An Approach to Correcting Parallel Loop Execution Results

Volodymyr Beletskyy, Bartlomiej Niaka
Faculty of Computer Science, Technical University of Szczecin, Zolnierska 49 st., 71-210 Szczecin, Poland
Bielecki@man.szczecin.pl

Abstract: A technique to execute the loop in parallel is considered. Firstly, all the loop iterations are executed in parallel. Then, the ends of the pairs of dependent iterations are re-executed in lexicographical order. The method requires no conversion of the source loop body, involving iteration indices to be converted into new ones, that appears to be a typical requirement in the most known loop parallelizing approaches. Possibilities of dynamic and static correction processes are discussed. A technique to correct parallel loop execution results is presented. The approach is applicable to imperfectly nested loops with non-uniform dependences. Experimental results are considered.

1 Introduction and related work

Since parallel and distributed computing have become increasingly popular, it is significant to develop such compilers that would automatically translate serial programs into effective parallel code.

Numerous methods of loop parallelizing have been proposed[1], [2], [4], [14], [15], [16], [17]. However, research in this field is still carried out as the problem of loop parallelizing appears extremely complicated. So far, comparative little work has been done for non-uniform dependence loops. According to an empirical study by Shen et. al. [13], subscripts that are linear functions of loop indices or coupled subscripts appear quite frequently in real applications. They observed that nearly 44% of two-dimensional array references are coupled and can generate non-uniform dependences.

The approach proposed in this paper aims at parallelizing loops with non-uniform dependences. First, all iterations are executed in parallel. Next, all the ends of the pairs of dependent iterations need to be corrected. The method requires no conversion of the source loop body, involving the source iteration indices to be converted into new ones, that appears to be a typical requirement in the most known loop parallelizing methods such as unimodular and non-unimodular linear transformation approaches[2], [3], [4], [5].

The method is different from the speculative parallel execution [6] in the following: 1) dependence analysis is fulfilled before loop execution; 2) if cross-iteration dependencies exist, then only the iterations that are the ends of the pairs of dependent iterations are re-executed.

The main objective of the paper is to present an approach to the correction of the parallel loop execution results, possibilities of its implementation, experimental results, and future research.
2 Analysis of Data Dependence

Our approach requires dependence analysis. It is considered in lots of papers, for example, in [7], [8], [9]. In this section, we attach the knowledge that is necessary to explain a concept of the correction approach and the scope of its use.

Consider the following generic nested loop

\[
\text{for } I_1 = L_1, U_1 \\
\text{for } I_2 = L_2(I_1), U_2(I_1) \\
\quad \ldots \\
\text{for } I_n = L_n(I_1, \ldots, I_{n-1}), U_n(I_1, \ldots, I_{n-1}) \\
\quad \quad H(I_1, \ldots, I_n) \\
\quad \text{end for} \\
\quad \ldots \\
\text{end for} \\
\text{end for,}
\]

where \( I_1, \ldots, I_n \) are the iteration indices; \( L_i \) and \( U_i \) - the lower and upper loop bounds that are linear functions of the iteration indices \( I_1, \ldots, I_{i-1} \); implicitly a stride of one is assumed; \( H \) is the body of the nested loop; array reference functions within the loop body are integer affine functions of the iteration indices. \( I = (I_1, \ldots, I_n) \) is called the iteration vector.

**Definition 1.** The set \( I \subseteq \mathbb{Z}^n \) such that

\[
I = \{(i_1, \ldots, i_n) | L_1 \leq i_1 \leq U_1, \ldots, L_n (i_1, \ldots, i_{n-1}) \leq i_n \leq U_n (i_1, \ldots, i_{n-1})\}
\]

is an iteration space.

Individual iterations are denoted by tuples of iteration indices.

The dependence relation between two statements constrains the order in which the statements may be executed. There are three types of data dependencies: flow, anti, and output dependence. The only true dependence is flow dependence. The other dependencies are the result of reusing the same location of memory and are hence called pseudo dependencies. They can be eliminated by renaming variables as well as by the techniques presented in [6], [10], [11]. The approach proposed works for all dependence types, but eliminating pseudo dependences allows the efficiency of the approach to be enhanced considerably.

For our approach, we need to know all the pairs of iterations that are dependent and the relationship between them, such as the dependence vector.

**Definition 2.** (Dependence Vector) For a pair of iterations \( i = (i_1, \ldots, i_n) \) and \( j = (j_1, \ldots, j_n) \) such that \( j \) is dependent on \( i \), the vector \( j - i = (j_1 - i_1, \ldots, j_n - i_n) \) is called the dependence distance vector.

Note that the coordinates of \( i, j \) can be linear or affine functions of free variables that are arbitrary integers. If, for a certain \( k \), \( j_k - i_k \) is not constant, we have the non-uniform dependence.

For our approach, definition 2 is valid not only for perfectly nested loops but also for imperfectly nested ones. In the end of this section, we show how vectors \( i, j \) may be found for imperfectly nested loops and that they are of size \( n \).

To find dependence vectors, equations should be built for each pair of the same named variables \( ID (A_1 I + B_1), ID (A_2 I + B_2) \) that are located in the loop body on both hand sides.
of assignment statements - the right and the left-or on the left-hand sides only, where $A_1$, $A_2$ are matrices, $B_1$, $B_2$ are vectors.

The equations mentioned can be written as follows:

$$
A_1 * i - A_2 * J = B_2 - B_1 ,
$$

$$
K = J - i ,
$$

K $\geq$ 0.

The solution to equations (1), if it exists, can be presented in the following form[12]:

$$
K = t_1 t * V_1 + t_{12} t * V_2 + ... + t_{1r} t * V_r + V_0 ,
$$

$$
i = t_{21} t * W_1 + t_{22} t * W_2 + ... + t_{2s} t * W_s + W_0 ,
$$

where: $V_0 , V_1 , ..., V_r , W_0 , W_1 , ..., W_s$ are vectors of dimension $n$ with constant elements; $t_{1q} , t_{2p} , q = 1, 2, ..., r ; p=1,2,...,s$ are free variables, values of which are arbitrary integer numbers, $0 \leq r, s \leq n$.

The bounds of free variables can be found by building corresponding linear inequalities, taking into account the loop bounds, and solving them with the Fourier-Motzkin algorithm[4].

The vector $i$ describes all the iterations that form the beginnings of the pairs of dependent iterations, while the vector $J = i + K$ describes the ends of those. For a pair of dependent iterations, the beginning is the iteration that is lexicographically less. To obtain correct results, all the dependent iterations have to be executed in lexicographical order.

Let us consider the example

for $i_1 = 1, N$

for $i_2 = 1, N$

$a(i_1,i_2) = a(i_1 - 2 , i_2 - 1)$.

Equations (1) for the loop above are as follows:

$$
i_1 - J_1 = -2 ,
$$

$$
i_2 - J_2 = -1 .
$$

The solution to these equations is: $K = (2, 1) , i = t_1 (1, 0) + t_2 (0, 1)$ , where $t_1 , t_2$ are free variables. For the vector $J$, we have

$$
J = i + K = t_1 (1, 0) + t_2 (0, 1) + (2, 1) .
$$

To find the limits of $t_1$ and $t_2$ for the vector $J$, we take into consideration the loop bounds and build the following inequalities

$$
(1,1) \leq J \leq (N, N) \quad \text{or} \quad (1,1) \leq t_1 (1, 0) + t_2 (0, 1) + (2, 1) \leq (N, N) .
$$

From (2) we have

$$
1 \leq t_1 \leq N-2 ,
$$

$$
1 \leq t_2 \leq N-1 .
$$

Most approaches to loop parallelization imply that the loop should be perfectly nested one[1] or transformed to such a form. The technique, proposed in the following section, does not have such a limitation.
Consider the following loop

for \( i_1 = 1, n_1 \)

\[ \text{s}_1 : a(i_1) = a(i_1 - 1); \]

for \( i_2 = 1, n_2 \)

\[ \text{s}_2 : b(i_2) = b(i_2 - 1); \]

To reveal loop carried flow dependencies stated with statement \( \text{s}_2 \), we build equations taking into account all the iteration indices and their lower and upper limits.

\[ i_1 = J_1 = t_1, 1 \leq t_1 \leq n_1, \]

\[ i_2 = J_2 - 1, \]

where \( t_1 \) is a free variable. The solution to these equations is as follows:

\[ i = (t_1, t_2), K = (0, 1), 1 \leq t_1 \leq n_1, 1 \leq t_2 \leq n_2. \]

To expose loop carried flow dependencies stated with statement \( \text{s}_1 \), we build equations taking into consideration the fact that the value of the non-surrounded iteration index must be set to its lower bound, that is,

\[ i_1 = J_1 - 1, \]

\[ i_2 = 1, J_2 = 1. \]

Here, the values of \( i_2, J_2 \) equal to their lower limits. The solution to these equations is

\[ i = (t_1, 1), K = (1, 0), 1 \leq t_1 \leq n_1. \]

Consider the more general loop below

for \( i_1 = l_1, u_1 \)

for \( i_2 = l_2, u_2 \)

... for \( i_k = l_k, u_k \)

\[ \text{s}_1 : A[f_{1k}(i_1, ..., i_k), ..., f_{kk}(i_1, ..., i_k)] = A[g_{1k}(i_1, ..., i_k), ..., g_{kk}(i_1, ..., i_k)]; \]

for \( i_{k+1} = l_{k+1}, u_{k+1} \)

... for \( i_n = l_n, u_n \)

\[ \text{s}_2 : A[f_{1n}(i_1, ..., i_n), ..., f_{kn}(i_1, ..., i_n)] = A[g_{1n}(i_1, ..., i_n), ..., g_{kn}(i_1, ..., i_n)]; \]

In this loop, statement \( \text{s}_1 \) textually precedes statement with the all surrounding indices \( \text{s}_2 \). To expose the loop carried flow dependencies, stated with statement \( \text{s}_1 \), we can write the following equations

\[ f_{1k}(i_1, i_2, ..., i_k) = g_{1k}(J_1, J_2, ..., J_k), \]

... \[ f_{kk}(i_1, i_2, ..., i_k) = g_{kk}(J_1, J_2, ..., J_k), \]

\[ i_{k+1} = J_{k+1} = l_{k+1}, \]

... \[ i_n = J_n = l_n. \]

In the above equations, we extend the iteration indices of statement \( \text{s}_1 \) to size \( n \) to make the definition of the dependence vector be valid.

To reveal the flow dependencies designated with statements \( \text{s}_1 \) and \( \text{s}_2 \), we can write the following two systems of equations
If there exist solutions to these equations, then we have the vectors  \( i \) and  \( J \), describing the beginnings and the ends of the pairs of dependent iterations, as well as the dependence vector  \( K \).

For imperfectly nested loops, in the case when the statements with the non-surrounding loop indices textually proceed the statements with the all surrounding indices, the non-surrounding indices should be set to their upper bounds.

So, we extended the iteration space of the statements that are beyond the innermost loop, and definition 2 is valid for imperfectly nested loops.

3 Technique for parallel loop execution

We propose the following approach to parallel loop execution.

1. All the iterations of a loop are executed in parallel applying only old values of variables (the values before the execution of iterations).
2. Given vectors  \( i \) and  \( K \), dependent iterations are determined and the correction of the results for these dependent iterations is executed. The correction means a procedure of repeating the execution of the ends of all the pairs of dependent iterations in lexicographical order with regard to the new(old) values of variables for flow and output(anti) dependences.

Let us consider the following loop:

\[
\text{for } i = 3, 6 \\
a(i) = a(i - 3) + 1;
\]

and suppose the values of the vector elements before loop execution are:  \( a(m) = 0, m=1,2,...,6 \).

When this loop is executed serially, the following values are yielded:  \( a(3) = 1, a(4) = 1, a(5) = 1, a(6) = 2 \).

Now, let the technique proposed be applied. After the first step (parallel execution of all iterations) we have:  \( a(3) = 1, a(4) = 1, a(5) = 1, \) and  \( a(6) = 1 \). The vectors  \( K \) and  \( i \) for the
loop considered are as follows: $K = 3$, $i = t_1 + 2$, $1 \leq t_1 \leq 4$. Here, we have the only pair of dependent iterations with its beginning in iteration 3 and its end in iteration 6 ($J = i + K = 3 + 3 = 6$). It means a single correction is only required: the result of iteration 6 is calculated using the new value of iteration 3:

$$A(6) = (3) + 1 = 1 + 1 = 2.$$  

The results obtained are the same as those obtained at serial loop execution.

Correction can be executed in various ways: serially (only one processor fulfills correction) or parallel (two or more processors execute correction simultaneously), statically (data for correction are prepared at compile time) or dynamically (data for correction are prepared at run time).

In the following section, we consider serial dynamic correction.

4 Correcting parallel loop execution results

An approach to correct results after the execution of all the loop iterations in parallel includes: 1) preparing data at compile-time; 2) fulfilling the correction at run-time.

Definition 3. A common iteration is the end of a pair of dependent iterations given with vector $J_i$ and simultaneously it is the beginning of one or more pairs of dependent iterations yielded with vectors $i_k, 1 \neq k, k=1,2,...,n$.

Preparing data consists of: 1) building equations to expose cross iteration dependencies; 2) solving these equations regarding $i$, $K$, $J$; 3) building inequalities for vectors $J_i$ meeting the limits of the iteration indices; 4) solving these inequalities with the Fourier-Motzkin algorithm[4] for free variables; 5) building and solving equations to find common iterations.

Tasks 1-4 were considered in section 2. To find common iterations, we should build the following equations

$$J_i = i_k, 1, k = 1,2,...,n, \ 1 \neq k . \tag{3}$$

The solution to each equation above, if it exists, exposes common iterations yielded with vectors $J_i$ and $i_k$.

Consider the following working loop example

```c
for(i=1; i<n; i++) {
    for(j=1; j<n; j++) {
        A[3*i][4*j] = f(A[i][j]);
        A[2*i-1][3*j] = g(A[i][j]);
    }
}
```

For this loop, we have the following vectors:

$$K_1 = (t_1, 2t_1), \ i_1 = (t_1 + 1, t_1), \ J_1 = (2t_1 + 1, 3t_1),$$

$$K_2 = (t_2, t_2), \ i_2 = (2t_2 - 1, 3t_2), \ J_2 = (3t_2 - 1, 4t_2),$$

$$K_3 = (2t_3, 3t_3), \ i_3 = (t_3, t_3), \ J_3 = (3t_3, 4t_3).$$

The free variables of the vectors $J$ have the limits as follows
$0 \leq t_1^1 \leq 4, \ 1 \leq t_2^1 \leq 3, \ 1 \leq t_3^1 \leq 2, \ 1 \leq t_1^2 \leq 3, \ 1 \leq t_2^2 \leq 2.$

Figure 1 shows the iteration space for the working loop example when $n=11$.

![Figure 1](image)

**Fig. 1.** The iteration space for the working loop example

There exists the only solution to the following equation stating common iterations

$$J_1 = i_2.$$  

This solution is as follows

$$t_1^1 = t_2^1 - 1, \ t_2^1 = t_3^1, \ 0 \leq t_1^2 \leq 2, \ 1 \leq t_2^2 \leq 2.$$  

In the general case, the correction process requires calculating the tuples of iteration indices with vectors $J$ and the vectors, yielding common iterations, in lexicographical order and includes the following steps.

1) With vectors $J_k, k=1,2,...,n$ calculate a set of iterations $\{j_k\}$ to be corrected firstly.

2) Calculate a set of common iterations using solutions to equations (3). Choose the lexicographically least common iteration. Denote it as $I_{com}$.

3) While there exists such $j_k$ that $j_k < I_{com}$ repeat the following steps

   3.1) execute $j_k$;

   3.2) calculate next iteration $j_k \in I$ with $J_k$.

4) Execute $I_{com}$; with $J_m$ calculate next iteration $j_m$ for which the condition $j_m = I_{com}$ holds.

5) If there exist following common iterations, calculate/choose the next lexicographically least common iteration and go to 3. Else go to 6.

6) Calculate and execute all the iterations that belong to the iteration space $I$ and are lexicographically more than $I_{com}$.

The correction process for the working example is as follows.
• Calculate the iterations to be corrected firstly: (1,3) with J₁, (2,4) with J₂, and (3,4) with J₃.

• Calculate and execute the first common iteration I_{com} = (1,3), calculate (1,6) with J₁ since J₁ = I_{com}.

• Calculate and execute next I_{com} = (1,6), calculate (1,9) with J₁, calculate next common iteration (3,3).

• Execute iterations: (1, 9), (2,4), calculate and execute (2,8) with J₂, calculate (3,3) with J₁, calculate (5,4) with J₂.

• Execute common iteration (3,3), calculate (3,6) with J₁, calculate next I_{com} = (3,6).

• Execute (3,4), calculate iteration (3,8) with J₃.

• Execute I_{com} = (3,6), calculate (3,9) with J₁, calculate next I_{com} = (5,3).

• Execute (3,9), calculate iteration (5,3) with J₁.

• Execute (3,8), calculate iteration (6,4) with J₃.

• Execute I_{com} = (5,3), calculate (5,6) with J₁, calculate next I_{com} = (5,6).

• Execute (5,4), calculate iteration (5,8) with J₂.

• Execute I_{com} = (5,6), it is the last common iteration, calculate (5,9) with J₁.

• Calculate and execute iterations (5,9), (7,3), (7,6), (7,9), (9,3), (9,6), (9,9) with J₁.

• Calculate and execute iterations (5,8), (8,4), (8,8) with J₂.

• Calculate and execute iterations (6,4), (6,8), (9,4), (9,8) with J₃.

There is a possibility to collect all the data required for the correction at compile-time (static correction). It will permit the time of loop execution to be decreased, but such an approach requires a lot of memory for implementation.

If a loop exposes anti-dependences or the loop body includes the statements of the form x=...x...(such statements induce reduction dependences), a correction process should be modified. Before the correction, the initial values of the array elements, located on the right hand sides of assignment statements as well as the values of reduction variables x within the loop body, should be restored (they must get the values that they have before parallel loop execution).

If a loop is imperfect, then the statements beyond the innermost loop that precede (proceed) the statements of the innermost loop are executed at the correction if and only if the non-surrounded loop indices equal to their lower (upper) loop bounds.

5 Experiments

The approach proposed was implemented as a C++ application compiled with GCC compiler in the Red Hat Linux 7.1 Operating System. Experiments were carried out on H400 server (Fujitsu Siemens Computers) with four Intel Pentium III Xeon processors (700Mhz, 2MB cache), 2GB RAM. A tool for building and solving equations (1) created at the Computer Science Department of the Technical University of Szczecin (available at detox.wi.ps.pl) was used for building the application.

The application executes the loop in parallel (independently all the loop iterations) and next corrects the results of the ends of the pairs of dependent iterations according to the approach proposed.
Experiments were carried out with a number of loops yielded such vectors $J$, each element of which depends on one free variable and all of those are independent (see the working example). In such a case, calculating the tuples of iteration indices in lexicographical order with vectors $J$ is simple enough. Changing the vector with free variables in lexicographical order causes calculating vectors $J$ in lexicographical order, too.

Figure 2 shows how the time of the execution of the working example loop (see section 4) depends on the number of iterations at different doing: serial and parallel execution as well as correcting all the ends of the pairs of dependent iterations. To reach the desired number of iterations, we proportionally extend the bounds of the loop indices. The sum of the times of parallel execution and correcting results yields the time of the loop execution with our approach. Parallel execution means all the loop iterations are executed independently.

As we can see from figure 2, the speedup at applying the approach proposed to the working example is about 3.

The results of applying the approach to a number of loops investigated permits the following conclusions to be made. Speedup significantly depends on the percentage of dependent iterations. For the loops investigated, speedup was more than 1 at growing the percentage of dependent iterations up to 35%.

6 Conclusion

The approach, enabling executing loops with non-uniform dependences in parallel, was presented. It involves two steps: the first step provides the parallel execution of all the iterations of a loop, while the second one fulfils the correction of the ends of the pairs of the dependent loop iterations.

The main scope of the approach applicability is loops with non-uniform dependences that expose a few percentage of dependent iterations.

There is a need in further research on the approach. The main work to be done is as follows: comparing the efficiency of the correction method with other known non-uniform loop parallelization techniques; enlarging the method to parallelize sparse codes; the examination of the method by means of standard benchmarks and different general-purpose applications.
References


