An Approach to Parallelizing Non-uniform Loops with the Omega Calculator

V. Beletskyy, R. Drazkowski, M. Liersz
Faculty of Computer Science, Technical University of Szczecin, Zolnierska 49 st.,
71-210 Szczecin, Poland, fax. (+4891) 487-64-39
vbeletskyy@wi.ps.pl, mliersz@wi.ps.pl

Abstract

An approach permitting loops with non-uniform dependences to be parallelized is presented. It does not require loop-carried dependences to be uniformized. Firstly, all the loop iterations are executed in parallel despite the presence of loop carried dependences, next all the sinks of the pairs of dependent iterations are re-executed to get correct results. A modification of this approach divides the iteration space into two sets. The first one includes independent iterations as well as all the sources of the pairs of dependent iterations that simultaneously are not the sinks of those. All iterations of this set can be executed in parallel. The second set holds all the sinks of the pairs of dependent iterations. The iterations of this set must be executed serially in lexicographical order. The approaches proposed have been implemented by means of the Omega calculator. Experimental results illustrate the scope of the approaches applicability.

Index Terms — loop parallelization, non-uniform dependences, Presburger formulas

1. Introduction

A lot of transformations have been developed to expose parallelism in loops, minimize synchronization, and improve memory locality in the past [1], [2], [5], [6], [7], [9], [10], [11], [12], [13], [18], [19]. Those transformations are successfully used for parallelizing loops with uniform dependences.

So far, comparative little work has been done for loops with non-uniform dependences. According to an empirical study by Shen et. al. [16], subscripts that are linear functions of loop indices or coupled subscripts appear quite frequently in real applications, which relate also electrical problems. They observed that nearly 44% of two-dimensional array references are coupled and can generate non-uniform dependences.

A number of publications have studied loop parallelization with non-uniform dependences [3], [4], [8], [14], [17], [20]. Techniques proposed are based on non-linear transformations, substitution of non-uniform dependences with uniform ones, or applying affine mappings. All those techniques have drawbacks. They reveal little parallelism, induce additional dependences, or are very time-consuming.

This paper presents new approaches to parallelizing loops with non-uniform dependences. Their basic merit is the easiness of implementation by means of the Omega project software [ftp://ftp.cs.umd.edu/pub/omega], which includes: the Omega library, a set of routines for manipulating linear constraints over integer variables, Presburger formulas (they are built by applying the first order logical connectives: ¬, ∧, ∨, ⇒, ∀ and ∃), and integer tuple relations and sets; the code generation library, a set of routines for generating code to scan the points in the union of a number of convex sets; the Omega calculator, a text-based interface to the Omega library; petit, an educational/research tool for analyzing array data dependences, a tool designed to answer the usual questions asked for dependence analysis [15].

2. Approach to Loop Parallelization

Our approaches are based on the dependence analysis proposed by Pugh and Wonnacott [15]. That analysis permits exact dependence information for any single structured procedure in which the expressions in the subscripts, loop bounds, and conditionals are affine functions of the loop indices and loop-independent variables, and the loop steps are known constants to be produced. Dependences are presented with dependence relations [15]. A dependence relation is a mapping from one iteration space to another, and is represented by a set of linear constraints on variables that represent the values of the loop indices at the source and sink of the dependence and the values of the symbolic constants.

The basic merits of the dependence analysis proposed by Pugh and Wonnacott are as follows:

1. it is exact;
2. it is valid for both perfectly and imperfectly nested loops;
3. it permits value-based dependences to be calculated.

A dependence between iteration S(I) and S(J) is value-based if: S(I) is executed before S(J); S(I) and S(J) refer...
to a memory location \( M \), and at least one of these references is a write; the memory location \( M \) is not written between iteration \( I \) and iteration \( J \), which are the source and sink, correspondingly.

The dependence analysis by Pugh and Wonnacott is implemented in Petit, a research tool for doing dependence analysis and program transformations on programs in the Petit language. To carry out dependence analysis manually for sources written in other programming languages, the Omega calculator can be applied.

If dependence between iterations \( I \) and \( J \) can be represented by a vector \( J-I \) which elements are constants, it is called the uniform dependence else it is the non-uniform one.

Let us consider the following loop

\[
\begin{align*}
\text{for}(i1 = 1; i1 <= 1000; i1++) \{ \\
\text{for}(i2 = 1; i2 <= 1000; i2++) \{ \\
\quad s: A[2*i2+3][i1+1] = A[2*i1+i2+1][i1+i2+3]; \\
\text{\} } \\
\text{\} }
\end{align*}
\]

(1)

Let \( I=(i1, i2)' \) and \( J=(j1, j2)' \) be dependent iterations and \( J > I \), that is, \( I, J \) represent the source and sink of a pair of dependent iterations, correspondingly.

To reveal anti dependences for the loop above, we can write the following relation by means of Presburger formulas

\[
R:=[[i1,i2] -> [j1,j2] : 2i1+i2 +1 = 2j2+3 &\&
\quad i1+i2+3=j1+1 &\&
\quad 1<= 2i1+i2+1, i1+i2+3
\quad 2j2+3, j1+1 <= 1000],
\]

where \( 2i1+i2 +1 = 2j2+3 \) and \( i1+i2+3 = j1+1 \) means that the subscripts of \( A \) on the right hand side of statement \( s \) in iteration \( I \) are the same as those on the left hand side in iteration \( J \); \( 1<= 2i1+i2+1, i1+i2+3, 2j2+3, j1+1 <= 1000 \) means that the subscripts of \( A \) satisfy the loop bounds, that is, they lie in the iteration space of the above loop.

Both the domain and the range of the above relation are valid in the loop iteration space.

To find all the sinks of the pairs of dependent iterations, we can write \( J= range R \) as the input of the Omega calculator and get

\[
J:=[[j1,j2] : 0 <= j1 <= 999 &\&
\quad 1 <= j2 <= 498];
\]

Using the instruction “\text{codegen } J” of the Omega calculator permits us to yield the following loop which scans all the sinks of the pairs of dependent iterations of loop (1)

\[
\begin{align*}
\text{for}(i1 = 0; i1 <= 999; i1++) \{ \\
\text{for}(i2 = -1; i2 <= 489; i2++) \{ \\
\quad s: A[2*i2+3][i1+1] = A[2*i1+i2+1][i1+i2+3]; \\
\text{\} } \\
\text{\} }
\end{align*}
\]

(3)

Analogously, we can write the relation to reveal flow dependences for the above loop and discover that there are no flow dependences in the iteration space.

The main idea of our approach is as follows:

1. Despite the presence of loop-carried dependences, all iterations of a loop are executed in parallel applying only the old values of the array elements (the values before the loop execution).

2. The correction of the results formed by all the iterations that are the sinks of the pairs of dependent iterations is executed in lexicographical order with regard to the new values of the arrays elements if these elements are involved into flow or output dependences and with regard to the old values of those if they are involved into anti or reduction (\( x=...x \) operation...) dependences.

In other words for the working example, firstly source loop (1) is executed in parallel despite the loop-carried dependences, next loop (3) is executed serially taking into account the old values of \( A \) since its elements are involved into the loop-carried anti-dependences.

The basic advantages of the technique presented are:

i) the simplicity of code generation;

ii) the regularity of the code that executes all iterations in parallel (such a code is simply a source loop);

iii) no transformations of the source loop body need, there is only a requirement to scan the sinks of all the pairs of dependent iterations in lexicographical order.

The main drawbacks of the technique are as follows:

i) the iterations that are the sinks of the pairs of dependent iterations are executed twice;

ii) the presence of loop carried anti-dependencies and statements with reduction variables (\( x=...x \)) within the loop body requires restoring the old values of the array elements and induction variables.

If these drawbacks are essential for a particular loop, for example, the percentage of dependent iterations is high enough, the following modification of the technique proposed can be applied.

We divide all the loop iterations into two sets. The first one includes independent iterations and the sources of the pairs of dependent ones that simultaneously are not the sinks of those. The second set consists of the sinks of the pairs of dependent iterations, i.e. it is the set \( J \).

The first set \( S1 \) can be generated as the difference of the set \( I0 \), including all the loop iterations, and the set \( J \). To get \( S1 \) for the working example, we can write the following instructions of the Omega calculator

\[
I0:=[[i01, i02] : 1 <= i01, i02 <= 1000];
S1:=I0 - J;
\]

and get the following result
S1:={\([1000,\text{In}_2]: 1 \leq \text{In}_2 \leq 1000\) union \([\text{In}_1,\text{In}_2]: 1 \leq \text{In}_1 \leq 999 \&\& 499 \leq \text{In}_2 \leq 1000\)}

By means of the instruction “codegen S1”, we get two loops scanning all independent iterations and the sources of the pairs of dependent iterations that simultaneously are not the sinks of those

\[
\begin{align*}
&\text{for}(i1 = 1; i1 \leq 999; t1++) \\
&\text{for}(i2 = 499; i2 \leq 1000; t2++) \\
&(s(i1,i2); \\
&\text{for}(i2 = 1; i2 \leq 1000; i2++) \\
&s(1000,i2);
\end{align*}
\]

Now, we can firstly execute all the iterations of loops (4) and (5) in parallel, then loop (3) is executed serially.

This modification also does not require the loop body transformation and each iteration is executed only one time, but the codes generated to scan the iterations of both the sets, are for-loops, including in the general case if-statements, that can induce overheads increasing the loop execution time.

The next step to reduce the time of the loop execution is to build a loop that will allow the sinks of the pairs of dependent iterations to be executed in parallel. This is a goal of our further research and is out of the scope of this paper.

3. Experiments

In this Section, to show the effectiveness of the approaches proposed, we consider loops that cannot be parallelized neither with the wavefront method [18] since they are not fully permutable nor with the hyperplane method [2] since one or two of the highest coordinates of dependence vectors are not constants. For our experiments, we have chosen loops from [17], [20]. Table 1 presents source loops, the percentage of the iterations to be corrected, and the theoretical speedup that is found as follows:

\[S_{\text{theoretical}} \leq 100/\sigma,\]

where \(\sigma\) is the percentage of the iterations to be corrected.

The formula above is based on Amdahl’s low, which states that speedup is limited by the time taken to do the serial calculations of the application. In our case, this time is defined by \(\sigma\) since the correction is carried out serially.

The approaches were implemented by means of the Omega calculator (generating the sets \(J, I0, S1\) as well as the codes scanning these sets in lexicographical order). Next, the codes, generated by the Omega calculator, were manually converted to C applications compiled with the Omni OpenMP Compiler Project as an extension to Cygwin compiler, permitting multithreading, in the WindowsNT Server Operating System for H400 server (Fujitsu Siemens Computers) with four Intel Pentium III Xeon processors (700Mhz, 2MB cache), 2GB RAM.

![](image)

### Table 1. Testing loops.

<table>
<thead>
<tr>
<th>Loop</th>
<th>Source loop</th>
<th>The percentag e of the iterations to be corrected</th>
<th>Theoretical speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>for(i1=1;i1&lt;=1500) for(i2=1;i1&lt;=1500) a[i1+i2][3<em>i1+i2+3] = ... =a[i1+i2+1][i1+2</em>i2+4];</td>
<td>(31.3%)</td>
<td>(3.19)</td>
</tr>
<tr>
<td>L2</td>
<td>for(i1=1;i1&lt;=1500) for(i2=1;i1&lt;=1500) a[2<em>i2+3][i1+1] = ... =a[2</em>i1+i2+1][i1+i2+3];</td>
<td>(25%)</td>
<td>(4)</td>
</tr>
<tr>
<td>L3</td>
<td>for(i1=1;i1&lt;=1500) for(i2=1;i1&lt;=1500) a[2<em>i2+3][i1+1] = ... =a[2</em>i1+i2+1][i1+i2+3];</td>
<td>(12.5%)</td>
<td>(8)</td>
</tr>
<tr>
<td>L4</td>
<td>for(i1=1;i1&lt;=1500) for(i2=1;i1&lt;=1500) a[2<em>i2][2</em>i2] = ... =a[i2+10][i1+i2+10];</td>
<td>(23.4%)</td>
<td>(4.27)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Loop</th>
<th>Aver. run time of one iterat</th>
<th>Time of seq. exec., s</th>
<th>Time of paral. exec. (all iter.), s</th>
<th>Time of paral. exec. (ind. iter.), s</th>
<th>Time of seq. correction, s</th>
<th>Spee dup (bas. meth)</th>
<th>Spee dup (mod. meth)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>1</td>
<td>0.118</td>
<td>0.045</td>
<td>0.038</td>
<td>0.075</td>
<td>0.983</td>
<td>1.044</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>1.034</td>
<td>0.240</td>
<td>0.228</td>
<td>0.322</td>
<td>1.840</td>
<td>1.880</td>
</tr>
<tr>
<td></td>
<td>500</td>
<td>4.562</td>
<td>1.141</td>
<td>1.042</td>
<td>1.379</td>
<td>1.810</td>
<td>1.884</td>
</tr>
<tr>
<td>L2</td>
<td>1</td>
<td>0.119</td>
<td>0.037</td>
<td>0.031</td>
<td>0.066</td>
<td>1.155</td>
<td>1.227</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>1.025</td>
<td>0.231</td>
<td>0.214</td>
<td>0.306</td>
<td>1.909</td>
<td>1.971</td>
</tr>
<tr>
<td></td>
<td>500</td>
<td>4.425</td>
<td>1.219</td>
<td>1.294</td>
<td>1.266</td>
<td>1.781</td>
<td>1.728</td>
</tr>
<tr>
<td>L3</td>
<td>1</td>
<td>0.109</td>
<td>0.040</td>
<td>0.031</td>
<td>0.031</td>
<td>1.535</td>
<td>1.758</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>0.159</td>
<td>0.036</td>
<td>0.031</td>
<td>0.125</td>
<td>0.988</td>
<td>1.019</td>
</tr>
<tr>
<td></td>
<td>500</td>
<td>0.987</td>
<td>0.282</td>
<td>0.243</td>
<td>0.273</td>
<td>1.778</td>
<td>1.913</td>
</tr>
<tr>
<td>L4</td>
<td>1</td>
<td>0.109</td>
<td>0.047</td>
<td>0.072</td>
<td>0.047</td>
<td>1.160</td>
<td>0.916</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>0.831</td>
<td>0.222</td>
<td>0.210</td>
<td>0.221</td>
<td>1.876</td>
<td>1.928</td>
</tr>
<tr>
<td></td>
<td>500</td>
<td>3.703</td>
<td>0.944</td>
<td>0.842</td>
<td>0.909</td>
<td>1.998</td>
<td>2.115</td>
</tr>
</tbody>
</table>

### Table 2. Sequential and parallel time.

At multithreading, speedup greatly depends on the volume and time of the thread execution that in turn
depend on the average run time of one iteration. Creating and synchronizing threads need certain time. If this time is comparable or more than the time of the thread execution, then the efficiency of parallel applications may be low enough. It is why we have investigated how speedup depends on the average run time of one iteration.

The results of the experiments are presented in Table 2, where the time of 1, 100, 500 means that the time of the execution of one iteration equals to the time of the execution of a corresponding statement in the loop body plus the time of the execution of the loop of dimension 1, 100, and 500 correspondingly with the empty loop body. In such a manner we simulated the average run time of one iteration.

Analyzing the experimental results received, we can make the following conclusions.

1) There is a little difference in the speedups at applying the basic and modified approaches. The possible reason is that parallel loops, which the Omega calculator generates for the modified approach, include numerous guards (if-statements) that induce overheads. Applying other code generators, eliminating the guards, can enhance the efficiency of the modified approach.

2) The average run time of one iteration substantially affects the speedups received. In some cases, this time has a greater effect than the percentage of the iterations to be corrected.

3) At a certain run time of one iteration, positive speedup can be reached even with 30% of the dependent iterations to be corrected.

4. Conclusion

The main advantage of the approach proposed and its modification is that they permit parallel code to be produced very easily with the Omega calculator. Other public available code generators (see, for example http://www.irisa.fr/cosi/ALPHA/welcome.html) can be used to generate parallel code in accordance with our approaches. The scope of the applicability of the approaches is non-uniform loops exposing a few percent of dependent iterations (equal or less than 30%). We plan to extend our approach to parallellizing sparse codes.

References

