Research Article

Loop Selection for Multilevel Nested Loops Using a Genetic Algorithm

Kai Nie ^{1,2} Qinglei Zhou ² Hong Qian ³ Jianmin Pang ¹ Jinlong Xu ¹
and Xiyan Li ¹

¹State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou 450001, China

²Zhengzhou University, Zhengzhou, Henan 450001, China

³Jiangnan Institute of Computing Technology, Wuxi, Jiangsu 214083, China

Correspondence should be addressed to Jinlong Xu; longkaizh@126.com

Received 12 November 2020; Revised 4 March 2021; Accepted 21 March 2021; Published 1 April 2021

Academic Editor: Weifeng Pan

Copyright © 2021 Kai Nie et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Loop selection for multilevel nested loops is a very difficult problem, for which solutions through the underlying hardware-based loop selection techniques and the traditional software-based static compilation techniques are ineffective. A genetic algorithm-(GA-) based method is proposed in this study to solve this problem. First, the formal specification and mathematical model of the loop selection problem are presented; then, the overall framework for the GA to solve the problem is designed based on the mathematical model; finally, we provide the chromosome representation method and fitness function calculation method, the initial population generation algorithm and chromosome improvement methods, the specific implementation methods of genetic operators (crossover, mutation, and selection), the offspring population generation method, and the GA stopping criterion during the GA operation process. Experimental tests with the SPEC2006 and NPB3.3.1 standard test sets were performed on the Sunway TaihuLight supercomputer. The test results indicated that the proposed method can achieve a speedup improvement that is superior to that by the current mainstream methods, which confirm the effectiveness of the proposed method. Solving the loop selection problem of multilevel nested loops is of great practical significance for exploiting the parallelism of general scientific computing programs and for giving full play to the performance of multicore processors.

1. Introduction

With the rapid development of multicore processor technology, how to effectively use multicore processors to improve the performance of general scientific computing programs is challenging. One way is to create multiple threads that can be executed in parallel. In general, in scientific computing programs, there is an enormous number of loops, which are well structured and take up a large amount of time for the program to execute. Therefore, many studies have focused on creating multiple threads to execute the loops in parallel to improve the overall performance of the program. However, due to the presence of complex dependencies, loops cannot be directly executed in parallel unless there is parallel transformation (e.g., loop distribution and loop exchange). With single-level loops or a small number of nested loop levels (≤ 3), parallelism can

usually be readily achieved through loop transformation. In the presence of many nested loop levels, it must be discussed separately. In one case, there are loop levels for which parallelism can be directly executed in the multilevel nested loops, and thus, they are moved to the outermost layer for parallel execution through loop exchange. Unfortunately, this situation is usually very rare. In another case, which is more common, there are no loop levels at which parallelism can be directly executed in multilevel nested loops, and a set of loop levels must be selected to be moved to the outermost layer for serial execution to expose possible new parallelism. As the number of nested loop layers increases, the dependencies between the loop iterations become very complicated, which makes it difficult to analyze and search for such a group of loop levels.

There are currently two approaches for the parallelism problem of multilevel nested loops. One approach is to use

the multithread implementation technology of the underlying hardware, in which the appropriate loop layer is chosen for multithread parallelization with the assistance of the underlying hardware, e.g., the thread-level speculation (TLS) technique and the transactional memory (TM) technique. However, the dependency analysis capability of the underlying hardware is limited and lacks flexibility, which makes it difficult for the TLS technique to analyze the complex dependencies of the multilevel nested loops and thus is impossible to provide effective parallelism [1–6]. The hardware TM technique has limitations on the transactional size (transactional buffer) and duration (operating system events and disruptions can abort the transaction); therefore, when applied to the thread-level parallelism of nested loops, it often leads to overflow of the underlying hardware resources due to excessive granularity of the transaction itself, which ultimately manifests poor parallel performance [7–12]. Another common approach is to use the traditional software-based static compilation techniques, in which the dependency analysis test module in the compiler is used to test whether there are dependencies between the loop iterations in a layerwise manner. However, as the number of levels of nested loops to be tested increases, the number of dependencies that must be tested increases exponentially or even factorially, which makes the compilation time unbearably long. When encountering this situation, mainstream compilers (e.g., gcc) usually adopt a conservative approach; i.e., they assume that dependencies are present between all iterations of the loops, and thus, parallelism cannot be implemented. Typically, scientific computing programs contain a large number of loops, with also a large number of nesting levels. Researchers from the University of Minnesota and Intel have made a survey on the maximum numbers of nested loop levels and loops in some scientific computing programs in SPEC2006, and the result is shown in Table 1 [13]. Obviously, the number of nested loop levels in the SPEC2006 standard test set is very large (e.g., the nesting depth of 445.gobmk reaches even 22 levels), with a large number of such loops. Therefore, solving the parallelism problem in nested loops can not only enhance the overall performance of general scientific computing programs but also be of great practical significance for expanding the applications scope of thread-level parallelism and making effective use of multicore processors.

Since common multilevel nested loops lack loop layers that can be directly executed in parallel, the key to solving the nested loop parallelism issue is to select a group of appropriate loop layers and move them to the outermost layer, to enable them to be executed in serial, for the purpose of exposing new parallelism. To expose parallelism with maximum granularity, the number of loop layers that are chosen to be exchanged to the outermost layer to be executed in serial must be as low as possible. With the deepening of the number of loop layers, the granularity of the parallelism becomes increasingly small. Therefore, to improve the parallelism performance of multilevel nested loops in general scientific computing programs, in this study, we propose a method that uses the least number of loops to expose the maximum parallelism in multilevel nested loops.

TABLE 1: SPEC2006 benchmarks.

Benchmark	Maximum loop nesting level	Number of loops
401.bzip2	11	232
429.mcf	5	52
445.gobmk	22	1265
456.hmmer	5	851
458.sjeng	10	254
462.libquantum	4	94
464.h264ref	15	1870
473.astar	6	116
433.milc	11	421
444.namd	4	619
453.povray	15	1311
470.lbm	3	23
482.sphinx3	8	609

Many researchers have informally pointed out that selecting the smallest set of loop layers from multilevel nested loops to obtain maximum parallelism is a nondeterministic polynomial complete (NPC) problem [14, 15]. As the number of nested loop levels increases, the scale of the problem grows very fast (in factorial order). Therefore, to solve the loop selection problem of multilevel nested loops, some heuristic methods can only be used to obtain approximate solutions, from which a satisfactory solution is then chosen. Compared with traditional heuristic methods, genetic algorithms (GAs) have a very strong search ability; they can find the global optimal solution of a problem with a high probability, and their inherent parallelism is more suitable for processing optimization problems [16–23]. Therefore, a GA was adopted in this study to solve the loop selection problem.

The main contributions of this study are as follows: (1) a formal specification and mathematical model of the loop selection problem of multilevel nested loops is presented; (2) a new idea for solving the loop selection problem of multilevel nested loops is provided; and (3) the initial population generation algorithm, chromosome repair algorithm, and the calculation method for a series of genetic parameters are designed for GAs to solve the multilevel nested loop selection problem. The experimental results show that compared with the current mainstream methods for the loop selection problem of multilevel nested loops (e.g., static compilation combined with a dynamic operation method and machine learning methods), the proposed method is more effective, which indicates that the parallel compilation of programs is still a key factor that restricts the capabilities of hardware multicore processors.

2. Overview of Solutions to the Loop Selection Problem of Multilevel Nested Loops

In this section, we outline the proposed solution to the loop selection problem of multilevel nested loops. First, we present the formal specification of the loop selection problem; second, we further abstract the mathematical model of the problem; last, we design the solving algorithm for the problem based on the mathematical model of the problem, as shown in Figure 1.

FIGURE 1: Solution to the loop selection problem of multilevel nested loops.

We first use a direction matrix to represent the complex dependencies of multilevel nested loops, and then, we symbolize the feasible solutions, constraints, and optimal solution of the problem with sets. Next, we use the 0-1 matrix to represent the direction matrix and the selection vector to represent the set of feasible solutions, and we give a mathematical representation of the constraints of the feasible solutions to further abstract the mathematical model of the problem.

Based on the mathematical model of the problem, we found that the solution space of the problem is determined and all possible solutions can be represented through coding. Moreover, we found that the fitness function of the possible solutions can be determined, thus meeting the preconditions of using GAs to solve a problem. Therefore, we opt to use GAs to solve the problem. In the following sections, we will present the formal specification and mathematical model of the problem and the specific steps of the GA when solving the problem, in detail.

3. Formal Specification and Mathematical Model of the Loop Selection Problem of Multilevel Nested Loops

3.1. Example Problem. Before giving the formal specification of the loop selection problem of multilevel nested loops, we first illustrate the difficulty in the problem with the following example, which is derived from simplifying a nested loop segment from a large scientific computing program, as shown in Figure 2. Figure 3 is the direction matrix based on the dependencies in the nested loop segment, in which the row represents the direction vector of the dependency of the syntaxes in the nested loop segment and the column represents the direction vector of a nested loop layer in all directions. According to the theorem proposed by Allen and Kennedy, it is safe to exchange loops in the nested loop segment [14], but all loop layers carry the dependencies (because “<” is present in each column of the direction matrix); as a result, this loop segment has no loop layer that can be directly parallelized.

In Figure 3, the set composed of loop layers in the nested loop segment is $L = \{l_1, l_2, l_3, l_4\}$, and the direction vector set

is $D = \{d_1, d_2, d_3, d_4, d_5, d_6\}$. At the position of the i^{th} row of the j^{th} column of the direction matrix, “<” indicates that the nested layer of l_j carries dependencies at the direction vector of d_i , and the serial execution of l_j can satisfy all of the dependencies carried in d_i ; i.e., l_j covers d_i . Table 2 shows the set of loop layers that cover all direction vectors. By randomly selecting a loop layer, respectively, from the set of $s_1, s_2, s_3, s_4, s_5, s_6$, a loop layer set that covers all direction vectors ($L_1 = \{l_1, l_2, l_3\}$, $L_2 = \{l_1, l_2, l_4\}$, $L_3 = \{l_3, l_4\}$, and so on) is obtained.

In Set L_1 , the safe parallel execution of the loop layer l_4 can be achieved by serial execution of loop layers l_1, l_2 , and l_3 . In Set L_2 , by serially executing l_1 and l_2 while loop-exchanging l_3 and l_4 , the safe parallel execution of the loop layer l_3 can be achieved. In Set L_3 , by changing l_3 and l_4 to the outermost layers for serial execution through the loop exchange technique, the safe parallel execution of the l_1 and l_2 loop layers can be achieved and so on. Finally, we found that, in this example, the maximum safe parallel execution granularity can be obtained through Set L_3 , as shown in Figure 4. Therefore, the loop selection problem of multilevel nested loops is essentially the search for the coverage of all direction vectors for maximum granularity of parallelism through the smallest nested loop layer set.

Through the description of this example, we found that, in the case of a small number of nested loops (e.g., three or fewer layers), the problem can be directly solved through enumeration. As the number of nested loops increases, a combinatorial explosion occurs, which makes the solving difficult. Wang et al. have informally argued for selecting the least number of loop layers in multilevel nested loops to cover all direction vectors, thus revealing that maximum parallelism is an NPC problem [15]. Premised under legitimate loop exchanges and based on a direction matrix, we propose a method of processing multilevel (≥ 4) nested loops. In the next section, we will present the formal specification of the loop selection problem of multilevel nested loops and then the mathematical model of the problem to provide a theoretical basis for the design of an algorithm for solving the problem.

```

l1: for(i = 2; i <= 1999; ++i) {
l2:   for(j = 2; j <= 1999; ++j) {
l3:     for(k = 2; k <= 1999; ++k) {
l4:       for(l = 2; l <= 1999; ++l) {
          a[i][j][k+1][l] = a[i][j][k][l-1] + a[i-1][j][k][l] + a[i][j-1][k][l];
          b[i+1][j+1][k][l+1] = b[i+1][j][k][l] + b[i][j][k-1][l+1] + b[i][j+1][k][l];
        }
      }
    }
  }
}

```

FIGURE 2: Nested loop segment in a scientific computing program.

	l_1	l_2	l_3	l_4
d_1	=	=	<	<
d_2	<	=	<	=
d_3	=	<	<	=
d_4	=	<	=	<
d_5	<	<	<	=
d_6	<	=	=	<

FIGURE 3: Direction matrix of the nested loop segment example.

TABLE 2: Set of loop layers covering all direction vectors.

Direction vectors d_i	The set of covering s_i
d_1	$s_1 = \{l_3, l_4\}$
d_2	$s_2 = \{l_1, l_3\}$
d_3	$s_3 = \{l_2, l_3\}$
d_4	$s_4 = \{l_2, l_4\}$
d_5	$s_5 = \{l_1, l_2, l_3\}$
d_6	$s_6 = \{l_1, l_4\}$

3.2. Formal Specification of the Loop Selection Problem of Multilevel Nested Loops. As shown in Figure 5, without losing generality, we assume that loop exchanges of nested loops are legitimate in nested loops with n layers that contain m dependencies ($m > n$, $n \geq 4$, $m \in N_+$, $n \in N_+$), with a direction matrix, as shown in Figure 6. The goal is to seek the set that has the least number of loop layers that cover all direction vectors.

The following two aspects should be noted here. First, the multilevel nested loops to be solved must have legitimate loop exchanges. According to Allen and Kennedy, loop selection must be based on legal loop exchange [14]; if the legality of the loop exchange cannot be guaranteed, then the loop selection technique does not exist. Therefore, in this study, it is required that the loop exchange of multilevel nested loops be legal. Second, in this study, we focus on loop

layers that cannot be directly executed in parallel, i.e., each loop layer of the n -layer nested loops carries at least one dependency, and therefore, it is required that $m > n$.

Next, we proceed to the formal specification on the loop selection problem for the n -layer nested loops shown in Figure 5. As shown in Figure 6, for the set that is composed of all nested loop layers ($L = \{l_1, l_2, \dots, l_n\}$) and the set that is composed of all direction vectors ($D = \{d_1, d_2, \dots, d_k\}$), assuming that there are Q ($Q \geq 1$, $Q \in N_+$) subsets ($L_1, L_2, \dots, L_Q$) of Set L that cover all direction vectors ($d_1, d_2, \dots, d_k$) and the sets that are composed of the loops that cover the direction vectors of $d_1, d_2, \dots, d_k$ are $s_1, s_2, \dots, s_k$, respectively, and then the subsets $L_1, L_2, \dots, L_Q$ must satisfy the following:

$$\begin{cases} |s_1 \cap L_q| \geq 1, & q = 1, 2, \dots, Q, \\ |s_2 \cap L_q| \geq 1, & q = 1, 2, \dots, Q, \\ \dots \\ |s_k \cap L_q| \geq 1, & q = 1, 2, \dots, Q. \end{cases} \quad (1)$$

Furthermore, the subset L_w that satisfies the condition below is the desired set:

$$|L_w| = \min(|L_1|, |L_2|, \dots, |L_Q|), \quad w \in \{1, 2, \dots, Q\}. \quad (2)$$

3.3. Mathematical Model of the Loop Selection Problem of Multilevel Nested Loops. In the direction matrix of the n -layer nested loops shown in Figure 6, the position in the direction matrix with "<" is represented by 1 and that with "=" by 0, which gives rise to $(a_{ij})_{k \times n}$, and the 0-1 matrix that corresponds to the direction matrix, as shown in Figure 7. In the 0-1 matrix, $a_{ij} = 1$ means that the loop layer of l_j ($j \in \{1, 2, \dots, n\}$) covers the direction vector of d_i ($i \in \{1, 2, \dots, k\}$). To facilitate further specification, it is necessary to introduce the selection vector.

Definition 1. (selection vector). Let $L_t = \{l_{t1}, l_{t2}, \dots, l_{tn_t}\}$ ($n_t \geq 1$, $n_t \in N_+$) be a set that is composed of n_t nested loop layers (i.e., $\forall l_{tu} \in L, u \in \{1, 2, \dots, n_t\}$), and let $x_{t1}, x_{t2}, \dots, x_{tn_t}$ be the corresponding 0-1 selection numbers of the nested loop layers of $l_1, l_2, \dots, l_n$. If $l_j \in L_t$, then $x_{tj} = 1$; if $l_j \notin L_t$, then $x_{tj} = 0$. The vector $x_t = [x_{t1}, x_{t2}, \dots, x_{tn_t}]$, which is

```

l3: for(k = 2; k <= 1999; ++k) {
l4:   for(l = 2; l <= 1999; ++l) {
l1:     parallel for(i = 2; i <= 1999; ++i) {
l2:       parallel for(j = 2; j <= 1999; ++j) {
          a[i][j][k+1][l] = a[i][j][k][l-1] + a[i-1][j][k][l] + a[i][j-1][k][l];
          b[i+1][j+1][k][l+1] = b[i+1][j][k][l] + b[i][j][k-1][l+1] + b[i][j+1][k][l];
        }
      }
    }
  }
}

```

FIGURE 4: Parallelism representation of a scientific computing program segment.

```

l1: for (iv1 = init1; iv1 < up1; iv1++)
l2:   for (iv2 = init2; iv2 < up2; iv2++)
...
ln:     for (ivn = initn; ivn < upn; ivn++) {
      a[iv1 + const11][iv2 + const12]...[ivn + const1n] = b[iv1 + const21][iv2 + const22]...[ivn + const2n] + ...
      ...
      c[iv1 + const(j-1)1][iv2 + const(j-1)2]...[ivn + const(j-1)n] = d[iv1 + constj1][iv2 + constj2]...[ivn + constjn] + ...
      ...
      e[iv1 + constm1][iv2 + constm2]...[ivn + constmn] = f[iv1 + const(m+1)1][iv2 + const(m+1)2]...[ivn + const(m+1)n]
    }
  }
}

```

FIGURE 5: An example of nested loops with n layers that contain m dependencies.

$$\begin{array}{c|cccc}
 & l_1 & l_2 & \dots & l_n \\
\hline
d_1 & < & = & \dots & = \\
d_2 & = & < & \dots & < \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
d_k & < & = & \dots & <
\end{array}$$

FIGURE 6: The direction matrix of nested loops with n layers that contain m dependencies.

$$(a_{ij})_{k \times n} = \begin{vmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 0 & \dots & 1 \end{vmatrix}$$

FIGURE 7: The 0-1 matrix of the n -layer nested loops.

composed of $x_{t1}, x_{t2}, \dots, x_{tm}$, is the selection vector that corresponds to L_t .

According to the definition of the selection vector, in the example described in Section 3.1, the selection vectors of $L_1 = \{l_1, l_2, l_4\}$, $L_2 = \{l_1, l_2, l_3\}$, and $L_3 = \{l_3, l_4\}$ are $x_1 = [x_{11}, x_{12}, x_{13}, x_{14}] = [1, 1, 0, 1]$, $x_2 = [x_{21}, x_{22}, x_{23}, x_{24}] = [1, 1, 1, 0]$, and $x_3 = [x_{31}, x_{32}, x_{33}, x_{34}] = [0, 0, 1, 1]$, respectively. According to the mutual exclusiveness of sets, the set that is composed of the different loop layers has its unique corresponding selection vector, and thus, the above three selection vectors are not identical. At the same time, the size of the set can be calculated through the selection vectors; for example, $|L_1| = x_{11} + x_{12} + x_{13} + x_{14} = 1 + 1 + 0 + 1 = 3$, $|L_2| = x_{21} + x_{22} + x_{23} + x_{24} = 1 + 1 + 1 + 0 = 3$, and $|L_3| = x_{31} + x_{32} + x_{33} + x_{34} = 0 + 0 + 1 + 1 = 2$.

Next, Q groups of sets $(L_1, L_2, \dots, L_Q)$ that cover all direction vectors $(d_1, d_2, \dots, d_k)$ are represented by the selection vector. The selection vector of L_1 is represented by $x_1 = [x_{11}, x_{12}, \dots, x_{1n}]$, with a size of $|L_1| = x_{11} + x_{12} + \dots + x_{1n} = \sum_{j=1}^n x_{1j}$; the selection vector of L_2 is represented by $x_2 = [x_{21}, x_{22}, \dots, x_{2n}]$, with a size of $|L_2| = x_{21} + x_{22} + \dots + x_{2n} = \sum_{j=1}^n x_{2j}$; the selection vector of L_Q is represented

by $x_Q = [x_{Q1}, x_{Q2}, \dots, x_{Qn}]$, with a size of $|L_Q| = x_{Q1} + x_{Q2} + \dots + x_{Qn} = \sum_{j=1}^n x_{Qj}$.

According to the definitions of the selection vector $(x_q (q \in \{1, 2, \dots, Q\}))$ and the 0-1 matrix $((a_{ij})_{k \times n})$, we have

$$\begin{aligned} (a_{ij})_{k \times n} \times (x_q)_{n \times 1}^T &= \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{k1} & a_{k2} & \cdots & a_{kn} \end{bmatrix} \times \begin{bmatrix} x_{q1} \\ x_{q2} \\ \vdots \\ x_{qn} \end{bmatrix} \\ &= \left[\sum_{j=1}^n a_{1j}x_{qj} \quad \sum_{j=1}^n a_{2j}x_{qj} \quad \cdots \quad \sum_{j=1}^n a_{kj}x_{qj} \right]. \end{aligned} \quad (3)$$

From the constraint formula (1), we have

$$\begin{cases} \sum_{j=1}^n a_{1j}x_{qj} \geq 1, & q = 1, 2, \dots, Q, \\ \sum_{j=1}^n a_{2j}x_{qj} \geq 1, & q = 1, 2, \dots, Q, \\ \dots \\ \sum_{j=1}^n a_{kj}x_{qj} \geq 1, & q = 1, 2, \dots, Q. \end{cases} \quad (4)$$

According to formula (2), the selection vector x_w that satisfies the equation below is the selection vector that corresponds to the desired set:

$$\min \left(\sum_{j=1}^n x_{1j}, \sum_{j=1}^n x_{2j}, \dots, \sum_{j=1}^n x_{Qj} \right) = \sum_{j=1}^n x_{wj}, \quad (5)$$

$w \in \{1, 2, \dots, Q\}$.

3.4. Formal Specification and Mathematical Model. In this section, we show the graphical comparison of the formal specification and mathematical model of the loop selection problem of the n -layer nested loops, as shown in Figure 8.

4. Solving the Loop Selection Problem of Multilevel Nested Loops Using GAs

In this section, we will present the steps of solving the loop selection problem of multilevel nested loops using a GA. First, we will give a brief introduction to GAs; second, we will show the overall framework of the solution with GAs; last, we will describe the chromosome representation method and fitness function calculation method in the GA operation process, the initial population generation algorithm and chromosome improvement methods, the specific implementation method of genetic operators (crossover, mutation, and selection), and the offspring population generation method and the stopping criterion of the GA.

4.1. Basic Framework of the GA. Conceived based on the principle of biological evolution, GA is a bionic algorithm for searching for the optimal solution. It simulates the natural process of gene recombination and evolution and encodes the parameters of the problem to be solved into binary code or decimal code (or other numerical code), i.e., genes and multiple genes form a chromosome (individual). Paired crossover and mutation operations similar to natural selection are then performed on many chromosomes and iterated (i.e., generational inheritance) until the final optimization result is obtained [24]. The basic framework of the GA is as follows:

```

begin/*GA*/
  t ← 0;/* number of evolutionary generations*/
  Generate initial population P(t);
  Calculate the fitness value of each individual in the
  initial population P(t);
  while (the stopping criterion is not met) do
    /* Use the following operations to generate new
    individuals, and select better individuals to form a
    new population*/
    (1) Recombine individuals in population P(t)
    through reproduction, crossover or mutation to
    generate candidate population C(t);/* Note that C(t)
    does not include individuals of P(t) */
    (2) Calculate the fitness value of each individual in
    C(t);
    (3) Choose better individuals from C(t) and P(t) to
    form a new population P(t+1) according to the
    fitness values;
    (4) t ← t+1
  end while
  Choose the best individual in P(t) as the solution;
end begin

```

4.2. Basic Framework for Solving the Loop Selection Problem of Multilevel Nested Loops Using a GA. As shown in Figure 9, we describe the basic framework for solving the loop selection problem of multilevel nested loops using GAs, with the following steps:

- (1) Determine the chromosome representation scheme and the calculation method for the fitness function.
- (2) Generate the initial population. Calculate the size of the population (N_p) and input the loop layer set ($L = \{l_1, l_2, \dots, l_n\}$) and direction vector set ($D = \{d_1, d_2, \dots, d_k\}$) into Algorithm 1 to generate the initial population.
- (3) Crossover operation. Select parents using the tournament selection method; generate offspring chromosomes using the single-point crossover technique.
- (4) Mutation operation. Select $\lceil N_p/n \rceil$ individuals from offspring chromosomes to perform a mutation operation to generate new offspring chromosomes.

FIGURE 8: Formal specification and mathematical model of the loop selection problem.

FIGURE 9: Basic framework for solving the loop selection problem of multilevel nested loops using GAs.

(5) Chromosome repair. Perform feasibility judgment on offspring chromosomes generated through crossover and mutation; repair infeasible chromosomes using

Algorithm 2. Enter the repaired offspring chromosomes along with the parent chromosomes to the candidate set of the offspring population.

Input: $L = \{l_1, l_2, \dots, l_n\}, D = \{d_1, d_2, \dots, d_k\}$ / * all loop layer set L , all direction vector set D */
Output: S_{ini-p} / * chromosome set of the initial population */

```

(1) function INITIALISEPOPULATION ( $L, D$ )
(2)   Calculate the population size  $N_p = \mu\phi n$ 
(3)   for  $i = 1 \longrightarrow k$  do
(4)     Calculate the set of loop layers ( $s_i$ ) that cover the direction vector of  $d_i$ 
(5)   end for
(6)   for  $q = 1 \longrightarrow N_p$  do
(7)     for  $i = 1 \longrightarrow k$  do
(8)       Randomly choose a loop layer ( $l_j$ ) from  $s_i$  and add it to the  $\alpha_q$  set
(9)     end for
(10)    while  $\alpha_q \neq \emptyset$  do / * Delete the redundant loop layer */
(11)       $\alpha_q = \alpha_q - l_j$  ( $l_j \in \alpha_q$ ) / * randomly remove the  $l_j$  loop layer from the  $\alpha_q$  set */
(12)      for  $i = 1 \longrightarrow k$  do
(13)        Calculate the number of loop layers ( $v_i$ ) in the  $\alpha_q$  set that covers the direction vector of  $d_i$ 
(14)        if  $v_i < 1$  then
(15)           $L_q = L_q \cup l_j$ 
(16)          break;
(17)        end if
(18)      end for
(19)    end while
(20)    Generate Chromosome  $X_q$  that corresponds to  $L_q$  and add it to the  $S_{ini-p}$  set
(21)  end for
(22) end function

```

ALGORITHM 1: Initial population generation.

Input: $X_q, D = \{d_1, d_2, \dots, d_k\}$ / * chromosome to be repaired (X_q), the set of all direction vectors (D) */
Output: X'_q / * repaired chromosome (X'_q) */

```

(1) function REPAIRCHROMOSOME( $X_q$ )
(2)   Calculate the set of loop layers ( $\beta_q$ ) contained in Chromosome  $X_q$ 
(3)   for each  $l_j \in \beta_q$  do / * Calculate the set of all direction vectors ( $L_q$ ) covered by Chromosome  $X_q$  */
(4)     Calculate the set of direction vectors ( $D_j$ ) covered by  $l_j$ 
(5)      $L_q = L_q \cup D_j$ 
(6)   end for
(7)    $\gamma_q = D - L_q$  / * Calculate the set of direction vectors ( $\gamma_q$ ) not covered by Chromosome  $X_q$  */
(8)   if  $\gamma_q \neq \emptyset$  then / * chromosome repair */
(9)     for each  $d_i \in \gamma_q$  do
(10)      Calculate the set of loop layers ( $s_i$ ) that cover  $d_i$ 
(11)      Randomly select a loop layer ( $l_j$ ) from  $s_i$  and add it to the set of  $L_q$ 
(12)    end for
(13)  end if
(14)  while  $L_q \neq \emptyset$  do / * delete redundant genes */
(15)     $L_q = L_q - l_j$  ( $l_j \in L_q$ ) / * randomly delete a loop layer ( $l_j$ ) from the set of  $L_q$  */
(16)    for each  $d_i \in D$  do
(17)      Calculate the number of loop layers ( $v_i$ ) in the set of  $L_q$  that cover the direction vector of  $d_i$ 
(18)      if  $v_i < 1$  then
(19)         $L'_q = L'_q \cup l_j$ 
(20)        break;
(21)      end if
(22)    end for
(23)  end while
(24)  Generate Chromosome  $X'_q$  that corresponds to  $L'_q$ 
(25) end function

```

ALGORITHM 2: Chromosome repair.

- (6) Fitness evaluation. Form the candidate set of the offspring population using the repaired offspring chromosomes and parent chromosomes; calculate the fitness of all of the chromosomes in the candidate set; select N_p chromosomes with the best fitness to form the offspring population.
- (7) Check the stopping criterion. If it is not met, then return to step (3); otherwise, stop the algorithm and output the chromosome individual with the best fitness in the current population.
- (8) Generate the loop layer set that corresponds to the chromosome with the best fitness.

Next, we give a detailed description of the specific implementation method of each step.

4.2.1. Chromosome Representation Scheme and Calculation Method of the Fitness Function. To solve the loop selection problem of multilevel nested loops using GAs, we must first determine the representation scheme of the chromosomes. According to the mathematical model of the problem detailed in Section 4, the selection vector ($x_q = [x_{q1}, x_{q2}, \dots, x_{qn}]$) that corresponds to feasible solutions (L_q , the set of the loop layer that covers all of the direction vectors) is a 0-1 matrix, and thus, adopting the n -bit binary numbers to encode feasible solutions is a suitable chromosome representation scheme.

Analogous to the method of representing feasible solutions with the selection vector, n -bit binary coding ($b_{q1}b_{q2} \dots b_{qn}$) is used to represent the chromosomes of the feasible solution L_q , in which n is the number of nested loop layers, and the value of 1 at the j^{th} bit (b_{qj} ($j \in \{1, 2, \dots, n\}$)) represents that the l_j loop layer is in this set of solutions (i.e., $l_j \in L_q$), and the value of 0 at the j^{th} bit of b_{qj} represents that the l_j loop layer is not in this set of solutions (i.e., $l_j \notin L_q$). Figure 10 shows the X_q chromosome that corresponds to the L_q feasible solution.

Analogous to the method of using the x_q selection vector to calculate the size of the corresponding feasible solution set of L_q , Chromosome X_q is represented with a binary (0-1) code ($b_{q1}b_{q2} \dots b_{qn}$). Accordingly, the fitness function of Chromosome X_q can be calculated using the following formula:

$$f_q = \sum_{j=1}^n b_{qj}. \quad (6)$$

Table 3 shows the representation of the chromosome that corresponds to the feasible solutions and the calculation of the fitness values of the example program shown in Figure 2.

4.2.2. Generation of the Initial Population. After the chromosome representation scheme is determined, the generation of the initial population is performed. First, the size of the population is decided. If the size of the initial population is too large, the execution efficiency is low although the optimal solution can be obtained with relatively fast convergence (within a small number of generations); if the size

FIGURE 10: A binary coded chromosome.

of the initial population is too small, then the convergence will be too slow. Therefore, the population size has a large impact on the quality of the solution obtained using GA, and choosing an appropriate population size can improve the algorithm's efficiency.

In GAs, the product of the chromosome length and the population density (ϕ) is usually used as an approximate value of the population size, of which the population density refers to the ratio of the number of dependencies in the direction matrix to the size of the entire matrix [25]. In principle, it is required that chromosomes in an initial population must contain all of the genes (i.e., all of the nested layers). Assuming that the population size is N_p and the average number of genes contained in each chromosome in the population is $1/\phi$, then the total number of genes in the population is N_p/ϕ . To ensure that $\cup_{\alpha=1}^{N_p} L_{\alpha} = L$ (in which L_{α} is the set of nested layers that correspond to any chromosome in the population and L is the set of all nested loop layers), we must have $N_p/\phi \geq \mu n$ ($\alpha \geq 1$, and n is the total number of loops), and thus, we have $N_p = \mu \phi n$. Through repeated experiments, it was found that the determinant value of the direction matrix of the multilevel nested loops is low, and the dependence density varies greatly with different numbers of loops. Therefore, in actual operation, the size of μ is manually set according to the characteristics of the multilevel nested loops, to achieve a good outcome; for example, when there are four loop layers, $\mu = 2$; when there are 15 loop layers, $\mu = 6$.

After the size of the initial population is determined, we must also consider the feasibility of generating new chromosomes in the initial population (i.e., whether the set of loop layers represented by the chromosomes covers all direction vectors). The feasibility of the chromosomes in the initial population has an important impact on the quality of the solution and the convergence rate of the GA. Therefore, we require that the gene set contained in each chromosome in the initial population must cover all of the direction vectors.

By combining the initial population size and the feasibility of the chromosomes, we propose the generation algorithm for the initial population, as shown in Algorithm 1. The rationale of the algorithm is as follows. First, a loop layer is randomly chosen from the loop layer set (s_i ($i = 1, 2, \dots, k$)) of each direction vector (d_i ($i = 1, 2, \dots, k$)) and added to the α_q set in such a way that the α_q set covers all of the direction vectors. Second, the redundant loop layers in the α_q set are deleted to form the L_q set and, ultimately, the L_q that corresponds to Chromosome X_q . The above steps are iterated N_p times, which give rise to the initial population.

The deletion of the redundant loop layers of the α_q set is performed as follows. From the α_q set, a loop layer (l_j) is

TABLE 3: The feasible solutions, chromosome representation, and fitness value calculation of the sample program.

Feasible solution	Chromosome	Fitness calculation
$L_1 = \{l_1, l_2, l_3\}$	$X_1 = b_{11}b_{12}b_{13}b_{14} = 1110$	$f_1 = \sum_{j=1}^4 b_{1j} = 1 + 1 + 1 + 0 = 3$
$L_2 = \{l_1, l_2, l_4\}$	$X_2 = b_{21}b_{22}b_{23}b_{24} = 1101$	$f_2 = \sum_{j=1}^4 b_{2j} = 1 + 1 + 0 + 1 = 3$
$L_3 = \{l_3, l_4\}$	$X_3 = b_{31}b_{32}b_{33}b_{34} = 0011$	$f_3 = \sum_{j=1}^4 b_{3j} = 0 + 0 + 1 + 1 = 2$

randomly deleted, and then, the number of loop layers (v_i ($i = 1, 2, \dots, k$)) in the α_q set that covers the direction vector (d_i ($i = 1, 2, \dots, k$)) is examined. If $v_i < 1$, then the loop layer of l_j is not a redundant layer; then, it is included in the chromosome generation set (L_q); otherwise, the loop layer of l_j is redundant and is removed.

4.2.3. Crossover and Mutation. After the initial population is determined, we will discuss various genetic operations, including parent selection, crossover, and mutation.

- (1) Parent selection. Parent selection is the opportunity that each individual in the population is given to produce offspring, and it can be implemented in many ways, including proportional selection, tournament selection, and roulette selection [26]. In this study, we use the tournament selection method to select the parent individuals in the population. Specifically, two mating pools are generated, each of which consists of T ($T \in N^+$) individuals who are randomly selected from the population, and two individuals with the best fitness in their respective mating pools are chosen to mate.

In this study, we evenly divide the chromosomes in the population into two groups ($\lceil N_p/2 \rceil$ or $\lfloor N_p/2 \rfloor$), and then, we choose the chromosome with the best fitness in each group to mate until one group is empty or both groups are empty. The reason for taking this specific action is that it is relatively easy to select and efficient to implement, without having to calculate the probability of each individual in the population being selected, as in the case of the proportional selection method, which introduces additional computational overhead that affects the overall execution efficiency of the algorithm. In the actual experiments, we also found that the tournament selection method is more efficient than the proportional selection method.

- (2) Crossover operation. Crossover is the main method for introducing new individuals, and single-point crossover or multipoint crossover are often used in GAs. In both crossover operations, the crossover point is first set, and the chromosome segments of two parents are exchanged to produce two new offsprings. In this study, the single-point cross with a fixed crossover point is adopted. Specifically, let $X_1 = b_{11}b_{12}, \dots, b_{1k}b_{1(k+1)}, \dots, b_{1n}$ and $X_2 = b_{21}b_{22}, \dots, b_{2k}b_{2(k+1)}, \dots, b_{2n}$ be two parent chromosomes, and let the set crossover point be k ($k = \lceil n/2 \rceil$); then, the two offspring chromosomes that are generated are

$$\begin{aligned} X_1 &= b_{11}b_{12}, \dots, b_{1k}b_{2(k+1)}, \dots, b_{2n}, \\ X_2 &= b_{21}b_{22}, \dots, b_{2k}b_{1(k+1)}, \dots, b_{1n}. \end{aligned} \quad (7)$$

After setting up the single fixed-point crossover operation, we also must set the crossover rate to control the number of chromosomes in the mating pools that are participating in crossovers. In this study, the chromosomes in two pools are all subjected to the crossover operation. The reason is that the numbers of rows and columns of the direction matrix of the multilevel nested loops at issue in this study are not too high, and the size of the population is, thus, not large; to ensure the efficient convergence of the GA, the structure of the offspring chromosomes should not be vastly different from those of the parent chromosomes, and thus, the single-point fixed crossover method is used. Moreover, the single-point fixed crossover operation can ensure that new individuals are continuously produced in the mating pool, while avoiding the number of new individuals being so large that the genetic order is broken. Therefore, all individuals in the parent selection pools are subjected to the crossover operation.

- (3) Mutation operation. Mutation, as another way to generate new individuals, is an operation on offspring chromosomes that are produced by crossover operations; it is achieved by flipping a certain bit of a certain offspring chromosome (i.e., flipping 0 to 1 or flipping 1 to 0) at a low probability to produce new individuals in the population. It is usually regarded as a process in which genes on chromosomes mutate to generate genes that do not exist in the initial population, or they reintroduce genetic information lost due to crossover operations. The real meaning of mutation is to have a defense mechanism to prevent the population from entering convergence prematurely and failing to find a better solution [27].

The mutation rate is the ratio of the number of mutated genes to the total number of genes in a population. When applying a GA to solve various problems, the mutation rate of the selected operation is an important factor that affects the efficiency of the algorithm. A reasonable mutation rate allows the offspring to obtain new genes without losing the good traits inherited from both parents. According to an early study by Back, the mutation rate of a population cannot be lower than $1/n$ (n is the chromosome length); otherwise, it will affect the efficiency of the GA in finding the optimal solution [28]. Thus, in this study, we randomly choose $d \lceil N_p/n \rceil$ chromosomes from the offspring chromosomes produced through crossover to perform the mutation

operation at the “1” bit to generate new chromosomes and then determine the feasibility of the newly generated offspring chromosomes and repair infeasible chromosomes (the repair algorithm will be detailed in the next section), which are then added to the next generation of the candidate population. Figure 11 illustrates the mutation operation of two chromosomes.

4.2.4. Chromosome Improvement. Although Algorithm 1 ensures the feasibility of the chromosomes in the initial population, crossover and mutation will inevitably produce infeasible chromosomes. There can be two issues with the newly generated chromosomes. First, new chromosomes can cover all direction vectors but could have redundant genes, which must be deleted. Second, the newly generated chromosomes cannot cover all direction vectors. There are two ways to address these issues: abandon and repair. If the chromosomes are abandoned, then the optimal solution could also be lost, thereby slowing down the convergence speed. For this reason, we adopt the repair method to address the above-described chromosome issues. The chromosome repair algorithm is detailed in Algorithm 2.

Using Algorithm 2, we can repair infeasible chromosomes that are generated through crossover and mutation, to guarantee the feasibility of chromosomes in the population while removing redundant genes, to further enhance the efficiency of the algorithm.

4.2.5. Offspring Population Generation and Stopping Criterion

- (1) Method for generating new offspring population. Two methods are commonly used in producing offspring populations: the steady-state replacement method and the whole generation replacement method. In the steady-state replacement method, once a new generation of chromosomes is produced, the newly produced chromosomes and the chromosomes in the parent population are combined to form the candidate population, from which chromosomes with higher fitness are selected to form the new offspring population. In the whole generation replacement method, the newly generated offspring chromosomes directly replace the original parent chromosomes [29]. In this study, we adopt the steady-state replacement method because this method can not only use the newly generated offspring chromosomes for various operations (e.g., selection and crossover) but also ensure that the chromosome with the highest fitness will always be retained in the population. Hence, the steady-state replacement method converges faster with a higher efficiency in finding the optimal solution than the whole generation replacement method, an advantage that was also confirmed in our experiment.
- (2) Stopping criterion. In practice, GA needs to perform evolution over multiple generations until the fitness of the chromosomes in the population is stabilized,

FIGURE 11: Mutation operation of chromosomes.

or the optimal solution is found, or the preset maximum number of generations is reached; at any of these points, the evolutionary process ends. Because GAs do not include information such as the gradient of the objective function, it is impossible to determine the location of an individual in the solution space in the evolutionary process and a theoretical criterion of convergence [30]. A commonly used method is to stop the evolution when the predetermined number of evolutionary generations is reached. Given that the numbers of the rows and columns of the direction matrix of the multilevel nested loops are rather low, to reduce the execution time of the actual operation, the maximum number of generations was set to 100 after repeated experiments on operations for the search for optimal solution for nested loops of 20 or fewer layers. In the actual operation process, the parameters are set to determine whether the chromosomes have changed between two consecutive generations of the population, based on which the convergence of the algorithm is assessed and the iterative solution process can be ended early when the convergence is reached.

5. Experimental Evaluation

The GA proposed in this study to solve the loop selection problem of multilevel nested loops was integrated in the test version of the basic compiler of Sunway TaihuLight supercomputer, and it passed the correctness test of the multithreaded automatic parallel optimization process.

5.1. Test Platform and Content

5.1.1. Test Platform. The test platform was the Sunway TaihuLight supercomputer with the SW26010 heterogeneous many-core processor (with a total of 40,960 processors), with a peak system performance of 125.436 PFlops, under the operating system of Raise Linux, which was integrated with the basic compiler for the proposed method and equipped with complete compiler tool chain and performance analysis tools. The tests of this study were performed on one computing node of the Sunway TaihuLight supercomputer, with the following parameters:

CPU: 4* (1 master core + 64 slave cores)
Main frequency: 1.25 GHz
Memory: 8 G

L1 Cache: 32 KB private
L2 Cache: 256 KB private
Vector unit: 256 b

5.1.2. Test Content. We chose ten scientific computing programs of the SPEC CPU2006 benchmark test set and the NPB3.3.1 parallel computation test set as the test benchmark. Tables 4 and 5 show the brief description of the SPEC CPU2006 and NPB3.3.1 component suites. The input scales of the SPEC CPU2006 benchmark program test set are categorized, from small to large, into three scales, i.e., test, train, and reference. The reference scale, which is the largest, was adopted in this study. The input scales of the NPB3.3.1 parallel computing test set are categorized, from small to large, into six levels, i.e., S, W, A, B, C, and D. The B level was adopted in this study. Because the proposed method was integrated in the basic compiler of the Sunway TaihuLight supercomputer, we chose the single-node four-thread automatic parallelization performance improvement test.

To test the effectiveness of the proposed method, we conducted two sets of experiments on the Sunway TaihuLight supercomputer. One set is the comparative performance test of the basic compiler using the proposed method and the basic compiler using the loop selection method in combination with the underlying hardware techniques (the thread-level speculation technique and the TM technique), and the other set is the comparative performance test of the basic compiler using the proposed method and the basic compiler using the loop selection methods (i.e., those combining the compile time and runtime and those based on machine learning), which were mostly adopted in previous studies.

5.2. Comparison between the Proposed Method and the Underlying Hardware Techniques. On the Sunway TaihuLight platform, we conducted automatic parallelization performance tests of (1) the basic compiler (as a benchmark for the performance test), (2) the basic compiler in combination with the loop selection method based on the TM technique, (3) the basic compiler in combination with the loop selection method based on the thread-level speculation technique, and (4) the basic compiler using the proposed method, respectively. The test results are shown in Figures 12 and 13.

Figure 12 shows the automatic parallelization performance test results of the ten scientific computing programs chosen from the SPEC CPU2006 test set. All of the programs contain many multilevel nested loops. Examination of the printed intermediate representation files generated by the automatic parallelization process during compilation indicated that the conventional compiler technique cannot accurately analyze the complex dependencies in the nested loops, thus showing poor parallel acceleration performance. The dependency analysis ability of the TM and TLS techniques relies on the traditional compiler technique, and as a result, their parallel acceleration performance is basically identical to that of the traditional compiler technique. Because the proposed method can provide the parallelization

information of multiple loop layers after loop exchange, it enables the parallelization with multigranularity in the compilation process and achieves a remarkable speedup improvement. For example, using the proposed method, the 401.bzip2 and 433.milc compilers not only successfully parallelized the outer loops of the multilevel nested loops, which could not be parallelized before but also provided the automatic vectorization information of the inner loops. The acceleration performance of several other scientific computing programs was not as pronounced because the multilevel nested loops in the source code are not a hotspot of the program and only take up very little execution time of the entire program.

Figure 13 shows the automatic parallelization test results on the NPB3.3.1 test set. The proposed method showed significant acceleration improvement on five programs, i.e., BT, FT, LU, MG, and SP. The reason is that these programs contain many multilevel nested loops, some of which cannot be parallelized through the basic compiler but can be successfully parallelized by the proposed method, thus achieving a notable acceleration effect. In comparison, the acceleration effect of the automatic parallelization of five other programs, i.e., CG, DC, EP, IS, and LU-HP, was insignificant. The reason is that their loop structure is either irregular or based on operations with a complex pointer structure or contains an indirect array access mode, which is beyond the scope of this study.

5.3. Comparison of the Proposed Method with Existing Techniques. We selected two loop selection methods, Method A and Method B, which have been mostly investigated. Method A uses a loop selection method that combines the compile time and the runtime [31], and Method B uses a loop selection method based on machine learning [32]. The automatic parallelization performance of the basic compiler was again used as the benchmark of the test. Then, the ten scientific computing programs selected from the SPEC2006 test set and the NPB3.3.1 parallel computing test set were tested, with the results shown in Figures 14 and 15, respectively.

Figure 14 shows the automatic parallelization performance test results of the proposed method and Methods A and B. Method A greedily selected multiple different nested layers of the same loop as candidate speculation objects through the compiler and determined the nested loop layer with the best parallel performance based on the execution delay and the number of cancellations as dynamically monitored at runtime. Through the target program hotspot analysis tool and the printed automatic parallelization optimization result information during the compilation process, we found that Method A has only a limited capability in speculating nested loop layers and executing the model in parallel; as a result, some nested loop layers that could be executed in parallel did not give parallelization prompt information. Moreover, the scope of the runtime performance analysis cost model of Method A is fuzzy and, as a result, some of the nested loop layers that could be executed in parallel to obtain a limited benefit were not executed in

TABLE 4: Component of SPEC CPU2006.

Benchmark	Brief description
401.bzip2	Julian Seward's bzip2 version 1.0.3, modified to do most work in memory, rather than doing I/O
433.milc	A gauge field generating program for lattice gauge theory programs with dynamical quarks
434.zeusmp	ZEUS-MP is a computational fluid dynamics code developed at the Laboratory for Computational Astrophysics (NCSA, University of Illinois at Urbana-Champaign) for the simulation of astrophysical phenomena
437.leslie3d	Computational fluid dynamics (CFD) using large-eddy simulations with linear-eddy model in 3D; uses the MacCormack predictor-corrector time integration scheme
445.gobmk	Plays the game of go, a simply described but deeply complex game
453.povray	Image rendering; the test case is 1280×1024 antialiased image of a landscape with some abstract objects with textures using a Perlin noise function
456.hmmer	Protein sequence analysis using profile hidden Markov models (profile HMMs)
458.sjeng	A highly-ranked chess program that also plays several chess variants
464.h264ref	A reference implementation of H.264/AVC encodes a videostream using 2 parameter sets; the H.264/AVC standard is expected to replace MPEG2
481.wrf	Weather modeling from scales of meters to thousands of kilometers; the test case is from a 30 km area over 2 days

TABLE 5: Component of NPB3.3.1.

Benchmark	Brief description
BT	Block tridiagonal solver
CG	Conjugate gradient, irregular memory access, and communication
DC	Data cube
EP	Embarrassingly parallel
FT	Discrete 3D fast Fourier transform, all-to-all communication
IS	Integer sort, random memory access
LU	Lower-upper Gauss-Seidel solver
LU	Lower-upper Gauss-Seidel solver
MG	Multigrid on a sequence of meshes, long- and short-distance communication, memory intensive
SP	Scalar pentadiagonal solver

FIGURE 12: Performance tests of the proposed method and the underlying hardware techniques on the SPEC2006 test set.

FIGURE 13: Performance tests of the proposed method and the method based on the underlying hardware techniques on the NPB3.3.1 test set.

FIGURE 14: Performance tests of the proposed method and Methods A and B on the SPEC2006 test set.

parallel at runtime. Therefore, the number of parallelized multilevel nested loops obtained using Method A was lower than that using the proposed method.

Method B uses the K-nearest neighbor (KNN) algorithm in machine learning to focus on analyzing the thread granularity, data dependency distance, thread creation, and

the performance impact caused by excitation to determine the optimal loop nesting layer for parallel execution. We constructed a machine learning-based loop selection model using Method B, based on which compilation prompts were given in the compilation process. However, through experiments, we found that the nested loop layer selected by

FIGURE 15: Performance tests of the proposed method and Methods A and B on the NPB3.3.1 test set.

Method B was not necessarily the loop layer with the optimal benefit and thus did not achieve an ideal speedup improvement effect. We also found that the multilevel nested loop parallelization benefit is associated not only with characteristics such as the loop dependency distance and thread granularity of Method B but also with the software environment and hardware parameters of the actual machine; for example, the thread synchronization method, the division of task granularity, and the memory size of the operating system of the software environment affect the actual execution performance of the loop.

Based on the direction matrix constructed through dependency relationships, the proposed method successfully exposed more parallelism while ensuring the safe parallel execution of multilevel loops in multilevel nested loops. In the experiment, we found that the compiler that used the proposed method prompted not only the coarse-grained thread-level parallelization at the outer loop layers of the multilevel nested loops but also the fine-grained instruction-level parallelization of the inner loops, showing multi-granularity parallelization; for example, through the proposed method, the compiler guaranteed the correctness of the parallel execution and performed automatic vectorization and loop unrolling on inner-level loops to further gain the instruction-level parallelism while obtaining the thread-level coarse-grained parallelism in the outer-level loops. This property is also the reason that the proposed method is superior to Methods A and B in terms of automatic parallelization performance improvement.

Figure 15 shows that the acceleration effect of Methods A and B on the CG, DC, EP, IS, and LU-HP programs were superior to that of the proposed method on the NPB3.3.1 test set. The reason is that these five programs do not have the complex dependencies between multilevel nested loop

iterations, while the effect of parallelization mainly depends on the performance evaluation of the parallelization benefit of the loop layers. Method A combines the loop-layer parallelization benefit evaluation at compile time and runtime, and Method B, through the machine learning method, combines the information from a large number of nested loop layers to heuristically select beneficial loop layers for parallelization, while the proposed method was based only on the static analysis of the compiler's parallelization performance benefits for the loop layers. Thus, the acceleration effect of the proposed method was lower than that of Methods A and B. Moreover, in the experiment, when examining the assembly code that corresponds to the multilevel nested loop segment, we surprisingly found that, after ensuring the correctness of the dependency, the proposed method obtained a much larger instruction scheduling space in the subsequent optimization process by using techniques such as loop unrolling and register renaming while eliminating the instruction conflicts caused by interinstruction dependencies, thereby further improving the parallelism between the instructions. Therefore, the application of the proposed method to Methods A and B is expected to achieve better results.

6. Related Work

The authors in [1, 4, 33–37] obtained the profiling information that reflects the behavior of the program through multiple preexecution programs using the thread-level speculation technique in combination with the information on the program during preexecution, and then, the appropriate loop layer was selected for parallel execution based on the predicted performance. However, the dependence analysis ability of the hardware was limited, and thus, the

success rate of these techniques in stimulating multithreaded parallel execution in multilevel nested loops was not high [1, 4, 33–37]. In [31, 38–41], a multilevel nested loop thread-level parallelization scheme that combines static analysis and dynamic scheduling was adopted. Most of these methods used traditional compilation techniques to assess the benefits of the loops to obtain information on parallelization and made corresponding dynamic adjustments on various program behaviors caused by different platforms and different input data in actual operations, which is also the main direction of the current studies. However, reliance on only traditional compilation techniques cannot give accurate loop-level parallelization prompts for multilevel nested loops. Therefore, the methods described in these references lack generality [31, 38–41]. As reported in [32, 42–44] the relevant features, such as the number of loop iterations of multilevel nested loops and the number of nested layers, were extracted from the intermediate representation of the compiler to construct the loop selection assessment model of multilevel nested loops. Compared with the models based on the thread-level speculation technique, these models improve the parallelization effect of multilevel nested loops to some extent. However, as the number of nested loop levels of the program increases, the iteration dependency between loops becomes complicated, which makes them still unable to achieve the desired performance improvement [32, 42–44]. The authors in [45–47] proposed frameworks for misspeculation based on the loop cost in the compiler. Based on the control flow graph and data dependency graph in the compiler, these frameworks analyze the probability of parallel execution failure of loop layers in multilevel nested loops to further speculate on the cost of the parallel execution of the loop layers, based on which the multilevel nested loops are radically changed into loops executed in parallel [45–47]. These frameworks speculate that the accuracy of the parallel execution of the loops depends on the compiler's limited ability to analyze the dependencies between the loop iterations, and thus, they have little effect on the benefit of loop layer parallelization of multilevel nested loops. In this study, we proposed a solution for the parallel static compilation of multilevel nested loops using GAs, to employ the least number of loop layers to cover all loop-carried dependencies, thereby obtaining the highest parallel granularity. This method can be better combined with other methods and thus has a wider range of applications.

7. Discussion

Experimental results indicated that the proposed method can effectively solve the loop selection problem for multilevel nested loops and find a set of fewer loop layers that cover all direction vectors in a limited time to expose the parallelism of multilevel nested loops, to obtain a higher parallel granularity, and to further improve the performance of the entire program. In this study, the number of nested loop layers was used to represent the chromosome length, which is short in this case and proportional to the population size, and the preset stopping criterion ensured sufficient reproduction generations, while the settings of the parameters,

such as the chromosome crossover rate and mutation rate, prevented the population from falling into locally optimal solutions. Therefore, the optimal solution to the loop selection problem of the multilevel nested loops was successfully found by the proposed method multiple times.

Compared with the method that combines static compilation benefit analysis and a dynamic operating loop scheduling scheme [31], the proposed method can obtain a higher parallelism granularity. Compared with the multilevel nested loop parallelization model proposed by Liu et al. [32] based on the KNN algorithm of machine learning, the proposed method is simpler and more intuitive, which makes it easier to integrate with mainstream compilers. Given that Dice et al. [9] considered the actual running state of the multilevel nested loops to be closely related to the underlying hardware and that Li et al. [2] extracted the relevant parameters of multilevel nested loops from the intermediate representation of the Prophet compiler, which relies on the organizational structure of the specific compiler, the proposed method starts from the direction matrix of multilevel nested loops and thus has a higher applicability in mainstream compilers.

The proposed method can not only solve the loop selection problem of multilevel nested loops but also assist the compiler in finding the maximum parallel granularity in multilevel nested loops in a short time, thereby improving the overall performance of the program. This method also provides a new idea for solving the parallelism of multilevel nested loops with complex dependencies as well as a new method for mainstream compilers to explore the parallelism of multilevel nested loops.

8. Conclusions

Due to the complexity of the dependency analysis of the multilevel nested loops and the limitations of the static analysis capability of basic compilers, the loop selection problem of multilevel nested loops remains a very difficult problem. With the advent of the era of artificial intelligence, the multilevel nested loops that are present in a large number of machine learning algorithms seriously affect the overall performance of the programs. Therefore, solving the loop selection problem of multilevel nested loops and finding the thread-level parallelism with the maximum granularity in the multilevel nested loops are of great practical significance for making full use of multicore processors and even many-core processors, thus improving the overall performance of a program.

Data Availability

The SPEC2006 and NPB3.3 data used to support the findings of this study have been deposited in the SPEC and NPB website (<http://www.spec.org/benchmarks.html#cpu> and <https://www.nas.nasa.gov/publications/npb.html>).

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was supported by the National Key Research and Development Program “High-Performance Computing” Key Special (Grant no. 2016YFB0200503).

References

- [1] B. Liu, H. Yang, L. Yuancheng, L. Yuxiang, N. Dangdang, and L. Zhiming, “An improved programming model for thread-level speculation,” in *Proceedings-2019 IEEE International Conference on Parallel and Distributed Processing with Applications, Big Data and Cloud Computing, Sustainable Computing and Communications, Social Computing and Networking*, pp. 666–672, IEEE, Xiamen, China, December 2019.
- [2] Y. Li, Y. Zhao, L. Sun, and M. Shen, “A hybrid sample generation approach in speculative multithreading,” *The Journal of Supercomputing*, vol. 75, no. 8, pp. 4193–4225, 2019.
- [3] D. Matsunaga, A. Nunome, and H. Hirata, “Shelving a code block for thread-level speculation,” in *Proceedings - 20th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, pp. 427–434, IEEE, Toyama, Japan, July 2019.
- [4] Y. Li, D. Niu, Z. Zhang, and L. Zhang, “Tpaopia: a thread partitioning approach based on procedure importance in speculative multithreading,” in *Proceedings - 21st IEEE International Conference on High Performance Computing and Communications, 17th IEEE International Conference on Smart City and 5th IEEE International Conference on Data Science and Systems*, pp. 2339–2344, IEEE, Zhangjiajie, China, August 2019.
- [5] J. Salamanca and A. Baldassin, “A proposal for supporting speculation in the openmp taskloop construct,” in *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, LNCS, pp. 246–261, Springer, Berlin, Germany, 2019.
- [6] Z. Wang, H. Wang, and J. Li, “A speculative parallel optimization method for industrial big data algorithms,” in *Proceedings - IEEE International Conference on Industrial Internet Cloud, ICII 2019*, pp. 417–422, IEEE, Orlando, FL, USA, November 2019.
- [7] T. A. Tomeu, A. Gamez, and A. G. Salguero, “A parallel implementation for cellular potts model with software transactional memory,” in *Proceedings of the Advances in Intelligent Systems and Computing*, pp. 53–60, IEEE, L’Aquila, Spain, June 2020.
- [8] Q. Shen, C. Sharp, R. Davison et al., “A general purpose contention manager for software transactions on the gpu,” *Journal of Parallel and Distributed Computing*, vol. 139, pp. 1–17, 2020.
- [9] D. Dice, M. Herlihy, and A. Kogan, “Improving parallelism in hardware transactional memory,” *ACM Transactions on Architecture and Code Optimization*, vol. 15, no. 1, pp. 1–24, 2018.
- [10] C. Moumita, M. Anirban, and R. Sudipta, “Gossip based fault tolerant protocol in distributed transactional memory using quorum based replication system,” *Cluster Computing*, vol. 23, no. 2, pp. 1103–1124, 2020.
- [11] I. Shady, F. Pascal, and M. Alexander, “Extending hardware transactional memory capacity via rollback-only transactions and suspend/resume: power8 tm,” *Distributed Computing*, vol. 33, no. 3–4, pp. 327–348, 2020.
- [12] L. Xiuhong and G. Altenbek, “Optimised memory allocation for less false abortion and better performance in hardware transactional memory,” *International Journal of Parallel, Emergent and Distributed Systems*, vol. 35, no. 4, pp. 483–491, 2020.
- [13] V. Packirisamy, A. Zhai, W.-C. Hsu, P.-C. Yew, and T.-F. Ngai, “Exploring speculative parallelism in spec2006,” in *Proceedings of the ISPASS 2009-International Symposium on Performance Analysis of Systems and Software*, pp. 77–88, IEEE, Boston, MA, USA, April 2009.
- [14] R. Allen and K. Kennedy, *Optimizing Compilers for Modern Architectures*, Morgan Kaufmann, Burlington, MA, USA, 2001.
- [15] S. Wang, X. Dai, K. S. Yellajoyosula, A. Zhai, and P.-C. Yew, “Loop selection for thread-level speculation,” in *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, pp. 289–303, LNCS, Hawthorne, NY, United states, 2006.
- [16] C. Jianli, C. Zhikui, W. Yuxin, and G. He, “Parallel genetic algorithm for n-queens problem based on message passing interface-compute unified device architecture,” *Computational Intelligence*, vol. 36, no. 4, pp. 1–17, 2020.
- [17] Y. Sun, J. Li, X. Fu, H. Wang, and H. Li, “Application research based on improved genetic algorithm in cloud task scheduling,” *Journal of Intelligent & Fuzzy Systems*, vol. 38, no. 1, pp. 239–246, 2020.
- [18] S. Saroja and T. Revathi, “Hybrid dual-objective parallel genetic algorithm for heterogeneous multiprocessor scheduling,” *Cluster Computing*, vol. 23, no. 2, pp. 441–450, 2020.
- [19] M. Abbasi, M. Rafiee, M. R. Khosravi, A. Jolfaei, V. G. Menon, and J. M. Koushyar, “An efficient parallel genetic algorithm solution for vehicle routing problem in cloud implementation of the intelligent transportation systems,” *Journal of Cloud Computing*, vol. 9, no. 1, 2020.
- [20] N. Kantour and S. Bouroubi, “Cryptanalysis of merkle-hellman cipher using parallel genetic algorithm,” *Mobile Networks and Applications*, vol. 25, no. 1, pp. 211–222, 2020.
- [21] D. I. Arkhipov, D. Wu, T. Wu, and A. C. Regan, “A parallel genetic algorithm framework for transportation planning and logistics management,” *IEEE Access*, vol. 8, pp. 506–515, 2020.
- [22] M. K. Fallah, M. Mirhosseini, M. Fazlali, and M. Daneshtalab, “Scalable parallel genetic algorithm for solving large integer linear programming models derived from behavioral synthesis,” in *Proceedings-2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, pp. 390–394, IEEE, Vasterås, Sweden, March 2020.
- [23] J. Luo, D. El Baz, R. Xue, and J. Hu, “Solving the dynamic energy aware job shop scheduling problem with the heterogeneous parallel genetic algorithm,” *Future Generation Computer Systems*, vol. 108, pp. 119–134, 2020.
- [24] V. Jamshidi, V. Nekoukar, and M. H. Refan, “Analysis of asynchronous distributed multi-master parallel genetic algorithm optimization on can bus,” *Evolving Systems*, vol. 11, no. 4, pp. 673–682, 2020.
- [25] —, “Analysis of parallel genetic algorithm and parallel particle swarm optimization algorithm uav path planning on controller area network,” *Journal of Control, Automation and Electrical Systems*, vol. 31, no. 1, pp. 129–140, 2020.
- [26] V. Muttillio, P. Giammatteo, G. Fiorilli, and L. Pomante, “An openmp parallel genetic algorithm for design space exploration of heterogeneous multi-processor embedded systems,” in *Proceedings of the ACM International Conference Proceeding Series*, pp. 1–6, IEEE, Bologna, Italy, Januray 2020.

- [27] M. Ayani, L. MacGregor, and S. Mallick, "Inversion of marine controlled source electromagnetic data using a parallel non-dominated sorting genetic algorithm," *Geophysical Journal International*, vol. 220, no. 2, pp. 1066–1077, 2020.
- [28] T. Bäck, "Optimal mutation rates in genetic search," in *Proceedings of the 5th International Conference on Genetic Algorithms*, IEEE, Seattle, WA, USA, July 1993.
- [29] B. Kurniawan, W. Chandramitasari, A. A. Gozali, W. Weng, and S. Fujimura, "Triple-chromosome genetic algorithm for unrelated parallel machine scheduling under time-of-use tariffs," *IEEE Transactions on Electrical and Electronic Engineering*, vol. 15, no. 2, pp. 208–217, 2020.
- [30] Z. Wu, C. Zhao, and B. Liu, "Polygonal approximation based on coarsegrained parallel genetic algorithm," *Journal of Visual Communication and Image Representation*, vol. 71, 2020.
- [31] M.-R. Li, Y.-L. Zhao, Y. Tao, and Q.-M. Wang, "A static greedy and dynamic adaptive thread spawning approach for loop-level parallelism," *Journal of Computer Science and Technology*, vol. 29, no. 6, pp. 962–975, 2014.
- [32] B. Liu, J. He, Y. Geng, L. Huang, and S. Li, "Toward emotion-aware computing: a loop selection approach based on machine learning for speculative multithreading," *IEEE Access*, vol. 5, pp. 3675–3686, 2017.
- [33] R. P. Srivastava and G. Nandi, "Controlling multi thread execution using single thread event loop," in *Proceedings of the International Conference on Innovations in Control, Communication and Information Systems, ICICCI 2017*, pp. 1–7, IEEE, Greater Noida, India, August 2018.
- [34] T. Rauber and G. Runger, "How do loop transformations affect the energy consumption of multi-threaded runge-kutta methods?" in *Proceedings - 26th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pp. 499–507, IEEE, Cambridge, UK, March 2018.
- [35] X. Jin, Y. Zhou, B. Huang et al., "Qosmt: supporting precise performance control for simultaneous multithreading architecture," in *Proceedings of the International Conference on Supercomputing*, pp. 206–216, IEEE, Phoenix, AZ, USA, June 2019.
- [36] Y. Cui, S. Liu, N. Zou, and W. Wu, "A dynamic parallel strategy for doacross loops," in *Proceedings of the ACM International Conference Proceeding Series*, pp. 108–115, Tokyo, Japan, October 2018.
- [37] K. Jingu, K. Shigenobu, K. Ootsu, T. Ohkawa, and T. Yokota, "Directive based parallelization of for-loops at llvm ir level," in *Proceedings - 20th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, pp. 421–426, IEEE, Toyama, Japan, July 2019.
- [38] J. Jiang, Z. Wang, X. Liu et al., "Boyi: a systematic framework for automatically deciding the right execution model of opencl applications on fpgas," in *Proceedings of the FPGA 2020-2020 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pp. 299–309, IEEE, Seaside, CA, USA, February 2020.
- [39] B. Otero, O. Rojas, F. Moya, and J. Castillo, "Alternating direction implicit time integrations for finite difference acoustic wave propagation: parallelization and convergence," *Computers and Fluids*, vol. 205, 2020.
- [40] R. A. Caulk, E. Catalano, and B. Chareyre, "Accelerating yade's poromechanical coupling with matrix factorization reuse, parallel task management, and GPU computing," *Computer Physics Communications*, vol. 248, 2020.
- [41] R. Lim, Y. Lee, R. Kim, J. Choi, and M. Lee, "Auto-tuning gemm kernels on the intel knl and intel skylake-sp processors," *The Journal of Supercomputing*, vol. 75, no. 12, pp. 7895–7908, 2019.
- [42] B. Bylina and J. Bylina, "The parallel tiled WZ factorization algorithm for multicore architectures," *International Journal of Applied Mathematics and Computer Science*, vol. 29, no. 2, pp. 407–419, 2019.
- [43] F. Ghanim, U. Vishkin, and R. Barua, "Easy PRAM-based high-performance parallel programming with ICE," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 2, pp. 377–390, 2018.
- [44] I. Hadade, F. Wang, M. Carnevale, and L. di Mare, "Some useful optimisations for unstructured computational fluid dynamics codes on multicore and manycore architectures," *Computer Physics Communications*, vol. 235, pp. 305–323, 2019.
- [45] Z.-H. Du, C.-C. Lim, X.-F. Li, C. Yang, Q. Zhao, and T.-F. Ngai, "A cost-driven compilation framework for speculative parallelization of sequential programs," *ACM SIGPLAN Notices*, vol. 39, no. 6, pp. 71–81, 2004.
- [46] A. W. M. Checkaraou, A. Rousset, X. Besseron, S. Varrette, and B. Peters, "Hybrid MPI + OPENMP implementation of extended discrete element method," in *Proceedings-2018 30th International Symposium on Computer Architecture and High Performance Computing*, pp. 450–457, IEEE, Lyon, France, September 2018.
- [47] M. Hague and A. W. Lin, "Decidable models of integer-manipulating programs with recursive parallelism," *Theoretical Computer Science*, vol. 750, pp. 24–37, 2018.