_0}$, $\mathbf{j_k}$) where $\mathbf{j_o} \in B$.
In other words, an equivalent definition for GB_i considering the notion of *early*() is:
$GB_i=\{\mathbf{j} \in J^n \mid early(\mathbf{j}) = i\}$
Since an optimal schedule requires the search for all GB_i's, such search is performed through the longest path *long*($\mathbf{j_1}$, $\mathbf{j_i}$). A polynomial longest path algorithm based on topological ordering of a DAG is given in Appendix. Its total complexity is O (|V| + |E|) (see also [6] problem ND29). It suffices to visit every node and edge once to determine the "long" and "early" properties.

It is obvious, that section III refers to the decomposition of J^n into $|C_{max}|$ antichains at least. There exist many ways in which a poset (J^n,$<_D$) can be decomposed into $|C_{max}|$ antichains, where $|C_{max}|$ is the largest chain length. According to the above definitions and notes, a valid decomposition into $|C_{max}|$ antichains, is to find GB_i's through longest paths, and then consider A_i=GB_i. The methodology presented here, performs such antichain decomposition. This decomposition performs only the optimal time scheduling. In case of space mapping, a simple, near optimal mapping method leads in assigning every distinct element of any antichain $A_i = GB_i$ into a different processor. This means that $|A_{max}|$ processors are needed according to this antichain partitioning. Dilworth's dual theorem, establishes the ability to partition the V into no less than $|C_{max}|$ antichains. There exist many ways to partition in $|C_{max}|$ antichains, and the optimal one leads in antichains which all have approximately the same number of elements. The proposed "shuffling" algorithm, refines such a partitioning, to create well balanced antichains. It alters the contents of GB_i's to produce new, valid antichains which have the same or approximately the same elements. Subsection 5.C analyses this technique.

B. Reducing the Communication Delays:

Our objective, in this paper, is the assignment of chains of computations to the same processor so as to reduce the overall communication overhead imposed by the distributed memory target architecture. This means that we assign closely related computations which belong to different time partitions, to be executed into the same processor. Since the total number of communication links, which is the number of edges in the DAG modelling of the index space, is fixed, we try to reduce the overall communication overhead by increasing the intraprocessor communication requirements. In other words, we try to encapsulate into the same processor, computations which are closely related by a dependence vector. Given a node **j**, the proposed heuristic decides the processor $S_{PROC}(\mathbf{j})$, by checking where the $S_{PROC}(\mathbf{i})$ for every $\mathbf{i} \in IN(\mathbf{j})$. This means that *j* will be assigned to the same processor as one or more of its ancestors. This filtering is being iteratively done for every time step (see Fig. 2).

```
for every time instant t≥1
        for every processor p
                find a computation j from
TIME(t) such that:
   Task(p, t-1) belongs to IN(j)
    if (SPROC(j) > p) swap (Task(p, t-1), j)
end
```

Fig. 2: Reducing Communication algorithm

Lemma 5.1: For every **j**: early(**j**)=i≥1, it holds IN(**j**)≠∅

Proof

Suppose that there exists a **j**: early(**j**)=i≥1, and IN(**j**)=∅. Since IN(**j**)=∅, by corollary 3.1 we have **j**∈B, thus early(**j**)=0, which is impossible.

Lemma 5.1 shows that there always exist a computation **j** in the inner loop of the above algorithm. So after O(|V|) steps, in the worst case, the algorithm terminates. Since in every step it interchanges tasks of the same TIME(t), it does not increase total execution time. And finally in every iteration step it may reduce the total amount of interprocessor links E_{comm}, by assigning in the same processor p, nodes **i, j** with **i**∈IN(**j**). **End of Proof**

C. The refinement Technique:

Applying the Longest Path Algorithm (LPA) in O(|V|+|E|) we can trace the length of the longest chain in V. In addition to this, LPA determines for every index point the earliest time instance at which it can be executed. The obtained free time schedule is optimal in terms of time, but not always optimal in terms of processor utilisation. In other words *TP(A)* is not always *OTP(A)*.

The objective of the proposed refinement technique, is to improve the free schedule, by shuffling computations into time, from one antichain to another, so that while *OT(A)* is preserved, antichains have all equal or near equal size. Such scheduling is outlined in the PERT task graph execution problem. In our approach, for every indexed point **i** we compute two properties: *early*(**i**) and *late*(**i**). Since by section IV, a lower bound of *TP(A)* is known, we try to form antichains which have less or equal to this bound length. The key idea is the following:

Consider a node **i** for which *early*(**i**)=k and *late*(**i**)>k. Obviously, since *early*(**i**)=k ,then $\mathbf{i} \in A_k$. If $|A_k| > OTP(A)$ then **i** is a candidate node for shuffling. When checking an anti chain A_k, to reduce its size to *OTP(A)* (equivalently to MA_{max}) we scan all $i \in A_k$ to see which of them have *late*(**i**)>*early*(**i**). Definitely all **i**: *late*(**i**)=*early*(**i**) belong to MA_k and thus are non movable. From all the other nodes, we shuffle those which have the largest difference between *late*(**i**) and k, till the total number of elements in A_k is *OTP(A)*. For all the other shuffled points v_{sh}, we augment *early*(v_{sh})+=1. For every point which belongs to OUT() and check for the next A_{k+1}, where all previously shuffled points now belong. Hence, the algorithm shown in fig. 3 is proposed:

```
for every time instant t≥0
   proc=num_of_processors;
   A: while (proc≠0)
         choose k∈GBt: late(k)-early(k) =
         min{late(j)-early(j): j ∈GBt
         STIME(k) = t; SPROC(k)=proc;
            proc--;
            GBt = GBt-{k};
   B:  for  every i ∈ GBt  then
            early(i)=t+1;
       for every j ∈ DESCENDANT(i):
            early(j)+=1
      GBearly(j)= GBearly(j)-{j}
      GBearly(j)+1= GBearly(j)+1∪{j}
```

Fig. 3: The shuffling algorithm

The above presented SHUFFLING algorithm terminates, since the last checked antichain is $A_{|C_{max}|}$. Its correctness is easily derived by the fact that it shuffles computations into subsequent A_i's, thus preserving time execution validity. In addition to this, it does not increase the minimum makespan as lemma 5.2 proves. The above method was implemented and the results are presented in the following examples. Clearly, there is an impressive improvement in processor utilisation, provided that the algorithm permits it.

Lemma 5.2: The SHUFFLING algorithm preserves the minimum makespan.

Proof (sketch):
The minimum makespan is max $\{S_{TIME}(\mathbf{j}), \mathbf{j}\in T\}$. Let this makespan be t_{opt}. Since $\mathbf{j}\in T$, late($\mathbf{j}$)=long($\mathbf{j_0}$, $\mathbf{j_n}$)=early($\mathbf{j}$)=t_{opt}. Thus, late($\mathbf{j}$)-early($\mathbf{j}$)=0, so the while loop schedules $\mathbf{j}$ at time t_{opt}. Remember that procs≥ |T| by Section 4. This means that no $\mathbf{j}\in GB_{topt}$=T, will be executed by loop B above.
Consider an execution step where there exists a $\mathbf{j}$: late($\mathbf{j}$)-early($\mathbf{j}$)≥1, and we execute the step B, thus increasing early(j). We will show that for any $\mathbf{i}\in$DESCENDANT($\mathbf{j}$) early($\mathbf{i}$) cannot be bigger than $t_{opt.}$ Consider a $\mathbf{j_1}$: $\mathbf{j_1}\in$DESCENDANT($\mathbf{j}$) with early($\mathbf{j_1}$)=t_{opt} (before increasing it by early($\mathbf{j_1}$)+=1). Since $\mathbf{j}\in$ANCESTOR($\mathbf{j_1}$)⇒ early($\mathbf{j}$)-late($\mathbf{j}$)=0, then $\mathbf{j}$ could not have late($\mathbf{j}$)≠early($\mathbf{j}$), which is contradictory to hypothesis.

Optimal Scheduling Steps:

Step 1: From A(J^n, D) form the DAG G(V, E)
Step 2: Apply the topological ordering algorithm (Appendix) to find the *order*($\mathbf{j}$) ∀ $\mathbf{j}\in$V
Step 3: Apply the longest path algorithm (LPA, Appendix) from sources to end and from end to sources, to find early and late for every j in V
Step 4: Find num_of_processors = max{ $MA_{\max}, \left\lceil \dfrac{|J^n|}{|C_{\max}|} \right\rceil$ }, according to section 5A
Step 5: Apply the communication reduction algorithm to create fully connected chains which are assigned to processors.
Step 6: Apply the "shuffling" algorithm (Appendix C) and create equally balanced antichains.
As far as time complexity is concerned, it is clear that steps 1-2 i.e forming the DAG and ordering its nodes topologically have O(|V|+|E|) complexity each. In addition to this, finding longest paths from source and sink nodes costs again O(|V|+|E|). Finally, the shuffling algorithm is in the worst case O($|V|^2$) as it can be easily derived from Appendix C (total time in any UET DAG without communication is clearly O(|V|)).

Example 5.1:

Consider the following algorithm A(J^2,D)

```
For j0 = 0  to 4 do
   For j1 = 0 to 4 do
   a(j0, j1) = a(j0, j1-2) + a(j0-2, j1+1) +
   a(j0-2, j1-2)
   end
end
```

Fig. 4: The algorithm of example 5.1

Clearly $J^2=\{(j_0,j_1) \mid 0\le j_o,j_1\le 4\}$, D = {(0,2), (2, -1), (2,2)}

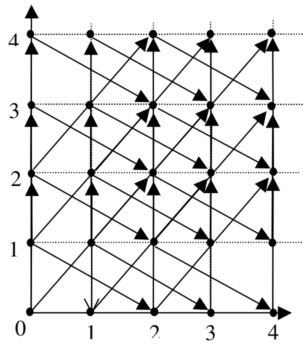

Fig. 5_a: Index space J^2

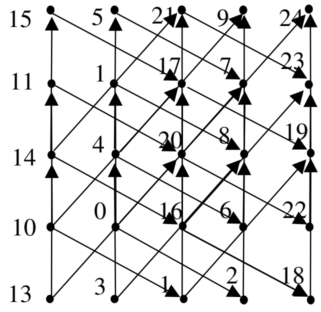

Fig. 5_b: Order of nodes (topol. order, Appendix A)

Table 1: Properties of V (Example 5.1)

order ($\mathbf{j_i}$)	long ($\mathbf{j_0},\mathbf{j_i}$)	long ($\mathbf{j_i},\mathbf{j_n}$)	early ($\mathbf{j_i}$)	late ($\mathbf{j_i}$)	GB
13	0	3	0	2	0
10	0	4	0	1	0
14	1	4	1	1	1
11	1	3	1	2	1
15	2	3	2	2	2
3	0	3	0	2	0
0	0	3	0	2	0
4	1	2	1	3	1
1	1	2	1	3	1
5	2	1	2	4	2
12	1	3	1	2	1
16	2	3	2	2	2
20	2	2	2	3	2
17	3	2	3	3	3
21	3	1	3	4	3
2	1	2	1	3	2
6	2	1	2	4	3
8	2	1	2	4	3
7	3	0	3	5	4
9	3	0	3	5	4
18	3	2	3	3	3
22	3	1	3	4	4
19	4	1	4	4	4
23	4	0	4	5	4
24	5	0	5	5	5

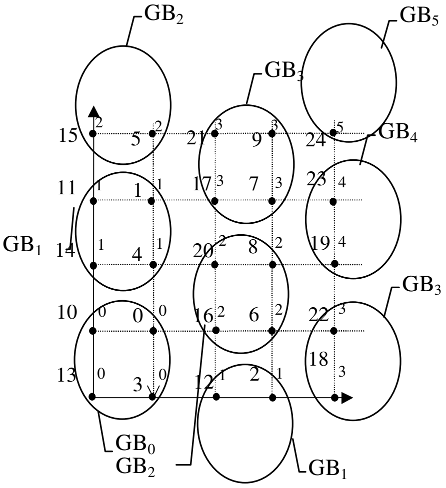

Fig. 6: GB$_i$'s in A(J^2,D)

In the following, the nodes are depicted by their property order(v), as found by topological ordering of V.

Clearly one of the longest chains is {13,14,15,17,19,24} with length 6. Thus at least 6 antichains are formed, for example:

A_0=GB_0={13,10,3,0}→0 A_1=GB_1={14,11,4,1,12,2}→1

A_2=GB_2={15,5,16,20,6,8}→2

A_3=GB_3={17,21,7,9,18,22}→3 A_4=GB_4={19,23}→4

A_5=GB_6={24}→5

Checking for every element v_j of A_i if early(v_j)=late(v_j) ⇒ v_j belongs to chain with max length, the following subsets MA_i are produced (see table 1, Appendix D):

MA_0=∅, MA_1={14}, MA_2={15,16}, MA_3={17,18}, MA_4={19}, MA_5={24} ⇒ $|MA_{max}|$=2.

Clearly, the free schedule produced by the A_i's needs 6 processors and executes the algorithm in optimal time (Table 3). Checking the processor utilisation in table we see that the free schedule has not the best results.

According to lower bounds of *TP(A):* max{ MA_{max}, $\left\lceil \frac{|J^n|}{|C_{max}|} \right\rceil$ }=max{2,5}=5 and TP(A)≤$|A_{max}|$ = 6.

Therefore 5 processors are adequate to execute A in optimal time. Communication costs are cut down by applying the filter of Section 5.B. as shown in table 4. Applying the "shuffling" algorithm of Section 5C to V, the schedule shown in table 5, optimal in terms of time and much more efficient in number of processors, is produced.

Table 2: Optimal linear schedule

	t_0	t_1	t_2	t_3	t_4	t_5
P_1	13	14	15	17	9	24
P_2	10	11	1	21	19	
P_3	3	0	5	8	23	
P_4		4	16	7		
P_5		12	20	18		
P_6			2	22		
P_7			6			

Table 3: free schedule

	t_0	t_1	t_2	t_3	t_4	t_5
P_1	13	14	15	17	19	24
P_2	10	11	5	21	23	
P_3	3	4	16	7		
P_4	0	1	20	9		
P_5		12	6	18		
P_6		2	8	22		

Table 4: Communication schedule

	t_0	t_1	t_2	t_3	t_4	t_5
P_1	13	14	15	17	19	24
P_2	10	11	20	21	23	
P_3	3	4	5	7		
P_4	0	1	8	9		
P_5		12	5	18		
P_6		2	16	22		

Table 5: Optimal "refined" schedule

	t_0	t_1	t_2	t_3	t_4	t_5
P_1	13	14	15	17	19	24
P_2	10	11	20	21	22	23
P_3	3	4	5	6	7	
P_4	0	1	16	18	9	
P_5		12	2	8		

By applying all equivalent methods presented in [1], [2], [4] or [5], the best, in terms of total execution time, linear schedule is given by Π = [3 2]. This linear schedule achieves the same total execution time. The main disadvantage of this linear schedule, is the poor processor utilisation that implies. The linear schedule, needs too many processors in order to be completed in the less possible time. Table 2 shows the linear scheduling achieved by Π.

Clearly we need at least 7 processors to execute this time schedule, whereas the free one requires 5 processors. Table 4 compares the processor utilisation for the three schedules. Finally table 7 shows a considerable decrease (36%) over the total communication requirements of the optimal linear and free schedule.

Table 6: Comparison of utilisation for the three schedules

time instance	% util. for optimal linear schedule	% util. for free schedule	% util. for optimal "refined" schedule
0	3/7	4/6	4/5
1	5/7	6/6	5/5
2	7/7	6/6	5/5
3	6/7	6/6	5/5
4	3/7	2/6	4/5
5	1/7	1/6	2/5
average utilisation	59.52%	69.44%	83.33%

Table 7: Comparison of communication for the three schedules

	optimal linear schedule	free schedule	optimal "refined" schedule
CCF	31	28	19

VI. CONCLUSION

In this paper a new optimal in terms of time and efficient in terms of communication overhead and processor utilisation, technique was presented to schedule multidimensional nested loops with loop-carried dependencies. It is obvious that such a method is general enough, and without any extensions, it can be applied to non-uniform nested loops as well. The proposed algorithm was implemented and exhaustively tested, and the outperforming results were reported. Future work includes the introduction of methods to avoid maximal path's complexity for the general class of uniform dependence loops, by exploiting the regularity of the index space. In [2] we have presented a method for unit uniform dependence loops which achieves low complexity. In addition to this, we will elaborate on the extension of the above methodology, in order to schedule in a different way every distinct assignment statement, instead of the more general loop instances as in affine by statement scheduling.

APPENDIX

The "Shuffling" algorithm

```
T=V;
  for i = 0 to max_time
  begin
     proc=num_of_processors;
     for every v_j ∈ T with early(v_j) = i
     begin
  while (proc≠0)
       {
  choose v_k :
  late(v_k)-early(v_k) = min{late(v_j)-
  early(v_j: v_j ∈ T ∧early(v_j)=i}
        S_TIME(v_k) = i
        S_PROC(v_k)=proc;
        TIME(i)=TIME(i)∪{v_k};
      /*set of nodes of time t */
     P(proc)=P(proc)∪{v_k};
     /*set of nodes for proc processor */
        T(proc, i)=v_k;      /*Task executed
          in i time by proc processor */
        proc--
        T = T-{v_k}
       }
        end
        for  every v_i ∈ T: if early(v_i)=i
  then {
  early(v_i)=i+1;
for every v_j ∈ OUT(v_i): early(v_j)+=1
}
end
end
```

The comm. filter algorithm

```
for j = 1 to max_time
begin
   for i = 1 to max_proc
   begin
      find v_x ∈ TIME(j): TASK(i,j-1)∈IN(v_x);
      if (S_PROC(v_x)>i) swap(T(i,j),v_x);
   end
end
```

Longest Path in O(|V|+|E|)

```
Begin
      P=∅;
      ∀ v∈V label(v)=order(v) from topological
         ordering
      Long_Path(v)=-∞ for all v∈V;
      Long_Path(s)=0 for all s∈B∩V;
   for i=1 to |V|
         begin
         let v∈V such that order(v)=i;
         for every u ∈ Adj(v) do
   if (Long_Path(u)<Long_Path(v)+c_uv) then
   Long_Path(u)=Long_Path(v)+c_uv;
   end;
In our case c_uv=1.
```

Topological Ordering Algorithm in O(|V|+|E|)

```
V  set of nodes of G(V,E);
E set of edges of G(V,E);

begin
for every v∈V do indegree(v)=0;
      for every (v,u) ∈ E do
         indegree(u)=indegree(u)+1;
      /* we have already found  ∀ v∈V
         indegree(v) */
      /* We will found order(v)(topological
         sorting) */
      LIST = ∅ ;
      next = 0;
      for every v∈V do
      if ( indegree(v)=0 ) then
      LIST = LIST ∪ {v};
      while ( LIST≠∅ )
      begin
      select a node v∈LIST;
      LIST=LIST-{v};
      next+=1;
      order(v)=next;
      for every u ∈ Adj(v) do
      begin
      indegree(u)=indegree(u)-1;
      if (indegree(u)=0) then
      LIST = LIST ∪ {u};
      end
         end;

         if (next < |V|) then G  has a cycle
         thus ERROR

         else label(v)=order(v) ∀ v∈V
```

REFERENCES

1. T. Andronikos, N. Koziris, Z. Tsiatsoulis, G. Papakonstantinou and P. Tsanakas, "Lower Time and Processor Bounds for Efficient Mapping of Uniform Dependence Algorithms into Systolic Arrays," *Journal of Parallel Algorithms and Applications,* vol. 10, no. 3-4, 1997

2. T. Andronikos, N. Koziris, G. Papakonstantinou and P. Tsanakas, "Optimal Scheduling for UET-UCT Generalized n-Dimensional Grid Task Graphs," *Proceedings of the 11th IEEE/ACM International Parallel Processing Symposium (IPPS97), pp 146-151, Geneva, Switzerland.*

3. P. Chretienne "Task Scheduling with Interprocessor Communication Delays," *Eur. J. Oper. Res.*, vol. 57, pp 348-354, 1992.

4. A. Darte, Y. Robert, "Mapping Uniform Loop Nests onto Distributed Memory Architectures," *Parallel Computing,* vol. 20, pp 679-710, 1994.

5. A. Darte and Y. Robert, "Constructive Methods for Scheduling Uniform Loop Nests," *IEEE Trans. Parallel Distrib. Syst*., vol. 5, no. 8, pp. 814-822, Aug. 1994.

6. M. R. Garey and D. S Johnson, "Computers and Intractability, a Guide to the Theory of NP-completeness," *W. H. Freeman and Company: New York, 1979.*

7. E. H. Hollander, "Partitioning and Labeling Loops by Unimodular Transformations," *IEEE Trans. on Parallel and Distributed Systems*, vol. 3, no. 4, pp. 465-476, Jul. 1992.

8. H. Jung, L. Kirousis and P. Spirakis "Lower Bounds and Efficient Algorithms for Multiprocessor Scheduling of DAGS with Communication Delays," *Proceedings of 1st ACM SPAA 1989, pp. 254-264.*

9. N. Koziris, G. Papakonstantinou and P. Tsanakas "Optimal Time and Efficient Space Free Scheduling For Nested Loops," *The Computer Journal*, vol. 39, no. 5, pp 439-448, 1996.

10. N. Koziris, G. Papakonstantinou, and P. Tsanakas, "Mapping Nested Loops onto Distributed Memory Multiprocessors," in *Proceedings of the 1997 IEEE International Conference on Parallel and Distributed Systems,* IEEE Press, Korea, Dec.1997.

11. L. Lamport, "The parallel execution of DO loops," *Commun. ACM*, vol.17, no.2, pp. 83-93, Feb. 1974.

12. D.I. Moldovan and J.A.B. Fortes, "Partitioning and Mapping Algorithms into Fixed Size Systolic Arrays," *IEEE Trans. Comput*., vol C-35, no 1, pp. 1-11, Jan. 1986.

13. C. Papadimitriou and M. Yannakakis, "Toward an Architecture-Independent Analysis of Parallel Algorithms," *SIAM J. Comput., vol. 19, pp. 322-328, 1990, Extended Abstract in Proceedings STOC 1988.*

14. W. Shang and J.A.B. Fortes, "Independent Partitioning of Algorithms with Uniform Dependencies," *IEEE Trans. Comput*., vol. 41, no. 2, pp. 190-206, Feb. 1992.

15. W. Shang and J.A.B. Fortes, "Time Optimal Linear Schedules for Algorithms with Uniform Dependencies," *IEEE Trans. Comput*., vol. 40, no. 6, pp. 723-742, June 1991.

16. W. Shang, E. Hodzic and Z. Chen, "On Uniformization of Affine Dependence Algorithms," *IEEE Trans. on Parallel Distributed Systems*, vol. 45, no. 7, pp. 827-840, July 1996.

17. J.-P. Sheu and T.-H. Tai, "Partitioning and Mapping Nested Loops on Multiprocessor Systems," *IEEE Trans. on Parallel Distributed Systems*, vol. 2, no. 4, pp. 430-439, Oct. 1991.

18. T. Tzen and L. Ni, "Dependence Uniformization: A Loop Parallelization Technique," *IEEE Trans. On Parallel Distributed Systems*, vol. 4, no. 5, pp. 547-558, May 1993.