UC Irvine
ICS Technical Reports

Title
Software pipelining of non-vectorizable loosely nested loops

Permalink
https://escholarship.org/uc/item/58w2r7f6

Authors
Kim, Ki-chang
Nicolau, Alexandru

Publication Date
1991

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

Z
699
C3
no. 91-04

Software pipelining of non-vectorizable loosely nested loops

LIBRARY
University of California
IRVINE

Ki-chang Kim and Alexandru Nicolau
Department of Information and Computer Science
University of California, Irvine
Irvine, CA. 92717

Technical Report #91-04
April 1991

Notice: This Material
may be protected
by Copyright Law
(Title 17 U.S.C.)

Software pipelining of non-vectorizable loosely nested loops

Ki-Chang Kim and Alexandru Nicolau
Computer Science Department
University of California - Irvine
Irvine, CA 927171
(714)856-4079;kkim@ics.uci.edu,nicolau@ics.uci.edu

Abstract

This paper presents a new technique to parallelize non-vectorizable loosely nested loops. Loosely nested loops represent the general form of nested loops. Previously, the attempt of parallelizing nested loops, e.g. the wavefront method, has been limited to tightly nested loops, a restricted class of general nested loops. Our method overcomes this limitation. It consists of two steps: computing the exact time step of each statement instance and capturing and expressing the parallel statement instances whose time steps are equal. We provide efficient algorithms for both steps and illustrate their practicality by parallelizing the well-known SOR algorithm. The parallel SOR was run on a Sequent machine with 1 to 7 physical processors resulting significant speed-ups.

Key words: parallelizing compiler, nested loop parallelization, fine grain parallelization, software pipelining, loop scheduling

1 Introduction

Parallelizing loops is essential to achieve a speed-up in parallel machines. Vectorization is a well-known technique to exploit parallelism out of loops. However, this technique is not applicable when the loop is not vectorizable. For non-vectorizable loops, only some limited forms of loops are parallelizable using the current loop parallelization techniques.

The first case is single loops – one-dimensional loops. A group of techniques, generally called *software pipelining* [SDWX87][Lam88][AiNi88], have been developed to parallelize single loops whether they are vectorizable or not. The basic method of these techniques is *compaction*. The loop is unfolded and compacted, exposing parallelism between statements. The unfolding process is not indefinite; it stops when the parallel form of the loop becomes predictable. One of them, *Perfect Pipelining* [AiNi88], successfully parallelizes loops with arbitrary control flow near-optimally; if there is no if-statement in the loop, it guarantees to parallelize the loop optimally.

The second case is tightly nested loops. An n-dimensional loop is tightly nested if all of its loops are iterating over the same set of statements. For this case, the general parallelization technique is wavefronting [Mura71][Lamp74][Wolf87][Bane90][LaWo90]. This method can decide which iterations can be done in parallel at each time step; these parallel iterations at each time step form the wavefront.

It is elegant and efficient, but it ignores the parallelism inside iterations and is limited to tightly nested form.

We propose a technique that can parallelize loosely nested loops whether they are vectorizable or not. Loosely nested loops represent the general form of n-dimensional loops. We note that DOACROSS [Cytr86] can parallelize loosely nested loops, too. However, its schedule shows only how to assign tasks (statements or iterations) to the processors; therefore, we know which processor executes which tasks, but we do not know which tasks can be done in parallel at a certain time step. As a result, synchronous machines, e.g. superscalar or VLIW machines, can accept our schedule but not that of DOACROSS. In fact, our method is best suited for VLIW/superscalar machines since it concentrates parallelism in innermost loops, but it applies to other asynchronous machines, too, as we can see in Section 4. Another technique that handles loosely nested loops is *Loop Quantization* [Nico87]. The loop is unwound by some amount for each dimension and compacted, exposing parallelism. However, the parallelism it exploits is limited by the unwinding sizes.

Our technique exposes parallelism without explicitly taking resource constraints into account. However, since the parallelism is concentrated in inner loops, it is relatively easy to map our schedule to fewer resources than those implied by the schedule initially produced by our algorithm.

Basically, we schedule every statement instance[1] individually at a certain time step and calculate which statement instances are at the same time step. Computing the time step of each statement instance (Section 3.2) and capturing and expressing the parallel statement instances whose scheduled time steps are all equal (Section 3.1) are the critical tasks in this process and the two main topics of this paper. Section 4 shows an application of our method to a real problem.

2 Definitions

Before going into the details of our method, we need to define a few terms. Since we are dealing with loosely nested loops, a new indexing scheme is introduced. We will use a tree, called *loop tree*, to capture the structure of a nested loop.[2] In this tree, each node corresponds to a loop in the nested loop. The index of each node, then, is represented by the *path* from the root to the corresponding node. We represent the index of the outermost loop (the root node) by I_1 (we assume there is only one outermost loop); therefore, the i_{th} child loop of this outermost loop has index I_{1i}, the j_{th} child loop of this child loop has index I_{1ij}, and so on. For example, the nested loop in Figure 1(a) will have the loop tree in Figure 1(b). In the figure, the index of each loop is shown next to the corresponding

[1] We distinguish a "statement instance" and a "statement". In a loop, a "statement" is executed a number of times with different indices at each execution. Each instance of this statement at each execution is called a "statement instance".

[2] We note that the structure of the *loop tree* is similar to the *control dependence graph*[KMC72]. The difference lies in index notation.

node.

As can be noted from the above example, we regard a simple statement as a loop with a single iteration. Therefore, the outermost loop, whose index is represented by I_1, has three child loops whose indices are I_{11}, I_{12}, and I_{13}. The first child is a statement; the other two, loops. Also, we assume all loops are normalized such that the lower bounds are always zero's. The upper bounds are represented by $N_{path} - 1$, where *path* shows the position of the corresponding loop in the loop tree, as in loop indices. Note that all leaf nodes in the loop tree are simple statements.

For each node in the loop tree, we define three values: H_{path}, d_{path}, and S_{path}. H_{path} is the number of child loops of loop I_{path} (or node I_{path}).[3] The second value, d_{path} is the amount of delay between the iterations of loop I_{path}. We will call this value the delay of loop I_{path}. Finally, S_{path} is the *size* of loop I_{path}, which is defined as,

$$S_{path} = \begin{cases} (N_{path} - 1)d_{path} + S_{1,p_1,\ldots,p_x,1} + \ldots + S_{1,p_1,\ldots,p_x,H_{path}} & \text{if node } I_{path} \text{ is a loop, not a statement} \\ 1 & \text{if node } I_{path} \text{ is a statement,} \end{cases}$$

when $path = (1, p_1, \ldots, p_x)$.

Note that

$$S_{1,p_1,\ldots,p_x,1} + \ldots + S_{1,p_1,\ldots,p_x,H_{path}}$$

is the sum of the *sizes* of all the child loops of loop I_{path}. We will use a short-hand representation $S_{1,p_1,\ldots,p_x,*}$ for this. For example, the size of loop I_{1ij} is

$$S_{1ij} = (N_{1ij} - 1)d_{1ij} + S_{1ij*}.$$

These values are needed to transform the loop correctly. Especially, the delays, d_{path}, are what preserve the semantics of the original loop after transformation. Since a statement can be surrounded by several loops, and each of them has a delay of its own, the time step to execute an instance of this statement can be computed by accumulating the delays it suffers at each loop. An example is given in Figure 2(a)-(b). In the figure, loop I_1 has delay of 2, while loop I_2 has delay of 1. Therefore, the execution of statement instance A23, for example, is delayed by 2*2 time steps at the first dimension and by 3*1 time steps at the second dimension.

A node in the loop tree is executed a number of times dictated by the upper bounds of its predecessors. For example, loop I_{121} is repeated by $N_1 \times N_{12}$. The *partial iteration vector* shows which copy is active; that is, it shows the index values of the currently active surrounding loops. Therefore, the instance of loop I_{121} at *partial iteration vector* (iv_1, iv_{12}) is its copy when $I_1 = iv_1$, and $I_{12} = iv_{12}$.

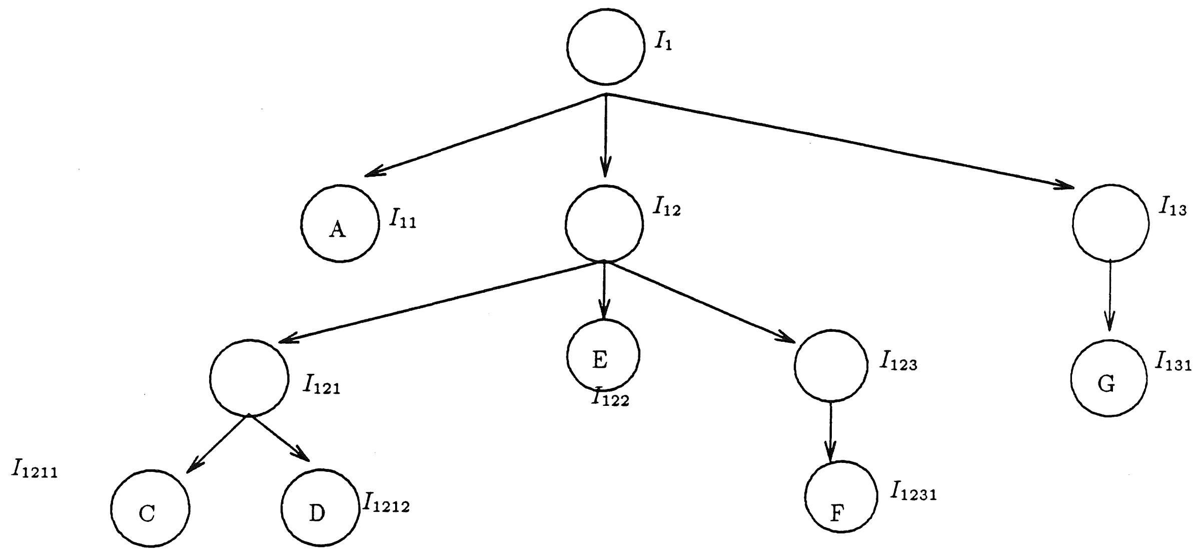
3 Parallelizing loosely nested loops

As explained in the introductory section, two steps are needed to parallelize loosely nested loops: computing the exact time steps of all statement instances and capturing and expressing the parallel

[3] We will use *node* and *loop* interchangeably throughout the paper.

```
For $I_1 = 0$ to $N_1 - 1$
    A
    For $I_{12} = 0$ to $N_{12} - 1$
        For $I_{121} = 0$ to $N_{121} - 1$
            C
            D
        Endfor
        E
        For $I_{123} = 0$ to $N_{123} - 1$
            F
        Endfor
    endfor
    For $I_{13} = 0$ to $N_{13} - 1$
        G
    Endfor
Endfor
```

(a) An example loosely nested loop



(b) Its loop tree

Figure 1: An example loosely nested loop and its loop tree.

```
For $i_1 = 0$ to $N_1 - 1$
    For $i_2 = 0$ to $N_2 - 1$
        A:A(i1,i2)=f(A(i1,i2-1),B(i1-1,i2))
        B:B(i1,i2)=g(A(i1,i2))
    Endfor
Endfor
```

(a) The source code

time step				schedule											
0	A00														
1	B00	A01													
2		B01	A02			A10									
3			B02	A03		B10	A11								
4				B03	A04		B11	A12			A20				
5					B04	A05		B12	A13		B20	A21			
6						B05	.		B13	A14		B21	A22		
7							. .			B14	A15		B22	A23	
8								. .			B15	.		B23	.
9									. .			. .			.

(b) The execution schedule

Figure 2: An example of execution schedule with delays.

statement instances whose time steps are equal. Computing time steps of the statement instances is the problem of finding efficient delays for all nodes in the loop tree because the time step of a statement instance depends on the delay values of its surrounding loops. We first assume all these delay values are computed, and show how the process of capturing and expressing parallel statement instances can be done. Then, we will present a method to compute efficient delay values for all nodes in the loop tree.

3.1 Transformation

All nodes (except the root and leaf nodes) in the loop tree are transformed into parallel form as follows. Suppose we want to transform node I_{1ij} into parallel form. Assume it has three child loops. Then, the I_{1ij} loop below,

```
.....
.....
 For $I_{1ij} = 0$ to $N_{1ij} - 1$
     For $I_{1ij1} = 0$ to $N_{1ij1} - 1$
       ....
     Endfor
     For $I_{1ij2} = 0$ to $N_{1ij2} - 1$
       ....
     Endfor
     For $I_{1ij3} = 0$ to $N_{1ij3} - 1$
       ....
     Endfor
 Endfor
...... ......
```

will be transformed into

```
.....
.....
 Forall $I_{1ij} = L_{1ij}$ to $U_{1ij}$
     Case $t_{1ij} - I_{1ij}d_{1ij}$ is
     0 to $S_{1ij1} - 1$ : For $I_{1ij1} = 0$ to $N_{1ij1} - 1$
                              ....
                           Endfor
     $S_{1ij1}$ to $S_{1ij1} + S_{1ij2} - 1$ : For $I_{1ij2} = 0$ to $N_{1ij2} - 1$
                                                  ....
                                               Endfor
     $S_{1ij1} + S_{1ij2}$ to $S_{1ij1} + S_{1ij2} + S_{1ij3} - 1$ : For $I_{1ij3} = 0$ to $N_{1ij3} - 1$
                                                                      ....
                                                                   Endfor
     Endcase
 Endforall
......
......
```

Note that the inner loops of loop I_{1ij} are not parallelized yet. They can be parallelized by applying the same process recursively. S_{path} is the size of the loop I_{path} as explained in Section 2. L_{1ij} and U_{1ij} are the new loop bounds. The value of t_{1ij} is computed by

$$t_{1ij} = t - TB(I_{1ij}, (iv_1, iv_{1i}))$$

where (iv_1, iv_{1i}) is the partial iteration vector of loop I_{1ij} (see Section 2 for the definition of a partial iteration vector).

TB is the starting time step of its argument loop at the designated partial iteration vector, whose computation will follow shortly, while t is the global time step. Because t_{1ij} represents the time passed since the instance of loop I_{1ij} at partial iteration vector (iv_1, iv_{1i}) began, we call it the *local time step* of loop I_{1ij}.

The computation of TB proceeds as follows. Assume we want to compute the TB for loop I_{1ij} at a partial iteration vector (iv_1, iv_{1i}). If we draw the surrounding loops of loop I_{1ij} , we get

```
For $I_1 = 0$ to $N_1 - 1$
    For $I_{11} = 0$ to $N_{11} - 1$
        ...
    Endfor
    For $I_{12} = 0$ to $N_{12} - 1$
        ...
    Endfor
    ...
    ...
    For $I_{1i} = 0$ to $N_{1i} - 1$
        For $I_{1i1} = 0$ to $N_{1i1} - 1$
            ...
        Endfor
        For $I_{1i2} = 0$ to $N_{1i2} - 1$
            ...
        Endfor
        ...
        ...
        For $I_{1ij} = 0$ to $N_{1ij} - 1$
            ...
        Endfor
        ...
        ...
    Endfor
    ...
    ...
Endfor.
```

We want to calculate the starting time step of the instance of loop I_{1ij} when $I_1 = iv_1$, and $I_{1i} = iv_{1i}$. Since each iteration of I_1 is delayed by d_1, the iv_1-th iteration will start at $iv_1 d_1$ time step. (Note the iteration count starts from zero.) At iv_1-th iteration, we have to wait until all previous loops before I_{1i} are executed. Therefore, $S_{11} + \ldots + S_{1,i-1}$ time steps should be passed. At this point, we again have to wait for the iv_{1i}-th iteration of I_{1i} loop. This adds $iv_{1i}d_{1i}$ times steps to the delay time accumulated so far. Finally, we have to wait until all the previous loops before I_{1ij} loop at the iv_{1i}-th iteration are executed. So, the starting time step of the desired instance of loop I_{1ij} is

$$TB(I_{1ij}, (iv_1, iv_{1i})) = iv_1 d_1 + S_{11} + \ldots + S_{1,i-1} + iv_{1i}d_{1i} + S_{1i1} + \ldots + S_{1,i,j-1}.$$

The last variables we need to compute are L_{1ij} and U_{1ij}, the new loop bounds. L_{1ij} is the iteration that spans t_{1ij}, the local time step of the current instance of loop I_{1ij}, for the first time. U_{1ij} is the last iteration that spans t_{1ij}. Therefore, if $L_{1ij} > 0$, the ending time step[4] of the iteration $L_{1ij} - 1$ should be strictly less than t_{1ij}, and the ending time step of the iteration L_{1ij} should be greater than

[4]Actually local ending time step. We are looking at only the current instance of loop I_{1ij}. Every time step here, while we are explaining the computation of L_{1ij} and U_{1ij}, refers to the local time step of the current instance of loop I_{1ij}.

$$TB(I_{1,p_1,\ldots,p_{x-1},p_x},(iv_1,iv_{1,p_1},\ldots,iv_{1,p_1,p_2,\ldots,p_{x-1}})) =$$

$$\begin{cases} 0 & \text{if the path } p_1,p_2,\ldots,p_{x-1},p_x \text{ is nil} \\ iv_1 d_1 + S_{11} + S_{12} + \ldots + S_{1,p_1-1} & \\ +iv_{1,p_1} d_{1,p_1} + S_{1,p_1,1} + S_{1,p_1,2} + \ldots + S_{1,p_1,p_2-1} & \\ +iv_{1,p_1,p_2} d_{1,p_1,p_2} + S_{1,p_1,p_2,1} + S_{1,p_1,p_2,2} + \ldots + S_{1,p_1,p_2,p_3-1} & \\ \ldots\ldots & \\ +iv_{1,p_1,\ldots,p_{x-1}} d_{1,p_1,\ldots,p_{x-1}} + S_{1,p_1,\ldots,p_{x-1},1} + \ldots + S_{1,p_1,\ldots,p_{x-1},p_x-1} & \text{if the path } p_1,p_2,\ldots,p_{x-1},p_x \text{ is not nil} \end{cases}$$

$$L_{1,p_1,\ldots,p_x} =$$

$$\begin{cases} MAX(0,\lceil (t_{1,p_1,\ldots,p_x} + 1 - S_{1,p_1,\ldots,p_x,*})/d_{1,p_1,\ldots,p_x} \rceil) & \text{if } d_{1,p_1,\ldots,p_x} > 0 \\ 0 & \text{if } d_{1,p_1,\ldots,p_x} = 0 \end{cases}$$

$$U_{1,p_1,\ldots,p_x} =$$

$$\begin{cases} MIN(N_{1,p_1,\ldots,p_x} - 1, \lfloor t_{1,p_1,\ldots,p_x}/d_{1,p_1,\ldots,p_x} \rfloor) & \text{if } d_{1,p_1,\ldots,p_x} > 0 \\ N_{1,p_1,\ldots,p_x} - 1 & \text{if } d_{1,p_1,\ldots,p_x} = 0 \end{cases}$$

$$S_{1,p_1,\ldots,p_x} = \begin{cases} (N_{1,p_1,\ldots,p_x} - 1)d_{1,p_1,\ldots,p_x} + S_{1,p_1,\ldots,p_x,1} + \ldots + S_{1,p_1,\ldots,p_x,H_{path}} & \text{if node } I_{1,p_1,\ldots,p_x} \text{ is a loop} \\ 1 & \text{if node } I_{1,p_1,\ldots,p_x} \text{ is a statement.} \end{cases}$$

$$\text{MAX-GLOBAL-TIME-STEP} = S_1 - 1$$

Figure 3: Transformation formula.

or equal to t_{1ij}. Also if $U_{1ij} < N_{1ij} - 1$, the starting time step of U_{1ij} should be greater than or equal to t_{1ij}, while that of $U_{1ij} + 1$ should be strictly greater than t_{1ij}. Therefore, when $L_{1ij} > 0$ and $U_{1ij} < N_{1ij} - 1$, we get the following inequalities to be satisfied.

$$(L_{1ij} - 1)d_{1ij} + S_{1ij*} < t_{1ij} \leq L_{1ij} d_{1ij} + S_{1ij*}$$

$$U_{1ij} d_{1ij} \leq t_{1ij} < (U_{1ij} + 1)d_{1ij}$$

Solving these with the constraints that L_{1ij} is an integer greater than or equal to zero, and U_{1ij} is an integer less than or equal to $N_{1ij} - 1$, we get

$$L_{1ij} = MAX(0, \lceil (t_{1ij} + 1 - S_{1ij*})/d_{1ij} \rceil),$$

and

$$U_{1ij} = MIN(N_{1ij} - 1, \lfloor t_{1ij}/d_{1ij} \rfloor).$$

Now, the same parallelization process can be repeated for all the intermediate nodes. The parallelization of the leaf nodes is simple: just leave them untouched. For the root node (the outermost loop), we parallelize it following the above process, but this time add another loop on top of it. The new outermost loop is a sequential loop, and its index is the global time step. At each global time step, the sequential outermost loop specifies which statements of which loops can be executed in parallel.

The general formula for TB, L, U, and S are in Figure 3. The derivations of the first three are straightforward from the above explanations, and that of the last is borrowed from Section 2. Note

```
For $t = 0$ to MAX-GLOBAL-TIME-STEP
   Forall $I_1 = L_1$ to $U_1$
        Case $t_1 - I_1 d_1$ is
        0 : A
        1 to $S_{12}$ :
            Forall $I_{12} = L_{12}$ to $U_{12}$
                 Case $t_{12} - I_{12} d_{12}$ is
                 0 to $S_{121} - 1$:
                     Forall $I_{121} = L_{121}$ to $U_{121}$
                          Case $t_{121} - I_{121} d_{121}$ is
                          0: C
                          1: D
                          Endcase
                     Endforall
                 $S_{121}$: E
                 $S_{121} + 1$ to $S_{121} + S_{123}$:
                        Forall $I_{123} = L_{123}$ to $U_{123}$
                               F
                        Endforall
                 Endcase
            Endforall
        $S_{12} + 1$ to $S_{12} + S_{13}$:
               Forall $I_{13} = L_{13}$ to $U_{13}$
                      G
               Endforall
        Endcase
   Endforall
Endfor
```

Figure 4: Parallel form for the loop in Figure 1(a).

that $TB(I_1) = 0$, which is the case when the path $p_1, p_2, \ldots, p_{x-1}, p_x$ is nil, because by definition it is the starting time step of the outermost loop. For completeness, we have included the formula for MAX-GLOBAL-TIME-STEP in the figure. MAX-GLOBAL-TIME-STEP is the time step of the last statement executed, or 1 time step less than the size of the root node (the outermost loop).

We show an example to illustrate the whole process of transformation. The loop in Figure 1(a) will be transformed into the loop in Figure 4. The outermost loop (with index t) is the sequential loop; all loops inside are parallel. Assuming that $d_1 = 3, d_{13} = 1$, and all other delays are zero's (the computation of delays are discussed in Section 3.2), the unknown variables in Figure 4 can be calculated as follows.

First, let's compute all S_{path} values. Since a single statement has a size of 1, $S_{11} = S_{1211} = S_{1212} = S_{122} = S_{1231} = S_{131} = 1$. And at the next level, $S_{121} = (N_{121} - 1)d_{121} + 2, S_{123} = (N_{123} - 1)d_{123} + 1$, and $S_{13} = (N_{13} - 1)d_{13} + 1$. Based on these values,

$$S_{12} = (N_{12} - 1)d_{12} + (N_{121} - 1)d_{121} + 2 + 1 + (N_{123} - 1)d_{123} + 1,$$

and finally

$$\begin{aligned} S_1 &= (N_1 - 1)d_1 + 1 + (N_{12} - 1)d_{12} + (N_{121} - 1)d_{121} + 2 + 1 + (N_{123} - 1)d_{123} + 1 + (N_{13} - 1)d_{13} + 1 \\ &= (N_1 - 1)d_1 + (N_{12} - 1)d_{12} + (N_{121} - 1)d_{121} + (N_{123} - 1)d_{123} + (N_{13} - 1)d_{13} + 6. \end{aligned}$$

Then,

$$\text{MAX-GLOBAL-TIME-STEP} = (N_1 - 1)3 + (N_{13} - 1) + 6 - 1 = 3N_1 + N_{13} + 1,$$

$$t_1 = t - TB(I_1) = t,$$

$$L_1 = MAX(0, \lceil (t + 1 - S_{1,*})/3 \rceil) = MAX(0, \lceil (t - N_{13} - 4))/3 \rceil),$$

$$U_1 = MIN(N_1 - 1, \lfloor t/3 \rfloor),$$

$$t_{12} = t - TB(I_{12}, (I_1)^5) = t - (I_1 d_1 + S_{11}) = t - 3I_1 - 1,$$

$$L_{12} = 0, U_{12} = N_{12} - 1,$$

$$t_{121} = t - TB(I_{121}, (I_1, I_{12})) = t - 3I_1 - 1,$$

$$L_{121} = 0, U_{121} = N_{121} - 1,$$

$$t_{123} = t - TB(I_{123}, (I_1, I_{12})) = t - (I_1 d_1 + S_{11} + I_{12} d_{12} + S_{121} + S_{122}) = t - (3I_1 + 1 + 3) = t - 3I_1 - 4,$$

$$L_{123} = 0, U_{123} = N_{123} - 1,$$

$$t_{13} = t - TB(I_{13}, (I_1)) = t - (I_1 d_1 + S_{11} + S_{12}) = t - (3I_1 + 1 + 4) = t - 3I_1 - 5,$$

$$L_{13} = MAX(0, \lceil (t - 3I_1 - 5 + 1 - S_{1,3,*})/1 \rceil) = MAX(0, t - 3I_1 - 5),$$

$$U_{13} = MIN(N_1 - 1, t - 3I_1 - 5).$$

Therefore, the final instantiated parallel loop is as shown in Figure 5. The original sequential version of this loop requires $((2N_{121} + 1 + N_{123})N_{12} + 1 + N_{13})N_1$ time steps. The parallel one requires $3N_1 + N_{13} + 1$ time steps (given a sufficient number of processors).

3.2 Computing delays

As can be seen in Section 3.1, the parallel execution time of our transformed loop, MAX-GLOBAL-TIME-STEP, depends on the delays and loop bounds of the nesting loops. Since loop bounds are not adjustable, we need to compute a set of delays that will minimize the parallel execution time.

To ensure correctness, the delays should satisfy a set of inequalities, explained as follows. Suppose loop I_{1ik} depends on loop I_{1ij}[6] with dependence distance vector (v_1, v_{1i}).[7] Then, the instance of loop I_{1ik} at some iteration vector $(iv_1 + v_1, iv_{1i} + v_{1i})$ can not start until the instance of loop I_{1ij} at (iv_1, iv_{1i}) completes its execution. Therefore, we have

$$TB(I_{1ij}, (iv_1, iv_{1i})) + S_{1ij} - 1 < TB(I_{1ik}, (iv_1 + v_1, iv_{1i} + v_{1i})).$$

[5] The partial iteration vector of the current instance of loop I_{12} is given by index I_1

[6] Actually, some statement in loop I_{1ik} depends on some other statement in loop I_{1ij}.

[7] By [Wolf82], dependence information is available only for the common nesting loops of the involved tasks; in this case, we are looking at the dependence between loop I_{1ik} and I_{1ij}, and the common nesting loops of both are loop I_1 and loop I_{1i}. Therefore, the dependence distance vector between loop I_{1ik} and I_{1ij} can have only two elements in it corresponding to the two common loops.

```
For $t = 0$ to $3N_1 + N_{13} + 1$
    Forall $I_1 = MAX(0, \lceil (t - N_{13} - 4))/3 \rceil)$ to $MIN(N_1 - 1, \lfloor t/3 \rfloor)$
          Case $t - 3I_1$ is
          0 : A
          1 to 4:
                Forall $I_{12} = 0$ to $N_{12} - 1$
                      Case $t - 3I_1 - 1$ is
                      0 to 1:
                            Forall $I_{121} = 0$ to $N_{121} - 1$
                                  Case $t_{121} - I_{121}d_{121}$ is
                                  0: C
                                  1: D
                                  Endcase
                            Endforall
                      2: E
                      3 to 4:
                        Forall $I_{123} = 0$ to $N_{123} - 1$
                              F
                        Endforall
                      Endcase
                Endforall
          5 to $4 + N_{13}$:
                Forall $I_{13} = MAX(0, t - 3I_1 - 5)$ to $MIN(N_1 - 1, t - 3I_1 - 5)$
                      G
                Endforall
          Endcase
    Endforall
Endfor
```

Figure 5: Fully instantiated parallelized form.

Assuming $k \leq j$,

$$TB(I_{1ij}, (iv_1, iv_{1i}))$$

$$= TB(I_{1,i,j-1}, (iv_1, iv_{1i})) + S_{1,i,j-1}$$

$$= TB(I_{1,i,j-2}, (iv_1, iv_{1i})) + S_{1,i,j-2} + S_{1,i,j-1}$$

$$............$$

$$= TB(I_{1,i,j-(j-k)}, (iv_1, iv_{1i})) + S_{1,i,j-(j-k)} + \ldots + S_{1,i,j-1}$$

$$= TB(I_{1,i,k}, (iv_1, iv_{1i})) + S_{1,i,k} + \ldots + S_{1,i,j-1}.$$

Furthermore, from Figure 3,

$$TB(I_{1ik}, (iv_1 + v_1, iv_{1i} + v_{1i}))$$

$$= (iv_1 + v_1)d_1 + S_{11} + \ldots + S_{1,i-1} + (iv_{1i} + v_{1i})d_{1i} + S_{1i1} + \ldots + S_{1,i,k-1}$$

$$= iv_1 d_1 + S_{11} + \ldots + S_{1,i-1} + iv_{1i}d_{1i} + S_{1i1} + \ldots + S_{1,i,k-1} + v_1 d_1 + v_{1i}d_{1i}$$

$$= TB(I_{1ik}, (iv_1, iv_{1i})) + v_1 d_1 + v_{1i}d_{1i}.$$

Therefore, after simplification, the inequality to be satisfied is

$$v_1 d_1 + v_{1i}d_{1i} > S_{1ik} + S_{1,i,k+1} + \ldots + S_{1,i,j-1} + S_{1ij} - 1.$$

Now, if $k > j$,

$$TB(I_{1ij}, (iv_1, iv_{1i}))$$
$$= TB(I_{1,i,k}, (iv_1, iv_{1i})) - S_{1,i,k-1} - \ldots - S_{1,i,j+1} - S_{1ij}.$$

Therefore, the inequality in this case is

$$v_1 d_1 + v_{1i} d_{1i} > -S_{1,i,k-1} - S_{1,i,k-2} - \ldots - S_{1,i,j+2} - S_{1,i,j+1} - 1.$$

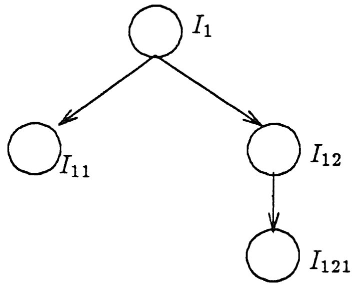
In general, the inequality to be satisfied due to the dependence between $I_{1,p_1,\ldots,p_x,j}$ and $I_{1,p_1,\ldots,p_x,k}$ with a dependence distance vector $(v_1, v_{1,p_1}, \ldots, v_{1,p_1,\ldots,p_x})$ is, assuming the latter depends on the former,

$$v_1 d_1 + v_{1,p_1} d_{1,p_1} + \ldots + v_{1,p_1,\ldots,p_x} d_{1,p_1,\ldots,p_x} >$$
$$\begin{cases} S_{1,p_1,\ldots,p_x,k} + S_{1,p_1,\ldots,p_x,k+1} + \ldots + S_{1,p_1,\ldots,p_x,j-1} + S_{1,p_1,\ldots,p_x,j} - 1 & \text{if } k \leq j \\ -S_{1,p_1,\ldots,p_x,k-1} - S_{1,p_1,\ldots,p_x,k-2} - \ldots - S_{1,p_1,\ldots,p_x,j+2} - S_{1,p_1,\ldots,p_x,j+1} - 1 & \text{if } k > j \end{cases} \quad (1)$$

The derivation is straightforward and tedious, so it is omitted.

For each dependence, we should have a separate inequality as above. Subject to this set of inequalities, the delays should minimize the MAX-GLOBAL-TIME-STEP. Note that the MAX-GLOBAL-TIME-STEP is a function of all delays. This is an integer programming problem. We show an efficient method to solve this problem.

For convenience, let's rename the delays for the purpose of this section. Instead of indexing them with their *paths*, we number them by top-to-bottom and left-to-right order in the loop tree. For example, the following loop tree



will have a delay set (d_1, d_2, d_3, d_4) which corresponds to $(d_1, d_{11}, d_{12}, d_{121})$ in our old notation. With this new indexing scheme, since S_{path} is a linear combination of the delays (see Section 2), Inequality 1 can be rewritten

$$v_1 d_1 + v_2 d_2 + \ldots + v_m d_m > c$$

where m is the cardinality of the new delay set, and c is some constant. Assuming M dependences to be satisfied, we will have M inequalities below.

$$u_{11} d_1 + u_{12} d_2 + \ldots + u_{1m} d_m > c_1$$

Algorithm. dvector.
Input. $\mathbf{u}_I^J$ and $\mathbf{c}_I$, where $\mathbf{u}_I^J$ is a set of u_{ij} such that $i \in I$ and $j \in J$, and $\mathbf{c}_I$ is a set of c_i such that $i \in I$. I and J are two integer sets with cardinalities MAXI and MAXJ, respectively.
Output. $\mathbf{d}_J$, a set of d_j such that $j \in J$.
Comment. In the algorithm, c'_{I_i} is $c_{I_i} - \sum_{x \in (J-J_1)} u_{I_i x} d_x$. Initially the algorithm starts with $I = (1, 2, \ldots, M)$ and $J = (1, 2, \ldots, m)$, where M is the number of dependences, and m is the cardinality of the delay set, $\mathbf{d}$.
Method.

```
1. If all the elements of the column vector $\mathbf{u}^{J_j}$ are positive,
     $d_{J_j} = MAX(\lceil c_{I_1}/u_{I_1 J_j} \rceil, \ldots, \lceil c_{I_{MAXI}}/u_{I_{MAXI} J_j} \rceil)$
     $d_x = 0, \forall x \in J$ except $J_j$
   Else
     $I' = \{I_k \text{ s.t. } u_{I_k J_1} = 0\}$
     $J' = \{J_k, \forall k \neq 1\}$
     call algorithm dvector with $\mathbf{u}_{I'}^{J'}$ and $\mathbf{c}_{I'}$ to calculate $\mathbf{d}_{J'}$.
     $d_{J_j} = MAX(\lceil c'_{I_1}/u_{I_1 J_1} \rceil, \ldots, \lceil c'_{I_{MAXI}}/u_{I_{MAXI} J_1} \rceil)$.
```

Figure 6: Algorithm to compute the delays.

$$u_{21}d_1 + u_{22}d_2 + \ldots + u_{2m}d_m > c_2$$

$$\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots$$

$$\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots$$

$$u_{M1}d_1 + u_{M2}d_2 + \ldots + u_{Mm}d_m > c_M \tag{2}$$

Now, let's look at how u_{ij} are formed for a particular row, i. From the formula in Inequality 1, the first few terms of u_{ij} exactly correspond to $v_1, v_{1,p_1}, \ldots, v_{1,p_1,\ldots,p_x}$, repectively. The rest of the coefficients will be supplied from the right-hand side of the formula. Since the first few terms are supplied from a dependence distance vector, and they are leading coefficients of this row, vector u_{ij} can be regarded as a legal dependence distance vector. A dependence distance vector is legal if its first non-zero element is positive.

Therefore, we can apply the algorithm in [KiNi91] to solve the set of inequalities in Inequalities 2. We have copied the algorithm in Figure 6. Basically, the algorithm works in divide-and-conquer manner. The inequalities are divided into two groups: those with positive leading coefficient and those with zero leading coefficient. The latter group is solved first. Note that the latter problem has one less variables than the original one. The leading delay corresponding to the leading coefficient is missing here. Once this group is solved, the solutions are substituted to the the former group of inequalities to compute the missing leading delay. The solving process of the latter group is a recursive application of the same divide-and-conquer strategy.

For example, take a look at the example loop in Figure 1. Suppose loop I_{121} needs data from loop I_{122}(statement E) with dependence distance vector (1,1). Also suppose loop I_{131}(statement G) needs data from itself with dependence distance vector (0,1). For the first dependence, we have

$$1 \times d_1 + 1 \times d_{12} > S_{121} + S_{122} - 1,$$

and from the second,

$$0 \times d_1 + 1 \times d_{13} > S_{131} - 1.$$

$S_{121} = (N_{121} - 1)d_{121} + S_{1211} + S_{1212} = (N_{121} - 1)d_{121} + 2$, and $S_{122} = S_{131} = 1$, assuming all statements take one time step. Therefore, after simplification, we have the following inequalities.

$$d_1 + d_{12} + (1 - N_{121})d_{121} > 2$$

$$d_{13} > 0$$

With renaming, we have

$$d_1 + 0d_2 + d_3 + 0d_4 + (1 - N_{121})d_5 + 0d_6 + 0d_7 > 2$$

$$0d_1 + 0d_2 + 0d_3 + d_4 + 0d_5 + 0d_6 + 0d_7 > 0$$

Applying the above algorithm to compute the delay set, we have $d_1 = 3, d_4 = 1$, and all other delays are zeros. Note d_4 corresponds to d_{13} in old notation.

4 Example – SOR algorithm

We have applied our algorithm to parallelize SOR (Successive Over-Relaxation) algorithm [PFTV86], which is widely used to solve large linear systems. The parallelized SOR was actually run on a Sequent machine with very favorable results.

Figure 7 shows the SOR algorithm. We will use this algorithm to solve the following *Poisson's equation* for u on the unit square $0 \le x \le 1, 0 \le y \le 1$

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 4.$$

On the boundary, $u(x, y)$ satisfies the condition

$$u(x, y) = x^2 + y^2 + y.$$

The difference equation corresponding to the above *Poisson's equation* is

$$U(J+1, L) + U(J-1, L) + U(J, L+1) + U(J, L-1) - 4U(J, L) = \frac{4}{JMAX^2},$$

when we chose the grid size to be $JMAX \times JMAX$, and let $J = x \times JMAX, L = y \times JMAX$. Then, in Figure 7, $F = \frac{4}{JMAX^2}$. ρ in the figure is also an input as the spectral radius of the Jacobi iteration, or an estimate of it, whose value is computed by $1 - \frac{\pi^2}{1 \times JMAX^2}$. ABS() is a function that returns the absolute value of its argument.

```
anormf=zero
For J=2, JMAX - 1
    For L=2, JMAX - 1
        anormf = anormf + ABS(F(J,L))
    Endfor
Endfor
omega = 1
For N=1, MAXITS
    anorm = 0
    For J=2, JMAX - 1
        For L=2, JMAX - 1
            If MOD(J+L,2) = MOD(N,2) then
              resid=U(J+1,L)+U(J-1,L)+U(J,L+1)+U(J,L-1)-4U(J,L)-F(J,L)
              anorm=anorm + ABS(resid)
              U(J,L) = U(J,L) + $omega \times resid/4$
            Endif
        Endfor
    Endfor
    If N = 1 then
      $omega = 1/(1-(1/2) \times \rho^2)$
    Else
      $omega = 1/(1-(1/4) \times \rho^2 \times omega)$
    Endif
    If ($N > 1$ and $anorm < eps \times anormf$) then RETURN
Endfor
```

Figure 7: The SOR algorithm.

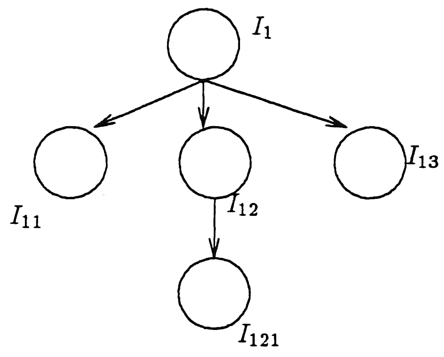
We have modified the ordering of statements in the above algorithm and performed *scalar expansion* [Padu79] on variables *anorm* and *omega*, to facilitate the exploitation of parallelism. Refer [AlKe87][KiNi91] for more information on the reordering of statements. Also, we removed the temporary variable *resid* by substituting all of its appearances by its definition. This is to remove dependences involving the variable *resid*. This change is only for the purpose of a clear explanation. A temporary variable like *resid* does not cause dependence in real situation because it can be declared as a local variable on each processor. Finally, we have normalized all loops such that all indices start from zero's with stride 1. Note that these are standard optimizations that are not specific to our transformation system. The transformed loop is in Figure 8.

In the figure, the main loop starts at statement 0 ending at statement 16, whose time complexity is $O(Q \times JMAX^2)$, where Q is the number of iterations for the outermost loop (indexed by N) when the condition at statement 15 is met. We will parallelize this loop. We regard statement 1 through 6 as a single statement, and the innermost loop at line number 8 to 13 as another single statement. This is to increases the granulity of the scheduling unit because the Sequent machine is not suited well for fine-grain parallelism – it uses fork-join mechanism to handle parallelism, and each forking takes considerable amount of CPU time. With a finer grain machine, we may parallelize down to simple statement level, exploiting more parallelism. However, this example shows that our technique can adjust to medium or coarse grain machines.

The loop tree for this loop is

```
anormf=zero
For J=2, JMAX - 1
    For L=2, JMAX - 1
        anormf = anormf + ABS(F(J,L))
    Endfor
Endfor
omega[1] = 1
0:For N=0, MAXITS-1
1:     anorm[N+1] = 0
2:     If N+1 = 1 then
3:         $omega[N+2] = 1/(1-(1/2)\rho^2)$
4:     Else
5:         $omega[N+2] = 1/(1-(1/4)\rho^2 omega[N+1])$
6:     Endif
7:     For J=0, JMAX - 3
8:         For L=0, JMAX - 3
9:             If MOD(J+L+4,2) = MOD(N+1,2) then
10:                anorm[N+1]= anorm[N+1] + ABS(U(J+3,L+2)+U(J+1,L+2)
                               +U(J+2,L+3)+U(J+2,L+1)-4U(J+2,L+2)-F(J+2,L+2))
11:                U(J+2,L+2) = U(J+2,L+2) + omega[N+1]×(U(J+3,L+2)
                               +U(J+1,L+2)+U(J+2,L+3)+U(J+2,L+1)-4U(J+2,L+2)-F(J+2,L+2))/4
12:            Endif
13:        Endfor
14:    Endfor
15:    If $(N+1 > 1$ and $anorm[N+1] < eps \times anormf)$ then RETURN
16:Endfor
```

Figure 8: SOR algorithm with statement reordering and scalar expansion.



Node I_{11} in the loop tree corresponds to statement 1 through 6 in Figure 8, and node I_{121} in the loop tree to statement 8 through 13 in the same figure. Now, node I_{11} has dependence on itself with distance vector (1) because of statement 5. Node I_{121} also has dependence on itself with distance vector (0,1), (1,0), and (1,-1). The distance vector (0,1) is due to statement 10. The distance vector (1,0) and (1,-1) arises from the self-dependence on statement 11. Statement 11 at iteration vector (N,J) needs data from the result of statement 11 at iteration vector (N-1,J) and (N-1,J+1). Finally, there are in-loop dependences between node I_{11} and I_{12}, and between node I_{12} and I_{13}. Since node I_{13} is an exit-test, strictly speaking, we have dependences from node I_{13} to I_{11} and I_{12} with distance vector (1); however, since the algorithm converges, there is no need for node I_{11} and I_{12} to wait for the result of the node I_{13} in the previous iteration. So, these dependences due to the exit-test are

```
For $t = 0$ to MAX-GLOBAL-TIME-STEP
   Forall $I1 = L_1$ to $U_1$
          Case $t_1 - I_1 d_1$ is
          0: anorm[N+1] = 0
             If N+1 = 1 then
                $omega[N+2] = 1/(1-(1/2)\rho^2)$
             Else
                $omega[N+2] = 1/(1-(1/4)\rho^2 omega[N+1])$
             Endif
          1 to $S_{12}$:
            Forall $I_{12} = L_{12}$ to $U_{12}$
               For L=0 to JMAX-3
                  If MOD(J+L+4,2) = MOD(N+1,2) then
                    anorm[N+1]= anorm[N+1] + ABS(U(J+3,L+2)+U(J+1,L+2)
                           +U(J+2,L+3)+U(J+2,L+1)-4U(J+2,L+2)-F(J+2,L+2))
                    U(J+2,L+2) = U(J+2,L+2) + omega[N+1]×(U(J+3,L+2)
                           +U(J+1,L+2)+U(J+2,L+3)+U(J+2,L+1)-4U(J+2,L+2)-F(J+2,L+2))/4
                  Endif
               Endfor
            Endforall
          $S_{12}+1$: If ($N+1 > 1$ and $anorm[N+1] < eps \times anormf$) then RETURN
          Endcase
   Endforall
Endfor
```

Figure 9: The template of a parallel SOR algorithm.

ignored.

The inequalities to be satisfied, therefore, are

$$1 \times d_1 > S_{11} - 1 = 0,$$

$$1 \times d_{12} > S_{121} - 1 = 0,$$

$$1 \times d_1 > S_{121} - 1 = 0,$$

and

$$1 \times d_1 + (-1) \times d_{12} > S_{121} - 1 = 0.$$

Note that $S_{11} = S_{121} = 1$ since we regard node I_{11} and node I_{121} as a single statement each. The in-loop dependences already satisfy the inequalities required (see Section 3.2). Solving the above inequalities using the algorithm in Figure 6, we have

$$d_1 = 2, d_{12} = 1.$$

Now, from Section 3.1, the transformed loop will have the form in Figure 9. To compute the unknown values in this figure, we need the size of each loop first. From the formulas in Figure 3 and the values of the delays above,

$$S_{11} = S_{13} = S_{121} = 1,$$

$$S_{12} = (JMAX - 3)d_{12} + 1 = JMAX - 2,$$

```
For $t = 0$ to $2 \times MAXITS + JMAX - 3$
    Forall $I1 = MAX(0, \lceil (t+1-JMAX)/2 \rceil)$ to $MIN(MAXITS-1, \lfloor t/2 \rfloor)$
        Case $t - 2I_1$ is
        0: anorm[$I_1$+1] = 0
           If $I_1$+1 = 1 then
              omega[$I_1$+2] = $1/(1-(1/2)\rho^2)$
           Else
              omega[$I_1$+2] = 1/(1 - (1/4)$\rho^2$omega[$I_1$+1])
           Endif
        1 to $JMAX - 2$:
          $I_{12} = t - 2I_1 - 1$
          For L=0 to JMAX-3
            If MOD($I_{12}$+L+4,2) = MOD($I_1$+1,2) then
              anorm[$I_1$+1]= anorm[$I_1$+1] + ABS(U($I_{12}$+3,L+2)+U($I_{12}$
                  +1,L+2)+U($I_{12}$+2,L+3)+U($I_{12}$+2,L+1)-4U($I_{12}$+2,L+2)-F($I_{12}$+2,L+2))
              U($I_{12}$+2,L+2) = U($I_{12}$+2,L+2) + omega[$I_1$+1](U($I_{12}$+3,L+2)
                  +U($I_{12}$+1,L+2)+U($I_{12}$+2,L+3)+U($I_{12}$+2,L+1)-4U($I_{12}$+2,L+2)-F($I_{12}$+2,L+2))/4
            Endif
          Endfor
        $JMAX - 1$: If ($I_1 + 1 > 1$ and anorm[$I_1$+1] < $eps \times anormf$) then RETURN
        Endcase
    Endforall
Endfor
```

Figure 10: Parallel SOR algorithm.

and

$$S_1 = (MAXITS-1)d_1 + 1 + JMAX - 2 + 1 = 2 \times MAXITS + JMAX - 2.$$

Then,

$$\text{MAX-GLOBAL-TIME-STEP} = S_1 - 1 = 2 \times MAXITS + JMAX - 3,$$

$$t_1 = t,$$

$$L_1 = MAX(0, \lceil (t+1-S_{1,*})/d_1 \rceil) = MAX(0, \lceil (t+1-JMAX)/2 \rceil),$$

$$U_1 = MIN(MAXITS-1, \lfloor t/2 \rfloor),$$

$$t_{12} = t - TB(I_{12}, (I_1)) = t - (I_1 d_1 + 1) = t - 2I_1 - 1,$$

$$L_{12} = MAX(0, t-2I_1-1), U_{12} = MIN(JMAX-3, t-2I_1-1).$$

Therefore, the final transformed loop is as shown in Figure 10. Note that, in the figure, $0 \leq t - 2I_1 - 1 \leq JMAX - 3$, when $1 \leq t - 2I_1 \leq JMAX - 2$, so we wrote

$$I_{12} = t - 2I_1 - 1$$

instead of

Forall $I_{12} = MAX(0, t-2I_1-1)$ to $MIN(JMAX-3, t-2I_1-1)$.

We ran the sequential and parallel version of SOR algorithm on Sequent machine using 1, 2, 4,and 7 physical processors for various values of $JMAX$ until the result converges under $eps = 0.00001$. The experimental data are summarized in Table 11. When only one processor is used, the parallel SOR runs slower than the sequential version because of the overhead to fork parallel tasks. Otherwise, the speed-ups are very promising. Especially, the speed-up of parallel SOR over sequential version improves as the problem size (represented by variable JMAX) increases.

Note that the SOR algorithm is not vectorizable for any loop. In Figure 8, the loop indexed by N is not vectorizable because of the self-dependence on statement 5, and the loops indexed by J and L are not vectorizable either because of the self-dependence on statement 10. Applying the wavefront method to the SOR algorithm (actually to the loop indexed by N which was parallelized by our method) is not easy because it is loosely nested. We may transform the loop N into a tightly nested form by bringing statement 1 to 6 and statement 15 inside the innermost loop L, using if statements, and apply the wavefront method. This strategy seems to work for this particular example, but, in general, transforming a loosely nested loop into a tightly nested form is not always possible. For example, suppose the statement 15 in Figure 8 were a loop, say loop K, instead of a simple statement. Then, to transform loop N into a tightly nested form, we need first to fuse loop K with loop J. However, loop fusion is not always possible [Wolf89]. Therefore, applying the wavefront method to loosely nested loops by changing them to tightly nested forms is not always possible.

Finally, applying DOACROSS to the SOR algorithm is not promising, either. Let's look at Figure 8 to see how DOACROSS can be applied. Since the loops indexed by J and L (from statement 7 to 14) are sequential because of statement 10, we may apply DOACROSS to the outermost loop (indexed by N). Then, since the code section from statement 7 to 14 has to be ordered between iterations, the asymptotic execution time is $O(Q \times JMAX^2)$, compared to $O(Q \times JMAX)$ in our case (see Figure 10), where Q is the expected number of iterations to satisfy the convergence test in statement 15. To improve the DOACROSS schedule, we may remove the strict ordering of the code section from statement 7 and 14. That is, we no longer regard the inner loops J and L as a single statement. Then, to preserve the semantics of the original loop, every instance of statement 10 and 11 of loops J and L (there are rougly $JMAX^2$ of them) at each iteration of loop N needs to wait for a synchronization signal from the previous iteration of loop N. This means about $JMAX^2$ of synchronizations are needed between any two iterations of loop N, which is not a trivial problem from implementation point of view, especially because the value of $JMAX$ could vary. At least, in the Sequent machine, it seems impossible to implement such a large and varying number of synchronization signals.

5 Conclusion

In this paper, we showed a technique that can parallelize non-vectorizable loosely nested loops. Since loosely nested loops represent the general form of nested loops, our method has a very wide appli-

	no. of pe's in parallel SOR				
jmax	1	2	4	7	seq SOR
20	2.6	1.6	1.1	0.8	2.4
40	20.4	11.5	6.0	4.0	19.3
60	69.1	37.9	20.5	13.1	64.9
80	167.8	90.2	46.4	28.9	158.1
100	331.6	176.4	92.7	54.9	310.9

(a) Comparison of execution times (seconds)
between parallel SOR and sequentail SOR

	no. of pe's in parallel SOR			
jmax	1	2	4	7
20	0.9	1.5	2.1	3.0
40	0.9	1.6	3.2	4.7
60	0.9	1.7	3.2	5.0
80	0.9	1.7	3.4	5.5
100	0.9	1.8	3.4	5.7

(b) Speed-up of parallel SOR over sequential SOR

Figure 11: The results of experiments on SOR algorithm.

cability. Previously, loop parallelization was limited to single loops or tightly nested loops when the loops are not vectorizable.

Two steps are needed in our method: computing the exact time step of each statement instance and capturing and expressing the parallel statement instances whose time steps are equal.

Computing the time steps of the statement instances is the problem of computing the delays of all nesting loops, because the time step of a statement instance depends on the delay values of its surrounding loops. The delay values should be chosen such that they not only satisfy the dependences between statements but also minimize the total execution time. We provide an efficient method to find such delay values.

The second step is capturing and expressing parallel statement instances. At each time step, we compute which statement instances can be done in parallel. We introduced the notion of *loop tree* to explain this process. For each node (which corresponds to each nesting loop) in the loop tree, the iterations that span the given time step are calculated using a simple relationship between the given time step and the iteration number. Once these parallel iterations are calculated for all nodes in the loop tree, we can transform the loop into a parallel form.

We showed the practicality of our algorithm by successfully parallelizing the SOR algorithm using our technique. We ran the parallel version of it on Sequent machine with 1 to 7 physical processors and obtained significant speed-ups.

References

✓ [AiNi88] Aiken,A. and Nicolau,A., *Perfect Pipelining: a new loop parallelization technique*, Proc. of the 1988 European Symposium on Programming, pp 221-235, Springer Verlag Lecture Notes in Computer Science no. 300, March 1988.

[AlKe87] Allen,J.R. and K.Kennedy, *Automatic Translation of Fortran Programs to Vector Form*, ACM Transactions on Programming Languages and Systems, Vol. 9, No. 4, pp.491-542, Oct. 1987.

[Bane90] Banerjee,U., *Unimodular Transformations of Double Loops*, 3rd Workshop on Programming Languages and Compilers for Parallel Computing, Irvine, California, August, 1990.

[Cytr86] Cytron,R.G., *Doacross: Beyond Vectorization for Multiprocessors*, Proc. of the 1986 International Conference on Parallel Processing, Hwang, Jacobs, Swartzlander(eds), IEEE Computer Society Press, Los Angeles, CA, pp 836-844, Aug. 1986.

[KiNi91] Kim,K.C. and Nicolau,A., *Fine Grain Software Pipelining of Non-Vectorizable Nested Loops,* to be published in the Proc. of the International Symposium on Shared Memory Multiprocessing, 1991.

[KMC72] Kuck,D., Muraoka,Y., and Chen,S., *On the number of operations simultaneously executable in FORTRAN-like programs and their resulting speed-up*, IEEE Transactions on Computers, C-21, No. 12, Dec., 1972, pp. 1293-1310.

✓ [Lam88] Lam, M.S., *Software Pipelining: An Effective Scheduling Technique for VLIW Machines,* In Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation, pp. 318-328, Atlanta, Georgia, June 1988.

[Lamp74] Lamport, L., *The parallel execution of DO loops*, Comm. of the ACM, 17(2):83-93, Feb. 1974.

[LaWo90] Lam,M. and Wolf,M., *Maximizing Parallelism Via Linear Loop Transformations*, 3rd Workshop on Programming Languages and Compilers for Parallel Computing, Irvine, California, August, 1990.

[Mura71] Muraoka, Y., *Parallelism Exposure and Exploitation in Programs,* Ph.D. Thesis, Univ. of Illinois at Urbana-Champaign, 1971.

[Nico87] Nicolau,A., *Loop Quantization or Unwinding Done Right*, Proc. Supercomputing 1st International Conference, June 1987.

[Padu79] Padua,D.A.H., *Multiprocessors: Discussions of some Theoretical and Practical Problems,* Ph.D. thesis, Univ. Of Ill. at Urbana-Champaign, Urbana, Ill. 1979.

[PFTV86] Press,W.H.,Flannery,B.P.,Teukolsky,S.A., and Vetterling,W.T., *Numerical Recipes*, pp.647-659, Cambridge university press, 1986.

[SDWX87] Su, B., Ding, S., Wang, J., and Xia, J., *GURPR – A Method for Global Software Pipelining*, Proc. of the 20th Microprogramming Wordshop (MICRO-20), pp.97-105, Colorado Springs, CO, Dec. 1987.

[Wolf82] Wolfe,M.J., *Optimizing Supercompilers for Supercomputers*, Ph.D. Thesis, Univ. of Ill. at Urbana-Champaign, Dept. of Comp. Sci. Rpt. No. 82-1105, Oct. 1982.

[Wolf87] Wolfe,M., *Loop Skewing: The Wavefornt Method Revisited*, Tech. Rpt., Univ. of Illinois at Urbana-Champaign, April 1987.

[Wolf89] Wolfe,M.J., *Optimizing Supercompilers for Supercomputers*, pp.92-94, The MIT Press, Cambridge, 1989.

3 1970 00882 4523