Available online at www.sciencedirect.com
SCIENCE DIRECT®
J. Parallel Distrib. Comput. 64 (2004) 578–590
Journal of Parallel and Distributed Computing
http://www.elsevier.com/locate/jpdc

# Data dependent loop scheduling based on genetic algorithms for distributed and shared memory systems

Jose L. Aguilar[a,*] and Ernst L. Leiss[b]

[a] CEMISID. Dpto. de Computación, Facultad de Ingeniería, Universidad de los Andes, Merida 5101, Venezuela
[b] Department of Computer Science, University of Houston, Houston, TX 77204-3010, USA

Received 18 December 2000; revised 10 June 2003

**Abstract**

Many approaches have been described for the parallel loop scheduling problem for shared-memory systems, but little work has been done on the data-dependent loop scheduling problem (nested loops with loop carried dependencies). In this paper, we propose a general model for the data-dependent loop scheduling problem on distributed as well as shared memory systems. In order to achieve load balancing and low runtime scheduling and communication overhead, our model is based on a loop task graph and the notion of critical path. In addition, we develop a heuristic algorithm based on our model and on genetic algorithms to test the reliability of the model. We test our approach on different scenarios and benchmarks. The results are very encouraging and suggest a future parallel compiler implementation based on our model.


*Keywords:* Loop scheduling; Loop-carried dependence; Parallel algorithms; Genetic algorithms; Performance optimization

## 1. Introduction

Generally, loops are the richest source of parallelism in parallel applications. One way to exploit this parallelism is to execute loop iterations in parallel on different processors, thereby reducing the running time. Consider the case in which a loop of $m$ iterations is executed in a multiprocessor system with $P$ processors. The goal of scheduling is to distribute these $m$ iterations to $P$ processors in the most equitable manner and with the least amount of overhead. In the case where there are no data dependencies between tasks in different iterations (i.e., parallel loops) there is no need for synchronization. If there are data dependencies between tasks in different iterations, the communication delay may slow down the execution. Previous approaches have attempted to achieve the minimum completion time for the parallel loop scheduling problem only by distributing the workload as evenly as possible while minimizing the number of synchronization operations required and the communication overhead caused by access to non local data on shared-memory systems [2,3,5,6,11,18]. Other authors have studied the parallelism across iterations to consider loop carried dependencies, proposing different techniques to improve this parallelism [4,8,12–14,16,17]: cyclo-compaction scheduling, loop pipelining, etc.

In this paper, we study the problem of scheduling a set of $n$ nested loops, with data dependencies among the loops/iterations (that is, with loop carried dependencies), on distributed or shared memory machines. Our technique falls in the static scheduling and software pipelining category. In the presence of data dependencies between tasks in different iterations, we need a better representation than the traditional task graph model to be able to represent the data dependencies. We represent the data dependencies among tasks in loops using a more specific type of task graph, the *loop task graph*, whose nodes represent the tasks on different iterations and whose arcs represent the dependence relationship between tasks. We also need a scheduling approach that can exploit parallelism within each iteration and among different loop iterations. We solve this problem using the loop unrolling technique and the critical path concept [1,5]. The basic idea is to unroll the loop to allow several iterations and tasks in the same

*Corresponding author. Fax: +587-440-2872.
*Email-addresses:* aguilar@cemisid.ing.ula.ve (J.L. Aguilar), coscel@cs.uh.edu (E.L. Leiss).

0743-7315/$ - see front matter © 2004 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2004.03.004

iteration to overlap in execution in a way that minimizes the loop execution time. Then, we introduce a model based on the notion of *critical path* to schedule this loop task graph. The critical path is the minimum execution time of the program modeled by the loop task graph. The execution time will always be at least as large as the critical path when the number of processors is finite rather than unlimited. Our general approach for the data dependent loop scheduling problem, on distributed as well as shared memory systems, achieves load balancing, low runtime scheduling and communication overhead, based on the loop task graph, the unrolling technique and the notion of critical path. Finally, because data dependent loop scheduling is NP-hard, we propose a heuristic algorithm based on our model and on genetic algorithms to test the reliability of our approach. We define a set of specific genetic operators in order to implement an efficient search on the solution space for this problem.

The organization of this paper is as follows: Section 2 presents the data-dependent loop-scheduling problem. Section 3 presents the main ideas of our model. Section 4 presents our heuristic algorithm based on genetic algorithms. Section 5 reports our experiments. Then, we present our conclusions and further work.

## 2. The data-dependent loop scheduling problem

The term "scheduling" has sometimes been used loosely in the literature, with different interpretations in different application domains. Loop scheduling can be viewed as part of the general problem of scheduling tasks on multiprocessor systems so as to minimize the completion time of parallel applications [5,9]. In this context, loop scheduling is analogous to process scheduling, which also has been studied extensively. Process scheduling is concerned with many of the same issues involved in loop scheduling, including concerns about load imbalance, synchronization overhead, and communication overhead. But, completion time is not always the only target when making a schedule. For example, power can also be a factor to consider; so can be memory access constraints. We do not consider these aspects in this work.

In general, an efficient loop scheduling algorithm should optimize its performance by trading off adaptively scheduling overhead (synchronization overhead, loop allocation overhead, scheduler execution time overhead, etc.), load imbalance overhead, and data communication overhead. Many approaches have been proposed for the parallel loop scheduling problem for shared-memory systems [2,3,5,6,11,18]. All of these parallel loop scheduling algorithms attempt to achieved the minimum completion time by distributing the workload as evenly as possible, by minimizing the number of synchronization operations required, or by minimizing communication overhead caused by access to non-local data. Each of the algorithms assumes that we work on a shared-memory system. Many other works have studied the parallelism across iterations to consider the loop carried dependencies [4,8,12–14,16,17]. Refs. [12,14] present a technique, which is applied to a communication sensitive data flow graph that represents a nested loop. This technique takes into account the data transmission time, the loop carried dependencies, and the target architecture to schedule the set of nodes of the graph. It implicitly uses the loop pipelining technique and a task remapping procedure to allocate nodes. In [4] a method is proposed combining the loop pipelining technique with data prefetching, called partition scheduling with prefetching. In this algorithm, the iteration space is first divided into regular partitions. Then a two-part scheduler is used to do load balancing and to produce high throughput. Another interesting approach was proposed by Passos and Sha [8], namely a retiming technique applied to get optimal execution rates in parallel and/or pipeline systems. It is a common transformation tool in one dimensional problems, when loops are modeled as multidimensional data flow graphs (MDFGs). They prove that they can obtain full-parallelism for MDFGs with more than one dimension. They extend their work to schedule data flow graphs with conditional branches. In general, these techniques do not consider the communication costs or the memory latencies, or cannot exploit pipelining across loop boundaries.

There are two basic loop scheduling methods used to assign iterations of a loop to processors [2,5]:

- *Static scheduling*: Static policies depend on the average behavior of the system and not on its current state; they are usually applied to uniformly distributed loops. They assign iterations to processors statically, minimizing run-time synchronization overhead but do not always balance the load. The simplest static scheduling algorithm assigns a given number of loop iterations among the available processors as evenly as possible, in the hope that each processor receives about the same amount of work. If all tasks do not take the same amount of time, load imbalances may arise, which will cause some processors to be idle while other processors continue to execute loop iterations. Thus, workload imbalance is its major disadvantage.
- *Dynamic scheduling:* Whenever a processor becomes available, one or more iterations will be assigned to that processor. In these algorithms, the processors maybe exchange load information (only in distributed queues) and iterations periodically. These methods can achieve better load balancing in the presence of unpredictable transient loads and

non-uniformly distributed loops. Clearly, some scheduling overhead will be involved at run time.

### 2.1. Optimization criteria

The classic optimization criteria are [2,5,18]:

- *Minimize loop imbalance:* In this case, the idea is to distribute the workload as evenly as possible. We can minimize it by favoring fine grained allocation of loop iterations in order to minimize the effects of uneven assignment.
- *Minimize communication cost:* The communication cost is a main factor. In the case of shared memory some memory may be not equidistant from all processors (such as cache memory). In the case of distributed memory the existence of local memory implies that some processors are closer to the data required by an iteration than others. We can minimize this cost by exploiting the processor affinity that favors the allocation of loop iterations close to their data.
- *Minimize synchronization problems:* In this case, we must minimize the waiting time of the processors due to dependent iterations assigned to different processors.
- *Minimize scheduling overhead:* This is the time to allocate the remaining iterations. Normally, we include the synchronization overhead in this cost.

### 2.2. Loop task graph

In general, data dependence is a consequence of the flow of data in a program. A task that uses a variable in an expression is data dependent upon the task that computes the value of the variable. Data dependence in loops can be classified as [2,5]:

- *Loop carried dependence*: when data are passed between different iterations.
- *Loop independent dependence*: when data are passed from one task to another within the same iteration.

Clearly, loop-independent dependencies can be easily represented using task graphs, but loop carried data dependencies are more difficult to represent in task graphs since they express dependencies among tasks in different loop iterations. We first give the following definition [5]:

- *Iteration vector*: When some tasks are contained in $n$ nested loops, we refer to separate instances of their execution using an iteration vector. A vector $I = \{i_1, i_2, \ldots, i_n\}$ is called an iteration vector if the loop body is executed in the period when the $j$th level loop is in the $i_j$th iteration, $1 \leqslant j \leqslant n$. Elements of iterations vectors are numbered from outermost to innermost, as are the loops.
- *Distance vector* ($D_{wv}$): Using iteration vectors, we can define a distance vector for each dependence between tasks. Suppose that $v$ and $w$ are two tasks enclosed in $n$ nested loops. If $w$ during the iteration identified by iteration vector $I_w$ is dependent on $v$ during the iteration identified by iteration vector $I_v$, the distance vector for this dependence is $D_{wv} = I_w - I_v$. Task $w$ is loop carried data dependent on task $v$ iff $I_w \neq I_v$. Otherwise, the dependence is called loop-independent ($D_{wv} = 0$).
- *Dependence pair*: We define a dependence pair between two tasks $w$ during the iteration $I_w$ and $v$ during the iteration $I_w$, as $(D_{wv}, W_{wv})$, where $D_{wv} = I_w - I_v$ is the distance vector and $W_{wv}$ is the size of the message that $w$ receives from $v$. Task $w$ can have more than one data dependence pair from task $v$.
- *Dependence set* ($DDS_{wv}$): The set of all dependence pairs between two tasks.
- *Upper bound vector*: For $n$ nested loops, it is $\{b_1, b_2, \ldots, b_n\}$, where $b_i$ is the upper bound of the loop at the $i$th level.
- *Unrolling vector:* If it is equal to $\{u_1, \ldots, u_n\}$, this means that the $i$th loop is unrolled $u_i$ times.

We assume that loops are normalized to iterate from 1 to some upper bound in steps of 1. We also assume perfect (tightly) nested loops, which means all the tasks are enclosed in the innermost loop. Methods of transforming non-normalized loop into normalized loop are presented in [5]. Some of the methods of transforming imperfectly nested loop into perfect ones are [5]: loop distribution and non-basic-to basic loop transformation. The last assumption implies that all distance vectors have only non-negative values which guarantees an acyclic loop-carried data dependence graph.

#### 2.2.1. Definition of a loop task graph

Loop carried and loop-independent data dependencies among a set of tasks that is enclosed in a nested loop can be represented using a *loop task graph* [5]. The graph is a weighted directed graph $G = (V, A)$ where $V$ is a set of nodes and $A$ is a set of arcs. A node in the *loop task graph* represents a task, and an arc between two nodes $u$ and $v$ represents the dependence between $u$ and $v$. There are weights on the nodes and the arcs. The weight on a node represents the amount of computation of the task represented by this node. The weight on an arc $(u, v)$ is the dependence set between tasks $u$ and $v$. The loop task graph may contain cycles. To illustrate these concepts, consider the following toy example which will be used throughout this section since its small size allows us to work out the details explicitly:

*For* $i = 1$ *to* 2
  *For* $j = 1$ *to* 2

$T1(i,j)$
$T2(i,j)$
where

Task $T1(i,j)$:
$X[i,j] = F1(V[i-1,j], X[i-1,j-1])$
$Z[i,j] = \text{constant1}$
$V[i,j] = \text{constant2}$

Task $T2(i,j)$:
$Y[i,j] = F2(Z[i,j], Y[i-1,j])$

Clearly, the data dependence in the above code segment can be represented using a loop task graph with only two nodes $(T1, T2)$. The weights on the nodes can be computed as follows: assume the amount of computation of the functions $F1$ and $F2$ are 4 and 8, respectively, and that an assignment statement takes 1 unit of computation. In this way, the amount of computation of task $T1$ is 7, and of task $T2$ is 9 (the weight of tasks $T1$ and $T2$). In order to obtain the weights on the arcs of the loop task graph, we assume that the arrays $X, Y, Z$, and $V$ take 10, 12, 20 and 5 units of storage, respectively. Task $T1$ uses during iteration $\langle i,j\rangle$ $V[i-1,j]$ which is computed by task $T1$ during iteration $\langle i-1,j\rangle$, and $X[i-1,j-1]$ which is computed by task $T1$ during iteration $\langle i-1,j-1\rangle$. The distance vectors of these two dependences from task $T1$ to itself are $\langle 1,0\rangle(\langle i,j\rangle - \langle i-1,j\rangle)$ and $\langle 1,1\rangle(\langle i,j\rangle - \langle i-1,j-1\rangle)$, respectively. According to the size of an element in $V$ and $X$, the dependence set from $T1$ to itself is equal to $\{(\langle 1,0\rangle, 5), (\langle 1,1\rangle, 10)\}$. In the same way, we can calculate the arcs for task $T2$, in one case, because during iteration $\langle i,j\rangle T2$ uses $Z[i,j]$ (the dependence set from $T1$ to $T2$ is equal to $\{(\langle 0,0\rangle, 20)\}$), and in the other case because it uses $Y[i-1,j]$ (the dependence set from $T2$ to itself is equal to $\{(\langle 1,0\rangle, 12)\}$).

### 2.2.2. Loop unrolling

This is a process of replacing the iterations of a loop with non-iterated straight-line code [5]. The basic idea is to unroll the loop in order to uncover loop-carried dependencies that allow several iterations to overlap in execution. For example, the loop:

*For* $i = 1$ *to* 4
$X[i+2] = X[i+1] + X[i]$
is unrolled once to

*For* $i = 1$ *to* 4 *step* 2
$X[i+2] = X[i+1] + X[i]$
$X[i+3] = X[i+2] + X[i+1]$

Unrolling reveals independent iterations that we can group together in a manner that allows some parallelism. The resulting groups can be scheduled to take advantage of the available parallelism. When a single loop with upper bound $b$ is unrolled $u$ times, $u+1$ copies of the body are replicated, the loop control variable is adjusted for each copy, and the step value of the loop is multiplied by $u+1$. Similarly, when a set of $n$ nested loops with upper bound vector $\{b_1, \ldots, b_n\}$ is unrolled using unrolling vector $u = \{u_1, \ldots, u_n\}$, $\prod_{i=1}^{n}(u_i+1)$ copies of the body are replicated, the loop variables are adjusted for each copy, and the step value of the $i$th loop is multiplied by $u_i + 1$.

### 2.2.3. Replicated task graphs

Given a loop task graph $G = (V, A)$, we define a *replicated task graph* $G_u = (V_u, A_u)$ as follows [4]: the graph $G_u$ is an acyclic task graph that represents the body of the loop after being unrolled using unrolling vector $u = \{u_1, \ldots, u_n\}$. The set of nodes $V_u$ is the set of replicated tasks according to the unrolling process $(|Vu| = |V| * \prod_{i=1}^{n}(u_i+1))$. The set of arcs is the set that represents the original loop-independent dependencies (these are the arcs whose weights (dependence sets) have distance vectors with zero elements in $G$, for example, the arc between node $T1$ and node $T2$ in Fig. 1) and loop-carried dependencies that have become loop independent as a result of the unrolling. The weight of a replicated node remains the same as its original node. The weight on an arc in $G_u$ is the size of the message portion of the dependence pair of the dependence set on the original arc. Fig. 2 shows the replicated task graph $G_u$ for our example of Fig. 1, where $u = \{1,0\}$, i.e., the outer loop is unrolled once.

Now, if we suppose that $u = \{1,1\}$ (the completely unrolled loop), the new graph is (see Fig. 3)

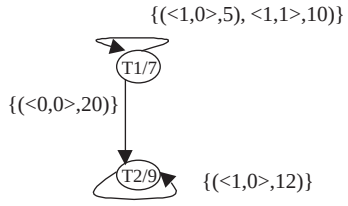


Fig. 1. A loop task graph.

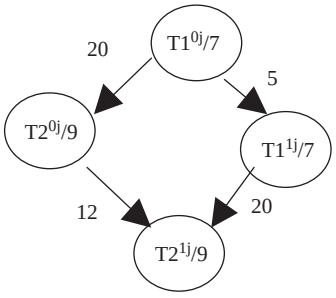


Fig. 2. Replicated task graph of the loop task graph of Fig. 1 with $u = \{1,0\}$ and $j = 0, 1$.

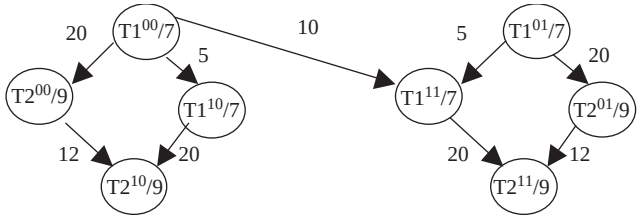


Fig. 3. Replicated task graph of the loop task graph of Fig. 1 with $u = \{1, 1\}$.

#### 2.2.4. Critical path of a task graph

A critical path of a graph is the longest path in the task graph [1]. The critical path is the minimum execution time of a program modeled by a task graph. The execution time will always be at least as large as the critical path when the number of processors is finite rather than unlimited.

## 3. Our scheduling loop task graph approach

Because revealing loop-carried dependencies may allow several iterations of a set of loops to overlap in execution, loop unrolling can help to exploit the parallelism that may exist among different iterations. Because unrolling entails unpleasant space/time complexities, this is a decision that should be made at compile time. One simple but usually impractical way to schedule a loop task graph is as follows:

1. Calculate the complete loop unrolling $u = \{b_1 - 1, \ldots, b_n - 1\}$ for the loop task graph $G$ in order to determine the set of $M$ tasks on the loop ($\mathrm{TS} = \{T_1^{0\ldots0}, \ldots, T_N^{0\ldots0}, \ldots, T_1^{b_1-1\ldots b_n-1}, \ldots, T_N^{b_1-1\ldots b_n-1}\}$), where $N$ is the set of tasks on the $n$ nested loops ($N = |V|$) and $M$ is the set of tasks to be allocated ($M = |V|^* \prod_{i=1}^{n} b_i$).
2. Each task on the loop is assigned to a ready queue: a ready queue is initialized for ready tasks by inserting every task that has no immediate predecessor.
3. As long as the ready queue is not empty do the following ($M$ times):
   (3.1) Obtain a task from the front of the queue.
   (3.2) Select an idle processor to run the task.
   (3.3) When all the immediate predecessors of a particular task are executed, their successor is ready to be inserted into the ready queue.

The time complexity of this algorithm is $O(M)$ and the maximum size of the ready queue is $M$ (for the case of a parallel loop). This scheduler does not give a good scheduling because it does not consider the communication cost between different tasks. Clearly, the loop execution time will depend on the scheduling, since all tasks which have been assigned to the same processor will execute sequentially, and some tasks on different processors will also have to execute sequentially due to the precedence relations. The exact computation of this execution time is not trivial. However an optimistic estimate can be obtained. We redefine step (3.1) of our scheduler according to the following ideas based on the critical path notion: In general, to obtain a good scheduling we need to minimize the execution time of each task of the loop. The sequential execution time of a nest of $n$ loops is

$$T_s = \sum_{I^1=0}^{b_1-1} \cdots \sum_{I^n=0}^{b_n-1} \sum_{j=1}^{N} T_j^{I^1 \ldots I^n}$$

assuming that all $M$ tasks reside on the same processor and exchange information through common memory (which is assumed to incur no additional communication cost). The parallel execution time ($T_p$) of the loop on $K$ processors, according to a given assignment $\prod(g)$ of the tasks of the loop, where $g$ is one of the possible assignments of the tasks ($g = 1, \ldots, K^M$, where $K$ is the number of processors), is the earliest time at which a given task $T_j^{I^1 \ldots I^n} \in \mathrm{TS}$ will terminate.

$$T_p\left(\prod(g)\right) = \max_{T_j^{I^1 \ldots I^n} \in TS} \left\{ T_j^{I^1 \ldots I^n}\left(\prod(g)\right) \right\} \quad \forall j = 1, \ldots, N,$$

$$\forall I^1 = 1, \ldots, b^1, \ldots, \ \forall I^n = 1, \ldots, b^n.$$

This is the best possible execution time of the loop with the assignment $\prod(g)$. The specific assignment $\prod(g)$ of the tasks of the loop ($\prod(g) = \{\prod_1^{0\ldots0}(g), \ldots, \prod_N^{b_1-1\ldots b_n-1}(g)\}$) is defined as

$$\prod_j^{I^1 \ldots I^n}(g) = l \quad \text{if task } T_j^{I^1 \ldots I^n} \text{ is allocated to processor } l.$$

The instant at which $T_j^{I^1 \ldots I^n}$ will terminate, according to the current assignment $\prod(g)$, can be defined as follows:

$$\begin{aligned} T_j^{I^1 \ldots I^n}\left(\prod(g)\right) = T_j^{I^1 \ldots I^n} &+ \max\left\{ X_i^{I'^1 \ldots I'^n}\left(\prod(g)\right) \right\} \\ &+ \Bigg| \max_{T_i \in R} \left\{ T_i^{I'^1 \ldots I'^n}\left(\prod(g)\right) \right. \\ &\left. + \lambda_{ij}^{I^1 \ldots I^n, I'^1 \ldots I'^n}\left(\prod(g)\right) \right\} \\ &- \max\left\{ X_i^{I'^1 \ldots I'^n}\left(\prod(g)\right) \right\} \Bigg|_{\text{if(COND=true)}}. \end{aligned} \tag{1}$$

The first expression of the previous equation is the execution time of the task $T_j$. The second one is the completion time of the last task that has been executed on processor where $T_j$ will start its execution (it is the maximum completion time between the tasks already executed on this processor). The last expression represents the relationship among $T_j$ and its predecessors. $T_j$ can not start before its predecessors. COND determines the overhead due to the communication time of the predecessor task of $T_j$ that has finished last when this time plus its completion time is bigger than the

completion time of the last task executed on the same processor where $T_j$ will start its execution. Specifically, $X_i^{I'^1 \ldots I'^n}(\prod(g))$ is a vector that gives the completion time of the tasks already executed on the processor where $T_j$ has been allocated.

$$X_i^{I'^1 \ldots I'^n}\left(\prod(g)\right) = \begin{cases} T_i^{I'^1 \ldots I'^n}(\prod(g)) & \text{if } \prod_i^{I'^1 \ldots I'^n}(g) = \prod_j^{I^1 \ldots I^n}(g), \\ 0 & \text{otherwise,} \end{cases}$$

$R$ defines the data dependence between the tasks $T_j^{I^1 \ldots I^n}$ and $T_i^{I'^1 \ldots I'^n}$:

$$R = \{B^{I^1 \ldots I^n, I'^1 \ldots I'^n} \in \mathrm{DDS}_{ij}\} \quad \text{and} \quad B^{I^1 \ldots I^n, I'^1 \ldots I'^n} = \langle I^1 - I'^1, \ldots, I^n - I'^n \rangle$$

$B^{I^1 \ldots I^n, I'^1 \ldots I'^n}$ is the distance vector between $T_j^{I^1 \ldots I^n}$ and $T_i^{I'^1 \ldots I'^n}$. We search for this vector in the dependence set of these tasks ($\mathrm{DDS}_{ij}$). If we find this vector, that means $T_i^{I^1 \ldots I^n}$ must be executed before $T_j^{I^1 \ldots I^n}$. In this case, we need to consider the communication time between them if they are allocated on different processors. $\lambda_{ij}^{I^1 \ldots I^n, I'^1 \ldots I'^n}$ is the communication time when two tasks are allocated on different processors, assuming that tasks residing on the same processor incur no communication cost,

$$\lambda_{ij}^{I^1 \ldots I^n, I'^1 \ldots I'^n}\left(\prod(g)\right) = \begin{cases} \sum_{B^{I^1, \ldots, I^n, I'^1, \ldots, I'^n} \in \mathrm{DDS}_{ij}} W_{ij} & \text{if } \prod_j^{I^1 \ldots I^n}(g) \neq \prod_i^{I'^1 \ldots I'^n}(g), \\ 0 & \text{if } \prod_j^{I^1 \ldots I^n}(g) = \prod_i^{I'^1 \ldots I'^n}(g). \end{cases}$$

In this case, we need to sum the size of the messages ($W_{ij}$) of the different dependence pairs with the distance vector $B^{I^1 \ldots I^n, I'^1 \ldots I'^n}$ that belong to the dependence set of these tasks ($\mathrm{DDS}_{ij}$). Finally, we need to calculate the overhead due to the communication time between these tasks when the next condition is true

$$\begin{aligned} \mathrm{COND} = \max_{T_i^{I^1 \ldots I^n} \in R} & \left\{ T_i^{I^1 \ldots I^n}\left(\prod(g)\right) + \lambda_{ij}^{I^1 \ldots I^n, I'^1 \ldots I'^n}\left(\prod(g)\right) \right\} \\ & > \max\left\{ X_i^{I'^1 \ldots I'^n}\left(\prod(g)\right) \right\}. \end{aligned}$$

Ideally, we want to choose an assignment $\prod(g)$ which minimizes $T_p$. For our problem, we define the objective function as follows:

$$\begin{aligned} \text{Cost Function} &= \min_g \left\{ T_p\left(\prod(g)\right) \right\} \\ &= \min_g \left\{ \max_{T_j^{I^1 \ldots I^n} \in ST} \left\{ T_j^{I^1 \ldots I^n}\left(\prod(g)\right) \right\} \right\} \\ &\quad \forall g = 1, \ldots, L. \end{aligned} \tag{2}$$

We have redefined the data dependent loop scheduling problem as a *min–max problem* where $L$ is the number of the possible different assignments of the tasks ($L = K^M$). The complexity of this expression is $O(M^4)$. The next table summarizes the set of formulae:

| | |
|---|---|
| $\mathrm{TS} = \{T_1^{0\ldots0}, \ldots, T_N^{0\ldots0}, \ldots, T_1^{b_1-1\ldots b_n-1}, \ldots, T_N^{b_1-1\ldots b_n-1}\}$ | The set of $M$ tasks on the loop after calculating the complete loop unrolling |
| $T_s = \sum_{I^1=0}^{b_1-1} \cdots \sum_{I^n=0}^{b_n-1} \sum_{j=1}^{N} T_j^{I^1 \ldots I^n}$ | The sequential execution time of a nest of $n$ loop |
| $\prod(g) = \{\prod_1^{0\ldots0}(g), \ldots, \prod_N^{b_1-1\ldots b_n-1}(g)\}$ | A given assignment of the tasks of the loop, where $g$ is one of the possible assignments of the tasks |
| $\prod_j^{I^1 \ldots I^n}(g)$ | Variable for assigning $T_j^{I^1 \ldots I^n}$ |
| $T_p(\prod(g)) = \max_{T_j^{I^1 \ldots I^n} \in TS} \{T_j^{I^1 \ldots I^n}(\prod(g))\}$ | The parallel execution time of the loop on $K$ processors |
| $T_j^{I^1 \ldots I^n}(\prod(g)) = T_j^{I^1 \ldots I^n} + \max\{X_i^{I'^1 \ldots I'^n}(\prod(g))\} + \left\| \max_{T_i \in R}\{T_i^{I'^1 \ldots I'^n}(\prod(g)) + \lambda_{ij}^{I^1 \ldots I^n, I'^1 \ldots I'^n}(\prod(g))\} - \max\{X_i^{I'^1 \ldots I'^n}(\prod(g))\} \right\|_{\text{if(COND=true}}$ | The instant at which $T_j^{I^1 \ldots I^n}$ will terminate |
| $X_i^{I'^1 \ldots I'^n}(\prod(g)) = \begin{cases} T_i^{I'^1 \ldots I'^n}(\prod(g)) & \text{if } \prod_i^{I'^1 \ldots I'^n}(g) = \prod_j^{I^1 \ldots I^n}(g) \\ 0 & \text{otherwise} \end{cases}$ | Completion time of the tasks already executed on the processor where $T_j^{I^1 \ldots I^n}$ has been allocated. |
| $B^{I^1 \ldots I^n, I'^1 \ldots I'^n} = \langle I^1 - I'^1, \ldots, I^n - I'^n \rangle$ | Distance vector between $T_j^{I^1 \ldots I^n}$ and $T_i^{I^1 \ldots I^n}$ |
| $R = \{B^{I^1 \ldots I^n, I'^1 \ldots I'^n} \in \mathrm{DDS}_{ij}\}$ | Data dependence between the tasks $T_j^{I^1 \ldots I^n}$ and $T_i^{I'^1 \ldots I'^n}$ |

| | |
|---|---|
| $W_{ij}$ | Size of the messages of the different dependence pairs with the distance vector $B^{I^1 \ldots I^n, I'^1 \ldots I'^m}$ that belong to the dependence set $\mathrm{DDS}_{ij}$ |
| $\lambda_{ij}^{I^1 \ldots I^n, I'^1 \ldots I'^m}(\prod(g)) =$ $\begin{cases} \sum_{B^{I^1,\ldots,I^n,I'^1,\ldots,I'^m} \in \mathrm{DDS}_{ij}} W_{ij} & \text{if } \prod_j^{I^1 \ldots I^n}(g) \neq \prod_i^{I'^1 \ldots I'^m}(g) \\ 0 & \text{if } \prod_j^{I^1 \ldots I^n}(g) = \prod_i^{I'^1 \ldots I'^m}(g) \end{cases}$ | Communication time when two task are allocated on different processors |
| $\mathrm{COND} = \max_{T_i^{I^1 \ldots I^n} \in R} \{T_i^{I^1 \ldots I^n}(\prod(g)) + \lambda_{ij}^{I^1 \ldots I^n, I'^1 \ldots I'^m}(\prod(g))\} > \max\{X_i^{I'^1 \ldots I'^m}(\prod(g))\}$ | Overhead due to the communication time between dependent tasks |

### 3.1. Our heuristic scheduling algorithm

We define a heuristic approach based on genetic algorithms (GA) to solve the data-dependent loop scheduling problem. In our heuristic, we study the different assignments of the loop tasks, such as to optimize the cost function (2). The input of our heuristic is the *loop task graph*, and the output is the scheduling of the loop. The main phases of our algorithm are:

1. Calculate the complete loop unrolling graph ($G_u$) for the loop task graph $G$ in order to determine the set of $M$ tasks on the loop.
2. Call our scheduling algorithm based on GAs.

A GA is an optimization algorithm based on the principles of evolution in biology. A GA follows an "intelligent evolution" process for individuals based on the utilization of evolution operators such as mutation, inversion, selection and crossover [1]. The idea is to find the best local optimum, starting from a set of initial solutions, by applying the evolution operators to successive solutions so as to generate a new and better local minimum. The procedure continues until it converges to a minimum (which may be local or global). Previous work that uses GAs in loop optimization problems has been studied in [7]. Nisbet proposed a genetic algorithm parallelisation system (GAPS) compiler framework. The GA is used to determine the restructuring transformation applied to each statement and its associated iteration space. The hypothesis is that GAs can be used to determine the sequence of restructuring transformations which are better, or as good as those produced by more conventional compiler search techniques.

The GA applied in our problem follows the next procedure: we define a search space of $M^*n$ vectors, each representing an individual, and every individual representing a possible solution. An individual represents the assignment of the different nodes of the replicated task graph $G_u$ that represents the complete loop unrolling of the original loop task graph $G$. We number the set of tasks of the replicated task graph $G_u$ from 1 to $M$ (the number of tasks in the $n$ nested loops). In this way, each individual is composed of $M$ elements. The $i$th element has two values: a value between 1 and $K$, indicating the processor where task $i$ will be allocated, and another value indicating its execution order on this specific processor (see Fig. 4).

For example, assume the individual of the Fig. 5:

This means that task 1 will be the first task executed on processor 1, then task 3; task 2 will be the first task executed on processor 3, and so forth. That is, if we do not consider data dependence among the tasks the schedule table is:

| Processors \ time | 1 | 2 |
|---|---|---|
| 1 | $T1$ | $T3$ |
| 2 | $T4$ | $T5$ |
| 3 | $T2$ | |

Furthermore, we use the objective function (2) to calculate the cost of each individual. According to the execution order, we may encounter deadlock (a task is scheduled before its predecessor). The cost for the individuals that represent this situation will be infinite (wrong individuals). We begin with an initial population of individuals randomly selected and we choose the individuals with minimal cost for generating new individuals using our specific genetic operators. Since the population size is constant, we replace the worst individuals of the initial solution by the best individuals generated. The procedure stops if we exceed a given number of generations without finding a better solution or for a given number of iterations. The general genetic algorithm is defined as:

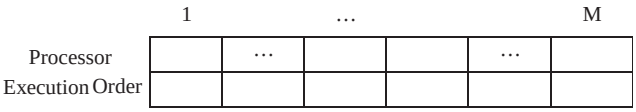


Fig. 4. Data structure of an individual.

*Generation of individuals which represent potential solutions*
*Repeat until system convergence or termination*
*Evaluation of every individual*
*Selection of the best individual for reproduction*
*Reproduction of the individual using evolutive operators*
*Replacement of the worst old individuals by the best new individuals*

Each iteration of our algorithm is of time complexity $O(M)$. If there are $t$ iterations, the time complexity of the entire algorithm is $O(tM)$. We use the following specific genetic operators:

- *Mutation of the processor allocation (MA)*: With this operator we can redefine a new allocation $k$ for a given task $i$, where $1 \leqslant k \leqslant K$ and $1 \leqslant i \leqslant M$ are chosen randomly. Its order of execution ($o$) will be the same as the current. For the tasks already assigned to processor $k$ which have an execution order bigger than or equal to $o$, we must increase their execution order by 1. In addition, for the tasks assigned in the initial allocation of the task $i$ that have been reallocated, we must decrease their execution order by 1 if these are bigger than or equal to $o$. The complexity of this operation is $O(M)$. Assume the individual of if the second element is chosen to mutate to 2, the new individual is shown in Fig. 6:
- *Permutation of the execution order (PEO):* For a given value of the processor field (for example, tasks assigned to processor $k$), we permute the execution order of the tasks assigned to it. The complexity of this operation is $O(1)$. Assume the individual of Fig. 5 and $k = 2$, when we apply this operator, the next individual is shown in Fig. 7:
- *Permutation of the allocation (PA):* For a given value of the execution order field (for example, task to be executed in the position $o$), we permute the processor where they will be assigned. The complexity of this operation is $O(1)$. Assume the individual of Fig. 5 and $o = 2$, the next individual is shown in Fig. 8:
- *Partial crossover (PC):* This operator takes the part of two individuals that correspond to a given execution order (for example, $o = 1$) or processor (for example, $k = 1$); it compares if they are similar (if each individual has the same number of elements with this execution order or processor assignment). Then, it exchanges these specific parts of information between them. That is, we are going to exchange the columns from each individual with the same given value of $o$ or $k$. The complexity of this operation is $O(M)$. For example, assume the two individuals of Fig. 9: and $o = 1$. PC yields these two new individuals (Fig. 10):

| | | | | |
|---|---|---|---|---|
| 1 | 3 | 1 | 2 | 2 |
| 1 | 1 | 2 | 1 | 2 |

Fig. 5. An individual.

| | | | | |
|---|---|---|---|---|
| 1 | **2** | 1 | **2** | **2** |
| 1 | **1** | 2 | **2** | **3** |

Fig. 6. Mutation of the processor allocation.

| | | | | |
|---|---|---|---|---|
| 1 | 3 | 1 | **2** | **2** |
| 1 | 1 | 2 | **2** | **1** |

Fig. 7. Permutation of the execution order.

| | | | | |
|---|---|---|---|---|
| 1 | 3 | **2** | 2 | **1** |
| 1 | 1 | **2** | 1 | **2** |

Fig. 8. Permutation of the allocation.

Similarly, for $k = 2$, PC yields the individuals of Fig. 11:

This operator is extended for a set of execution orders (for example, $o = \{2, 3\}$) or set of processors (for example, $k = \{1, 3\}$) and for this partial information we execute the same exchange. In this way, we can exchange partial good solution between individuals. For example, assume the parent individuals of Fig. 12:

For $k = \{1, 3\}$, the new individuals as shown in Fig. 13:

With our operators we can combine good partial results and guarantee that each individual represents a possible solution to our problem (maybe some repre-

| | | | | |
|---|---|---|---|---|
| 1 | 3 | 1 | 2 | 2 |
| 1 | 1 | 2 | 1 | 2 |

and

| | | | | |
|---|---|---|---|---|
| 2 | 2 | 1 | 3 | 1 |
| 2 | 1 | 1 | 1 | 2 |

Fig. 9. Parent individuals.

| | | | | |
|---|---|---|---|---|
| **2** | **1** | 1 | **3** | 2 |
| **1** | **1** | 2 | **1** | 2 |

and

| | | | | |
|---|---|---|---|---|
| 2 | **1** | **3** | **2** | 1 |
| 2 | **1** | **1** | **1** | 2 |

Fig. 10. New individuals for $o = 1$.

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 3 | 1 | **2** | **2** | and | **2** | **2** | 1 | 3 | 1 |
| 1 | 1 | 2 | **2** | **1** | | **1** | **2** | 1 | 1 | 2 |

Fig. 11. New individuals for $k = 2$.

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 2 | **1** | **1** | **3** | 2 | and | 2 | **1** | **3** | 2 | **1** |
| 1 | **1** | **2** | **1** | 2 | | 2 | **1** | **1** | 1 | **2** |

Fig. 12. Parent individuals.

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 2 | **1** | **3** | **1** | 2 | and | 2 | **1** | **1** | 2 | **3** |
| 1 | **1** | **1** | **2** | 2 | | 2 | **1** | **2** | 1 | **1** |

Fig. 13. New individuals for $k = \{1, 3\}$.

senting a deadlock). Initially, we start with solutions without deadlock to guarantee that the GA works in a space with valid solutions.

## 4. Results analysis

In this section, we discuss experiments for both applications where we know the optimal scheduling and well known benchmarks. Then, we perform some experiments for parallel loops. The performance of our approach is evaluated by comparing the resulting schedule length. For the set of experiments we test the percentage of individuals of the final population that represent the optimal scheduling, the execution time to obtain it, the schedule length of the best individuals of our model (we compare that with previous results [4,12,18]) or the number of iterations to obtain optimal results. In our experiments, we assume a fully connected homogeneous distributed-memory system composed of 8 processors ($K = 8$) and each operation (task) takes 1 time unit. We have used in our experiments a Pentium III, 250 MHz, 128M RAM Dell machine.

### 4.1. Test of the optimal scheduling

For this test, the kernel application programs used for the performance evaluation were carefully selected from different classes of loops (kinds of workload). The set of experiments was defined using the next algorithms:

Program 1: The algorithm proposed in Section 2.2.1, where the loops are iterated $n$ times (we replace 2 by $n$)

Program 2

For $i = 1$ to $n$ do
  For $j = 1$ to $n$ do
    $T : X[i,j] = F1(X[i+2, j-3])$

Program 3

For $i = 1$ to $n$ do
  For $j = 1$ to $n$ do
    $T1 : A[i] = B[i] + C[i]$
    $T2 : D[i] = A[i]/E[i]$
    $T3 : E[i+1] = \mathrm{SQRT}(D[i] + D[i+1])$
    $T4 : F[i] = F[i-1]/D[i]$

We assume five sets of parameters for our heuristic (see Table 1), where the value of each element represents the probability to use that operator. The initial results are presented in Table 2, where ET is the execution time (in seconds) for the GA convergence and $P$ the percentage of the individuals of the final population that represent an optimal scheduling.

Our results show that in order to obtain the optimal scheduling for this problem, the execution time of the heuristic depends on the values of the parameters of it. The main genetic operator is the PC operator because it allows the exchange of partial good solutions (Set 2). The overhead due to our operators is very important (see Set 4). A good combination of the operators can give a good execution time of the heuristic and a large number of optimal individuals in the last population (Set 5). In general, for Set 4 the percentage of optimal solutions in the final solution is smaller than for Sets 1 and 5 when the number of tasks in the nested loop is important (programs 2 and 3). In this part, the reason to study the percentage of optimal individual in the last population (when the GA converges) is because some of the optimal individuals can be different. That can be interesting for the operating system in order to choose a schedule that improves the overall performance on the system taking into account the current workload. Remember that it is a static scheduling; in this way we minimize the degradation of the performance on the system in the future. We must find a compromise between the execution time of the heuristic scheduler (it is executed during compile time) and the number of optimal solutions at the end of the procedure (that can help to optimize the performance in our system). Another important criterion is the number of iterations

Table 1
Sets of parameters

| Parameter | Set 1 | Set 2 | Set 3 | Set 4 | Set 5 |
|---|---|---|---|---|---|
| MA | 0.9 | 0.1 | 0.9 | 0.9 | 0.5 |
| PEO | 0.1 | 0.1 | 0.1 | 0.9 | 0.5 |
| PA | 0.1 | 0.1 | 0.1 | 0.9 | 0.5 |
| PC | 0.9 | 0.9 | 0.1 | 0.9 | 0.5 |

Table 2
Results for different numbers of iterations

| Program | Iterations | Set 1 | | Set 2 | | Set 3 | | Set 4 | | Set 5 | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | ($n$) | ET | $P$ | ET | $P$ | ET | $P$ | ET | $P$ | ET | $P$ |
| 1 | 1000 | 130 | 40 | 100 | 65 | 70 | 42 | 340 | 48 | 150 | 63 |
| | 500 | — | — | 82 | 73 | — | — | 278 | 51 | 102 | 65 |
| | 100 | — | — | 62 | 75 | — | — | 198 | 52 | 58 | 73 |
| 2 | 1000 | 260 | 23 | 220 | 49 | 180 | 21 | 610 | 26 | 270 | 51 |
| | 500 | — | — | 165 | 54 | — | — | 443 | 31 | 211 | 54 |
| | 100 | — | — | 112 | 62 | — | — | 321 | 38 | 146 | 63 |
| 3 | 1000 | 260 | 22 | 230 | 43 | 180 | 28 | 640 | 31 | 250 | 55 |
| | 500 | — | — | 189 | 47 | — | — | 467 | 36 | 179 | 66 |
| | 100 | — | — | 123 | 59 | — | — | 312 | 46 | 124 | 71 |

Table 3
Number of iterations of the GA to obtain the optimal scheduling for $n = 100$

| Program | Set 1 | Set 2 | Set 3 | Set 4 | Set 5 |
|---|---|---|---|---|---|
| 1 | 21 | 19 | 32 | 11 | 25 |
| 2 | 29 | 27 | 43 | 15 | 32 |
| 3 | 25 | 26 | 51 | 16 | 37 |

of the GA to obtain an optimal scheduling. Table 3 shows these results.

With a small number of iterations (11, 15, 16) we can obtain an optimal scheduling. Of course, these results dependent on the quality of the initial individuals (in our case, we have guaranteed not to have an optimal solution in the initial population). In general, with Set 4 we need fewer iterations because the genetic operators allow to explore all the solution space, but its execution time is very large (see Table 2) because we use very frequently the different genetic operators. Set 3 carries out a random search at the beginning; this is the reason of its large number of iterations to find an initial optimal individual. If we determine the optimal combination of the operators, we will reduce the number of iterations to find an optimal solution. In addition, if we need a fast optimal solution, we must minimize the number of iterations of the heuristic scheduler.

In Table 4 we see that the convergence time of the GA increases when then number of iterations of the programs increases. We can reduce this execution time supposing a given number of iterations for the GA, but the main criteria of the system convergence (when the GA can not obtain a best solution) guarantee a good solution (maybe the global optimal). If we reduce the number of iterations of the GA we can no longer guarantee that. Remember that the optimal scheduling is calculated only once, that is, for a new execution of the programs 1, 2 or 3, we are going to use the same scheduling.

Table 5 shows the execution time for the GA convergence for different $n$ and $K$. The convergence time depends of them because the size of the solution space is determined by them. The number of processors to use depends of the degree of parallelism of the applications. If with more processors we can improve the execution time of the applications, the cost for the execution time of our approach is low (we introduce this cost only one time because we execute our scheduler only at compile time).

### 4.2. Tests with benchmarks

In this section, we evaluate the effectiveness of our approach by running a set of simulations on digital signal processing (DSP) benchmarks, in order to

Table 4
Execution time (in seconds) for Set 2 and different numbers of iterations ($n$)

| Program | $n = 10.000$ | $n = 100.000$ | $n = 1.000.000$ | $n = 10.000.000$ |
|---|---|---|---|---|
| 1 | 342 | 614 | 1821 | 4542 |
| 2 | 621 | 1024 | 4041 | 8432 |
| 3 | 619 | 931 | 3857 | 7841 |

Table 5
Execution Time for Set 2, program 1 and different numbers of iterations ($n$) and processors ($K$)

| $K$ | $n = 500$ | $n = 1.000$ | $n = 10.000$ | $n = 100.000$ | $n = 1.000.000$ |
|---|---|---|---|---|---|
| 4 | 58 | 79 | 221 | 478 | 722 |
| 8 | 82 | 100 | 342 | 614 | 1821 |
| 16 | 102 | 138 | 567 | 1378 | 3081 |
| 32 | 121 | 188 | 703 | 2012 | 4971 |

Table 6
Results of the second set of experiments for $K = 12$.

| Benchmark | $n$ | Set 2 | | Set 4 | | List | | PSP | |
|---|---|---|---|---|---|---|---|---|---|
| | | Len | Ratio(%) | Len | Ratio(%) | Len | Ratio(%) | Len | Ratio(%) |
| WDF | 4 | 2 | 100 | 2 | 100 | 3 | 66 | 2 | 100 |
| | 12 | 4 | 100 | 4 | 100 | 6 | 66 | 5 | 80 |
| | 64 | 24 | 91 | 22 | 100 | 27 | 81 | 24 | 91 |
| | 128 | 45 | 95 | 43 | 100 | — | — | — | — |
| IIR | 16 | 6 | 100 | 6 | 100 | 8 | 75 | 6 | 100 |
| | 64 | 16 | 81 | 13 | 100 | 20 | 65 | 16 | 81 |
| | 128 | 42 | 85 | 38 | 94 | — | — | — | — |
| DPCM | 16 | 6 | 100 | 6 | 100 | 7 | 85 | 7 | 85 |
| | 64 | 19 | 100 | 19 | 100 | 23 | 82 | 21 | 91 |
| | 128 | 45 | 80 | 39 | 92 | — | — | — | — |
| Floyd | 16 | 6 | 100 | 6 | 100 | 11 | 54 | 6 | 100 |
| | 64 | 14 | 100 | 14 | 100 | 18 | 77 | 16 | 87.5 |
| | 128 | 44 | 81 | 42 | 85 | — | — | — | — |

compare the performance of our approach with previous works [4,12]. Table 6 shows our scheduling results. The first column gives the benchmarks' names. The abbreviations WDF, IIR, DPCM and Floyd stand for Wave Digital Filter, Impulse Response filter, Differential Pulse-Code Modulation device and Floyd-Steinberg algorithm, respectively. They are typical algorithms with data dependence loops. For example the WDF kernel is:

For $i = 1$ to $n$ do
  For $j = 1$ to $n$ do
    $D[i,j] = B[i-1,j+3]^{*} C[i-1,j-1]$
    $A[i,j] = D[i,j] + 0.5$
    $B[i,j] = A[i,j] + 1$
    $C[i,j] = A[i,j] + 2$

In Table 6, the second column is the number of nodes ($n$) in each benchmark. The final best schedule of our heuristic is shown in the next two columns for Sets 2 and 4 of parameters. Specifically, we also show the results using list scheduling and partition scheduling with prefetching (PSP) [4,12]. PSP is one of the best scheduling algorithms for DSP problems. The results are shown in the columns *list* and *PSP*, respectively, where the sub-column *len* is the schedule length and the sub-column *ratio* is the ratio comparing the optimal schedule length with that of the different scheduling algorithms. Here, a value equal to 100% indicates that the scheduler obtains the optimal solution and one smaller than 100% that the result is worse.

As we can see, list scheduling (list) rarely achieves the best schedule length because list scheduling is not well balanced. Our approach obtains similar results as PSP. Our approach consistently produces good results (the average ratios of the schedule length for list scheduling, PSP, our approach-set 2 and our approach-set 4 are

Table 7
Results of the second set of experiments for WDF, $n = 64$

| $K$ | Set 2 | | Set 4 | |
|---|---|---|---|---|
| | Len | Ratio(%) | Len | Ratio(%) |
| 4 | 30 | 80 | 30 | 80 |
| 12 | 24 | 91 | 22 | 100 |
| 32 | 24 | 91 | 22 | 100 |

Table 8
Results for the last set of experiments (for 100 iterations of the GA)

| Benchmark | Parameters | Set 2 | Set 4 | AA |
|---|---|---|---|---|
| Type 1 | $n = 1024$ $L = 500$ | 52 | 52 | 52 |
| Type 2 | $n = 1024$ $L = 500$ | 55 | 55 | 55 |
| Type 3 | $n = 512$ | 47 | 47 | 47 |

72.3%, 90.6%, 93.3% and 97.7%, respectively. But, there are four cases for which list scheduling and PSP do not have results. These values are not considered in the average, which by the way makes our results look worse than they are. If we restricted the average for Set 2 and Set 4 to only those results for which list scheduling and PSP have values, then our averages are much better). The GA is executed a fixed number of iterations (the number of iterations is equal to 20), for this reason we do not obtain the optimal scheduling for certain cases.

Table 7 shows that the solution depends of the number of processors, but for a given number of processors we can not improve the scheduling. That is, increasing the number of processors beyond a certain point will not improve the schedule. Like the previous one, the GA is executed a fixed number of iterations (the number of iterations is equal to 20).

### 4.3. Parallel loop test

In this section, we compare the performance of our approach to scheduling parallel loops. A parallel loop is a particular case of a data dependent loop, since there are no precedence relationships between the tasks. Thus, Eq. (1) is simplified because COND will be always false. We have tested the effect on our model, characterizing the parallel loops according to the distribution of loop execution time:

*Type 1: Loops with potential affinity and balanced workload.* We have chosen the successive over-relaxation (SOR) algorithm. In this case, all iterations take about the same time to execute and each iteration always accesses the same data. Here is the kernel:

```
do i = 1, L
  do parallel j = 1, n
    do k = 1, n
      a[j,k] = update(a,j,k)
```

*Type 2: Loops with unpredictable workload.* In this case, we have used the Jacobi Iteration algorithm. The iterations have a different workload. Here is the kernel:

```
for i = 1, L
  for parallel j = 1, n
    for k = 1, n
      if a[j,k] ≠ 0 and j ≠ k then
        x1[j] = x1[j] + a[j,k]*xo[k]
    x1[j] = (b[j] − x1[j])/a[j,j]
  for l = 1, n
    xo[l] = x1[l]
```

*Type 3: Loops with non-affinity and balanced workload.* In this case, we have used a matrix multiplication algorithm. This algorithm does not have affinity to exploit (in this case, iterations do not always access the same data). Here is the kernel:

```
for parallel i = 1, n
  for parallel j = 1, n
    for k = 1, n
      c[i,j] = c1[i,j] + a[i,k]*b[k,j]
```

The performance metric we use to compare our model is the execution time of the applications (see Table 8). Execution time measures how differently the scheduling algorithms work. For this case, we assume a shared memory system with 8 processors. We compare our algorithm with the adaptive affinity algorithm (AA) [18], the best approach of that paper.

This result show that our approach can be used like a static parallel loop scheduling algorithm. Of course, with this approach we have the same disadvantage as the static approach in the presence of unpredictable transient loads (load imbalance during runtime). But the main advantages are that our approach can take into account the current workload to distribute the iterations, it minimizes the run-time synchronization overhead, and, if not all iterations take the same amount of time (especially when loops are non-uniformly distributed), our approach can achieve good load balancing.

## 5. Conclusions

We have proposed a data dependent loop scheduling model. Our model is suitable for a wide range of applications and for different situations (parallel loops, data dependent loops, distributed and shared memory systems, etc.); thus, our approach is very versatile and provides a generalized model of the loop scheduling problem. In general, the heuristic scheduler based on our model is affected by the loop size (number of tasks, number of nested loops, etc.), its parameters

(probability to use each genetic operator, number of iterations, etc.) and the parallel/distributed platform (number of processors). We have defined specific genetic operators for our problem. We have compared our model with some of the best parallel loops scheduling algorithms and data dependent loops algorithms and we have obtained similar or better results. The results are very encouraging and suggest a parallel compiler implementation based on our model. Moreover, the execution time of our GA can be improved by developing a parallel version of it. For example, we can develop a parallel version of our algorithm that improves the search at the level of the solution space. At the same time, we can execute a parallel version of our algorithm for different nested loops of a given application. As we have stated previously, our approach is executed at compile time. Then, the system can reuse the results for a new execution of the same application with different input. A dynamic compile framework could improve the quality of results but at the cost of a large execution time. Finally, we plan to test the performance of our approach using other optimization techniques.

## Acknowledgments

This work was partially supported by the CDCHT-ULA Grant I-620-98-02-AA. Jose Aguilar was supported by a CONICIT-Venezuela grant (subprograma de pasantías postdoctorales) and CONICIT grant.

## References

[1] J. Aguilar, E. Gelenbe, Task assignment and transaction clustering heuristics for distributed systems, Inform. Sci.: Inform. Comput. Sci. 97 (1) (1997) 199–219.
[2] J. Aguilar, E. Leiss, A general adaptive parallel loop scheduling for distributed and shared memory systems, Technical Report, Department of Computer Sciences, University of Houston, June 2000.
[3] A. Bull, Feedback Guided Dynamic Loop Scheduling: Algorithms and Experiments, Lecture Notes in Computer Science, Vol. 1470, Springer, Berlin, 1998, pp. 377–382.
[4] F. Chein, T. O'Neil, E. Sha, Optimizing overall loop schedules using prefetching and partitioning, IEEE Trans. Parallel Distributed Systems 11 (6) (2000) 604–614.
[5] H. El_Rewini, T. Lewis, H. Ali, Task Scheduling in Parallel and Distributed Systems, Prentice-Hall, Englewood Cliff, NJ, 1994.
[6] G. Nanliken, G. Belloch, Space-efficient scheduling of nested parallelism, ACM Trans. Programming Language Systems 21 (1) (1999) 138–169.
[7] A. Nisbet, GAPS: genetic algorithm optimized parallelisation, Technical Report, Department of Computer Sciences, University of Manchester, UK, 1998.
[8] N. Passos, M. Sha, Achieving full parallelisation using multi-dimensional retiming, IEEE Trans. Parallel Distributed Systems 7 (11) (1996) 1150–1163.
[9] A. Srinivasan, J. Anderson, A simple proof technique for priority scheduled systems, Inform. Process. Lett. 77 (2–4) (2001) 63–70.
[10] Z. Szczerbinski, Optimal Distribution of Loops Containing no Dependence Cycles, Lecture Notes in Computer Science, Vol. 1595, Springer, Berlin, 1999, pp. 1254–1257.
[11] S. Tongsima, E. Sha, N. Passos, Communication-sensitive loop scheduling for DSP applications, IEEE Trans. Signal Process. 45 (5) (1997) 1309–1322.
[12] S. Tongsima, C. Chantrapornchai, E. Sha, Probabilistic Loop Scheduling Considering Communication Overhead, Lecture Notes in Computer Science, Vol. 1459, Springer, Berlin, 1998, pp. 158–179.
[13] S. Tongsima, E. Sha, C. Chantrapornchai, D. Surma, N. Luiz, Probabilistic loop scheduling for applications with uncertain execution time, IEEE Trans. Comput. 49 (1) (2000) 65–80.
[14] J. Xue, Communication-minimal tiling of uniform dependence loops, J. Parallel Distributed Comput. 42 (1997) 42–59.
[15] C. Wang, K. Parhi, Resource-constrained loop list scheduler for DSP algorithms, J. VLSI Signal Process. 11 (1–2) (1995) 75–96.
[16] Y. Yan, C. Jin, X. Zhang, Adaptively scheduling parallel loops in distributed shared-memory systems, IEEE Trans. Parallel Distributed Systems 8 (1) (1997) 70–81.

**Jose L. Aguilar** Professor Jose Aguilar received the B. S. degree in System Engineering in 1987 from the Universidad de los Andes-Venezuela, the M. Sc. degree in Computer Sciences in 1991 from the Universite Paul Sabatier-France, and the Ph. D degree in Computer Sciences in 1995 from the Universite Rene Descartes-France. He was a Postdoctoral Research Fellow in the Department of Computer Sciences at the University of Houston (1999-2000). He is a Titular Professor in the Department of Computer Science at the Universidad de los Andes, researcher of the High Performance Computing Center of Venezuela (CeCalCULA) and of the Center of Studies in Microelectronics and Distributed Systems (CEMISID). He has published more than 150 papers in journals, books and proceedings of international conferences in the field of parallel and distributed systems, and computational intelligence. He has received several awards and some of his papers have received special awards. Dr. Aguilar has been a visiting research/professor in different universities and laboratories in France, USA, and Spain. He has been Chairman of Symposia, Workshops, etc.; editor of proceedings and books, and member of Program Committees for different International Conference. He has been the coordinator of different research projects supported by the Venezuelan Scientific Office, the French Scientific Office, and INTEVEP (Venezuelan Institute of research in oil).

**Ernst L. Leiss** is Professor of Computer Science at the University of Houston. He received graduate degrees in Computer Science and Mathematics from the University of Waterloo and the Technical University of Vienna. He (co)authored over 130 peer-reviewed papers and wrote Principles of Data Security (1982, Plenum), Software Under Siege: Viruses and Worms (1990, Elsevier), Parallel and Vector Computing: A Practical Introduction (McGraw-Hill, 1995), and Language Equations (Springer, 1999). He has been (co)PI on research funded at over 3 million. His research interests range from vector/parallel computing to data security and formal language theory. He has supervised 13 Ph. D. dissertations and approximately 100 M. S. theses. He is a member of ACM, a Senior Member of IEEE, and an Active Member of SEG.