Nested Loop Parallelization over a Distributed Memory Environment using PVM

A. Pappas T. Andronikos N. Koziris G. Papakonstantinou
P. Tsanakas

Digital Systems and Computers Laboratory
Computer Science Division
Department of Electrical and Computer Engineering
National Technical University of Athens
Zographou Campus, 15773, Athens, Greece

Abstract

The most important issue in sequential program parallelization is the efficient assignment of computations into different processing elements. The most extensive, in terms of time execution, part of a program is the nested loops. The process of parallelizing a nested loop involves two subsequent steps: First the "time" partitioning of the index space into appropriate groups and second the assignment of these concurrent partitions to different processors. This paper presents a new method and an environment for the parallelization of nested for-loops onto distributed memory architectures. The time partitioning of the index space is done with the wavefront method. The space mapping, which guarantees the minimum number of processors, is done using PVM message passing calls for interprocessor communications while preserving the minimum number of processors. We present an integrated environment which accepts as input a C-like nested loop algorithm and automatically produces the corresponding parallel implementation over a cluster of workstations using PVM. This tool incorporates the theoretical framework of loop parallelization with the generation of portable PVM code running various kinds of distributed memory parallel architectures.

Index Terms — Algorithm partitioning, algorithm transformations, hyperplane method, distributed architectures, PVM, message passing.

1 Introduction

Experience has shown that any program that requires a significant amount of computer time, spends most of this time executing one or more loops. For a large class of programs these loops are nested FOR-loops. Consequently, a lot of research has been done over the past years concerning the parallelization of FOR-loops. This research was further focused, and given practical results, for a specific category of FOR-loops, those which contain uniform dependencies among their computed statements.

The primary task in parallelizing a FOR-loop, is finding a schedule of computations in time, while preserving the data dependencies of the initial lexicographic loop order. The most frequently used time scheduling method is the hyperplane method originated by Lamport [11]. The main notion behind such scheduling is the organisation of indexed computations into defined distinct groups called wavefronts or hyperplanes. Many methods were proposed in literature to find the optimal partitioning into hyperplanes. Some of them are based on the solution of diofantine equations ([12], [9]) and others on the use of linear programing approaches ([13],[5]). In [1] another approach was presented, based on the geometric properties of the index space, with polynomial complexity for doubly nested loops.

Time scheduling of loop iterations does not ensure the efficient execution on parallel architectures as is shown in [10]. Researchers presented in the past several methods for the assignment of the concurrent partitions of the index space into different processing elements considering various special purpose architectures. In [12] a systematic methodology for mapping the nested loop index space into fixed size systolic arrays was presented and further extended in [9]. This method was based on space transformation S which transformed the original dependencies to meet the communication links of the available hardware. Since the target architecture is synchronously operating, there is no need for communication efficient mapping. Systolic cells communicate without additional time delays, and the main criterium for optimality is the total number of processors. However, in message passing multiprocessor systems, communication delays play the most important role, since they can increase dramaticaly the total execution time. Researchers are trying to minimize the interprocessor communication, by dividing the index space into as much as possible independent groups of computations. Shang in [14] has presented a method 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 [15], presented a systematic method of partitioning the index space into groups and assigning them to different processing elements. However their method does not minimize the communication delays between adjacent groups and is applicable only for fixed size hypercubes.

As far as tools for automatic parallelization of loops onto distributed memory systems is concerned, few research has given practical results. In [8] Loopo, a tool for nested loop parallelization Loopo is presented. It uses the wavefront method for time scheduling. As far as space mapping is concerned it uses linear transformations found in Feautrier [7], Darte-Vivien [6] by applying integer programming approaches (NP-complete in the worst case). These mappings do not necessarily use the minimum number of processors and do not minimize the inter-processor communication. Loopo generates C and Fortran programs using Parix routines which is less portable than other message passing platforms like PVM, MPI. In [4] the Barnett compiler is presented. It generates distributed memory parallel code from nested for-loops and the code generator creates programs which can be translated to Occam. This tool is based on mapping to systolic architectures which is not always communication efficient.

Most of the mapping methods presented in the literature, achieve optimal (or near optimal) time, without setting strict criteria for the number of processors. This is a natural consequence of the lack of theoretical lower bounds on the number of processors necessary to achieve optimal time. This also means that the proposed methodologies can not be evaluated in terms of processor utilization, i.e., processors with significant idle time are obligatory used. Furthermore, communication minimization is also correlated to processor optimality because any schedule that minimizes communication uses the optimal number of processors.

In our paper we present Loopdep a tool which maps the time partitions of the index space onto clusters of workstations. We have developed a complete environment for the parallelization of for-loops and their parallel execution on any machine that supports PVM. Our tool takes as input a program containing C-like for-loops, and creates for every loop a PVM program. The parallelized program achieves maximum processor utilization while using the minimum number of processors. To the best of our knowledge, there does not exist a tool which generates executable code of parallelized for-loops for distributed memory machines that is optimal in time, processor number and processor utilization and which is widely portable. Loopdep is very modular and can easily be embedded to an existing compiler to parallelize nested loops.

Loopdep consists of several parts:

- The parser accepts a C-like grammar consisting out of variable declarations, for-loops and assignments and generates a syntax tree.
- The dependence analyser takes the syntax tree, and creates a dependence matrix containing the dependencies in the loop. Only uniform dependencies are allowed.

- The scheduler takes the dependence matrix and calculates the time at which each iteration will be executed.
- The mapper maps a schedule to a number of tasks. The introduction of lower bounds on the number of tasks allows optimal space mappings. For the moment the number of tasks must be equal to the above lower bound.
- The code generator generates code for each task using PVM routines for communication.

In the following sections we will describe in more detail each of the above items.

2 Notation

Throughout this paper we use the following notation:

- N is the set of naturals,
- Z is the set of integers,
- UDL is the class of uniform dependence loops,
- n is the number of nested for-loops,
- $J^n \subset Z^n$ is the index space,
- m is the number of dependence vectors,
- $\mathbf{d}_i = (d_{i1}, ..., d_{in}), 1 \leq i \leq m$, is a dependence vector and
- $DS = \{\mathbf{d}_1, \mathbf{d}_2, ..., \mathbf{d}_m\}$ is the dependence set of the dependence vectors.

We focus on UDL, which are of the following form:

for $(i_1 \;=\; l_1;\; u_1)$
...
for $(i_n \;=\; l_n;\; u_n)$
$AS_1(I)$
...
$AS_p(I)$

1. $l_i, u_i \in Z, 1 \leq i \leq n$.
2. $\mathbf{I} = (i_1, ..., i_n)$.
3. $AS_1, ..., AS_p$ are assignment statements of the form $V_0(\mathbf{I}) = E(V_1(\mathbf{J}_1), ..., V_k(\mathbf{J}_k))$, where V_0 is an output variable, E is an expression of the input variables $V_1, ..., V_k$, and $\mathbf{d}_i = \mathbf{I} - \mathbf{J}_i, 1 \leq i \leq k$, are constant vectors.
4. The index space J^n is the set $(i_1, ..., i_n) | i_i \in Z \wedge l_i \leq i_i \leq u_i, 1 \leq i \leq n$. Each point in this n-dimensional integer space is a distinct instantiation of the loop body. The geometrical form of the index space is that of a hyperparallelepiped; the exact shape depends on the indices' lower and upper bounds. The cardinality of J^n is $\prod_{i=1}^{n}(u_i - l_i + 1)$.

3 Parser

The parser accepts all programs written in Loopdep language which is a subset of C. The language is left recursive and the parser was generated with bison (GNU's yacc). Loopdep's grammar is shown in the appendix. It contains only the necessary structures for describing nested-loop algorithms (i.e. for-loops, assignments) but can easily be adapted to include other structures as well. As datatypes, only integers and array of integers are used. There are three basic data structures, symbols which correspond to variable declarations, expressions, and statements.
To better illustrate the data structures created by the parser, consider the following small program:

```
program test1

int b[11][11], s[11][11];

int i, j;
{
  for (i = 2; 10;)
    for (j = 2; 10;) {
      b[i][j] = s[i-1][j];
      s[i][j] = s[i][j-1] + b[i][j-2];
    }
```

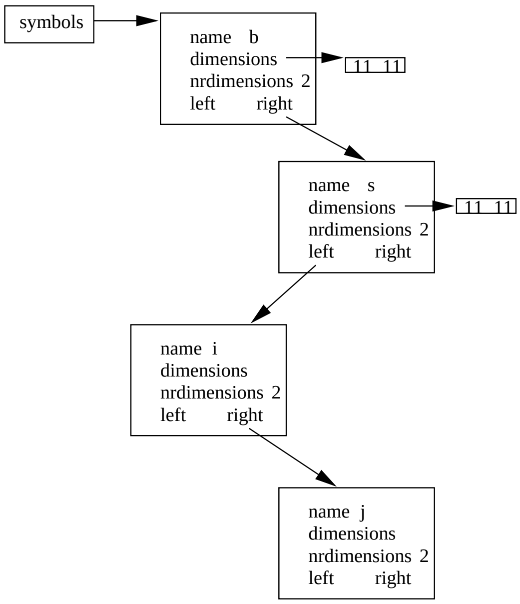

Figure 1: The symbol tree

One can see in figure 1 the symbol table generated from the avove variable declarations and in figure 2 the syntax tree generated from the above UDL. The dependence analyzer will use the syntax tree to calculate the dependencies.

4 Dependence Analyzer

When parallelizing UDLs, we are only concerned with data dependencies among assignment statements. If an assignment statement AS involves a variable V, whose value $V(\mathbf{j})$ on the j-th iteration

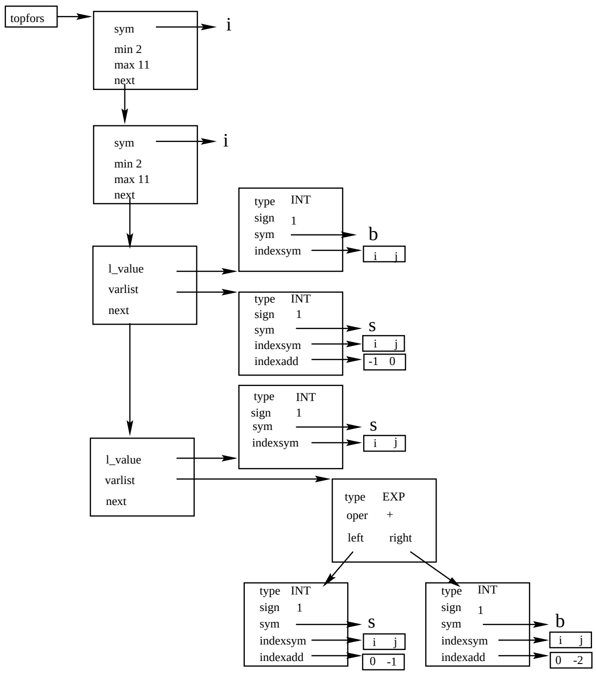

Figure 2: The syntax tree

of the loop body is given by an assignment statement AS' executed on the i-th iteration, then we say that AS depends on AS'. This qualitative notion of dependence is quantified by a dependence vector $\mathbf{d} = \mathbf{j} - \mathbf{i}$. In general, the dependence vectors traverse the various index points forming regular patterns inside the index set.

1. In our model all dependence vectors are uniform, i.e., their elements are constants and not functions defined over index sets. The theory we apply here is based on this assumption.

2. By definition, the dependence vectors are always greater than $\mathbf{0}$, where $\mathbf{0} = (0, ..., 0)$ and $>$ is the lexicographic ordering. It should be obvious that a dependence vector < 0 would indicate the absence of any dependence. This implies that there are no cycles in the index space, that is, J^n viewed as a grid is a directed acyclic graph (or DAG, for short). From our point of view this is crucial because it means that the longest path from the origin to each point of J^n is finite and well-defined. Consequetly, the wavefront is fully defined.

3. For the above class of algorithms J^n is a finite set. However, in order to derive general conclusions, we shall allow J^n to increase without bound, i.e., be infinite.

4. The points of J^n are ordered lexicographically; the usual symbol $<$ is used to denote this (linear) ordering.

4.1 Implementation details

Given the syntax tree of a for-loop, the dependence analyzer creates a dependence matrix.
The dependendence analyzer looks for each left value if it appears on the right hand side of a statement, and if yes, a dependence is created, by subtracting the indices of the right hand side variable from the indices of the left value. For the example program given above, b appears at the left side of statement 1 and at the right side of statement 2. Therefore there is a dependence for b. The dependence matrix is given in table 1.

variable	i	j
b	0	2
s	0	1
s	1	0

Table 1: dependence matrix

5 Scheduler

The dependencies of a UDL express precedence constraints among the points of the index space. The existence of the above constraints can be constructed as a partial ordering induced over the index space; a particular loop iteration has to wait the results of certain other iterations before it can be executed. For every UDL the partial ordering depends only on the existing dependencies and not on the size of the index space. Furthermore, the aforementioned ordering must be preserved if the correctness of the parallel execution is to be ensured no matter what the scheduling policy is.
When determining the scheduling of the points of the index space, we assume that:

- Execution begins at instant 1, and
- All iterations in the index space are completed in the same amount of time.

The index space can be partitioned into disjoint sets according to the following two criteria:

- The earliest possible execution time of each point, and
- The latest possible execution time of each point, assuming that the optimal makespan is achieved.

We denote GB_i the set of index points for which the earliest possible execution time is i.
It should be obvious that:

- Always $|GB_0| \geq 1$, otherwise no computation would ever be initiated.
- If the optimal parallel time of the algorithm is t, then the index space is partitioned into the following sets: $GB_0, GB_1, ..., GB_t$.

We denote GT_i the set of index points for which the latest possible execution time is i, assuming that the optimal makespan is achieved. If the optimal parallel time of the algorithm is t, then:

- $|GT_t| \geq 1$, (at least one computation is executed on instant t)
- The index space is partitioned into the following sets: $GT_0, GT_1, ..., GT_t$.

5.1 Scheduling Policy

Consider an arbitrary index point $\mathbf{j} \in GB_i$. This means that point $\mathbf{j}$ will be available for execution from instant i; it does not mean that $\mathbf{j}$ has to be executed at instant i. Symmetrically, assume that $\mathbf{j} \in GT_{t-r}$. This means that j can not be executed later than t-r; it must be executed at instant t-r or even earlier. If i=t-r, then $\mathbf{j}$ has no alternative but to be scheduled at time i, i.e., as soon as it is available. Such points that are executed immediately as they become available are characterized as 'crucial' points (those are exactly the points that belong to a maximal chain; see [3]). If $(t-r)-i = k > 0$, then the execution of $\mathbf{j}$ can be delayed for at most k time instants without causing the delay of the entire algorithm; specifically $\mathbf{j}$ can be executed at any of the following instants: i, i+1, ..., i+k. Since we strive for optimal time, any optimal schedule will have to focus on those points that admit delay.

0:	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1:	1	1	1	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2
2:	1	1	2	2	2	2	2	2	3	3	3	3	3	3	3	3	3	3
3:	1	1	2	2	2	2	3	3	3	3	3	3	4	4	4	4	4	4
4:	1	2	2	2	3	3	3	3	3	3	4	4	4	4	4	4	5	5
5:	1	2	2	2	3	3	3	3	4	4	4	4	4	4	5	5	5	5
6:	1	2	2	3	3	3	4	4	4	4	4	4	5	5	5	5	5	5
7:	1	2	2	3	3	3	4	4	4	4	5	5	5	5	5	5	6	6
8:	1	2	3	3	3	4	4	4	5	5	5	5	5	5	6	6	6	6
9:	1	2	3	3	3	4	4	4	5	5	5	5	6	6	6	6	6	6
10:	1	2	3	3	4	4	4	5	5	5	6	6	6	6	6	6	7	7
11:	1	2	3	3	4	4	4	5	5	5	6	6	6	6	7	7	7	7
12:	1	2	3	4	4	4	5	5	5	6	6	6	7	7	7	7	7	7
13:	1	2	3	4	4	4	5	5	5	6	6	6	7	7	7	7	8	8
14:	1	2	3	4	4	5	5	5	6	6	6	7	7	7	8	8	8	8
15:	1	2	3	4	4	5	5	5	6	6	6	7	7	7	8	8	8	8
16:	1	2	3	4	5	5	5	6	6	6	7	7	7	8	8	8	9	9
17:	1	2	3	4	5	5	5	6	6	6	7	7	7	8	8	8	9	9

Table 2: wavefront, with total 9 time steps

The scheduler takes the dependence matrix, calculates the wavefront, and creates the schedule, i.e. for each timestep, which iterations to execute. The wavefront is a matrix of the index space and is calculated recursively. An iteration, which does not depend on any iteration in the index space is executed at timestep 1. Otherwise, an iteration is executed one timestep more than the greatest timestep of the iterations on which it depends on. Consider following for-loop:

```
for (i = 5; i <= 22; i++) {
  for (j = 5; j <= 22; j++) {
    a[i][j] = a[i - 1][j - 4] + b[i - 4][j - 1];
    b[i][j] = a[i - 3][j - 5] + a[i - 5][j - 3] + b[i - 2][j - 2];
  }
}
```

Its dependence matrix is:

variable	i	j
a	1	4
a	1	1
a	3	5
a	5	3
a	2	2

The wavefront for the above dependence matrix is given in table 2. Each element in the matrix corresponds to a point in the index space and its value gives the earliest timestep it can execute. The schedule is an array of sets. Each timestep has a set which consists of the iterations that have the same earliest possible execution time. The schedule of the above wavefront is given in table 3.

6 Mapper

The mapper assigns iterations to processors, by preserving the dependencies between the iterations. A mapping is a matrix, that gives for each processor and each timestep, the iteration to execute. A simple mapping would be to assign each column of the above schedule to one processor. Then, the resulting mapping is given in table 4.
This mapping achieves minimum execution time but uses 52 processors, which is not the minimum possible number of processors. We have developed an algorithm, that maps a schedule to the minimum possible number of processors. The minimum number of processors is calculated in [2], [3]. The idea of the algorithm is, that certain iterations can be postponed, because their results are not needed immediately. So the critical iterations that can not be postponed are mapped first. If there are still processors free, iterations with many dependencies are mapped first and then iterations with less dependencies. If there are not enough processors, some or all of them are postponed.

```
Schedule (GBIs)
time iterations
1:   5  5    5  6    5  7    5  8    5  9    5 10    5 11    5 12    5 13    5 14    5 15    5 16    5 17    5 18    5 19    5 20    5 21    5 22
     6  5    6  6    6  7    6  8    7  5    7  6    8  5    8  6    9  5   10  5   11  5   12  5   13  5   14  5   15  5   16  5   17  5   18  5
    19  5   20  5   21  5   22  5
2:   6  9    6 10    6 11    6 12    6 13    6 14    6 15    6 16    6 17    6 18    6 19    6 20    6 21    6 22    7  7    7  8    7  9    7 10
     7 11    7 12    8  7    8  8    8  9    8 10    9  6    9  7    9  8   10  6   10  7   10  8   11  6   11  7   12  6   12  7   13  6   14  6
    15  6   16  6   17  6   18  6   19  6   20  6   21  6   22  6
3:   7 13    7 14    7 15    7 16    7 17    7 18    7 19    7 20    7 21    7 22    8 11    8 12    8 13    8 14    8 15    8 16    9  9    9 10
     9 11    9 12    9 13    9 14   10  9   10 10   10 11   10 12   11  8   11  9   11 10   12  8   12  9   12 10   13  7   13  8   13  9   14  7
    14  8   14  9   15  7   15  8   16  7   16  8   17  7   18  7   19  7
    20  7   21  7   22  7
4:   8 17    8 18    8 19    8 20    8 21    8 22    9 15    9 16    9 17    9 18    9 19    9 20   10 13   10 14   10 15   10 16   10 17   10 18
    11 11   11 12   11 13   11 14   11 15   11 16   12 11   12 12   12 13   12 14   13 10   13 11   13 12   14 10   14 11   14 12   15  9   15 10
    15 11   16  9   16 10   16 11   17  8   17  9   17 10   18  8   18  9   18 10   19  8   19  9   20  8   20  9   21  8   22  8
5:   9 21    9 22   10 19   10 20   10 21   10 22   11 17   11 18   11 19   11 20   11 21   11 22   12 15   12 16   12 17   12 18   12 19   12 20
    13 13   13 14   13 15   13 16   13 17   13 18   14 13   14 14   14 15   14 16   15 12   15 13   15 14   16 12   16 13   16 14   17 11   17 12
    17 13   18 11   18 12   18 13   19 10   19 11   19 12   20 10   20 11   20 12   21  9   21 10   21 11   22  9   22 10   22 11
6:  12 21   12 22   13 19   13 20   13 21   13 22   14 17   14 18   14 19   14 20   14 21   14 22   15 15   15 16   15 17   15 18   15 19   15 20
    16 15   16 16   16 17   16 18   17 14   17 15   17 16   18 14   18 15   18 16   19 13   19 14   19 15   20 13   20 14   20 15   21 12   21 13
    21 14   22 12   22 13   22 14
7:  15 21   15 22   16 19   16 20   16 21   16 22   17 17   17 18   17 19   17 20   17 21   17 22   18 17   18 18   18 19   18 20   19 16   19 17
    19 18   20 16   20 17   20 18   21 15   21 16   21 17   22 15   22 16   22 17
8:  18 21   18 22   19 19   19 20   19 21   19 22   20 19   20 20   20 21   20 22   21 18   21 19   21 20   22 18   22 19   22 20
9:  21 21   21 22   22 21   22 22
```

Table 3: schedule

proc nr	timesteps								
0	5 5	6 9	7 13	8 17	9 21	12 21	15 21	18 21	21 21
1	5 6	6 10	7 14	8 18	9 22	12 22	15 22	18 22	21 22
2	5 7	6 11	7 15	8 19	10 19	13 19	16 19	19 19	22 21
3	5 8	6 12	7 16	8 20	10 20	13 20	16 20	19 20	22 22
4	5 9	6 13	7 17	8 21	10 21	13 21	16 21	19 21	
5	5 10	6 14	7 18	8 22	10 22	13 22	16 22	19 22	
6	5 11	6 15	7 19	9 15	11 17	14 17	17 17	20 19	
7	5 12	6 16	7 20	9 16	11 18	14 18	17 18	20 20	
8	5 13	6 17	7 21	9 17	11 19	14 19	17 19	20 21	
9	5 14	6 18	7 22	9 18	11 20	14 20	17 20	20 22	
10	5 15	6 19	8 11	9 19	11 21	14 21	17 21	21 18	
11	5 16	6 20	8 12	9 20	11 22	14 22	17 22	21 19	
12	5 17	6 21	8 13	10 13	12 15	15 15	18 17	21 20	
13	5 18	6 22	8 14	10 14	12 16	15 16	18 18	22 18	
14	5 19	7 7	8 15	10 15	12 17	15 17	18 19	22 19	
15	5 20	7 8	8 16	10 16	12 18	15 18	18 20	22 20	
16	5 21	7 9	9 9	10 17	12 19	15 19	19 16		
17	5 22	7 10	9 10	10 18	12 20	15 20	19 17		
18	6 5	7 11	9 11	11 11	13 13	16 15	19 18		
19	6 6	7 12	9 12	11 12	13 14	16 16	20 16		
20	6 7	8 7	9 13	11 13	13 15	16 17	20 17		
21	6 8	8 8	9 14	11 14	13 16	16 18	20 18		
22	7 5	8 9	10 9	11 15	13 17	17 14	21 15		
23	7 6	8 10	10 10	11 16	13 18	17 15	21 16		
24	8 5	9 6	10 11	12 11	14 13	17 16	21 17		
25	8 6	9 7	10 12	12 12	14 14	18 14	22 15		
26	9 5	9 8	11 8	12 13	14 15	18 15	22 16		
27	10 5	10 6	11 9	12 14	14 16	18 16	22 17		
28	11 5	10 7	11 10	13 10	15 12	19 13			
29	12 5	10 8	12 8	13 11	15 13	19 14			
30	13 5	11 6	12 9	13 12	15 14	19 15			
31	14 5	11 7	12 10	14 10	16 12	20 13			
32	15 5	12 6	13 7	14 11	16 13	20 14			
33	16 5	12 7	13 8	14 12	16 14	20 15			
34	17 5	13 6	13 9	15 9	17 11	21 12			
35	18 5	14 6	14 7	15 10	17 12	21 13			
36	19 5	15 6	14 8	15 11	17 13	21 14			
37	20 5	16 6	14 9	16 9	18 11	22 12			
38	21 5	17 6	15 7	16 10	18 12	22 13			
39	22 5	18 6	15 8	16 11	18 13	22 14			
41		20 6	16 8	17 9	19 11				
42		21 6	17 7	17 10	19 12				
43		22 6	18 7	18 8	20 10				
44			19 7	18 9	20 11				
45			20 7	18 10	20 12				
46			21 7	19 8	21 9				
47			22 7	19 9	21 10				
48				20 8	21 11				
49				20 9	22 9				
50				21 8	22 10				
51				22 8	22 11				

Table 4: conventional mapping with 52 tasks

proc nr	timesteps								
0	5 5	6 9	8 11	10 13	12 15	8 19	6 20	5 21	9 22
1	5 6	7 7	9 9	11 11	13 13	9 19	7 20	6 21	10 22
2	6 5	7 8	9 10	11 12	13 14	10 19	8 20	7 21	11 22
3	6 6	8 7	10 9	12 11	14 13	14 17	9 20	8 21	12 22
4	5 7	8 8	10 10	12 12	14 14	15 15	10 20	9 21	13 22
5	5 8	9 6	11 8	13 10	15 12	15 16	11 20	10 21	14 22
6	5 9	6 10	7 13	8 17	11 17	16 15	12 20	11 21	15 22
7	6 7	6 11	7 14	9 15	12 16	16 16	13 20	12 21	16 22
8	6 8	6 12	8 12	9 16	12 17	17 14	14 20	13 21	17 22
9	7 5	7 9	8 13	10 14	13 15	19 6	16 19	14 21	18 22
10	7 6	7 10	8 14	10 15	13 16	19 7	17 17	15 21	19 21
11	8 5	7 11	9 11	10 16	13 17	19 8	17 18	16 21	19 22
12	8 6	8 9	9 12	11 13	14 15	19 9	18 17	17 21	20 21
13	9 5	8 10	9 13	11 14	14 16	19 10	18 18	18 21	20 22
14	5 10	9 7	10 11	11 15	15 13	15 17	19 16	19 19	21 19
15	5 11	9 8	10 12	12 13	15 14	16 17	20 6	19 20	21 20
16	5 12	10 6	11 9	12 14	16 12	17 15	20 7	20 19	21 21
17	10 5	10 7	11 10	13 11	16 13	17 16	20 8	20 20	21 22
18	11 5	10 8	12 8	13 12	16 14	13 18	20 9	21 5	22 5
19	12 5	11 6	12 9	14 10	17 11	14 18	20 10	21 6	22 6
20	5 13	11 7	12 10	14 11	17 12	15 18	20 11	21 7	22 7
21	5 14	12 6	13 7	14 12	17 13	16 18	20 12	21 8	22 8
22	5 15	6 13	13 8	15 9	7 18	18 13	20 13	21 9	22 9
23	13 5	6 14	13 9	15 10	8 18	18 14	20 14	21 10	22 10
24	14 5	6 15	14 7	15 11	9 18	18 15	17 19	21 11	22 11
25	15 5	7 12	14 8	16 8	10 18	18 16	19 17	21 12	22 12
26	5 16	12 7	7 15	16 9	11 18	11 19	15 20	21 13	22 13
27	5 17	13 6	7 16	16 10	12 18	12 19	16 20	21 14	22 14
28	16 5	14 6	7 17	17 7	18 7	13 19	17 20	21 15	22 15
29	17 5	15 6	8 15	17 8	18 8	14 19	18 19	21 16	22 16
30	5 18	6 16	8 16	9 17	18 9	15 19	18 20	21 17	22 17
31	18 5	6 17	9 14	10 17	18 10	19 11	19 18	21 18	22 18
32	5 19	16 6	14 9	11 16	18 11	19 12	20 15	5 22	22 19
33	19 5	17 6	15 7	16 11	18 12	19 13	20 16	6 22	22 20
34	5 20	6 18	15 8	17 9	6 19	19 14	20 17	7 22	22 21
35	20 5	18 6	16 7	17 10	7 19	19 15	20 18	8 22	22 22

Table 5: optimal processor mapping with 36 tasks

The optimal mapping uses just 36 processors and is given in table 5.
The specific algorithm (SA-version 1.0) is the one we have presented in [3] adapted for mapping onto distributed memory architectures. It is based on a Decision SA-algorithm(P), which decides if the UDL can be scheduled using P processors. For this class of algorithms, there is a usefull lower and a usefull upper bound.

- Lower Bound: Given a UDL, let t be the optimal parallel time. Then at least $\lceil \frac{|J^n|}{|t|} \rceil$ are needed. One can see that with fewer processors in t time instants the total number of possible computations would be less than $|J^n|$.
- Upper Bound: Given a UDL, let $|GB_M|$ be $max\{|GB_i| 0 \leq i \leq t\}$. Then at most $|GB_M|$ are needed. Clearly, $|GB_M|$ processors are adequate because at no time would there be more than $|GB_M|$ vertices to be executed in parallel.

6.1 The SA Algorithm

In this subsection, we shall formally state the scheduling algorithm, SA.

- Input: $A(J^n, DS), GB_i, GT_i, 0 \leq i \leq t$, where t is the optimal parallel time.
- Output: P (the optimal number of processors) and an optimal schedule.

Perform a binary search with $\lceil \frac{|J^n|}{|t|} \rceil$ and $|GB_M|$ as lower and upper bounds respectively, and return the smallest natural P for which Decision-SA(P) returns YES.

	test1	test2
serial	99.994133	323.991976
parallel	10.089632	9.919886

Table 6: run times

6.2 The Decision-SA Algorithm (ver. 1.0)

- Input: $GB_i, GT_i, 0 \leq i \leq t$, and P.
- Output: YES (and a schedule) iff P PEs execute $A(J^n, DS)$ in t.

7 Code Generator, Run Time Results

Given the mapping, the code generator creates an spmd (single program multiple data) program which uses PVM for task creation and communication. The program spawns the necessary tasks and sends the initial values to the appropriate tasks. Each task receives the data for one iteration, calculates one iteration, sends the data to the processors needing it and proceeds to the next iteration. Communication from a task to itself is eliminated, by passing the values on memory.
We ran some experiments on the following two for-loops:

```
/* test 1 */
for (i = 1; 10;)
  for (j = 1; 10;) {
    a[i][j] = a[i-1][j-1];
  }

/* test 2 */
for (i = 5; i <= 22;)
  for (j = 5; j <= 22;) {
    a[i][j] = a[i - 1][j - 4] + b[i - 4][j - 1]
    b[i][j] = a[i - 3][j - 5] + a[i - 5][j - 3] + b[i - 2][j - 2];
  }
}
```

For test 1 and test 2, the minimum number of tasks is 10 and 36 respectively. At the innermost loop, we introduced a delay of 1 second, which corresponds to a big amount of computation. The numbers in table 6 show, that for test 1 a speedup of 10, and for test 2 a speedup of 32 is achieved, which is almost equal to the number of tasks. Because we did not have 36 processors, we implemented the delay with a sleep statement and used only 10 processors. That allows several tasks to execute in parallel on one processor, because some of them are sleeping. Naturally, when the delay is a real computation 36 processors are required.

8 Conclusion

Loopdep has been implemented to parallelize for-loops. Its purpose is to show the feasibility of for-loop parallelization for distibuted memory machines. Assuming that the for-loops contain a big amount of computation, an almost linear speedup is achieved. Future work will be done in communication minimization and in embedding this program to a bigger compiler.

References

[1] T. Andronikos, N. Koziris, Z. Tsiatsoulis, G. Papakonstantinou and P. Tsanakas, "An Efficient Algorithm for the Optimal Linear Schedule of Uniform Dependence Algorithms," *Proceedings of the First ECPD International Conference on Advanced Robotics and Intelligent Automation*, pp. 140-145, Sep. 1995.

[2] 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 ," to appear in *Journal of Parallel Algorithms and Applications.*

[3] T. Andronikos, N. Koziris, Z. Tsiatsoulis, G. Papakonstantinou and P. Tsanakas, "Optimal Scheduling of Uniform Dependence Loops in Polynomial Time," *Second ECPD International Conference on Advance Robotics and Intelligent Automation,* pp. 605-610, Sep. 1996.

[4] M. Barnett, C. Lengauer, "A systolizing compilation scheme for nested loops iwth linear bounds.," *In P.E. Lauer, editor, Functional Programming, Concurrency, Simulation and Automated Reasoning, Lecture Notes in Computer Science 693*, pp. 374-398

[5] A. Darte and Y. Robert, "Constructive Methods for Scheduling Uniform Loop Nests," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 8, pp. 814–822, Aug. 1994.

[6] A. Darte and F. Vivien, "Automatic parallelization based on multidimensional scheduling," *Laboratoire de l'Informatique du Parallelisme, Ecole Normale Superieure de Lyon* Technical Report 94-24, September 1994

[7] P. Feautrier, "Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time," *Int. J. Parallel Programming*, vol. 21, no. 5, pp. 313-348, Oct 1992.

[8] M. Grieble and C. Lengauer, "The Loop Parallelizer LooPo—Announcement" in *Ninth Workshop on Languages and Compilers for Parallel Computing", Lecture Notes in Computer Science*, 1996.

[9] N. Koziris, G. Papakonstantinou and P. Tsanakas, "Automatic Loop Mapping and Partitioning into Systolic Architectures," *Proceedings of the 5th Panhellenic Conference on Informatics, Athens*, pp. 777–790, Dec. 1995.

[10] 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.

[11] L. Lamport, "The parallel execution of DO loops," *Comm. 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] W. Shang and J.A.B. Fortes, "Time optimal linear schedules for algorithms with uniform dependencies," in *IEEE Trans. Comput.*, vol. 40, no. 6, pp. 723–742, June 1991.

[14] W. Shang and J.A.B. Fortes, "Independent partitioning of algorithms with uniform dependencies," in *IEEE Trans. Comput.*, vol. 41, no. 2, pp. 190–206, Feb. 1992.

[15] J.-P. Sheu and T.-H. Tai, "Partitioning and Mapping Nested Loops on Multiprocessor Systems," in *IEEE Trans. Parallel Distrib. Syst.*, vol. 2, no. 4, pp. 430–439, Oct. 1991.

A Grammar

The EBN syntax of the Loopdep language is given below:

```
program:
       program_headline
       definition_part
       compound_statement
       ;

program_headline:
       PROGRAM IDENT ;

definition_part:
       | definition_part INT list_of_var ';'

list_of_var:
         variable_def
       | list_of_var ',' variable_def
       ;

variable_def:
         IDENT
       | variable_def '[' INTCONST ']'
       ;

compound_statement:
       '{' for_statements '}' ;

for_statements:
       | for_statement for_statements
       ;

for_statement:
       FOR '(' IDENT '=' INTCONST ';'  INTCONST ')'
       for_body
       ;

for_body:
       | for_statement
       | '{' assign_statements '}'
       ;

assign_statement:
       l_value '=' exp ';'
       ;

assign_statements:
         assign_statement
       | assign_statement assign_statements
       ;

l_value:
         IDENT
```

```
        | l_value '[' simple_exp ']'
        ;

simple_exp:
        IDENT addit
        ;

addit:
        | addop INTCONST
        ;

addop:
        '+' | '-' ;

exp:      INTCONST
        | l_value
        | exp addop exp
        | exp '*' exp
        | exp '/' exp
        | exp '%' exp
        | '-' exp
        | '(' exp ')'
        ;
```

B Brief Biographies

Alexandros Pappas obtained a M.Sc. in Computer Science at the Swiss Federal Institute of Technology in Zurich in 91 and is now a PhD candidate at the National Technical University of Athens (NTUA). His research interests are parallelization techniques, compilers and parallel programming.

Theodore Andronikos obtained a M.Sc. in Computer Science at NTUA in 91 and is finishing now his PhD at NTUA. He is focusing on the theory for parallelization of sequential algorithms.

Nektarios Koziris obtained a M.Sc. in Computer Science at NTUA in 93 and is working towards a PhD. He is focusing on automatic parallelization of nested loop Algorithms into special purpose parallel architectures.

George Papakonstantinou is a Professor at the Computer Science section of NTUA and head of the Digital Systems and Computers Laboratory. His research interests include knowledge engineering, syntactic pattern recognition, multimedia technology, as well as parallel architectures and languages.

Panagiotis Tsanakas is an associate Professor at the Computer Science section of NTUA. His research interests include parallel/distributed computing systems and applications, automated synthesis of VLSI architectures, image/signal processing, and intelligent learning systems.