

Coarse-Grained Automatic Parallelization Approach for Branch Nested Loop

Hui Liu^{a,b,c}, Jinlong Xu^{c,*}, and Lili Ding^c

^aCollege of Computer and Information Engineering, Henan Normal University, Xinxiang, 453007, China

^bEngineering Technology Research Center for Computing Intelligence and Data Mining in Henan Province, Xinxiang, 453007, China

^cState Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou, 450001, China

Abstract

GCC compiler is a retargetable compiler program that was developed to increase the efficiency of programs in the GNU system. In recent years, compiler optimization based on data dependency analysis has become an important research area of modern compilers. Existing GCC compilers can only conduct dependency analysis on perfect nested loops. In order to better explore the coarse-grained parallelism of the nested loops, we propose a dependence test method that can deal with the branch nested loops. Firstly, we identify the branch nested loop in the programs. Then, we analyze the relationship between the array subscript and the outer index variable of the branch nested loop. Finally, we calculate the distance vector of the outer loop index variable and determine whether the loop has dependence through distance vector detection. Experimental results show that our method can correctly and effectively analyze the dependence relationship of branch nested loops.

Keywords: data dependence analysis; GCC; perfect nested loop; branch nested loops

(Submitted on September 15, 2019; Revised on September 30, 2019; Accepted on October 12, 2019)

© 2019 Totem Publisher, Inc. All rights reserved.

1. Introduction

Due to the endless pursuit of higher arithmetic speed, the operating rate of CPU has been constantly improved. For supercomputers, operating rates have already realized petaflops per second. For instance, the floating point arithmetic speed of Sunway TaihuLight supercomputers was as high as 125.436 petaflops per second [1]. However, in practice, the operating rate of supercomputers can only reach 30% of the theoretical peak velocity, or even lower [2]. Upon the analysis of numerous research in the computer field, the reason why the result is reached lies in the fact that the development of parallel software lags behind hardware development, making it unable to make full use of the computing resources. Currently, an important method for resolving this problem is to recognize the potential parallelism in the serial program with a parallelizing compiler system and convert it to a parallel program. The essence of program parallelization is the reordering transformation of statements in the program, but not all programs can keep the constant semantics after reordering. It is necessary to carry out the dependence analysis of programs before executing the transformation. The dependence analysis technology is a method of judging whether statements rely on each other in the program, as well as a basic step of developing program parallelization with the parallelizing compiler. As for the parallelizing compiler, the strength of the dependence analysis can directly impact its excavation of parallelization.

The GCC (GNU compiler collection) compiler developed by GUN can carry out the dependence analysis of a perfect nested loop in the serial program and jump over an imperfect nested loop directly. The automatic parallelizing compiler SUIF developed by Stanford University can recognize perfect nested loops in serial programs, but its parallel recognition of imperfect nested loops is also limited [3]. The development of these systems calls for research on dependence detection technology to reach a climax, and various kinds of detection methods have been put forward. The dependence test methods of the single array index mainly include the GCD test method and the Banerjee inequality test method [4-5]. The dependence test methods of the coupling array subscript mainly include the delta test method and the omega test method [6-7].

* Corresponding author.

E-mail address: xujinlong_pla@126.com

However, these compilers have a common defect, namely the weak dependence analysis ability of imperfect loop nests, and some compilers can only test the dependence relation of perfect loop nests.

Although loop distribution technology is often applied to decompose a branch nested loop into a perfect nested loop, in order to compensate for the defect that compilers that cannot directly conduct the dependence analysis of branch nested loops, the method has the following problems:

- Not all branch nested loops can be processed by the loop distribution.
- After the loop distribution transformation of the branch nested loop, the total iterations and synchronization times of executing the parallel statements of the loop would be multiplied.
- The locality of data may be destroyed, and the cache hit rate may be reduced.

Daniel Berlin, one of the developers of GCC [8] put forward the Tree-SSA-based loop transformation and data dependence analysis framework applicable for perfect nested loops. This framework greatly improved automatic parallelization and vectorization performance. Zeng et al. [9] realized the data dependence analysis method of split recursive chains and reinforced the ability of GCC in data dependence analysis. Zhang et al. [10] proposed a method of conducting OpenMP automatic parallelization by aiming at loops containing the cross-iteration data dependence. Zhao designed the plug-in Cit to test the data dependence characteristics of parallel programs during compilation. The plug-in can further analyze the dependence of May Conflict and present the dependence information as a visual form to programmers [11]. The above studies boosted the parallelization performance of GCC to a certain degree, but they failed to resolve the problem that the GCC compiler cannot conduct the dependence analysis of branch nested loops. Aiming at this problem, a dependence test method of dealing with branch nested loops is proposed on the basis of GCC 5.1 in this paper.

In this paper, the nested loop is divided into a perfect nested loop, branch nested loop, and complicated nested loop according to the structural characteristics of nested loops. Then, the dependence analysis module of GCC5.1 is further analyzed. Finally, the dependence relation analysis in the GCC compiler is supplemented, realizing the dependence test method aiming at the branch nested loop. The experiment results show that this method can improve the parallelization efficiency of the program by more than 14%.

2. Related Concepts

The loop is the main target of the parallelization analysis of the parallelizing compiler system and the basic unit facing the shared storage structure OpenMP parallelization [12]. In the parallelizing compiler system, when the loop is judged to be free from dependence or involved in loop-independent dependence, it can conduct parallel execution. In this section, the definition of a branch nested loop is given, and then the basic definition of data dependence is introduced.

2.1. Classification Method of Nested Loop

There are various kinds of nested loop structures in applications, and the forms decide whether the loop can conduct parallel execution. The traditional classification method of nested loop divides the loop into a perfect nested loop and an imperfect nested loop. Most parallelizing compiler systems can only explore the potential parallelization of perfect nested loops effectively. However, there are many branch nested loops in scientific computing programs, such as NPB (NAS parallel benchmark) [13] and SPEC (standard performance evaluation corporation) [14]. Parallelization is a kind of coarse-grained optimizing strategy, and the initialization of parallel code and overhead of synchronous operation also prevent the improvement of program performance [15]. Therefore, the parallelization in the outer layer of loop should be explored as much as possible to achieve better performance improvement. In order to better understand the structure of branch nested loops, the definition of a nested loop is given below.

Definition 1 If the innermost loop body of an N-layer nested *for*-loop is formed by one or several non-loop assignment statements, and these statements satisfy no procedure call and procedure return statements, while the loop body of all other loops is only formed by one *for*-loop statement, the loop of each layer of the nested loop is a perfect nested loop (PNL).

Definition 2 If there are at least two perfect nested loops in the k^{th} -layer ($0 < k \leq N$) loop of the N-layer nested *for*-loop, the loop is a branch nested loop (BNL).

Definition 3 If an N-layer nested *for*-loop is neither the perfect nested loop nor the branch nested loop, it is a complex nested loop (CNL).

2.2. Definition of Data Dependence

Related theories and technologies of dependence are the foundation of guaranteeing the maintenance of semantic equivalence before and after transformation. The dependence relation is a kind of partial ordering relation that can determine the instruction execution sequence, and it can be divided into data dependence relation and control dependence relation. The data dependence is mainly caused by storage access, while the control dependence is mainly caused by the control flow in programs. In most conditions, the control statements of the loop body can be converted to data dependence for treatment through if-conversion. However, in practice, especially in scientific computing programs, the loop body seldom contains the control flow statements. Therefore, the data dependence is the emphasis of our research. The dependence stated in this paper refers to the data dependence if there are no special illustrations. Next, the definition of dependence will be introduced.

Definition 4 There is data dependence from statements $S1$ to $S2$ (statement $S2$ is dependent on $S1$), if and only if:

- Two statements access the same memory unit, and at least one statement is stored in this unit;
- There is a possible execution route for operation from $S1$ to $S2$.

Definition 5 Supposing there is dependence from $S1$ of iteration i to $S2$ of iteration j in the n -layer nested loop, the dependence distance vector $d(i, j)$ is defined as the vector whose length is n , which enables $d(i, j)_k = j_k - i_k$ ($1 \leq k \leq n$);

Definition 6 Supposing there is dependence from $S1$ of iteration i to $S2$ of iteration j in the n -layer nested loop, the dependence direction vector $D(i, j)$ is defined as the vector whose length is n , which enables:

$$D(i, j) = \begin{cases} "<", & \text{if } d(i, j)_k > 0 \\ "=" & \text{if } d(i, j)_k = 0, 1 \leq k \leq n \\ ">", & \text{if } d(i, j)_k < 0 \end{cases}$$

3. GCC Data Dependence Analysis

The GCC compiler is a widely-applied open source compiler that can explore potential parallelization in perfect nested loop applications and produce the OpenMP parallel code facing the shared storage structure [16-17]. As the foundation of deciding whether programs can be executed, dependence analysis should not only consider the subscript form of array reference quoted by arrays in the loop body, but also take the structure of loops into account. In this section, the GCC compiler framework is introduced at first, and organizational flows of loop transformation and data dependence analysis are described. Later, a detailed process of the data dependence analysis of GCC is introduced, and the method of calculating conflict iterations in the subscript-by-subscript form is analyzed emphatically.

3.1. Compiler Framework

The GCC compiler can accept several kinds of source language, like C/C++, Fortran, and JAVA, and parse them into a general abstract syntax tree, namely Generic tree through the corresponding front-end of all source languages in GCC. Afterwards, Generic transforms the code to a GIMPLE three-address intermediate code through Gimplify, and GIMPLE is processed by a GIMPLE-SSA analyzer and transformed to a tree-static single assignment (Tree-SSA) represented by a tree node. Data dependence analysis is conducted on Tree-SSA, and GCC executes a series of optimization passes in the intermediate language level and optimizes all kinds of functions. This process is the middle-end of GCC, and the source code is transformed to GIMPLE middle code again after various kinds of optimization of the Tree-SSA and then to a bottom-based RTL language. At this moment, GCC executes a series of optimization passes in RTL language and conducts the register assignment and code generation. The series of this process is called back-end. The GCC compiler framework is shown in Figure 1.

The execution of optimization pass based on Tree-SSA and RTL intermediate representation in GCC is dispatched by the `execute_one_pass` function in the source code file `Passes.c`. For instance, as shown in Figure 1, loop optimization operations like loop transformation, vectorization, and parallelization are based on the Tree-SSA intermediate representation. The loop transformation (according to the calling sequence of pass) in GCC includes loop unrolling, if hoisting, loop distribution, loop subsection, and loop interchange. The main purpose of loop transformation is to improve the instruction locality and data locality, change the loop dependence relation, and explore more parallelization. Therefore, the pass of loop transformation is executed earlier than the pass of parallelization in general conditions. However, in order to gain more

3.2.2. Classification of Subscript

The term subscript refers to one of the subscript positions in a pair of array references. Since a pair of array references will always be taken into account in the dependence test, the term subscript can be used to indicate the position of a pair of subscripts. For instance, in Figure 2, two subscripts are included in array a: the first subscript is $\langle i, i \rangle$, and the second is $\langle i, j \rangle$. In Figure 3, the first subscript is $\langle i, i \rangle$, and the second is $\langle j, k \rangle$ in array a. The more the loop indexes occur in one subscript, the more complex the dependence test will be. Therefore, in order to test these subscripts much more simply and efficiently, they should be classified.

```

for (i = L1; i < U1; i++)
{
    for(j=L2;j<U2; j++)
        a[i][j] = b[i][j] + D;
    for(k=L3 ;k<U3; k++)
        c[i] [j] = a[i][k] + 8;
}

```

Figure 3. P Branch nested loop

The subscript that does not include the loop index variable is ZIV, the subscript that includes only one loop index variable is SIV, and the subscript that includes more than one loop index variables is MIV. For instance, $\langle 5, 6 \rangle$ is ZIV, $\langle i, i \rangle$ is SIV, and $\langle j, k \rangle$ is MIV.

When testing a multi-dimensional array, divisibility should be used to describe whether subscripts may impact each other. Therefore, subscripts should be classified according to their relation with each other. When the loop index variable of one subscript does not occur in other indexes, it is believed that this subscript is a single array subscript and can be divided. If the same loop index variable occurs in different subscripts, they would be coupled, known as the coupled array subscript.

3.2.3. Calculation of Conflict Iteration in the Subscript-by-Subscript Form

The data dependence analyzer of GCC mainly works out the conflict iteration of each subscript of the same array with the traditional diophantine equation and subscript-by-subscript form, and it applies the recursive chain to indicate the results. Its basic thought is to calculate the direction vector matrix and distance vector matrix of the loop by making use of the conflict iteration, conduct the matrix transformation, and decide whether the loop can carry out the parallelization by judging whether the transformation is legal.

At first, judge whether the equation has solutions simply and rapidly with the GCD test method. If there are no solutions, the non-existence of dependence is proven, and the algorithm should be terminated immediately. The SIV, MIV, and Banerjee inequation test methods can be used to calculate the conflict iteration of the subscript, which should be represented by the recursive chain. The calculation of conflict iteration with the index-by-index form can only transfer the loop dependence distance among each dimension of the array access. Calculate the distance vector matrix and direction vector matrix according to the conflict iteration. Carry out the matrix transformation for the distance vector and direction vector. It is feasible if the transformation is legal; otherwise, it is not. The method of judging whether the loop can be parallelizable through the matrix transformation is only effective for the perfect nested loop, so the data dependence analysis of GCC can only carry out the dependence analysis for perfect nested loops.

It can be learned from the above analysis that the data dependence analyzer of GCC can carry out the dependence analysis of perfect nested loops rapidly and effectively, but it fails to explore the potential parallelization of branch nested loops. Aiming at this condition, a correlation dependence test method is proposed based on the improvement of GCC dependence analysis. This method can explore the parallelization in branch nested loops much more concisely and effectively, and it can improve the performance of programs greatly. Next, the algorithm will be introduced in detail, and examples will be cited to illustrate how to estimate the dependence relation of branch nested loops with this method.

4. Dependence Detection Technology Facing the Branch Nested Loop

The dependence test method based on the branch nested loop no longer adopts the method of detecting the relationship between array subscripts by subscript-by-subscript form in GCC; instead, it mainly judges the correlation between the array subscript and the current loop index. This method is much more effective than the calculation of conflict iteration in the subscript-by-subscript form. Besides, this method only needs to test if the distance vector in the outermost loop is a zero

vector to judge whether the loop can be parallelizable, and there is no need to test if the distance matrix and direction matrix transformation is legal. Consequently, this method is much simpler and more understandable.

4.1. Basic Thoughts of the Algorithm

The basic unit of parallelization is the loop, so only the loop-carrying dependence should be considered, namely, parallelization can be executed if the loop is free from dependence or there is loop-independent dependence. The type of loop dependence can be judged according to the value of the distance vector of the current layer. If the distance vector is a zero vector, it is loop-independent dependence; otherwise, it is loop-carrying dependence. In accordance with this principle, this method only needs to calculate the distance vector of the current layer of branch nested loop. Since only the distance vector of the current loop needs to be calculated, there is no need to calculate the conflict iteration functions of all loop indexes contained in the array subscript by the subscript-by-subscript form, and only the relation between the loop index variable and the current layer needs to be observed to decide whether the conflict iteration should be calculated or not.

4.2. Subscript Correlation Analysis

The correlation analysis of subscripts mainly tests whether the index variable of outer loop is contained in the subscript. When the subscript tested does not contain the index variable, the array subscript index will not change with the loop iteration of the outer layer. At this moment, regardless of whether the subscript contains other loop index variables, there is no need to work out the conflict iteration, and the dependence distance of the outer layer can be set as zero. When the subscript tested only contains the loop index variable of the outer layer, only the dependence distance of the outer layer should be calculated. If different subscripts show different dependence distances carried by the outer layer, there is no dependence, and the algorithm should be terminated. When the subscript tested includes the loop index variable of the outer layer, as well as other index variables, it will be complicated, and the algorithm proposed in this paper cannot deal with it accurately. Instead, conservative analysis can be adopted, and it should test whether the dependence distance vector of the outer loop is a zero vector.

As shown in Figure 2, the reference of array a forms two pairs of subscripts, namely $\langle i, i \rangle$ and $\langle j, k \rangle$. At first, it is found through the analysis of $\langle i, i \rangle$ that it only contains the outer loop index variable, and its dependence distance of the outer layer is calculated as 0. Afterwards, according to the analysis of index $\langle j, k \rangle$, it does not contain the outer loop index variable, and the value of the index would not change with the outer loop index. Therefore, the dependence distance of this subscript can also be set as 0. Consequently, the distance vector of the outer layer of this branch nested loop is $(0, 0)$, and there is no loop-carrying dependence, so the outer loop can be parallelizable. Next, the algorithm will be introduced specifically.

4.3. Specific Steps of the Algorithm

The algorithm proposed in this paper mainly aims at the branch nested loop, so the algorithm of judging whether the loop is a branch nested loop is given. Then, the method of dependence test whether the branch nested loop can be parallelizable is introduced. This algorithm is added to the GCC source code file *tree-data-ref.c*.

Step 1: Judge Whether the Input Loop is a Branch Nested Loop

Since GCC has the filtering algorithm of complicated nested loops, there is no need to consider the process jump or call while judging whether the nested loop is the branch nested loop. It only needs to traverse the loop and test whether there are branches in the same layer of the loop. This algorithm is added to the function *compute_data_dependences_for_loop*, and the specific steps are shown in Figure 4.

Step 2: Process the Array Reference Information

The main flow of the algorithm is shown below: firstly, the alias of array is analyzed, since there is no dependence in arrays that do not share the alias. Then, the expression of the same array subscript is analyzed. If there are subscripts of nonlinear expression containing the loop index, there is independence, and the dependence test should be stopped. Next, the relation between the array subscript and index variable of the outer loop of the branch nested loop is analyzed, and the dependence distance is calculated. Finally, the distance vector of the outer loop of the branch nested loop is analyzed. If it is a zero vector, the loop can be executed in parallel, or it can be determined that there is loop-carrying dependence. The specific algorithm is shown in Figure 5.

```

input □ loop
output □ If The loop is the branch nested loop , then return true □
else return false, and put the loop into the stack Loop_nest.
Procedure find_loop_BNL(loop,loop_nest)
{
1 □ loop_nest->safe_push(loop);    //put the loop into the stack
2: if(loop->inner)                  //whether the loop is nested loop
3:   loop=loop->inner
4:   while(loop)
5 □   if(loop->next)                //whether the loop has brothers
node
6 □   return true □
7:   loop=loop->inner
8:   end while
9:   end if
10: return false;
}

```

Figure 4. The algorithm of judging whether the loop is a branch nested loop

```

Input: the vector Datarefs which record the array reference information.
Output: no dependency return true, else return false.
Procedure computer_BNL_dependence(Datarefs,loop)
{
1: while Datarefs do
2:   if array a&&b is read operation then
3:     continue;
4:   end if
5:   if array a and array b don't share alias then
6:     continue;
7:   end if
8:   if the subscript of array a and array b is nonlinear expression then
9:     return false ;
10:  end if
11:  while the subscripts pair formed by array a and array b do
12:    //Subscripts corresponding to the arrays a and b are chrec_a and chrec_b
13:    //subscript is ZIV
14:    if chrec_a and chrec_b is ZIV then
15:      td <- chrec_a-chrec_b ;
16:      if td !=0 then
17:        d <- 0 ; //dependence distance to the outer layer is 0, store into vector d
18:        go back to the step1 ;
19:      end if
20:      tempd <-0 ;//dependence distance is 0, store into vector tempd
21:    end if
22:    //subscript is SIV
23:    if chrec_a和chrec_b is SIV then
24:      if chrec_a or chrec_b is outer layer index variable expression then
25:        tempd <- chrec_a-chrec_b ;
26:      else if chrec_a and chrec_b neither outer layer index expression
27:        tempd <- 0 ; //dependence distance is 0, store into vector tempd
28:      end if
29:    end if
30:    //subscript is MIV
31:    if chrec_a and chrec_b is MIV
32:      if chrec_a or chrec_b has outer layer index variable
33:        return false ;
34:      else
35:        tempd <- 0 ;
36:      end while
37:      if all values in tempd is 0, or there are two different values that are not 0
38:        d<- 0 ;
39:      else return false;
40:      end if
41:    end while
42:    if d is zero vector
43:      return true;
}

```

Figure 5. Dependence detection algorithm facing the branch nested loop

This method has two advantages:

- It is more convenient and efficient. Instead of calculating the conflict iteration of all subscripts through subscript-by-subscript means, it only calculates the conflict iteration of subscripts related to the outer loop index variable after deciding the correlation between the subscript and the outer loop index variable.
- It not only features low time complexity but also is more understandable. There is no need to calculate the direction vector and distance vector of the entire loop, conduct the matrix transformation, or estimate whether the loop can be parallelizable according to its legality. Instead, it is only necessary to calculate the distance vector of the outer loop and judge whether it is a zero vector.

However, this method still has some restricted conditions when dealing with branch nested loops:

- The subscript of array reference must be the linear expression of loop index variables.
- When the array subscript contains the outer loop index variable, it cannot contain other loop index variables.

The method put forward in this paper holds conservative tests for loops failing to meet the above conditions and judges the loop as not parallelizable.

For algorithms dealing with the loop array reference dependence information, the time cost mainly concentrates on the comparison of all array subscripts pairs. Therefore, the time complexity of the algorithm is $O(n(n-2)/2)$, among which $0 < i, j < n, i < j, n$ is the number of arrays, $(n(n-2)/2)$ is the times of array comparison, and a_{ij} is the number of array subscripts pairs formed by subscripts i and j and stored in *datarefs*.

5. Instance Analysis and Test

5.1. Experimental Platform

In this paper, the dependence analysis of branch nested loop is implemented on the basis of the GCC 5.1 open source compiler, and the compiler system is the Linux operating system, Red Hat Enterprise Linux Server release 5.5 (Tikanga). The experimental platform adopts the intel Xeon processor 5500, and its memory is 4G. The basic frequency is 1600.00Hz, the L1 data cache is 32KB, and the L2 cache is 256KB.

5.2. Program Analysis and Test

To verify the correctness and effectiveness of the algorithm, the NPB 3.3.1 standard test set is used for experimentation. NPB is a parallel benchmark test program developed by NAS, and it is aimed at comparing the performance of parallel machines. NPB3.3.1 has ten programs. The dependence test method proposed in this paper is applied to analyze the nested loop in the ten programs, and the results are shown in Table 1. Afterwards, the implementation of the algorithm and its speedup are analyzed specifically with the MG program, as shown in Figure 6. MG (multi-grid) mainly works out the discrete periodic approximate solution of a 3D Poisson equation with four V-loop multi-grid algorithms, and its kernel program is a branch nested loop, which can display functions of this algorithm explicitly.

Table 1. The results of NPB programs with our dependence test method

Program	Total nested loops (multi-layer)	BNL	Total BNL that can be processed	Total BNL that cannot be processed
MG	18	6	6	0
CG	10	3	2	1
SP	38	5	5	0
BT	35	8	5	3
LU	44	9	7	2
UA	68	1	1	0
EP	1	0	0	0
FT	8	0	0	0
DC	11	0	0	0
IS	0	0	0	0

According to the results in Table 1, it is clear that in the NPB 3.3.1 standard test set, there are 32 branch nested loops. There are 26 branch nested loops where our algorithm can analyze the dependence relations and conduct parallel execution,

and six branch nested loops cannot be processed. The reason why they cannot be processed is that the subscript tested covers the outer loop index variable and some inner layer loop index variables.

Since there are no branch nested loops in programs EP, FT, IS, and DC, and the branch nested loop in program UA is not included in the hotspot function, the algorithm has no speedup effect on these programs. However, in programs CG, SP, BT, and LU, the speedup can reach 8%. Since the kernel loop of the hotspot function of program MG is a branch nested loop, our algorithm receives the most evident speedup effect on it, and it can reach 60% of the manual parallel code in four threads. Next, the MG program will be analyzed.

The hotspot function in MG is *resid* and *psinv*, which accounts for 53.35% and 27.24% of the serial execution time, respectively. The two functions are similar in structure and only contain a three-layer nested loop, which is also a branch nested loop. The loop code is shown in Figures 6(a) and 6(b).

```

do i3=2,n3-1
  do i2=2,n2-1
    do i1=1,n1
      u1(i1) = u(i1,i2-1,i3) + u(i1,i2+1,i3)
    >      + u(i1,i2,i3-1) + u(i1,i2,i3+1)
      u2(i1) = u(i1,i2-1,i3-1) + u(i1,i2+1,i3-1)
    >      + u(i1,i2-1,i3+1) + u(i1,i2+1,i3+1)
    enddo
    do i1=2,n1-1
      r(i1,i2,i3) = v(i1,i2,i3)
    >      - a(0) * u(i1,i2,i3)
    >      - a(2) * ( u2(i1) + u1(i1-1) + u1(i1+1) )
    >      - a(3) * ( u2(i1-1) + u2(i1+1) )
    enddo
  enddo
enddo

```

(a) Resid kernel loop code

```

do i3=2,n3-1
  do i2=2,n2-1
    do i1=1,n1
      r1(i1) = r(i1,i2-1,i3) + r(i1,i2+1,i3)
    >      + r(i1,i2,i3-1) + r(i1,i2,i3+1)
      r2(i1) = r(i1,i2-1,i3-1) + r(i1,i2+1,i3-1)
    >      + r(i1,i2-1,i3+1) + r(i1,i2+1,i3+1)
    enddo
    do i1=2,n1-1
      u(i1,i2,i3) = u(i1,i2,i3)
    >      + c(0) * r(i1,i2,i3)
    >      + c(1) * ( r(i1-1,i2,i3) + r(i1+1,i2,i3)
    >      + r1(i1) )
    >      + c(2) * ( r2(i1) + r1(i1-1) + r1(i1+1) )
    enddo
  enddo
enddo

```

(b) Psinv kernel loop code

Figure 6. *Resid* and *psinv* kernel loop code

Since the loop structures in Figures 6(a) and 6(b) are similar, the kernel loop of function *resid* is analyzed. The loop in Figure 6(a) is a branch nested loop. GCC cannot conduct the dependence analysis for the loop, only conservative judgment. It only conducts further dependence tests on two parallel loops on the third layer, concludes that there is no dependence, and conducts the parallel treatment. It is clear that this practice fails to fully explore the parallelization of the loop, and it also brings more synchronous operation, which greatly impacts the performance of the program. Furthermore, the algorithm put forward in this paper can not only conduct the dependence analysis for the loop, but also correctly analyze that there is no loop-carrying dependence in the outer loop. The specific analysis process is the following: array $u1(i1)$ and array $u(i1,i2-1,i3)$ are not aliases of each other, there is no loop-carrying dependence in the outer layer, and its dependence distance is zero. The subscript formed by array $u1(i1)$ and array $u1(i1-1)$ is $(i1,i1-1)$, excluding the expression of the outer loop index variable. Therefore, the dependence distance of its outer layer is zero, and the distance vector of this subscript to the outer layer is a zero vector. The direction vector of the outer loop is "=", and the outer loop can be vectorized. Figure 7 shows the speedup of program MG with different thread counts, while using the manual parallel code and other different parallel analysis methods.

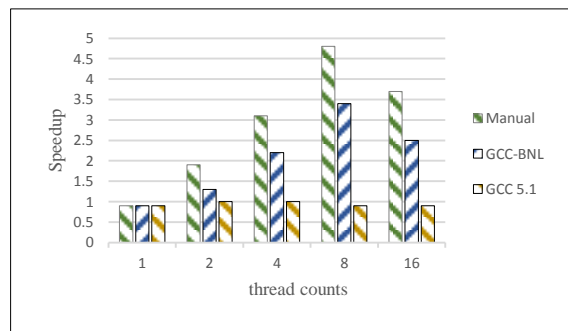


Figure 7. MG program speedup

Since GCC 5.1 can only analyze the dependence relation of the perfect nested loop in a branch nested loop and parallelize the inner loop, it results in too many synchronous operations during the parallel execution of the program. For the loop in Figure 6(a), GCC can only parallelize the innermost layer, and the times of synchronization include at least $2(n_2 - 2)(n_3 - 3)$. Due to the excessive synchronization cost, the parallelization is hardly accelerated.

A conclusion can be drawn from Table 1 and Figure 6 that our dependence analysis method aiming at the branch nested loop can analyze the potential parallelization of such loops rapidly, which may enhance the possibility of code parallelization and accelerate the program execution speed effectively.

6. Conclusions

Perfect nested loops are a kind of nested loop with compact structure, but branch nested loops often occur in applications. If dependence analysis cannot be conducted for such loops, the opportunity of improving the program performance would be lost. Consequently, a dependence test technology based on branch nested loops is put forward, and it aims at the condition that GCC fails to carry out dependence analysis for branch nested loops. According to the experiment results, the algorithm put forward in this paper can rapidly and accurately analyze the dependence relation of branch nested loops and enhance the dependence analysis ability of GCC. However, this algorithm also has some conservative test results, and its performance remains to be further improved; these will be studied in future works.

Acknowledgements

This work was financially supported by the National Key Research and Development Program "High-Performance Computing" Key Special (No. 2016YFB0200503) and the Youth Science Fund Project of Henan Normal University (No. 2015QK21).

References

1. H. H. Fu, J. F. Liao, J. Z. Yang, L. N. Wang, Z. Y. Song, X. M. Huang, et al., "The Sunway TahuLight Supercomputer: System and Applications," *Science China Information Sciences*, Vol. 59, No. 7, pp. 1-16, 2016
2. L. Han, "Research on Consistent Optimization Techniques of Parallel Decomposition for Distributed Memory Architecture," PLA Information Engineering University, 2008
3. M. W. Hall, S. P. Amarasinghe, B. R. Murphy, S. W. Liao, and M. S. Lam, "Interprocedural Parallelization Analysis in SUIF," *ACM Transactions on Programming Languages and Systems*, Vol. 27, No. 4, pp. 662-731, 2005
4. R. Allen and K. Kennedy, "Optimizing Compilers for Modern Architectures," Morgan Kaufmann, 2002
5. R. D. Venkatasubramanyam, "Array Access Analysis in Open64," University of Houston, 2004
6. N. Nethercote and J. Seward, "How to Shadow Every Byte of Memory used by a Program," in *Proceedings of the 3rd International Conference on Virtual Execution Environments*, pp. 65-74, 2007
7. Q. Zhao, D. Bruening, and S. Amarasinghe, "Umbra: Efficient and Scalable Memory Shadowing," in *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pp. 22-31, 2010
8. D. Berlin and D. Edelsohn, "High-Level Loop Optimizations for GCC," in *Proceedings of GCC Developers Summit*, pp. 37-54, 2004
9. L. Y. Zeng, C. Q. Yang, and C. Huang, "Analysis and Improvement of the GCC 4.1 Data Dependence Analyzer," *Computer Engineering and Science*, Vol. 28, No. 10, pp. 104-106, 2006
10. Q. S. Zhang, Y. Li, and Z. D. Fan, "Automatic Parallelization for Loops Carried Data Dependence Between Iteration," *Journal of Chinese Mini-Micro Computer Systems*, Vol. 35, No. 6, pp. 1293-1297, 2014
11. "Cit: A GCC Plugin for the Analysis and Characterization of Data Dependencies in Parallel Programs," (http://cas.et.tudelft.nl/pubs/Kumar_DCIS_2013.pdf, last accessed on March 10, 2018)
12. Z. Wang, G. Tournavitis, B. Franke, M. F. P. O'boyle, "Integrating Profile-Driven Parallelism Detection and Machine-Learning-based Mapping," *ACM Transactions on Architecture and Code Optimization*, Vol. 11, No. 1, pp. 1-26, 2014
13. D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, et al., "The NAS Parallel Benchmarks," *International Journal of High Performance Computing Applications*, Vol. 5, No. 3, pp. 63-73, 1991
14. J. L. Henning, "SPEC CPU2006 Benchmark Descriptions," *Acm Sigarch Computer Architecture News*, Vol. 34, No. 4, pp. 1-17, 2006
15. P. F. Huang, R. C. Zhao, Y. Yao, and J. Zhao, "Parallel Cost Model for Heterogeneous Multi-Core Processors," *Journal of Computer Applications*, Vol. 33, No. 6, pp. 1544-1547, 2013
16. C. H. Liao, "A Compile-Time OpenMP Cost Model," University of Houston, 2007
17. Z. J. Guo and H. Liu, "A New Compiler Framework based on Superword Level Parallel," *International Journal of Performability Engineering*, Vol. 14, No. 10, pp. 2511-2521, 2018

Hui Liu graduated from the State Key Laboratory of Mathematical Engineering and Advanced Computing at PLA Information Engineering University with a Ph.D. She is currently a lecturer in the College of Computer and Information Engineering at Henan Normal University. Her research interests include high performance computing, program performance optimization, and compiler optimization.

Jinlong Xu graduated from the State Key Laboratory of Mathematical Engineering and Advanced Computing at PLA Information Engineering University with a Ph.D. He currently works there as a lecturer. His research interests include high performance computing, program performance optimization, and compiler optimization.

Lili Ding graduated from the State Key Laboratory of Mathematical Engineering and Advanced Computing at PLA Information Engineering University with a master's degree. Her current research interests include high performance computing and program performance analysis.