ELSEVIER

The Journal of Systems and Software 56 (2001) 279–297

The Journal of Systems and Software

www.elsevier.com/locate/jss

ODCHP: a new effective mechanism to maximize parallelism of nested loops with non-uniform dependences

Der-Lin Pean, Cheng Chen *

Department of Computer Science and Information Engineering, National Chiao Tung University, 1001 Ta Hsueh Road, Hsinchu, Taiwan 30050, People's Republic of China

Received 9 December 1999; received in revised form 12 April 2000; accepted 25 April 2000

Abstract

There are many methods for nested loop partitioning. However, most of them perform poorly when partitioning loops with non-uniform dependences. This paper proposes a generalized and optimized loop partitioning mechanism to exploit parallelism from nested loops with non-uniform dependences. Our approach, based on dependence convex theory, will divide the loop into variable size partitions. Furthermore, the proposed algorithm partitions a nested loop by using the copy-renaming and the optimized partitioning techniques to minimize the number of parallel regions of the iteration space. Consequently, it outperforms the previous partitioning mechanisms of nested loops with non-uniform dependences. Many optimization techniques are used to reduce the complexity of the algorithm. Compared with other popular techniques, our scheme shows a dramatic improvement in the preliminary performance results. © 2001 Elsevier Science Inc. All rights reserved.

Keywords: Compilers; Non-uniform dependence; Loop parallelization; Parallel compiler; Parallel processing; Dependence convex hull

1. Introduction

The dependence of loops can be classified into two categories: uniform dependences and non-uniform dependences (Banerjee, 1988). A pattern of dependence vectors (namely, distance vectors), which are expressed by constants, will be known as uniform dependence vectors. Other dependence vectors in a regular pattern cannot be expressed by constants and belong to non-uniform dependences. Example 1 (a) below explicates a non-uniform dependence loop, which has non-uniform dependences in the iteration space (see Fig. 1(a)). Example 1 (b) below shows an uniform dependence loop, which has uniform dependences in the iteration space (see Fig. 1(b)).

Example 1. (a) A non-uniform dependence loop. (b) An uniform dependence loop.

(a)

```
forI = 1, 10
  for J = 1, 10
    A(2 * I + 3, J + 1) = ...
    ... = A(2 * J + I + 1, I + J + 3)
  endfor
endfor
```

(b)

```
for I = 1, 10
  for J = 1, 10
    A(I, J) = ...
... = A(I + 1, J) + A(I − 1, J) + A(I, J + 1) + A(I, J − 1)
  endfor
endfor
```

Because there is a rich parallelism of loops in scientific programs, current parallelizing compilers have been written to exploit the parallelism (Banerjee, 1988). However, most of them fail in parallelizing nested loops with non-uniform dependences because of irregular and complex dependence constraints. Although there have been many analyses for identifying the cross-iteration dependences in nested loops, most of them fail when analyzing with coupled subscripts (Berger, 1987).

* Corresponding author. Tel.: +86-35712121 ext 54734; fax: +86-35724176.
E-mail addresses: dlpean@csie.nctu.edu.tw (D.-L. Pean), cchen@csie.nctu.edu.tw (C. Chen).

0164-1212/01/$ - see front matter © 2001 Elsevier Science Inc. All rights reserved.
PII: S 0 1 6 4 - 1 2 1 2 (0 0) 0 0 1 0 3 - 5

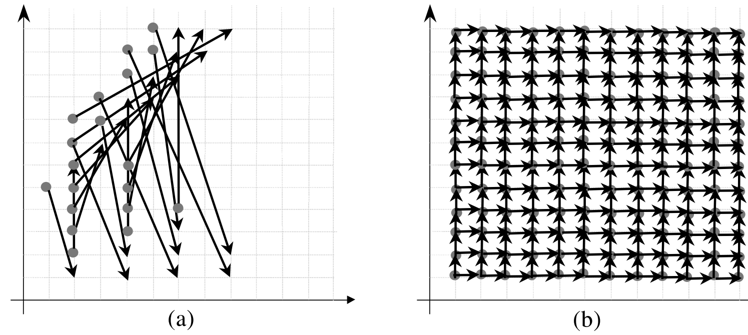

Fig. 1. Iteration space of Example 1: (a) A non-uniform dependence loop, (b) An uniform dependence loop.

According to an empirical study on array subscripts and data dependences (Shen et al., 1989), nearly 45% of two-dimensional array references are coupled, and most of them generate non-uniform dependences. In this paper, we emphasize the parallelization of nested loops with non-uniform dependences. Loop partitioning is an important optimization issue and requires exact and effective data dependence analysis (Zima and Barbara, 1990). However, irregularity in the dependence pattern makes dependence analysis very difficult for nested loops. A number of techniques based on convex hull theory have been proposed, such as dependence uniformization (Tzen and Ni, 1993; Shang et al., 1996), minimum dependence distance tiling (Punyamurtula and Chaudhary, 1994; Tseng et al., 1992; Chen and Yew, 1996; Tseng et al., 1996; Punyanurtula et al., 1997) three-region partition (Zaafrani and Ito, 1994), unique set oriented partitioning (Ju and Chaudhary, 1996), and improved three-region partitioning (ITRP) (Cho and Lee, 1997). However, all of these do not extract total parallelism from non-uniform dependence loops.

We employ a mechanism called optimized dependence convex hull partitioning (ODCHP), which divides the iteration space into many and variable sized parallel regions. The ODCHP mechanism can be combined with other non-uniform parallelization techniques such as three-region partitioning, ITRP, and unique set oriented partitioning mechanisms to exploit as much parallelism as possible. Our approach provides more accurate information about the iteration space and finds more parallelism. It is based on Convex Hull theory in order to reduce the complexity of our algorithm. Because dependence must appear in the dependence convex hulls, the complexity of searching algorithm for dependences must be bounded by the size of dependence convex hulls. On the other hand, some dependences are eliminated results from the execution of previous partitions. This property makes the partition of our method effective. Finally, the iterations that must be executed in lexicographical execution order result in the reduction of the checking procedure for dependences. Combining with integer programming technique, our mechanism is effective in partitioning iteration apace with non-uniform dependences.

We evaluate our scheme in the real multiprocessor system CONVEX SPP-1000 with eight processors. Our scheme performs much better than other famous mechanisms such as uniformization, minimum dependence distance tiling, ITRP, and ITRP combined with minimum dependence distance tiling mechanisms as the loop bounds increase from 10 to 100. However, as the loop bounds increase to 1000, our approach performs slightly better than other mechanisms due to large parallelism exploited and small number of available processors. In order to make effective use of large parallelism exploited, we construct a multiprocessor system environment called SEESMA (Su et al., 1996) that likes CONVEX SPP series machines to show the performance of our mechanism. Mean while, we also implement famous different mechanisms in four popular models and two real program kernel code segments to see the effectiveness of our scheme. In our SEESMA system with 128 processors, our scheme also performs better than other mechanisms. Thus, our mechanism is superior than any other existed method.

The rest of this paper is organized as follows. Section 2 describes our program model and reviews the concept of dependence convex hull, unique head and tail sets, and several related work. Section 3 presents the concept and principle of our new partitioning mechanism, the ODCHP. The preliminary performance results are illustrated in Section 4. Finally, in Section 5 we give some conclusions with suggestions for future work.

2. Preliminaries and related work

Most loops with complex array subscripts are two-dimensional loops (Shen et al., 1989). For a simplification of the explanation, the program under consideration in this paper is a doubly nested loop with coupled subscripts. The

```
for I=L_I, U_I
  for J=L_J, U_J
        .........
    S_d:  A(f_1(I, J), f_2(I, J)) = ....
    S_u:            ...... = A(f_3(I,J), f_4(I,J))
        .........
  endfor
endfor
```

Fig. 2. A doubly nested loop program model.

solution to multilevel nested loops can be obtained by enhancing our mechanisms. The model of a doubly nested loop is depicted in Fig. 2, where $f_1(I,J)$, $f_2(I,J)$, $f_3(I,J)$, and $f_4(I,J)$ are all linear functions of loop variables. The dimension of the nested loop is equal to the number of nested loops in it. In loop $I(J)$, $L_I(L_J)$ and $U_I(U_J)$ indicate the lower and upper bounds, respectively. Both the lower and upper bounds of indices should be known at compile time. At first, we will define a program and its data dependence formally in the following.

Definition 1. A sequential program is represented as $P = \langle \Im, \sigma, \vartheta, \wp, \omega, \gamma, \eta \rangle$, where

- $\Im$ is the set of instructions. An instruction is an indivisible unit such as a simple arithmetic operation on program variables.
- σ_s is the depth, or the number of surrounding loops, of instruction s.
- $\vartheta_s(i)$ is an affine expression derived from the loop bounds such that i is a valid loop index for instruction s, if and only if $\vartheta_s(i) \geqslant 0$.
- $\wp_{zsr}(i)$ is the affine array index expression in the rth array reference to array z in instruction s.
- ω_{zsr} is true if and only if the rth array reference to array z in instruction s is a write operation.
- γ_{zsr} is true if and only if the rth array reference to array z in instruction s is a read operation.
- $\eta_{ss'}$ is the number of common loops shared by instruction s and s'.

The access patterns in a program define the constraints of program transformations. The notion of data dependences is well understood. Informally, there is a data dependence from an access function $\wp$ to another access function $\wp'$, if and only if some instance of $\wp$ uses a location that is subsequently used by $\wp'$, and one of the accesses is a write operation. A data dependence set of a program contains all pairs of data-dependent access functions in the program. We define them formally as below. The formal definitions of flow- and anti-dependence are shown in Definitions 3 and 4, respectively. The data dependence set of a program is given in Definition 5.

Definition 2. [Zima and Barbara, 1990] We define $\prec$ to be the 'lexicographically less than' operator for program $P = \langle \Im, \sigma, \vartheta, \wp, \omega, \gamma, \eta \rangle$ such that $i \prec_{ss'} i'$ if and only if the iteration i of instruction s is executed before the iteration i' of s' in P. That is,

$$i \prec_{ss'} i' \equiv \begin{cases} i_{1:\eta_{ss'}} = i'_{1:\eta_{ss'}} \wedge s < s' & \text{or} \\ i_{1:m} = i'_{1:m} \wedge i_{m+1} < i'_{m+1} & \text{for} \quad 0 \leqslant m < \eta_{ss'}. \end{cases}$$

Definition 3. The flow-dependence set of a program $P = \langle \Im, \sigma, \vartheta, \wp, \omega, \gamma, \eta \rangle$, denoted as R^{f}_{δ}, can be defined as:

$$R^{\mathrm{f}}_{\delta} = \left\{ \begin{array}{r} \langle \wp_{zsr}, \wp_{zs'r'} \rangle | (\omega_{zsr} \wedge \gamma_{zs'r'}) \wedge ((\exists i \in \Im^{\sigma s}, i' \in \Im^{\sigma s'}) \ni ((i \prec_{ss'} i') \wedge \\ (\wp_{zsr(i)} - \wp_{zs'r'}(i') = 0) \wedge (\vartheta_s(i)0 \wedge \vartheta s'(i') \geqslant 0))) \end{array} \right\}.$$

Definition 4. The anti-dependence set of a program $P = \langle \Im, \sigma, \vartheta, \wp, \omega, \gamma, \eta \rangle$, denoted as R^{a}_{δ}, can be defined as:

$$R^{\mathrm{a}}_{\delta} = \left\{ \begin{array}{r} \langle \wp_{zsr}, \wp_{zs'r'} \rangle | (\gamma_{zsr} \wedge \omega_{zs'r'}) \wedge ((\exists i \in \Im^{\sigma s}, i' \in \Im^{\sigma s'}) \ni ((i \prec_{ss'} i') \wedge \\ (\wp_{zsr}(i) - \wp_{zs'r'}(i') = 0) \wedge (\vartheta_s(i)0 \wedge \vartheta_{s'}(i') \geqslant 0))) \end{array} \right\}.$$

Definition 5. The data dependence set of a program $P = \langle \Im, \sigma, \vartheta, \wp, \omega, \gamma, \eta \rangle$, denoted as R can be defined as:

$$R = \left\{ \begin{array}{l} \langle \wp_{zsr}, \wp_{zs'r'} \rangle | (\omega_{zsr} \vee \omega_{zs'r'}) \wedge ((\exists i \in \Im^{\sigma s}, i' \in \Im^{\sigma s'}) \ni ((i \prec_{ss'} i') \wedge \\ (\wp_{zsr}(i) - \wp_{zs'r'}(i') = 0) \wedge (\vartheta s(i) \geqslant 0 \wedge \vartheta s'(i') \geqslant 0))) \end{array} \right\}.$$

Usually, an iteration denoting a series of statements in the loop body is a unit of work assigned to a processor. Therefore, the dependence constraints inside the iteration can be ignored when parallelizing a nested loop. The dependence constraints between different iterations, called cross-iteration dependence, are our major concern. All the dependences discussed in this paper include only cross-iteration dependences. In our program model, shown in Fig. 2, statement S_d defines elements of array A, and statement S_u uses them. Dependence exists between S_d and S_u whenever both refer to the same element of array A. If the element defined by S_d is used by S_u in a subsequent iteration, there is a cross-iteration flow dependence between S_d and S_u and will be denoted by $(S_d,\ S_u) \in R^{\mathrm{f}}_{\delta}$. On the other hand, if the element used in S_u is defined by S_d at a later iteration, the dependence is called cross-iteration anti-dependence and will be denoted by $(S_u,\ S_d) \in R^{\mathrm{a}}_{\delta}$.

One of the most common method for computing data dependences involves solving a set of linear Diophantine equations (Zima and Barbara, 1990) with a set of constraints formed by the iteration boundaries as shown in Definition 5. The data dependence set R contains two pairs of access functions: $\langle \wp_{b11}, \wp_{b21} \rangle$ and $\langle \wp_{a11}, \wp_{a21} \rangle$. The constraints are $\vartheta_{su} \geqslant 0$ and $\vartheta_{sd} \geqslant 0$, respectively. The loop in Fig. 2 carries cross-iteration dependences if and only if there exists four integers i_1, j_1, i_2, j_2 satisfying the system of linear Diophantine equations given by (1) and the system of inequalities given by Eq. (2).

$$f_1(i_1, j_1) = f_3(i_2, j_2) \quad \text{and} \quad f_2(i_1, j_1) = f_4(i_2, j_2), ((i_1, j_1) \quad \text{and} \quad (i_2, j_2) \in (I, J)), \tag{1}$$

$$L_1 \leqslant i_1, i_2 \leqslant U_1 \quad \text{and} \quad L_2 \leqslant j_1, j_2 \leqslant U_2. \tag{2}$$

The dependence convex hull (DCH) (Tzen and Ni, 1993) is a convex polyhedron and a subspace of the solution space. There are two approaches for solving the system of Diophantine equations in (1). One way is to set i_1 to x_1, j_1 to y_1, and then solve i_2 and j_2, respectively. Here, i_1, j_1, i_2, j_2 and its inequalities can be represented as shown in Eq. (3), which forms DCH and is denoted by DCH1.

$$\begin{array}{l} (i_1, j_1, i_2, j_2) = (x_1, y_1, g_1(x_1, y_1), g_2(x_1, y_1)) \\ L_1 \leqslant x_1, g_1(x_1, y_1) \leqslant U_1 \quad \text{and} \quad L_2 \leqslant y_1, g_2(x_1, y_1) \leqslant U_2 \end{array} \tag{3}$$

The other way is to set i_2 to x_2, j_2 to y_2 and then solve i_1 and j_1, respectively. Here, i_1, j_1, i_2, j_2 and its inequalities can be represented as shown in Eq. (4), which forms DCH and is denoted by DCH2.

$$\begin{array}{l} (i_1, j_1, i_2, j_2) = (g_3(x_2, y_2), g_4(x_2, y_2), x_2, y_2), \\ L_1 \leqslant g_3(x_2, y_2), x_2 \leqslant U_1 \quad \text{and} \quad L_2 \leqslant g_4(x_2, y_2), y_2 \leqslant U_2. \end{array} \tag{4}$$

Clearly, if we have a solution i_1, j_1 in DCH1, we will have a solution i_2, j_2 in DCH2 because each of them is derived from the same set of equations. If iteration (i_2, j_2) is dependent on iteration (i_1, j_1), we will have a dependence vector $D(x, y)$ with $d_i(x, y) = i_2 - i_1$ and $d_j(x, y) = j_2 - j_1$. Therefore, for DCH1, we have

$$d_i(i_1, j_1) = g_1(i_1, j_1) - i_1 \quad \text{and} \quad d_j(i_1, j_1) = g_2(i_1, j_1) - j_1. \tag{5}$$

For DCH2, we have

$$d_i(i_2, j_2) = i_2 - g_3(i_2, j_2) \quad \text{and} \quad d_j(i_2, j_2) = j_2 - g_4(i_2, j_2). \tag{6}$$

Here we briefly describe some techniques for solving non-uniform dependence problems. The dependence uniformization scheme (Tzen and Ni, 1993) constructs two basic dependence vector sets, using dependence slope theory, and adds them to every iteration in the iteration space. The loop can then be parallelized by current parallel compilation techniques to uniform dependence loops. It is parallelized according to the two uniform dependence vectors, resulting in a do across type of loop execution. However, this mechanism always imposes too many additional dependences on the iteration space.

The minimum dependence distance tiling method (Punyamurtula and Chaudhary, 1994; Punyanurtula et al., 1997) exploits the parallelism using minimum distances computed from the dependence vectors of the integer DCH (IDCH) extreme points. Minimum distances are used to partition the iteration space into tiles of regular size and shape, but irregularity of non-uniform dependence distances is ignored.

The three-region partitioning technique (Zaafrani and Ito, 1994) divides the iteration space into two parallel regions and one serial region. The first region represents the part of iteration space where the anti-dependence exists. Hence, iterations in this area can be fully executed in parallel, provided that copy-renaming is performed. The second region represents the part of iteration space with flow-dependence heads and corresponding tails in the first region. The second region will be executed in parallel after the first region is executed. The serial region then represents the rest of the iteration space to which the dependence uniformization scheme can be applied. However, if this serial region increases, the performance of the loop will be significantly degraded.

A unique set-oriented partitioning mechanism (Ju and Chaudhary, 1996) divides the iteration space into the dependence unique head and tail sets. In this partitioning method, there are various combinations of overlap of these sets. The execution order of partitioned regions depends on these combinations. The iterations within the unique set can be executed in parallel, but the unique tail sets must be executed before the unique head sets. This technique provides more accurate information for the iteration space. However, it also suffers from several disadvantages. First, it does not present an exact partitioning scheme, and therefore, it is difficult to know the best scheme at compile-time. Second, it inevitably leaves some parallelism unexplored since the method of minimum dependence distance is applied to parallelize the rest of the iteration space that contains both dependence tails and heads. Increasing the iterations in such a region will degrade the speedup of the method.

The optimized three-region partitioning (OTRP) scheme (Pean and Chen, 1999) and the improved three-region partition (ITRP) scheme (Cho and Lee, 1997) are similar to three-region partitioning in the sense that the iteration space is divided into two parallel regions and one serial region. The size of a parallel region in the ITRP and OTRP schemes are not less than for the three-region partitioning mentioned above. On the other hand, it is simple to divide the iteration space into three regions, where the execution order of partitions is always the same. However, each of these mechanisms still leaves some unexplored parallelism in the partitioned serial region. Moreover, variable size partitioning in the serial region also has high time complexity. In the next section, we will present the detailed description of our new mechanism.

3. Optimized dependence convex hull partitioning

In the previous section, we briefly discussed advantages and disadvantages of conventional techniques. Here, we present an effective technique, called optimized dependence convex hull partitioning (ODCHP), to improve the drawbacks of those techniques. We use an arrow to represent dependence in an iteration space. We call the arrow's head as the dependence head, and the arrow's tail is known as the dependence tail. In order to explain our method clearly, we will give formal definitions about unique head (tail) set (Ju and Chaudhary, 1996) as follows.

Definition 6 (*Flow-dependence head set*). The flow-dependence head set of a program $P = \langle \Im, \sigma, \vartheta, \wp, \omega, \gamma, \eta \rangle$, denoted $Head(R_\delta^{\mathrm{f}})$, is defined as

$$Head\ (R_\delta^{\mathrm{f}}) = \left\{ \begin{array}{c} \langle \forall i' \in \Im^{\sigma s'} \rangle | ((\omega_{zsr} \wedge \gamma_{zs'r'})) \wedge ((\exists i \in \Im^{\sigma s}, i' \in \Im^{\sigma s'}) \ni ((i \prec_{ss'} i') \wedge \\ (\wp_{zsr}(i) - \wp_{zs'r'}(i') = 0) \wedge (\vartheta_s(i) \geqslant 0 \wedge \vartheta_{s'}(i')0))) \end{array} \right\}.$$

Definition 7 (*Flow-dependence tail set*). The flow-dependence tail set of a program $P = \langle \Im, \sigma, \vartheta, \wp, \omega, \gamma, \eta \rangle$, denoted by Tail (R_δ^{f}), is defined as

$$Tail\ (R_\delta^{\mathrm{f}}) = \left\{ \begin{array}{c} \langle \forall i \in \Im^{\sigma s} \rangle | ((\omega_{zsr} \wedge \gamma_{zs'r'})) \wedge ((\exists i \in \Im^{\sigma s}, i' \in \Im^{\sigma s'})((i \prec_{ss'} i') \wedge \\ (\wp_{zsr}(i) - \wp_{zs'r'}(i') = 0) \wedge (\vartheta_s(i) \geqslant 0 \wedge \vartheta_{s'}(i') \geqslant 0))) \end{array} \right\}.$$

Definition 8 (*Anti-dependence head set*). The anti-dependence head set of a program $P = \langle \Im, \sigma, \vartheta, \wp, \omega, \gamma, \eta \rangle$, denoted by $Head(R_\delta^{\mathrm{a}})$, is defined as

$$Head\ (R_\delta^{\mathrm{a}}) = \left\{ \begin{array}{c} \langle \forall i' \in \Im^{\sigma s'} \rangle | ((\gamma_{zsr} \wedge \gamma_{zs'r'})) \wedge ((\exists i \in \Im^{\sigma s}, i' \in \Im^{\sigma s'})((i \prec_{ss'} i') \wedge \\ (\wp_{zsr}(i) - \wp_{zs'r'}(i') = 0) \wedge (\vartheta_s(i) \geqslant 0 \wedge \vartheta_{s'}(i') \geqslant 0))) \end{array} \right\}.$$

Definition 9 (*Anti-dependence tail set*). The anti-dependence tail set of a program $P = \langle \Im, \sigma, \vartheta, \wp, \omega, \gamma, \eta \rangle$, denoted by $Tail(R_\delta^{\mathrm{a}})$, is defined as

$$Tail\ (R_\delta^{\mathrm{a}}) = \left\{ \begin{array}{c} \langle \forall i \in \Im^{\sigma s} \rangle | ((\gamma_{zsr} \wedge \omega_{zs'r'})) \wedge ((\exists i \in \Im^{\sigma s}, i' \in \Im^{\sigma s'})((i \prec_{ss'} i') \wedge \\ (\wp_{zsr}(i) - \wp_{zs'r'}(i') = 0) \wedge (\vartheta_s(i) \geqslant 0 \wedge \vartheta_{s'}(i') \geqslant 0))) \end{array} \right\}.$$

Definition 10 (Ju and Chaudhary, 1996). (Unique head (tail) set). The unique head (tail) set is a set of integer points in the iteration space that satisfies the following conditions: (1) it is the subset of one of the DCHs (or is the DCH itself); (2) it contains all the dependence arrows' heads (tails), but does not contain any other dependence arrows' tails (heads).

We will first examine the concept of DCH1 and DCH2 because it can partition the iteration space into unique sets.

Lemma 1. *For a nested loop, DCH1 contains all flow-dependence tails and all anti-dependence heads (if they exist), and DCH2 contains all anti-dependence tails and all flow-dependence heads (if they exist). Thus,*

$$\{\forall i \in \Im^{\sigma s} | i \in (Tail(R_\delta^{\mathrm{f}}) \vee Head(R_\delta^{\mathrm{a}}))\} \subseteq DCH1 \quad \text{and}$$
$$\{\forall i \in \Im^{\sigma s} | i \in (Tail\ (R_\delta^{\mathrm{a}}) \vee Head(R_\delta^{\mathrm{f}}))\} \subseteq DCH2.$$

Proof. The iterations in DCH1 and DCH2 can be obtained from Eqs. (3) and (4). The dependence vectors of DCH1 and DCH2 can also be calculated from Eqs. (5) and (6). The iterations inside DCH1 can be constructed by using functions $g_1(i_1, j_1)$ and $g_2(i_1, j_1)$. Thus, if there is any iteration inside DCH1 and its corresponding array reference is a write reference, the array reference of the iteration must become a flow-dependence tail. Otherwise, if there is any iteration inside DCH1 and its corresponding array reference is a read reference, the array reference of the iteration must become an anti-dependence head. Hence DCH1 contains all flow-dependence tails and all anti-dependence heads (if they exist). Similarly, DCH2 contains all anti-dependence tails and flow-dependence heads (if they exist).

Lemma 1 tells us that DCH1 and DCH2 may contain more than one unique set and two kinds of unique sets in DCH1 and DCH2 are also given. On the contrary, the following lemma states the condition for DCH1 and DCH2 to be unique sets. □

Lemma 2. *For a nested loop if $d_i(x, y) = 0$ does not pass through any DCH, there will be only one kind of dependence, either flow- or anti-dependence, and DCH itself is the unique head set or the unique tail set.*

Proof. If $d_i(x, y) = 0$ does not pass through any DCH, then DCH is on the side of either $d_i(x, y) < 0$ or $d_i(x, y) > 0$. If DCH1 and DCH2 are on the side of $d_i(x, y) > 0$, then DCH1 and DCH2 contain flow-dependence unique tail and head sets, respectively, because based on Lemma 1 the iterations in DCH1 and DCH2 are derived from Eqs. (5) and (6), respectively. Similarly, if DCH1 and DCH2 are on the side of $d_i(x, y) < 0$, then DCH1 and DCH2 contain an anti-dependence unique head set and anti-dependence unique tail set, respectively, because based on Lemma 1 the iterations in DCH1 and DCH2 are derived from Eqs. (5) and (6), respectively. Thus, if $d_i(x, y) = 0$ does not pass through any DCH, there will be only one kind of dependence, either flow- or anti-dependence, and DCH itself is the unique head set or tail set. □

DCH1 and DCH2 are constructed from the same system of linear Diophantine equations and inequalities. Lemma 3 highlights their common attributes.

Lemma 3. *For a nested loop, if $d_i(x_1, y_1) = 0$ does not pass through DCH1, then $d_i(x_2, y_2) = 0$ will not pass through DCH2.*

Proof. If $d_i(x_1, y_1) = 0$ does not pass through DCH1, then there is only one kind of dependence, either flow- or anti-dependence according to Lemma 2. Now assuming $d_i(x_2, y_2) = 0$ passes through DCH2, then DCH2 contains both a flow-dependence head set and an anti-dependence tail set. Thus, there must be a corresponding flow-dependence tail set and an anti-dependence head set inside DCH1 by Lemma 1. Consequently, this means that $d_i(x_1, y_1) = 0$ does pass through DCH1 by Lemma 2. Henceforth, if $d_i(x_1, y_1) = 0$ does not pass through DCH1, then $d_i(x_2, y_2) = 0$ will not pass through DCH2. □

Lemma 4. *For a nested loop, if $d_i(x_1, y_1) = 0 (d_i(x_2, y_2) = 0)$ does not pass through DCH1 (DCH2), and DCH1 (DCH2) is on the side of $d_i(x_1, y_1) > 0 (d_i(x_2, y_2) > 0)$, then DCH1 (DCH2) is a flow-dependence unique tail (head) set. Otherwise, if*

*DCH*1 (*DCH*2) *is on the side of* $d_i(x_1,y_1) < 0(d_i(x_2,y_2) < 0)$, *then DCH*1 (*DCH*2) *is an anti-dependence unique head* (*tail*) *set*.

Proof. If $d_i(x_1,y_1) = 0(d_i(x_2,y_2) = 0)$ does not pass through DCH1 (DCH2), and DCH1 (DCH2) is on the side of $d_i(x_1,y_1) > 0(d_i(x_2,y_2) > 0)$, then DCH1 will be a flow-dependence unique tail set according to Lemmas 1 and 2. On the other hand, DCH2 is a flow-dependence unique head set by Lemma 3. Otherwise, if DCH1 (DCH2) is on the side of $d_i(x_1,y_1) < 0(d_i(x_2,y_2) < 0)$, then DCH1 will be an anti-dependence unique head set based on Lemmas 1 and 2. And thus DCH2 is an anti-dependence unique tail set by Lemma 3. □

Now, we have found that if $d_i(x_1,y_1) = 0$ does not pass through DCH1, then both DCH1 and DCH2 are unique sets and the points in them have the same property. DCH1 (DCH2) may contain dependence heads and tails when $d_i(x_1,y_1) = 0(d_i(x_2,y_2) = 0)$ passes through it. This makes it difficult finding unique head and tail sets. Lemmas 5 and 6 will show some common attributes when $d_i(x_1,y_1) = 0$ passes through DCH1 (DCH2).

Lemma 5. *For a nested loop*, *if* $d_i(x,y) = 0$ *passes through a DCH*, *it will divide DCH into a unique tail set and a unique head set*. *Furthermore*, $d_j(x,y) = 0$ *determines the inclusion of* $d_i\ (x,y) = 0$ *in one of the sets*.

Proof. If $d_i(x,y) = 0$ passes through DCH1 (DCH2), then the DCH1 (DCH2) contains both a flow-dependence tail (head) set and an anti-dependence head (tail) set, and the sets are divided by the line $d_i(x,y) = 0$ according to Eqs. (3)–(6). The points in the line $d_i(x,y) = 0$ can be further categorized into different sets according to the line $d_j(x,y) = 0$. If these points are on the side of $d_j(x,y) > 0$, they belong to a flow-dependence unique tail set (flow-dependence unique head set) in DCH1 (DCH2). Otherwise, if these points are on the side of $d_j(x,y) < 0$, they belong to an anti-dependence unique head set (anti-dependence unique tail set) in DCH1 (DCH2) according to the results obtained using Eqs. (5) and (6). Furthermore, if the point is in $d_i(x,y) = 0$ and $d_j(x,y) = 0$, it has no cross-iteration dependence. Thus, $d_j(x,y) = 0$ determines the inclusion of $d_i(x,y) = 0$ in one of the sets. □

Lemma 6. *For a nested loop*, *if* $d_i(x_1,y_1) = 0$ *passes through DCH*1 (*DCH*2), *then DCH*1 (*DCH*2) *is the union of a flow-dependence unique tail* (*head*) *set and an anti-dependence unique head* (*tail*) *set*.

Proof. If $d_i(x,y) = 0$ passes through DCH1 (DCH2), then DCH1 (DCH2) contains both a flow-dependence unique tail (head) set and an anti-dependence unique head (tail) set, and the sets are divided by the line $d_i(x,y) = 0$ according to Lemmas 1 and 5. Thus, if $d_i(x_1,y_1) = 0$ passes through DCH1 (DCH2), then DCH1 (DCH2) is the union of a flow-dependence unique tail (head) set and an anti-dependence unique head (tail) set. □

Based on the properties described above, there are various combinations of overlaps of these unique sets. We will illustrate these properties by the following example:

Example 2. Consider the doubly nested loop

```
forI = 1, 10
  forJ = 1, 10
    A(2 * J + 3, I + J + 5) = ...
      ... = A(2 * I + J − 1, 3 * I − 1)
  endfor
endfor
```

The set of inequalities and dependence distances of the loop in Example 2 are computed as follows:

$$
\begin{array}{ll}
DCH1: & DCH2: \\
1 \leqslant i_1 \leqslant 10 \text{ and} & 1 \leqslant i_2 \leqslant 10 \text{ and} \\
1 \leqslant j_1 \leqslant 10 \text{ and} & 1 \leqslant j_2 \leqslant 10 \text{ and} \\
1 \leqslant \frac{i_1}{3} + \frac{j_1}{3} + 2 \leqslant 10 \text{ and} & 1 \leqslant 2i_2 - \frac{j_2}{2} - 4 \leqslant 10 \text{ and} \\
1 \leqslant -\frac{2i_1}{3} + \frac{4j_1}{3} \leqslant 10 \text{ and} & 1 \leqslant i_2 + \frac{j_2}{2} - 2 \leqslant 10 \\
d_i(i_1,j_1) = -\frac{2i_1}{3} + \frac{j_1}{3} + 2, & d_i(i_2,j_2) = -i_2 + \frac{j_2}{2} + 4, \\
d_j(i_1,j_1) = -\frac{2i_1}{3} + \frac{j_1}{3} & d_j(i_2,j_2) = -i_2 + \frac{j_2}{2} + 2,
\end{array}
\tag{7}
$$

Fig. 3 shows DCHs and the unique head (tail) sets of the loop in Example 2. Clearly, $d_i(x_1,y_1) = 0$ divides DCH1 into two areas. The area on the side of $d_i(x_1,y_1) < 0$ is an anti-dependence unique head set, which is on the right side of

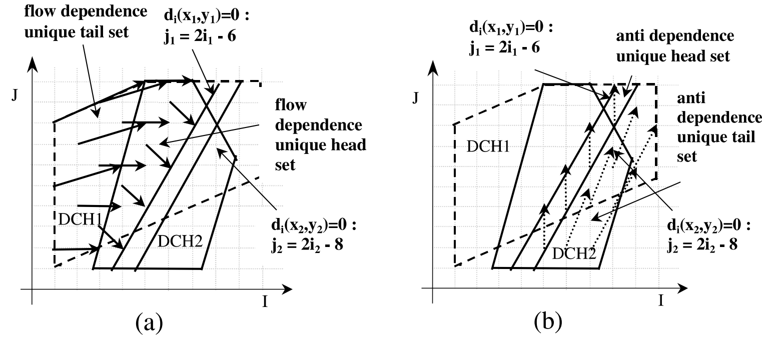

Fig. 3. Unique head sets and unique tail sets of: (a) flow-dependence, and (b) anti-dependence.

$d_i(x_1,y_1) = 0$ as shown in Fig. 3(b). Similarly, the area on the side of $d_i(x_1,y_1) > 0$ is a flow-dependence unique tail set, which is on the left side of $d_i(x_1,y_1) = 0$ as shown in Fig. 3(a). $d_i(x_2,y_2) = 0$ also divides DCH2 into two areas. The area on the side of $d_i(x_2,y_2) < 0$ is an anti-dependence unique tail set, which is on the left side of $d_i(x_2,y_2) = 0$ as shown in Fig. 3(b). The area on the side of $d_i(x_2,y_2) > 0$ is the flow-dependence unique head set, which is on the right side of $d_i(x_2,y_2) = 0$ as shown in Fig. 3(a).

Our approach is based on convex hull theory (Tzen and Ni, 1993). We use the lines $d_i(i,j) = 0$ and $d_j(i,j) = 0$ to partition the iteration space into four unique sets. All possible sets partitioned by $d_i(i,j)$ and $d_j(i,j)$ are summarized in Table 1 according to the above lemmas. Table 2 shows each set that is part of a line segment, which is partitioned according to the sign of $d_j(i_1,j_1)$ and $d_j(i_2,j_2)$.

At first, we use memory space to gain the benefits of parallel execution because the anti-dependence can be avoided by the concept of copy-renaming (Zima and Barbara, 1990). Lemma 7 below, introduces the condition where copy-renaming can be used.

Lemma 7. [Elimination of anti-dependence] *For a nested loop, if there is an anti-dependence between two statements S_d and S_u (denoted as $(S_d,S_u) \in R^{a}_{\delta}$) in the iteration space, these two statements can be executed in parallel after copy-renaming.*

Proof. If $(S_d,S_u) \in R^{a}_{\delta}$, we can assume that there are two accesses, $A_d \in S_d$ and $A_u \in S_u$, referencing the same memory location. For copy-renaming, we copy A_d into another memory location A'_d before execution of the iteration. Therefore, access A_u in the statement S_u can be changed into A'_u. Thus, the two accesses, $A_d \in S_d$ and $A'_u \in S_u$, will not refer to the same memory location and the two statements S_d and S_u can be executed in parallel.

Our goal is to develop an algorithm to identify non-uniform partitions with variable sizes so as to maximize parallelism from a doubly nested loop with non-uniform dependence. If we can find the dependence head iteration, (x,y), that occurs first (i.e., its execution order is lexicographically the first) in the range of the loop given by $l_1 \leqslant x \leqslant u_1$ and $l_2 \leqslant y \leqslant u_2$, then all the iterations, (i,j), in the range of $(l_1 \leqslant i \leqslant x-1)$ and $l_2 \leqslant j \leqslant u_2)$ plus $(i = x$ and $l_2 \leqslant j \leqslant y-1)$ can be executed in parallel since there are no dependences between these iterations. Thus, we can make the iterations in the range of $(l_1 \leqslant i \leqslant x-1$ and $l_2 \leqslant j \leqslant u_2)$ plus $(i = x$ and $l_2 \leqslant j \leqslant y-1)$ into a partition with size $(x-1_2)(u_2-l_2+1)+$

Table 1
The different sets partitioned by $d_i(i,j)$ and $d_j(i,j)$[a]

	$d_i(i_1,j_1) > 0$	$d_i(i_1,j_1) < 0$	$d_i(i_1,j_1) = 0$
$d_i(i_2,j_2) > 0$	*Head* (R^{f}_{δ}) *Tail* (R^{f}_{δ})	*Head* (R^{f}_{δ}) *Head* (R^{a}_{δ})	Refer to Table 2
$d_i(i_2,j_2) < 0$	*Tail* (R^{f}_{δ}) *Tail* (R^{a}_{δ})	*Tail* (R^{f}_{δ}) *Head* (R^{a}_{δ})	
$d_i(i_2,j_2) = 0$	Refer to Table 2		

[a] *Tail* (R^{f}_{δ}): flow dependence tail; *Head* (R^{f}_{δ}): flow dependence head; *Tail* (R^{a}_{δ}): anti dependence tail set; *Head* (R^{a}_{δ}): anti dependence head.

Table 2
The case of $d_i(i_1,j_1) = 0$ or $d_i(i_2,j_2) = 0$[a]

$d_j(i_1,j_1) > 0$	$d_j(i_1,j_1) < 0$	$d_j(i_1,j_1) = 0$	$d_j(i_2,j_2) > 0$	$d_j(i_2,j_2) < 0$	$d_j(i_2,j_2) = 0$
Tail (R^{f}_{δ})	*Head* (R^{a}_{δ})	No cross-iteration dependence	*Head* (R^{f}_{δ})	*Tail* (R^{a}_{δ})	No cross-iteration dependence

[a] *Tail* (R^{f}_{δ}): flow dependence tail; *Head* (R^{f}_{δ}): flow dependence head; *Tail* (R^{a}_{δ}): anti dependence tail set; *Head* (R^{a}_{δ}): anti dependence head.

$(y - l_2)$. The range of ($x = i = x - 1$ and $l_2 \leqslant j \leqslant u_2$) plus ($i = x$ and $y \leqslant j \leqslant u_2$) can then be partitioned in the same way. We can repeat this procedure for the rest of the loop iterations. However, there is no dependence if the tail of the dependence head is located at one of the previous partitions. Formally, we give the following definitions of ODCHP at first. And then some important properties are also investigated. □

Definition 11 (*Optimized dependence convex hull partition* (*ODCHP*)). For a nested loop, the kth partition of the ODCHP method, $ODCHP_k$, begins from the end of the next iteration of the previous partition to the previous iteration of the first head node with a tail node that belongs to the current partition. Formally, the new ODCHP partition of a program $P = \langle \Im, \sigma, \vartheta, \wp, \omega, \gamma, \eta \rangle$ is defined as

$$ODCHP_k = \left\{ \begin{array}{l} \langle \forall i \in \Im^{\sigma s} \rangle | ((\exists i_4, i_2 \in \Im^{\sigma s} \ni i_2 \prec_{s2s4} i_4) \wedge (\forall i_1 \in ODCHP_{k-1} \ni i_1 \prec_{s1s4} i_4) \\ \wedge (\nexists i_3 \in \Im^{\sigma s} \ni i_2 \prec_{s2s3} i_3 \prec_{s3s4} i_4)) \wedge ((\exists i_6 \in \Im^{\sigma s} \ni i_6 \prec_{s6s8} i_8) \wedge \\ (\nexists i_7 \in \Im^{\sigma s}, i_6 \prec_{s6s7} i_7 \prec_{s6s8} i_8) \wedge (\exists i_5 \in \Im^{\sigma s} \ni (i_5, i_8) \in R^{\mathrm{f}}_{\delta}) \wedge (i_2 \prec_{s2s5} i_5 \prec_{s5s8} i_8)) \\ (\nexists i_{10}, i_{11} \in \Im^{\sigma s} \ni i_2 \prec_{s2s10,11} i_{10}, i_{11} \prec_{s10,11s8} i_8 \wedge (i_{10}, i_{11}) \in R^{\mathrm{f}}_{\delta}) \wedge (i_2 \prec_{s2s} i \prec_{ss8} i_8) \end{array} \right\}.$$

Now, consider two processes, P_1 and P_2, executing iterations I_1 and I_2, respectively. These two processes can be executed in parallel and are denoted by $P_1|P_2$ if they are independent and do not create confusing results. Formally, the condition is $(I_1, I_2) \in R = \varnothing$.

Theorem 1. [Inclusion property] *For a nested loop, if all the corresponding dependence tails of dependence heads belong to the previous ODCHP partitions, then the iterations from the source node of the current ODCHP partition to the node of the last dependence head with lexicographical execution order can be executed in parallel. Formally, for a program* $P = \langle \Im, \sigma, \vartheta, \wp, \omega, \gamma, \eta \rangle$ *if*

$$\left\{ \begin{array}{l} (\forall i_1, i_2 \in \Im^{\sigma s} \ni (i_1, i_2) \in R^{\mathrm{f}}_{\delta}) \wedge \\ (\exists i_3, i_4 \in \Im^{\sigma s} (i_3, i_4) \in R^{\mathrm{f}}_{\delta}) \wedge (i_2 \prec_{s2s4} i_4) \\ (i_2, i_4 \in ODCHP_k) \Rightarrow ((i_1, i_3 \in ODCHP_{k1}) \wedge (K_1 < K)) \end{array} \right\} \Rightarrow \left\{ \begin{array}{l} ((\forall i_5, i_6 \in \Im^{\sigma s})(i_5, i_6 \prec_{s5,6s4} i_4)) \wedge \\ (i_5, i_6 \in ODCHP_k) \Rightarrow \\ P(i_5) \| P(i_6) \end{array} \right\}.$$

Proof. For a program $P = \langle \Im, \sigma, \vartheta, \wp, \omega, \gamma, \eta \rangle, \forall i \in Head(R^{\mathrm{f}}_{\delta})$ if their tails $j \in Tail(R^{\mathrm{f}}_{\delta})$ belong to previous ODCHP partitions and $(i, j) \in R^{\mathrm{f}}_{\delta}$, then as a result of Definition 2, their tails must be executed before the partition to which their heads belong. Thus, the iterations from the source node of the current ODCHP partition to the node of the last $k \in Head(R^{\mathrm{f}}_{\delta})$ with lexicographical execution order can be executed in parallel because the dependence of all heads has been resolved in the execution of previous partitions.

The iteration space in Example 1 (a) is partitioned by the minimum dependence distance tiling method as shown in Fig. 4(a). Even the corresponding dependence tails of dependence heads a, b, c, d belong to the previous tiles, they are also tiled with less dependence distance than the ODCHP scheme. By Theorem 1, we find that the number of iterations being executed in parallel are greatly increased by using ODCHP scheme. We explain the inclusion property and demonstrate it by partition the loop nest in Example 1 (a). As shown in Fig. 4(b), all the corresponding dependence tails of dependence heads a, b, c, d belongs to the previous ODCHP partition, $ODCHP_3$. Thus, the iterations from the

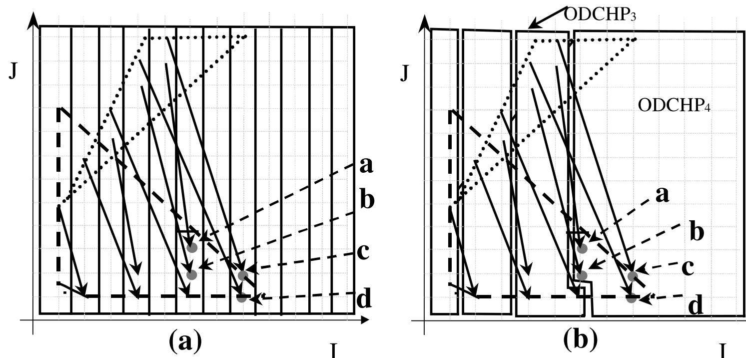

Fig. 4. (a) An example partitioned by the minimum dependence distance tiling without inclusion property, (b) an example of ODCHP partitioning with the inclusion property.

source node of the $ODCHP_4$ partition to the node of the last dependence head c with lexicographically execution order can be executed in parallel. □

Theorem 2. *For a nested loop, the iterations inside the same ODCHP partition can be executed in parallel. Formally, for a given program* $P = \langle \Im, \sigma, \vartheta, \wp, \omega, \gamma, \eta \rangle$ *if*

$$\left\{ \begin{array}{l} (\forall i_1, i_2 \in \Im^{\sigma s}) \wedge \\ (i_1, i_2 \in ODCHP_k) \end{array} \right\} \Rightarrow \{ P(i_1) \| P(i_2) \}.$$

Proof. As indicated by the inclusion property, for a program $P = \langle \Im, \sigma, \vartheta, \wp, \omega, \gamma, \eta \rangle$, the iterations from the source node of the ODCHP partition to the node of the last $i \in Head(R^f_\delta)$ that belong to the current partition with lexicographical execution order, can be executed in parallel. Thus, the following is a proof that the iterations from the last $i \in Head(R^f_\delta)$ to the node before the next head can be executed in parallel with the nodes that are included by the inclusion property. This is indicated as the iteration space $\Im^{\sigma s}$. The proof of this is as follows.

1. If there are dependence heads in the iteration space $\Im^{\sigma s}$, then it contradicts Definition 2.
2. If there are dependence tails in the iteration space $\Im^{\sigma s}$, then their heads must not be in the iteration space $\Im^{\sigma s}$ because there is no dependence head as indicated in Definition 2. Thus, due to the constraint of lexicographical execution order, their corresponding dependence heads must be in later ODCHP partitions.
3. There is no dependence head or tail in the iteration space $\Im^{\sigma s}$. □

As indicated above, the iterations in an ODCHP partition can be executed in parallel.

Theorem 3. *For a nested loop, the number of iterations inside each partition tiled by the ODCHP mechanism is a greedy maximum under the constraint of lexicographical execution order.*

Proof. If we tile the partition with more iterations than the ODCHP partition, then we have to include a dependence head with a tail inside the same ODCHP partition as shown in Theorem 2. Thus, the iterations inside the same ODCHP partition cannot be executed in parallel and the original ODCHP partition defined in Definition 2 is a greedy maximum. □

Theorem 4. [Bounded property] *For a given program* $P = \langle \Im, \sigma, \vartheta, \wp, \omega, \gamma, \eta \rangle, \forall i \in Head(R^f_\delta)$ *must be in the iteration space* $\{\{\{(x_2, y_2) | d_i(x_2, y_2) > 0\} \cup \{(x_2, y_2) | d_i(x_2, y_2) = 0 \text{ and } d_j(x_2, y_2) > 0\}\} \cap \{DCH2\}\}$. *In addition,* $\forall j \in Tail(R^f_\delta)$ *must be in the iteration space* $\{\{\{(x_1, y_1) | d_i(x_1, y_1) > 0\} \cup \{(x_1, y_1) | d_i(x_1, y_1) = 0 \text{ and } d_j(x_1, y_1) > 0\}\} \cap \{DCH1\}\}$. *Formally,*

$$\left\{ \begin{array}{l} (\forall i \in \Im^{\sigma s}, j \in \Im^{\sigma s})(i, j) \in R^f_\delta) \\ (i \in Head(R^f_\delta) \Rightarrow \\ i \in \{\{\{(x_2, y_2) | d_i(x_2, y_2) > 0\} \cup \{(x_2, y_2) | (di(x_2, y_2) = 0) \wedge (dj(x_2, y_2) > 0)\}\} \cap DCH2\} \\ (j \in Tail(R^f_\delta) \Rightarrow \\ j \in \{\{\{(x_1, y_1) | d_i(x_1, y_1) > 0\} \cup \{(x_1, y_1) | (d_i(x_1, y_1) = 0) \wedge (d_j(x_1, y_1) > 0)\}\} \cap DCH_1\} \end{array} \right\}.$$

Proof. As shown in Table 1 and above lemmas, for a given program $P = \langle \Im, \sigma, \vartheta, \wp, \omega, \gamma, \eta \rangle$, the flow-dependence head set can only appear in the set of $Head(R^f_\delta) = \{\{\{(x_2, y_2) | d_i(x_2, y_2) > 0\} \cup \{(x_2, y_2) | d_i(x_2, y_2) = 0 \text{ and } d_j(x_2, y_2) > 0\}\} \cap \{DCH2\}\}$, and the flow-dependence tail set can only appear in the set of $Tail(R^f_\delta) = \{\{\{(x_1, y_1) | d_i(x_1, y_1) > 0\} \cup \{(x_1, y_1) | d_i(x_1, y_1) = 0 \text{ and } d_j(x_1, y_1) > 0\}\} \cap \{DCH1\}\}$.

As depicted in Theorem 4, the set of $Head(R^f_\delta)$ is where the flow-dependence head may appear and $Tail(R^f_\delta)$ is where the flow-dependence tail may appear. Thus, if we want to check where the flow-dependence heads and tails might occur, we have only to check the set of $Head(R^f_\delta)$ and $Tail(R^f_\delta)$. This greatly reduces the complexity of the checking procedures. Fig. 5 is the dependence sets of Example 1 (a). The shadow area contains all the flow dependence head and tail sets and covers only a little part of the iteration space. Thus, if we want to check dependences for the ODCHP partitioning, we only have to check iterations in this shadow area. □

Theorem 5. [Forwarding property] *For a given program* $P = \langle \Im, \sigma, \vartheta, \wp, \omega, \gamma, \eta \rangle, \forall i \in Head\ (R^f_\delta$ *and* $\forall j \in Tail(R^f_\delta)$, *if we want to check the dependence relation between i and j, then we only have to check those iterations after the j iteration with lexicographical execution order as a consequence of the following property*:

$$\{(\forall i, j \in \Im^{\sigma s}) \wedge ((i \in Head(R^f_\delta), j \in Tail(R^f_\delta)) \wedge (i, j) \in R^f_\delta)\} \Rightarrow \{j \prec_{ss} i\}.$$

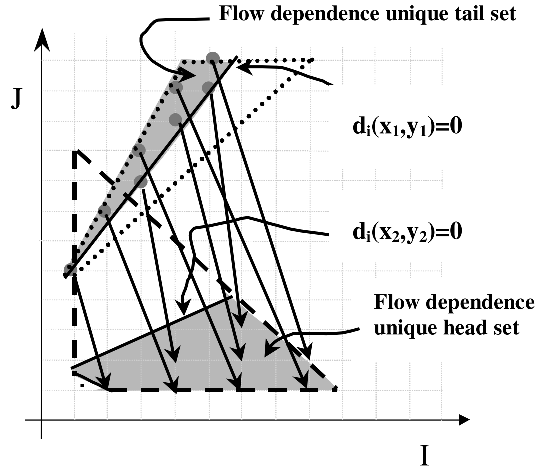

Fig. 5. The dependence relations of Example 1 with bounded property.

Proof. $\forall i \in Head(R^f_\delta)$ and $\forall j \in Tail(R^f_\delta)$, if there is dependence between iteration i and j, then iteration j must be executed before iteration i. On the other hand, due to the property of lexicographical execution order, iteration j must appear before iteration i. Thus, if we want to check the dependence relation between iterations i and j, then only the iterations that are after iteration j with lexicographical execution order must be checked. □

We explain the forwarding property and demonstrate it by partitioning the loop nest in Example 2. As shown in Fig. 6, the shadow area is the iteration space where we have to check for dependence after implementation of the forwarding property. The shadow area is decreasing as the partitioning algorithm is executing. Thus, the complexity of the ODCHP partitioning algorithm is successfully reduced.

By using an improved integer programming technique and concepts defined by theorems from Theorems 1–5, we obtain the generalized and optimized algorithm to exploit any parallelism available in nested loops with non-uniform dependences. On the other hand, before performing our improved integer programming technique, we should obtain

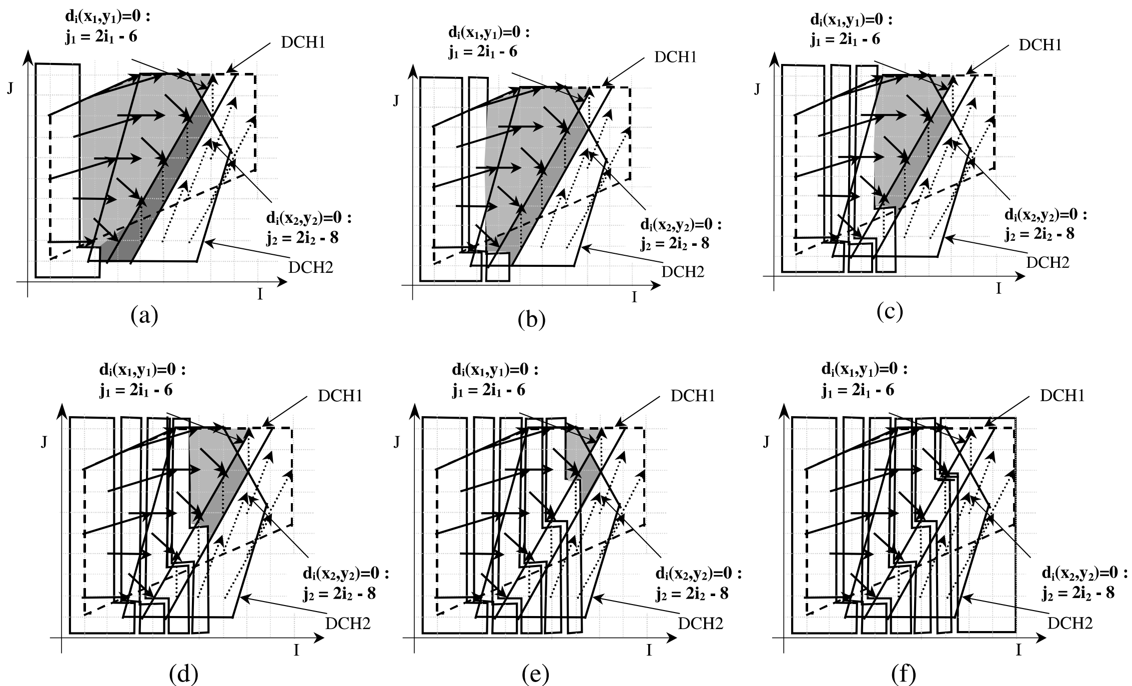

Fig. 6. The partitioning procedure of the ODCHP algorithm with the forwarding property of Example 2.

the general solution of dependence equations, the dependence vector function, DCH1, and DCH2. Let us consider the loop shown in Fig. 2. By applying Eqs. (3) and (4), we can obtain the general solution.

Hence, we extend the algorithm given by Tzen and Ni (Tzen and Ni, 1993) to form both DCH1 and DCH2. Their algorithm forms the convex hull as a ring connecting the extreme points (nodes of the convex hull). It starts with a large solution space and applies each half space from the set defined by Eqs. (3) and (4) and then they divide the iteration space to form two bounded dependence convex hulls. The extreme points of the convex hulls may have real coordinates because these points are intersections of a set of hyper-planes. Therefore, Punyamurtula and Chaudhary (1994) proposed an algorithm to convert these extreme points with real coordinates to extreme points with integer coordinates called the integer dependence convex hull (IDCH). An IDCH contains more accurate dependence information than a DCH. After constructing the initial DCH1 and DCH2, their algorithm checks if there are any real extreme points for the DCH1 and DCH2. If there are none, then DCH1 and DCH2 are IDCH1 and IDCH2, respectively.

By the above analysis, we can compute a dependence head iteration using an integer programming method combined with an inclusion property, a bounded property, and a forwarding property as indicated in Theorems 1, 4 and 5, respectively. Thus we can greatly reduce the complexity of our method by simplifying the detecting mechanism and skipping unnecessary constraints. Formally, these algorithms are shown in Fig. 7. The procedure Finding_IDCH_Flow_Head & Tail finds the minimum possible iteration space where dependence heads and tails may occur. It uses the concepts from Lemmas 1–7. Thus, the procedure Integer_Program_ij and Integer_Program_j finds dependence heads and tails of axis i and axis j only in the iteration space where the procedure

Procedure Finding_IDCH_Flow_Head&Tail
/* Input Parameter: A list of 18 half spaces
Output Parameter: I*DCH2*_Flow_Head & I*DCH1*_Flow_Tail */
Begin
HSs = { A list of 9 half spaces }
/* As indicated in Table 1
8 half spaces from Equation (3)
1 half space from Equation (5) for $d_i(x_1, y_1) > 0$ or ($d_i(x_1, y_1)=0$ and $d_j(x_1,y_1)>0$)*/
I*DCH1*_Flow_Tail = **call** Finding_IDCH(HSs);
HSs = { A list of 9 half spaces }
/* As indicated in Table 1
8 half spaces from Equation (4)
1 half space from Equation (6) for $d_i(x_2, y_2) > 0$ or ($d_i(x_2, y_2)=0$ and $d_j(x_2,y_2)>0$)*/
*IDCH2*_Flow_Head = **call** Finding_IDCH(HSs);
End Finding_IDCH_Flow_Head&Tail

Procedure Integer_Program_ij
/* Input parameter: *IDCH1_Flow_Head, IDCH2_Flow_Tail,* $f_i(x,y)$ for $1\le i \le 4$
Output parameter: the solution of IP: *minf, x, y,* " infeasible" if no solution */
Begin
Minf ρ Minimize $f_3(x,y)$
Subject to $f_1(x,y), f_2(x,y) \in$ *IDCH2_Flow_Tail*
$f_3(x,y), f_4(x,y) \in$ *IDCH1_Flow_Head*
where x,y are integers
End Integer_Program_ij
Procedure Integer_Program_j
/* Input parameter: *IDCH1_Flow_Head, IDCH2_Flow_Tail,* $f_i(x,y)$ for $1\le i \le 4$
Output parameter: the solution of IP: *minf, x, y,* " infeasible" if no solution */
Begin
Minf ρ Minimize $f_4(x,y)$
Subject to $f_1(x,y), f_2(x,y) \in$ *IDCH2_Flow_Tail*
$f_3(x,y), f_4(x,y) \in$ *IDCH1_Flow_Head*
where x,y are integers
End Integer_Program_j

Fig. 7. Algorithm of the enhanced integer programming mechanism.

```
Procedure Optimized_Dependence_Convex_Hull_Partitioning
/* To maximize parallelism from doubly nested loops with non-uniform dependences*/
Begin
/* to find the first iteration in the k^th partition (iP[k], jP[k]) */
  step 1: /* the process for dependence extraction */
      call Partition(fi(x, y) for 1≤ i ≤ 4, iP[], jP[], fk);
      /* To transform the loop into partitions with size,
       (jP[k-1], u2 ) plus ( iP[k+1] - iP[k]-1, u2 - l2 +1) plus (0, jP[k+1] - jP[k]) */
  step 2: /* code generation */
      If anti-exist = 1 /* there exists anti-dependence */
   /* generate code of copy-renaming */
      then Generate: " Dependence_Array' = Dependence_Array; "/* copying renaming */
      k = 1;
      While ( iP[k] [ u1 ) Do
       Iinc = iP[k+1] - iP[k];
      If (anti-exist = 1) then /* there exists anti-dependences*/
        { Generate:" Doall I = iP[k]-1, iP[k+1]
                    Doall J = l2 , u2
                                          If ((I= iP[k]-1) and (J<jP[k])) or((I= iP[k+1]) and (J>jP[k+1]))
                                                return;
                  ... = Dependence_Array'; /* copying renaming */
                           /* The instructions in the nested loop*/. "}
      else
          If (jP[k] = l2) then
            { Generate:" Doall I = iP[k], iP[k+1]-1
                   Doall J = l2 , u2
                                          If ((I= iP[k]-1) and (J<jP[k])) or((I= iP[k+1]) and (J>jP[k+1]))
                                                return;
                    /* The instructions in the nested loop*/. "}
        k = k +1;
      EndWhile
End Optimized_Dependence_Convex_Hull_Partitioning
```

Fig. 8. Algorithm of the ODCHP mechanism.

Finding_IDCH_Flow_Head & Tail exploits. Hence, the complexity of the main integer programming algorithm is the size of the dependence head iteration space multiply the size of the dependence tail iteration space where the procedure Finding_IDCH_Flow_Head & Tail exploits. The worst case of the algorithm is bounded by the general integer programming algorithm.

The ODCHP algorithm is described by the procedure called Optimized_Dependence_Convex_Hull_Plüeas shown in Fig. 8. The procedure called Partition in procedure Optimized_Dependence_Convex_Hull_Partitioning produces an array of the first iteration in each partition $(iP[k], jP[k])$ based on the concepts of inclusion property and the procedure Integer_Program_ij and Integer_Program_j. $fk - 1$ is the number of partitions for flow dependences. In the presence of anti-dependence, anti-exist is set to 1 and copy-renaming is implemented in step 2 according to Lemma 7. In addition, step 2 transforms the loop into partitions with size $(jP[k-1], u_2)$ plus $(iP[k+1] - iP[k] - 1, u_2 - l_2 + 1)$ and plus $(0, jP[k+1] - jP[k])$. Because the core procedure of the ODCHP algorithm is the procedure Integer_Program_ij and Integer_Program_j, its complexity is also bounded by these two procedures. By using both bounded property and forwarding property in it, the complexity is greatly reduced.

4. Performance evaluations

So far, we have illustrated the properties of the ODCHP mechanism in detail. In the following, performance evaluations are studied to practically verify effectiveness of our mechanism. The experimental programs include examples (Cho and Lee, 1997; Pean and Chen, 1999) discussed above, program models used by other related popular papers and some practical code segments. Table 3 shows four popular non-uniform loop models, which are widely used in several previous work (Tseng et al., 1992; Tzen and Ni, 1993; Shang et al., 1996; Tseng et al., 1996). Table 4 shows two popular code segments. They are Propogate code segment which are widely found in Linpack (Dongarra et al., 1979) and Swap code segment which serves as kernel of Fishpack Swarztrauber and Sweet, 1979). Linpack is a collection of Fortran programs for solving various types of linear systems and is supported on the Cray. Fishpack is a package of FORTRAN subprograms for solving those separable Elliptic partial differential equations and is also supported on the Cray.

```
Procedure Partition (fi(x, y) for 1 ≤ i ≤4, iP[], jP[], k);
/* get the solution of IP from Procedure Integer_Program_ij : minf, x, y, " infeasible" if no solution */
Begin
    lb1 = l1; ub1 = u1, lb2 = l2, ub2 = u2;
    k= 1; anti-exist = 0; iP[1] = l1; jP[1] = l2; inclusion = 0;
    While ( iP[k] [ u1 ) Do
    L1:Call Integer_Program_ij;
     Inclusion = 0;
     Call Inclusion_Check(Inclusion);
     If (Inclusion = 1) then goto L1;
     If " infeasible" then
        { If (k γ1) then /* the last partition */
          {k = k +1; iP[k] = u1 +1; jP[k] = l2; Retuen;}
     k = k +1; iP[k] = minf;
     /* Not in the same j dimension */
     If ( f1(x, y) γiP[k] ) then jB[k] = l2;
     /* In the same j dimension */
     else { jP[k] = l2; k = k + 1;
          iP[k] = iP[k-1]; jP[k] = f4(x, y);}
    L2: lb1 = iP[k]; lb2 = jP[k];
     /* Partition in the jth dimension and not the first one */
     If ( jP[k] γl2) then
      { L3: call Integer_Program_j;
          Inclusion = 0;
          Call Inclusion_Check(Inclusion);
          If (Inclusion = 1) then goto L3;
          k = k +1;
          If " infeasible" then
          { lb1 = lb1 + 1; lb2 = l2;
          iP[k] = lb1 ; jP[k] = l2; Continue;
          /* No Dependence in the j dimension */}
          iP[k] = lb1 ; jP[k] = minf; goto L2;
          /* Next j partition */ }
 EndWhile
End Partition

Procedure Inclusion_Check (Inclusion)
/* The property of this procedure is to discard dependences which is unnecessary in partitioning*/
Begin
 /* Tf(x, y) = Tail node of minf can be obtained by Equation (1) */
 Tfi(x, y) = f1(x, y);
 Tfj(x, y) = f2(x, y);
 /* This dependence is included in the previous partition and can be omitted
   as proved in Theorem */
 If (Tfi(x, y) < iP[k] ) or ((Tfi(x, y) = iP[k]) and (Tfj(x, y) < jP[k]))
 then inclusion = 1; Return;
End
```

Fig. 8. (Continued).

Table 3
The standard models

Model 1	Model 2	Model 3	Model 4
for $I = 1, N$ do	for $I = 1, N$ do	for $I = 1, N$ do	for $I = 1, N$ do
for $J = 1, M$ do	for $J = 1, M$ do	for $J = 1, M$ do	for $J = 1, M$ do
s1: $A(2I, 2J) = \dots$	s1: $A(I + J, 3I + J + 3) = \dots$	s1:$A(2J + 3, I + 1) = \dots$	s1: $A(3I, 5J) = \dots.$
s2: $\dots = A(J + 10, I + J + 6)$;	s2: $\dots = A(I + J + 1, I + 2J + 4)$;	s2:. $= A(I + J + 3, 2I + 1)$;	s2: $\dots\dots = A(I, J)$;
enddo	enddo	enddo	enddo
enddo	enddo	enddo	enddo

4.1. Overview of evaluation environment

The CONVEX Exemplar (Richardson, 1994) uses scalable parallel processing (SPP) technology, which is an implementation of massive parallel processing (MPP) technology expandable as customer needs increase. The processors

Table 4
The practical code segments

Propogate code segment	Swap code segment
DO $I = 1, Q$	DO $I = 1, 10$
DO $J = 1, R$	DO $J = 1, 10$
$AR(I,J) = AR(1,J)$	$Y(J,I) = Y(J,N+1-I)$
CONTINUE	CONTINUE
CONTINUE	CONTINUE

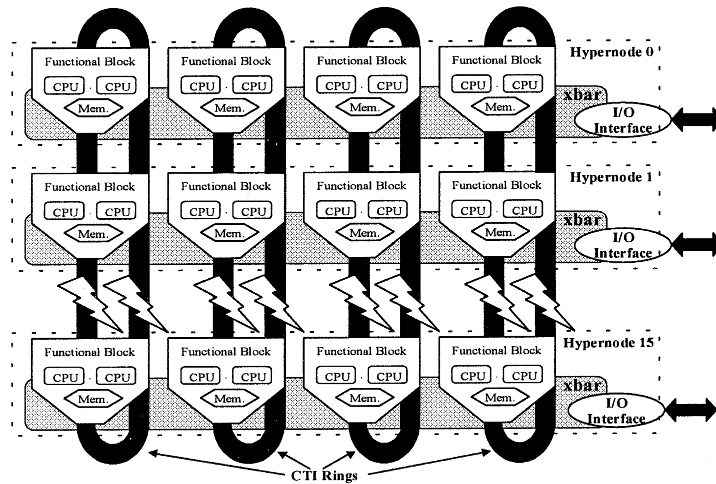

Fig. 9. Overview of Exemplar system architecture.

configured on the system are PA_RISC processors developed by Hewlett Packard; Exemplar configuration takes 4–128 PA-RISC processors. The configuration is shown in Fig. 9.

The fundamental building block in CONVEX SPP is hypernode as shown in Fig. 9. A hypernode is a symmetric multiprocessor (SMP), while an SPP is a group of hypernodes sharing a low-latency interconnect. In CONVEX SPP, each hypernode contains four CPU blocks, and each CPU block contains 2 PA-RISC processors associated with data and instruction caches, a CPU agent, as well as a CPU-private memory. For CONVEX SPP-1000, it has only one hypernode, and thus eight processors are available. Each hypernode also contains one or more hypernode private memories that can be accessed from any CPU within hypernode, but not accessible from others.

Nowadays, multiprocessor system is mostly configured in non-uniform memory access architecture, CONVEX SPP-1000 is a typical example. It is designed to achieve fast access latency of shared variables. Besides, it also supports a variety of directives and pragmas to ease the parallel programming. Users can easily familiarize with the parallel programming skill. C, C++ and Fortran compilers are supported as well. A friendly performance analyzer called CONVEX Performance Analyzer (CXpa) is provided to allow users profiling their running program. Being attracted by its amazing properties, such as popular architectural design, convenient in developing parallel program and easy profiling, we have eventually chosen CONVEX SPP-1000 as our target platform. We have constructed the ODCHP mechanism in the SUIF (Wilson et al., 1996) parallel compilation environment and then ran our experiments on a CONVEX SPP-1000.

In order to find whether our ODCHP technique performs better than existing partitioning mechanisms in a system with a large number of processors, we constructed a multiprocessor evaluation environment to measure their performance. Different mechanisms were implemented and the object code was evaluated on a simulator named SEESMA (a simulation and evaluation environment for shared-memory multiprocessor architecture) (Su et al., 1996), which is enhanced from MINT (Veenstra and Fowler, 1994). This system is a highly paralleled shared memory multiprocessor system environment. It is similar to the Exemplar system architecture with large number of processors.

4.2. Performance evaluation on program models

We have run the different mechanisms in both CONVEX SPP-1000 and SEESMA environments. Fig. 10 shows the speedup of our technique, ODCHP, versus ITRP, ITRP combined with Minimum Dependence distance Tiling (MDT),

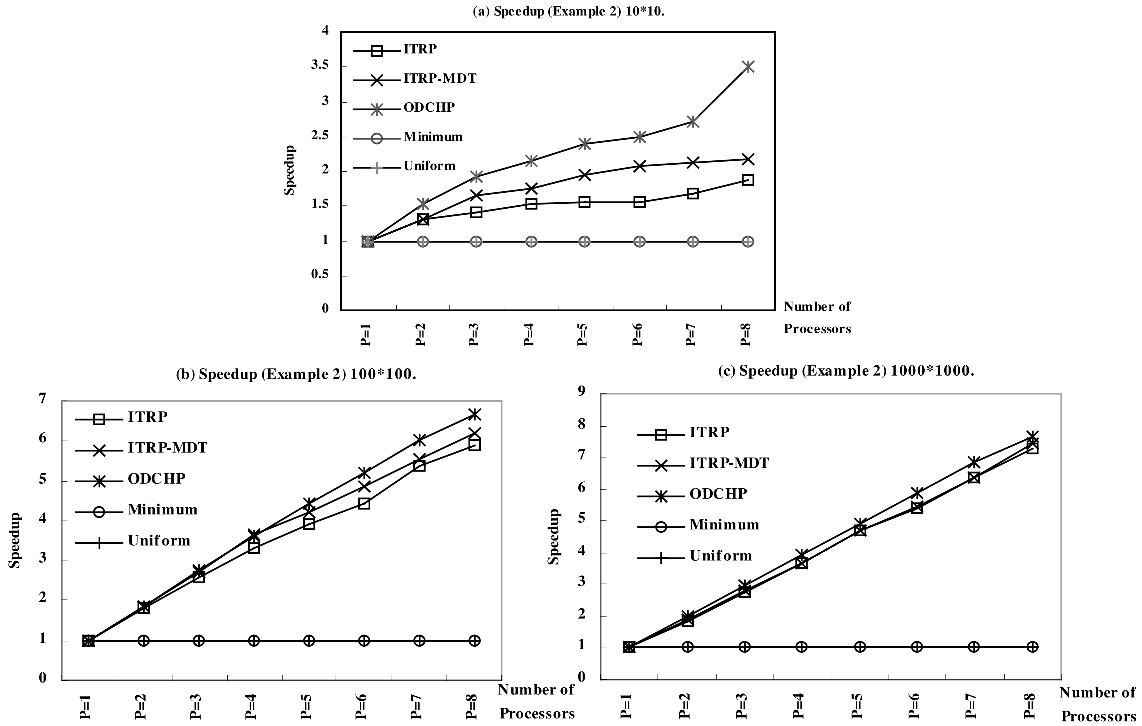

Fig. 10. Evaluation on CONVEX SPP-1000 for the program of Example 2.

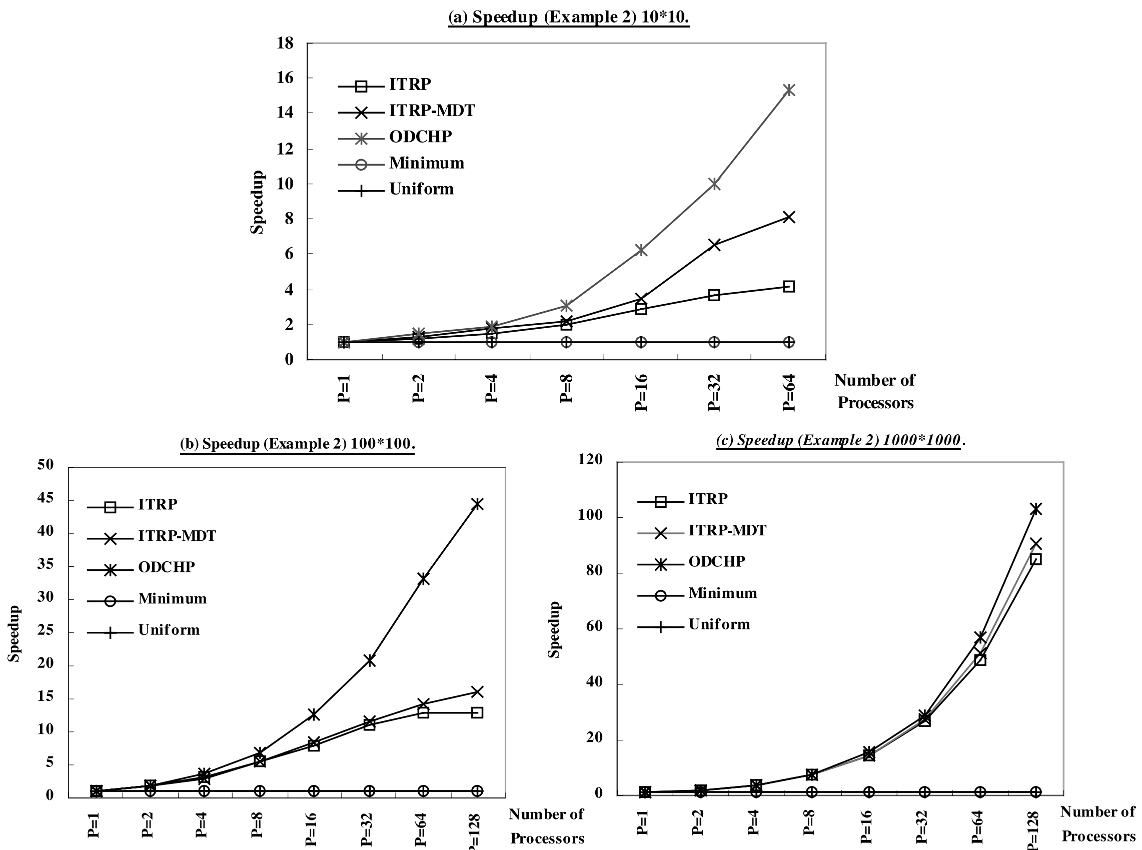

Fig. 11. Evaluation on the SEESMA environment for the program of Example 2.

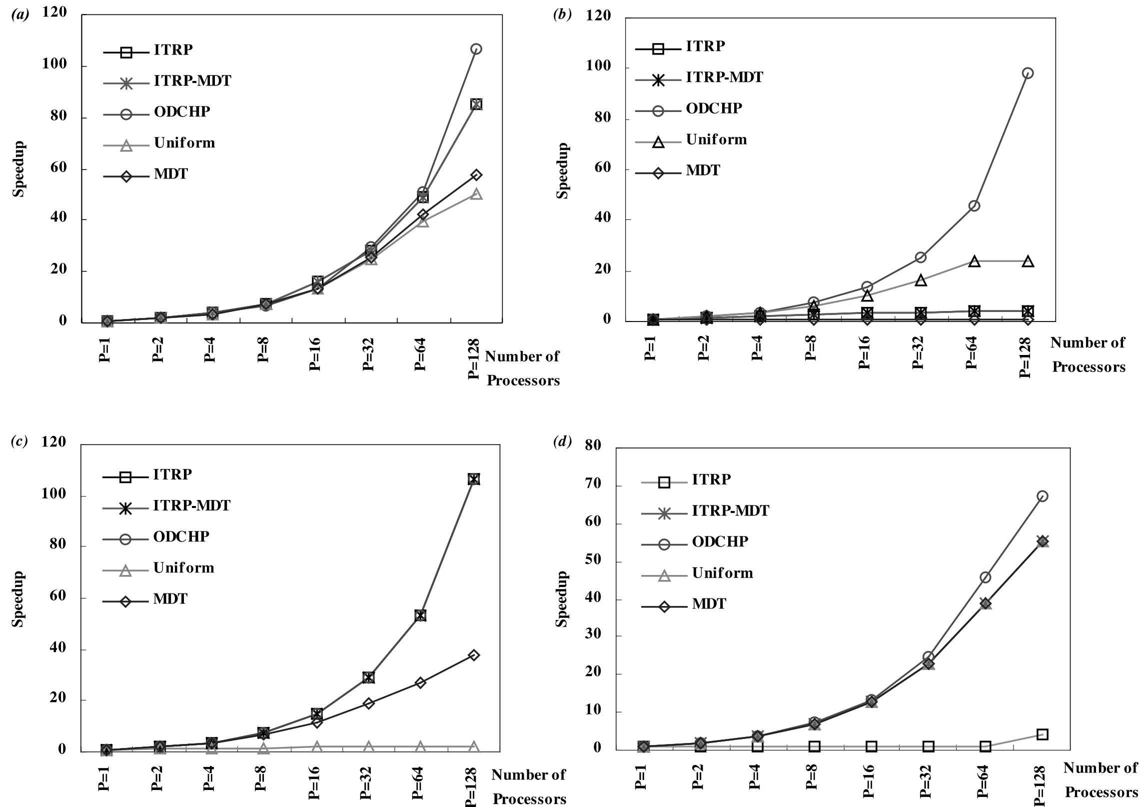

Fig. 12. Evaluation on the SEESMA environment for: (a) Model 1, (b) Model 2, (c) Model 3, (d) Model 4.

Minimum Dependence distance Tiling (MDT) and Uniformization techniques. Our technique delivers a better performance while the loop bounds of this example are set to 10, 100 and 1000 as shown in Fig. 10(a), (b) and (c), respectively. As the number of processors increases, the ODCHP mechanism performs even better than those of the other mechanisms. However, the speedup of different mechanisms is nearly linear, as shown in Fig. 10(c), due to the high degree of parallelism and the limited number of processors in the CONVEX SPP-1000 while the loop bound is set to 1000.

Fig. 11 shows the speedup comparisons in the SEESMA environment. Our technique delivers a better performance while the loop bounds for this example are set to 10, 100 and 1000 as shown in Fig. 11(a), (b) and (c), respectively. As the number of processors increases to 128, the performance of the ODCHP is much better than the other mechanisms because a larger number of processors can exploit a higher degree of parallelism.

Furthermore, Fig. 12 shows the speedup comparisons in the SEESMA environment for popular program models. Our technique delivers a better performance while the loop bounds for these models are set to 100 as shown in Fig. 12(a), (b), (c) and (d), respectively. As the number of processors increases to 128, the performance of the ODCHP is much better than the other mechanisms because a larger number of processors can exploit a higher degree of parallelism. For program model 1 and model 3, because there are only anti-dependence in the iteration space, our mechanism, ITRP and ITRP–MDT can detect effectively and exploits nearly the same parallelism. The MDT and uniformization method perform poorly in these two program models, because they cannot avoid anti-dependences. For program model 2, because the inclusion property can hide much unnecessary flow dependences, our mechanism performs extremely better than other mechanisms. For program model 4, because ITRP has large serial region in the iteration space, it also has poor performance. However, our mechanism not only performs better than other mechanisms but it also has low compilation time due to the bounded property and the forwarding property. The bounded property shows that the flow dependence tail exists only in a small region of the iteration space, while the forwarding property greatly reduces the checking area as the processing of our partitioning algorithm.

4.3. Performance evaluation on practical code segments

In order to show our mechanism is effective in real programs, we implement our mechanism in practical code segment in this section. Fig. 13 shows the speedup comparisons in the SEESMA environment. For Propogate code

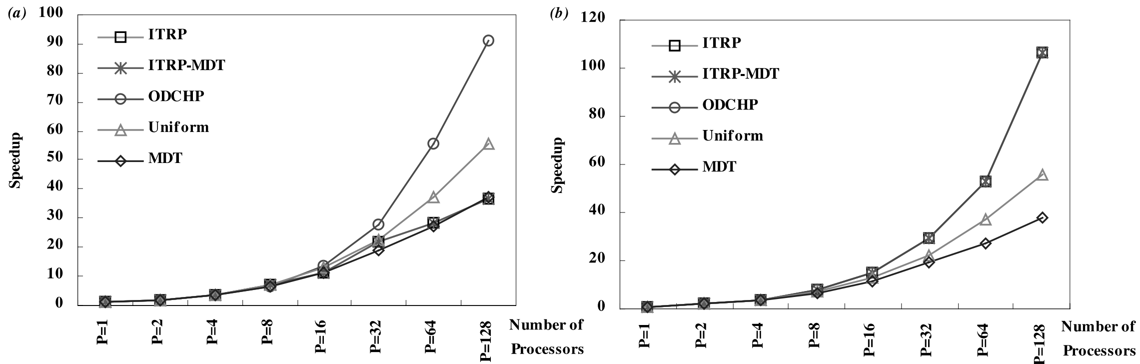

Fig. 13. Evaluation on the SEESMA environment for: (a) propogate, (b) swap code segments.

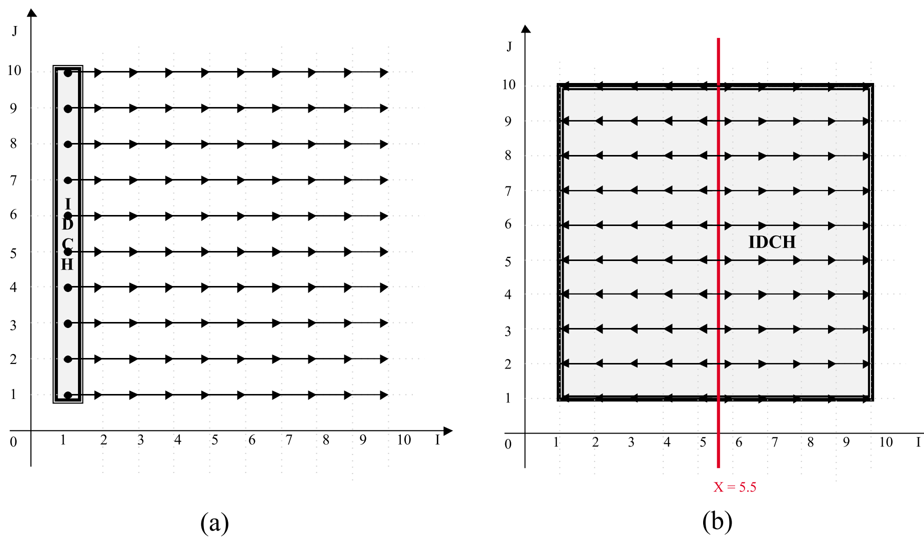

Fig. 14. Dependence graph: (a) propogate code segment, (b) swap code segment.

segment, our mechanism finds that the flow dependence tail only covers a small region of the iteration space due to bounded property as shown in Fig. 14(a). And after the first partitioning step, all the iteration space which is not partitioned is set to fully parallel due to inclusion property. Thus, in this program code segment our mechanism shows dramatic better performance than other mechanisms as shown in Fig. 13(a). For the Swap code segment, because the ITRP, ITRP–MDT and our ODCHP mechanisms partition the iteration space into two parallel regions as shown in Fig. 14(b), their performance is nearly the same and all effective as shown in Fig. 13(b).

In conclusion, we can find that our mechanism is better than any other existing popular partitioning mechanism. As the number of processor increases, the performance of the ODCHP is much better than the other mechanisms because a larger number of processors can exploit a higher degree of parallelism exploits by it.

5. Conclusion and future work

In this paper, we have studied the problem of transforming nested loops with non-uniform dependences in order to maximize parallelism by using a new scheme called ODCHP, which is based on convex hull theory. The ODCHP can easily divide the iteration space into several parallel regions using DCH bounds or lines of the dependence vector that are known in advance. Moreover, by combining the ODCHP with the properties of inclusion, bounded and forwarding, the complexity of the ODCHP can be greatly reduced and the parallelism exploited is more pronounced. In comparison with other partitioning methods based on convex hull theory or basic dependence theory, the ODCHP has

several advantages such as simple partitioning and high speedup. Comparison with other popular methods, our scheme shows dramatic better performance in not only popular program models but also real program code segments.

In the future, we will further extend the ODCHP method to multiple-dimensional iteration space and establish an appropriate dynamic data allocation mechanism to reduce data conflict. Our method will all be extended to work with more than one dependence in the nested loops as well.

References

Berger, M.J., 1987. A partitioning strategy for nonuniform problems on multiprocessor. IEEE Trans. Comput. C 36 (5), 570–580.

Banerjee, U., 1988. Dependence Analysis for Supercomputing, Kluwer Academic Publishers, 101 Philip Drive, Assinippi Park, Norwell, MA 02061, Boston.

Chen, D.K., Yew, P.C., 1996. On effective execution of nonuniform DOACROSS loop. IEEE Trans. Parallel Distrib. Sys. 7 (5), 463–476.

Cho, C.K., Lee, M.H., 1997. A Loop Parallelization Method for Nested Loops with Non-uniform Dependence. Dietz, Hank et al. (Eds.), International Conference on Parallel Processing, Los Alamitos, CA, 314–321.

Dongarra, J.J., Moler, C.B., Bunch, J.R., Stewart, G.W., 1979. LINPACK Users' Guide by SIAM, Philadelphia.

Ju, J., Chaudhary, V., 1996. Unique Sets Oriented Partitioning of Nested Loops with Non-uniform Dependences, Pingali K. (Ed.), International Conference on Parallel Processing, III, Los Alamitos, CA, 45–52.

Pean, D.L., Chen, C., 1999. An Optimized Loop Partition Technique for Maximize Parallelism of Nested Loops with Non-uniform Dependences. Lee, S.L. (Ed.), The Fifth Workshop on Compiler Techniques for High-Performance Computing, Chiayi, Taiwan, ROC, 18–19 March, 158–171.

Punyamurtula, S., Chaudhary, V., 1994. Minimum Dependence Distance Tiling of Nested Loops with Non-uniform Dependences, Proceedings of the Sixth IEEE Symposium on Parallel and Distributed Processing, 74–81.

Punyanurtula, S., Ju, J., Chaudhary, V., Roy, S., 1997. Compile time partitioning of nested loop iteration spaces with non-uniform dependences. Parallel Alg. Appl. 12, 113–141.

Richardson, 1994. Exemplar Architecture. Convex Computer Corporation, TX, USA.

Shang, W., Hodzic, E., Chen, Z., 1996. On uniformization of affine dependence algorithms. IEEE Trans. Comput. 7, 827–840.

Shen, Z., Li, Z., Yew, P.C., 1989. An Empirical Study on Array Subscripts and Data Dependencies, Ris, Fred and Kogge, Peter M. (Ed.), International Conference on Parallel Processing, Pennsylvania State University, pp. 145–152.

Su, J.P., Wu, C.C., Chen, C., 1996. Reducing the Overhead of Migratory-Shared Access for the Linked-Based Directory Coherent Protocols in Shared Memory Multiprocessor Systems. In: Wen-Shyong, Lionel M. Ni (Eds.), Proceedings of ICS'96 on Computer Architecture, Kaohsiung, Taiwan, ROC, pp. 160–167.

Swarztrauber, P.A., Sweet, R.A., 1979. Efficient fortran subprograms for the solution of separable elliptic partial differential equations. ACM Trans. Math. Software 5 (3), 352–364.

Tseng, S.Y., King, C.T., Tang, C.Y., 1992. Minimum dependence vector set: a new compiler technique for enhancing loop parallelism. In: International Conference on Parallel and Distributed Systems, HsinChu, Taiwan, National Tsing Hua University, 340–346.

Tseng, S.Y., King, C.T., Tang, C.Y., 1996. Profiling Dependence Vectors for Loop Parallelization. In: Proceedings of International Parallel Processing Symposium, Los Alamitos, CA, pp. 23–27.

Tzen, T.H., Ni, L.M., 1993. Dependence uniformization: a loop parallelization technique. IEEE Trans. Parallel Distrib. Syst. 4, 547–558.

Veenstra, J.E., Fowler, R.J., 1994. MINT Tutorial and User Manual, Technical Report, 452, University of Rochester, New York.

Wilson, R., Lam, M.S., Hennessy, J., 1996. An Overview of the SUIF Compiler System, Computer Systems Lab, Stanford University.

Zaafrani, A. and Ito, M., 1994. Parallel Region Execution of Loops with Irregular Dependences. In: Agrawal, Dharma P. (Ed.), Proceedings of the International Conference on Parallel Processing, London, 15–19 August, pp. 11–19.

Zima, H., Barbara, C., 1990. Supercompilers for Parallel and Vector Computers. Addison Wesley, New York.