A Release Combined Scheduling Scheme for Non-Uniform Dependence Loops

DER-LIN PEAN, HUEY-TING CHUA AND CHENG CHEN
Department of Computer Science and Information Engineering
National Chiao Tung University
Hsinchu, 300 Taiwan

In general, synchronization mechanisms can be used to preserve dependence constraints in any nested loop, and can be combined with a loop scheduling scheme to form a uniform framework to obtain the correct execution order and balance workload distribution. Most current scheduling mechanisms cannot handle non-uniform dependence loops. In this paper, we propose a new combined scheduling scheme called Release Combined Scheduling for Non-uniform Dependence Loops (RCS) to schedule non-uniform dependence doubly-nested loops in multiprocessor systems. It combines both static and dynamic scheduling mechanisms in order to optimize the system performance. In our approach, initialisation of a set of scheduling information is based on the concept of the minimum dependence distance. During runtime, scheduling information is used to adjust the number of parallelizable iterations. Our method is able to discover more parallelism from a given non-uniform dependence doubly-nested loop than is possible with previous approaches. The experimental results show that the RCS method reliably exploits parallelism and outperforms most of the existing non-uniform dependence loop scheduling schemes by 20.29%, on average.

Keywords: loop scheduling, multiprocessor, non-uniform dependence, hopping gate, hopping distance, synchronization, barrier

1. INTRODUCTION

The multiprocessor (MP) is one of the most important computer architectural designs addressing the demand for fast computational performance. To fully utilize an entire system, the workload on all the processors should be as evenly distributed as possible [1]. Ever since loops were identified as a major source of parallelism, the problem of loop scheduling has been studied in order to achieve equal and fair workload distribution, in addition to reducing synchronization, communication, and thread management overheads [2] in computer systems. In spite of the commonality of DOALL loop scheduling schemes [1-6], there are few non-uniform dependence loop scheduling schemes.

Non-uniform dependence loops, which exhibit irregular dependence on the iteration level, are mainly as a result of coupled subscripts [7], and Fortran numerical packages such as Linpack [19], Eispack [20], Itpack [21], and Fishpak [22] are typical examples of programs containing non-uniform dependence loops. These are library packages and can be frequently called on by a user in scientific and engineering computational programs. Henceforth, it is important to develop an efficient loop scheduling scheme for them, as shown in Fig. 1.

Received April 21, 2000; revised September 1, 2000; accepted December 5, 2000.
Communicated by Chu-Sing Yang.

```
        for I = 1, $U_I$
            for J = 1, $U_J$
                …
$S_d$:  A($f_1$(I, J), $f_2$(I, J)) = ...
                …
$S_u$: ... = A($f_3$(I, J), $f_4$(I, J))
                …
            endfor
        endfor
```

Fig. 1. Program model.

Among the existing approaches, the Staggered Distribution method [8] performs outstandingly in data-flow machines but is not adaptable for shared memory multiprocessor systems. On the other hand, in most of the current systems, acceptable scheduling schemes [9-13] introduce a significant delay overhead when preserving dependence correctness. In this paper, we develop a non-uniform dependence loop scheduling scheme, which is free from any evident delay overhead and is capable of achieving dynamically extracting parallelism. This newly developed method is called the release combined scheduling (RCS) method. The target platform is the shared-memory or distributed shared-memory MP system. It is not only able to exploit more parallelism than the compiler partitioning technique, but also can avoid causing any synchronization overhead or waiting time.

The RCS method originated in loop tiling techniques [14, 15] but differs from them in its method of maintaining parallelizable iterations. In loop tiling, a number of parallelizable iterations on each tile are determined during compilation time. During execution, this number is fixed, and before all the iterations in the current tile have been executed, no idle processors are allowed to execute iterations from subsequent tiles. To preserve the correct execution order, barrier synchronization is inserted at the end of each tile. In the RCS method, in contrast, a number of parallelizable iterations are initialized during compilation time, but these are updated after a certain number of dependence tails have been executed, such that parallelism can be exploited more dynamically. On the other hand, because we only derive scheduling information, we do not really partition the loop during the compilation stage. In fact, our method can exploit more parallelism while eliminating synchronizations. We propose a strategy that we can use to analyse statical dependent information in non-uniform dependence loops. We then can dynamically use this information to synchronize parallel-executed iterations such that improved parallel execution performance for non-uniform dependence loops can be obtained. To simplify our discussion, the program model shown in Fig. 1 is used for description and preliminary evaluation, as has been widely discussed by several other authors [10-16]. Performance evaluations were carried out using a CONVEX SPP-1000. According to our results, regardless of the number of processors available, the RCS method substantially reduces the delay overhead as well as multi-barrier synchronization. When eight processors are used and real benchmarks are considered, the RCS method performs better than the Index Synchronization Method [10] by 15.25%, on average. If our method is compared with loop partition techniques [14, 15] in terms of performance, then it is superior by 25.33%, on average. These amazing findings inspired for us to further extend the method.

The organization of this paper is as follows. Related work is discussed in section 2. Section 3 presents the basic concepts and principles of the RCS method. Section 4 provides further generalization of the method. Section 5 gives performance evaluations. Section 6 is a conclusion.

2. RELATED WORK

Conventionally, the performance of a loop scheduling mechanism is determined by five factors: (i) workload balancing; (ii) the scheduling overhead; (iii) the communication overhead; (iv) the thread management overhead; and (v) the synchronization overhead [2]. Among these, the synchronization, scheduling, and thread management overheads are our major concerns. The execution time of a loop body is assumed to be consistent for all iterations, so workload imbalance will only be caused by the scheduling mechanism.

Several previous works have been devoted to effectively scheduling non-uniform dependence loops, including: the Index Synchronization Method (ISM) [10]; the Group Synchronization Method [11]; and the Static Strip Scheme (SSS) [12,13]. ISM was proposed to schedule non-uniformly dependent two-way nested loops after partitioning using dependence uniformization method. The basic idea is to serially execute an inner loop while executing an outer loop and synchronization concurrently. This is done using a globally shared array, which incorporates a delayed operation. The performance of ISM will probably be constrained by the dependence uniformization method because an additional delay overhead is introduced. One of the variations of ISM is the Group Synchronization Method. Here also, a delay overhead is inevitable, and it may restrict the performance gain.

SSS is another approach that is also associated with the dependence uniformization method. A strip is a group of iterations that are to be executed sequentially. Cross strip dependences are preserved through explicit synchronization primitives, for example, post&wait. As the name implies, SSS is classified as static scheduling. Once again, it is constrained by the dependence uniformization technique, and different synchronization primitives result in distinct performance behaviours.

Another intuitive approach is scheduling the tiled loops, since a DOACROSS loop can be partitioned into a few totally parallelizable tiles. Dependence analysis is handled using loop partition techniques during compilation time. Any existing chunk size control functions can be applied to guide the scheduling process, such as Pure Self Scheduling, Chunk Self Scheduling, or Guided Self Scheduling [1]. Barrier synchronization is inserted at the end of each tile. In spite of the intuitiveness and simplicity of this method, scheduling of tiled loops is restricted by both the loop partition techniques and scheduling schemes. Multi barrier synchronization instructions are unavoidable at the end of each tile, and a poor choice of scheduling scheme can incur a further apparent overhead.

We will also introduce two related partitioning mechanisms for the purpose of performance evaluation. The minimum dependence distance tiling method [11, 14] exploits parallelism by using the minimum computed distances from the dependence vectors of the Integer Dependence Convex Hull (IDCH) extreme points. The minimum distances are used to partition the iteration space into tiles of regular size and shape. The Parallelization Part Splitting (PPS) [15] mechanism splits up the parallelization part of the iteration space for the purpose of parallel execution. As the parallelization part always occupies most of the iteration space, it can be partitioned in advance based on the concept of loop splitting in the uniform dependence loop partitioning method. This breaks the iteration space into three regions: two non-IDCH regions that are parallelizable and an IDCH region that must be incorporated using either a MDT or dependence uniformization method.

In the following section, we will present our new RCS method, which essentially separates the iteration space into either parallelizable or sequential blocks, while eliminating the delay instructions, synchronization primitives, and multi-barrier synchronization stages.

3. BASIC CONCEPTS AND PRINCIPLES OF THE RCS METHOD

Conventionally, exact techniques for analysing cross-iteration dependences are based on solving Diophantine equations [25] corresponding to array subscript expressions [10]. For a nested loop, as shown in Fig. 1, a dependence exists between statement S_d and statement S_u, if they both refer to the same element of array A, or if $f_1(i_1, j_1) = f_3(i_2, j_2)$, and $f_2(i_1, j_1) = f_4(i_2, j_2)$. In [14], it is stated that a cross-iteration dependency exists between S_d and S_u only if there is a set of integer solutions (i_1, j_1, i_2, j_2) to the Diophantine equation (1) and the following system of linear inequalities (2):

$$\begin{cases} f_1(i_1, j_1) = f_3(i_2, j_2), \\ f_2(i_1, j_1) = f_4(i_2, j_2), \end{cases} \quad \text{the Diophantine Eq.(1)}$$

$$\begin{cases} 1 \le i_1 \le U_I, \\ 1 \le j_1 \le U_J, \\ 1 \le i_2 \le U_I, \\ 1 \le j_2 \le U_J. \end{cases} \quad \text{linear inequalities (2)}$$

The Banerjee algorithm [26] can be applied to find general solutions (i_1, j_1, i_2, j_2) to equation (1). These general solutions can be expressed in terms of two integer variables x and y, except when $f_1(i_1, j_1) = f_3(i_2, j_2)$ is parallel to $f_2(i_1, j_1) = f_4(i_2, j_2)$, in which case, the solution is obtained in terms of three integer variables. Here, we consider the general solution with only two variables, $(i_1, j_1, i_2, j_2) = (S_1(x, y), S_2(x, y), S_3(x, y), S_4(x, y))$, and the dependence vector set is defined as either

$D(x, y) = \{(S_3(x, y)\text{- } S_1(x, y)), (S_4(x, y)\text{- } S_2(x, y))\}$ or
$d_i(x, y) = [S_3(x, y)\text{- } S_1(x, y)]$, $d_j(x, y) = [S_4(x, y)\text{- } S_2(x, y)]$.

Eq. (2) can then be rewritten as

$$\begin{cases} 1 \le S_1(x, y) \le U_I, \\ 1 \le S_2(x, y) \le U_J, \\ 1 \le S_3(x, y) \le U_I, \\ 1 \le S_4(x, y) \le U_J. \end{cases}$$

These inequalities form a convex polyhedron or Dependence Convex Hull (DCH). This can be represented as

$$\begin{aligned} \mathrm{DCH} = &\{(x, y) \mid 1 \le S_1(x, y) \le U_I\} \\ &\cap \{(x, y) \mid 1 \le S_2(x, y) \le U_J\} \\ &\cap \{(x, y) \mid 1 \le S_3(x, y) \le U_I\} \\ &\cap \{(x, y) \mid 1 \le S_4(x, y) \le U_J\}. \end{aligned}$$

A DCH-forming algorithm has been proposed by Tzen and Ni [10]. If the DCH is empty, then either there are no integer solutions (i_1, j_1, i_2, j_2) satisfying equation (2), or statements S_d and S_u in the program model are cross-iteration independent. Otherwise, dependence vectors exist for that iteration space. The extreme points of the convex hull may have real coordinates because these points are simply intersections of a set of hyperplanes. An algorithm for converting real coordinates into integer coordinates was proposed in [14], and the result is called the IDCH.

Thus far, the general Diophantine equations and dependence vector set of the program model shown in Fig. 1 can be expressed as

$$\begin{cases} i_1 = x \\ j_1 = y \\ i_2 = s_1 x + s_2 y + s_3 \\ j_2 = t_1 x + t_2 y + t_3 \end{cases} \quad \text{Diophantine equations (3)} \quad s_1, s_2, s_3, t_1, t_2, t_3 \in \mathbf{R}$$

$$\begin{cases} d_i(x, y) = (s_1 - 1)x + s_2 y + s_3 \\ d_j(x, y) = t_1 x + (t_2 - 1)y + t_3 \end{cases} \quad \text{dependence vector functions (4).}$$

Example 1 (L_1):

```
for I = 1, 10
     for J = 1, 10
Sd:A(3I, 5J) = …
Su:…= A(I, J)
     endfor
endfor
```

Fig. 2. Array assignment pattern of L_1.

L_1 is an example of a non-uniform dependence loop and is shown in Fig. 2. Its dependence graph is shown in Fig. 3. The Diophantine equation set of L_1 is

$$\begin{cases} i_1 = x, \\ j_1 = y, \\ i_2 = 3x, \\ j_2 = 5y. \end{cases}$$

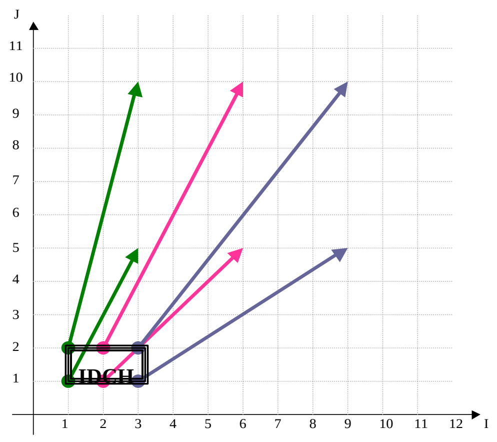

Fig. 3. Dependence graph of L_1.

The dependence vector set is {(3-1)x, (5-1)y } = {2x, 4y}, and the DCH is located at the intersection of eight half spaces [25], DCH(L_1) = $\{(x, y) \mid 1 \leq x \leq 10\} \cap \{(x, y) \mid 1 \leq y \leq 10\} \cap \{(x, y) \mid 1 \leq 3x \leq 10\} \cap \{(x, y) \mid 1 \leq 5y \leq 10\}$. It forms a rectangular shape in the dependence graph as shown in Fig. 3. If we express the dependence vector as an arrow, then the arrowhead is called the dependence head (destination), and the arrow's tail is called the dependence tail (source). For this example, iterations within the IDCH are called dependence tails. To preserve program correctness, iterations within the IDCH must be executed before their respective dependence heads are executed.

A feature of the RCS method is that a number of parallelizable iterations can hop across subsequent iterations as a result of the relaxation of dependence constraints. The basic idea of the RCS method is illustrated in Figs. 4 and 5. The letter M in these figures represents the number of parallelizable iterations in the two execution stages, as shown in Figs. 4 and 5. This is formally defined in Definition 3.3 below. The hopping gate is a specific iteration on which an iteration in a subsequent tile depends.

To achieve our goal, we allow for two global variables which can be used to track the dependence constraints: the hopping gate and hopping distance. The hopping gate is set to monitor the hopping occasion, while the hopping distance is used to define the distance that needs to be hopped. In Fig. 4, the hopping gate is initially set to iteration (1,2), and the hopping distance covers the first 20 iterations, ranging from (1,1) to (2,10). Once iterations (1,1) and (1,2) have been executed, the 30 iterations from (3,1) to (5,10) can be carried out. Afterwards, the hopping distance has been incremented by 30 iterations, and the hopping gate has been shifted by 10, as shown in Fig. 5. If any iterations before (2,2) have also been executed, the 30 more iterations ranging from (6,1) to (8,10) will be added to the current hopping distance.

3.1 The RCS Method for Growing Pattern Loops

The dependence vectors of a loop may increase backwards or forwards along a particular loop dimension. In [15], this type of loop is defined as having a growing pattern in

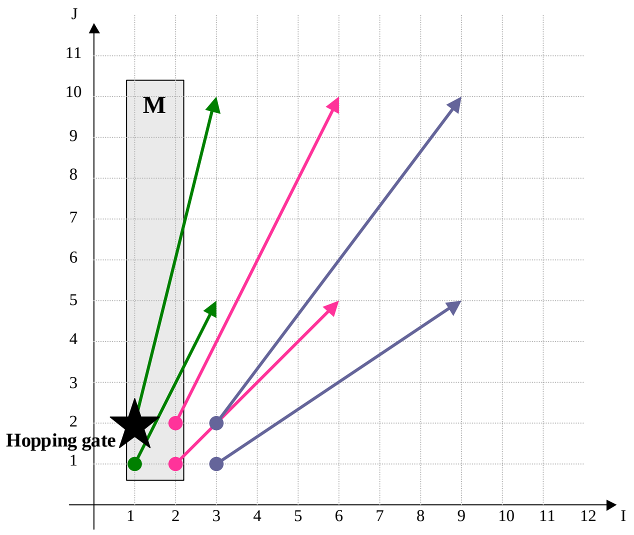

Fig. 4. The hopping gate and M before execution.

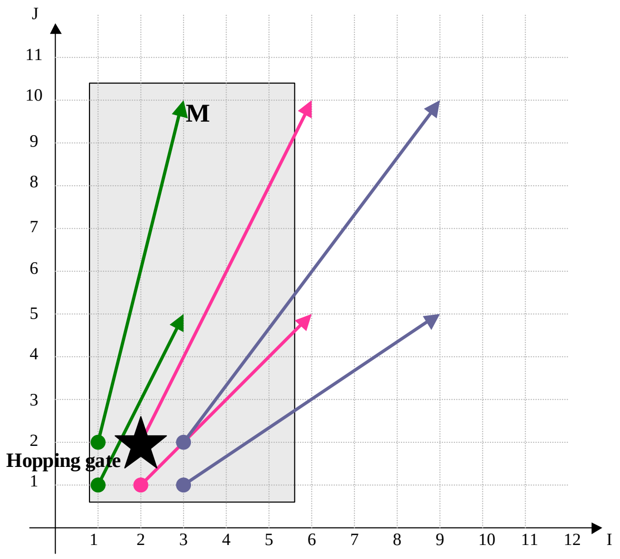

Fig. 5. The first hop of M.

the loop dimension. The authors in [15] took advantage of this feature and tiled the loop according to the dependence vector. As long as the dependence distance is increasing, the tile size will definitely grow, thus improving the parallelism. We use this specific loop in our illustration of the RCS method because of its simplicity and uniqueness. A formal definition of, and the condition for the growing pattern are given below.

Definition 3.1 (Growing Pattern Loop (GPL)): Given a two-way nested loop L, let L_I, and U_I be the lower and upper bounds on the loop index I, respectively, and let the dependence vector function V_I, be real on (L_I, U_I). If, $L_I < x < y < U_I$, then this implies that $|V_I(x)| \leq |V_I(y)|$ and the pattern of V_I has a growing pattern on (L_I, U_I), where L is called the Growing Pattern Loop (GPL). ❑

If a loop is a GPL, then $V_I = \{d_i(x, y) \mid d_i(x,y) = (s_1 - 1)x + s_3$ and $|(s_1 - 1)x_1 + s_3| \leq |(s_1 - 1)x_2 + s_3|$ for any $L_I \leq x_1 < x_2 \leq U_I\}$. L_1 in Example 1, is a typical example since the dependence vector set of L_1 is $\{2x, 4y\}$, or $d_i = 2x$. For any $1 \leq x_1 < x_2 \leq 10$, $|2x_1| < |2x_2|$ always holds; therefore, L_1 is said to have a growing pattern on loop index I. The following lemma further specifies the condition of a growing pattern.

Lemma 3.1: Let I and J be index variables. The pattern of the dependence vector is a growing pattern if the dependence vector functions V_I and V_J, contain only the linear functions of loop index variables I and J, respectively. ❑

Proof: If the dependence vector contains only one variable I, then it must be a*I + c, where $a,c \in \boldsymbol{R}$. We can differentiate this so that d(a*I + c)/dI = a. Clearly, $|a*x + c| \leq |a*y + c|$, $\forall$ x, y $\in \boldsymbol{Z}$ and $x < y$. Thus, the pattern of the dependence vector function V_I is a growing pattern. The dependence vector function V_J can also be proven to be a growing pattern in a similar manner. ❑

Furthermore, if the loop has a growing pattern on loop index I, then the corresponding dependence vector function carries a single index variable, I. This is formally expressed as $V_I = \{d_i(x,y) \mid d_i(x,y) = (s_1 - 1)x + s_3$ and $|(s_1 - 1)x_1 + s_3| \leq |(s_1 - 1)x_2 + s_3|, \forall L_I < x_1 < x_2 < U_I\}$. The growing pattern dependence vectors can be anti-dependent or flow-dependent. As any anti-dependence can be eliminated by means of array renaming, we are only concerned with flow-dependence loops with the growing pattern in the outermost loop level. These are called Backward Growing Pattern Loops (BGPLs), and a formal definition is given below.

Definition 3.2 Backward Growing Pattern Loop (BGPL): Given a two-way nested loop L, as shown in Fig. 1, the dependence vector function on the outermost loop level V_I, is positive and real. If $V_I(x) \leq V_I(y)$ for any x and y satisfy $1 < x < y < U_I$, then L is said to have a backward growing pattern on loop dimension I, and L is called the Backward Growing Pattern Loop (BGPL).

Also, if a loop has a backward growing pattern on the outermost loop level I, then $V_I = \{d_i(x, y) \mid d_i(x, y) = (s_1 - 1)x + s_3$ and $[(s_1 - 1)x_1 + s_3] \leq [(s_1 - 1)x_2 + s_3]$ for any $L_I \leq x_1 < x_2 \leq U_I\}$. Consider Example 1 again. L_1 has a growing pattern on loop index I, and it is also a BGPL because the dependences are all flow-dependent.

Lemma 3.2: Assume that the dependence vector function of the loop is V_I, and that the loop has the pattern of a BGPL for loop index I; then, the maximum dependence distance of the loop index variable $i \in$ I must be $V_I(i)$.

Proof: The pattern of a dependence vector function V_I for loop index I is the backward growing pattern. As $\forall\, x, y \in \boldsymbol{N}$, and $x < y$, this implies that $V_I(x) \le V_I(y)$. The maximum dependence distance of the loop index variable $i \in$ I must be $V_I(i)$ because $V_I(i) \le V_I(x)$, $\forall\, x \in (i + 1, i + V_I(i) - 1)$. ❑

Theorem 3.1 shows that if we use conventional partitioning methods to partition the iteration space, then the maximum number of parallelizable iterations we can exploit according the concept of the BGPL can be calculated.

Theorem 3.1: Assume that the pattern of the dependence vector function V_I for loop index I is a growing pattern, and that the dependence vector function is $V_I = s_1{*}I + s_3$, where $s_1, s_1, s_3 \in \boldsymbol{R}$, and I is the variable of the corresponding loop index. The parallelizable iterations P_n for the nth stage are $|s_1{*}(K_{n-1} + 1) + s_3|$, where K_n is equal to $\left|\sum_{i=0}^{n-1}(s_1+1)^i(s_1+s_3)\right|$.

Proof: The theorem can be proved by induction on k. We begin our induction at 1.

Basis: k = 1. Then, P_1 is equal to the absolute value of the dependence vector $V_I(1)$ and $P_1 = |V_I(1)| = |s_1 + s_3|$.

Induction: k > 1. By the induction hypothesis,

$$P_k = |s_1{*}(K_{k-1}+1)+1| = \left| s_1 * \left(\left|\sum_{i=0}^{k-2}(s_1+1)^i(s_1+s_3)\right| +1\right) + s_3\right|.$$ We must show that

$$P_{k+1} = |s_1{*}(K_k+1)+ s_3| = \left| s_1{*}\left(\left|\sum_{i=0}^{k-1}(s_1+1)^i(s_1+s_3)\right| +1\right) + s_3\right|.$$

Since $P_{k+1} = V_I(Kk + 1) = |s_1{*}(K_k + 1) + s_3|$, it follows that

$$\begin{aligned}
P_{k+1} &= |s_1{*}(K_{k-1} + 1) + s_3| \\
&= |s_1{*}(K_{k-1} + s_1(K_{k-1} + 1) + s_3 + 1) + s_3| \\
&= |s_1{*}((s_1 + 1)K_{k-1} + s_1 + s_3 + 1) + s_3| \\
&= \left|s_1{*}\left((s_1 + 1)\left(\left|\sum_{i=0}^{k-2}(s_1+1)^i(s_1+s_3)\right|\right) + s_1 + s_3 + 1\right) + s_3\right| \\
&= \left|s_1{*}\left((s_1 + 1){*}\left(\left|\sum_{i=0}^{k-2}(s_1+1)^{i+1}(s_1+s_3)\right|\right) + (s_1 + s_3) + 1\right) + s_3\right| \\
&= \left|s_1{*}\left(\left|\sum_{i=0}^{k-1}(s_1+1)^i(s_1+s_3)\right| + 1\right) + s_3\right| \\
&= |s_1{*}(K_k + 1) + s_3| \\
&= V_I(Kk + 1).
\end{aligned}$$

Thus, the proof is complete. ❑

Now, we will give a simple example to show how we can detect whether a loop is a GPL or not. We will consider loop L_1 in this example.

Example 2: In Fig. 2, loop L_1 is an example of a non-uniform dependence loop, and its dependence graph is shown in Fig. 3. The Diophantine equation set is $\{i_1 = x, j_1 = y, i_2 = 3x, j_2 = 5y\}$, and the dependence vector set is $\{2x, 4y\}$. In loop L_1, $1 \le i_1 < i_2 \le 10$, $2i_1<2i_2$; therefore, it is a BGPL.

From the definition of a BGPL, its basic properties can be summarized directly as follows.

Property 1: The IDCH [14] of a BGPL is always situated at the side of $d_i(x, y) > 0$, because the dependence vector function in the I dimension is always non-zero, positive, and real. Geometrically, this tells us that any dependence vectors that are within an outer loop index I_i or are anti-dependent are impossible.

Property 2: For any two dependence tails (i, j) and (i+1, j), their dependence heads are shifted rightward by s_1 and s_3, respectively, which represents the initial dependence offset along loop index I. Consider the dependence graph of L_1 shown in Fig. 3. The dependence heads of (1,1) and (2,1) are (3,5) and (6,5), respectively. Their dependence distance in the I dimension is 3.

Property 3: s_2 is always 0 for a GPL. The dependence heads of the two dependence tails (i, j) and (i, j+1) are located at the same outer loop index $i + d_i(x, y) = i + (s_1 - 1)x + s_3$. For example, the dependence tails (1,1) and (1,2) shown in Fig. 3, have dependence heads on (3,5) and (3,10), respectively. They are both situated in the same column, $I_i = 3$.

In the following subsection, we will determine some of the hopping information, in order to exploit as much parallelism as possible.

3.1.1 Determination of M

Given an iteration space, M represents the number of parallelizable iterations. We formally define M below.

Definition 3.3 (M): Given a normalized two-level nested loop L, let the lower and upper bounds on the outer loop index I be L_I and U_I, respectively. U_J is the upper bound on inner loop dimension J, and I_i to I_{i+j} are adjacent columns of iterations in the loop dimension I, where $L_I \le I_i \le I_{i+j} \le U_I$. For any iteration, $i \in [I_i, I_{i+j}]$ if it satisfies one of the following conditions:

(i) it is dependence free;
(ii) it is a dependence tail, but the corresponding dependence head has $i' \notin [I_i, I_{i+j}]$;
(iii) it is a dependence head, but the corresponding dependence tail has $i' \notin [I_i, I_{i+j}]$.

We say that these iterations are ***parallelizable*** and denote the number of parallelizable iterations as M, where $M = [(I_{i+j} - I_I + 1)*U_J]$. ❑

For DOALL loops, M is initially equal to the total number of iterations in the iteration space, or $[I_i, I_{i+j}] = [L_I, U_I]$ and $M = [(U_I - L_I + 1)*U_J]$. To initialize M, the number of parallelizable iterations before an IDCH and the first group of parallelizable iterations in an IDCH are computed. This is formally stated in Theorem 3.2. According to the PPS mechanism, a parallelizable region can be split from an IDCH by finding a non-IDCH region [15].

Definition 3.4 Non-IDCH region: For a nested loop L, let Γ be the iteration space with iterations i_k (k = 1, 2, 3, …) $\in \Gamma$. Γ(IDCH) represents the IDCH region. For any $i_k' \in \Gamma$, but not in Γ(IDCH), we say that they are in Γ(non-IDCH), and Γ(non-IDCH) is called the non-IDCH region of L.

The IDCH region shown in Fig. 6 is located from I_4 to I_7, while one of the non-IDCH regions is located from I_1 to I_3 and another from I_8 to I_{10}. Lemma 3.3 emphasizes the features of the non-IDCH regions.

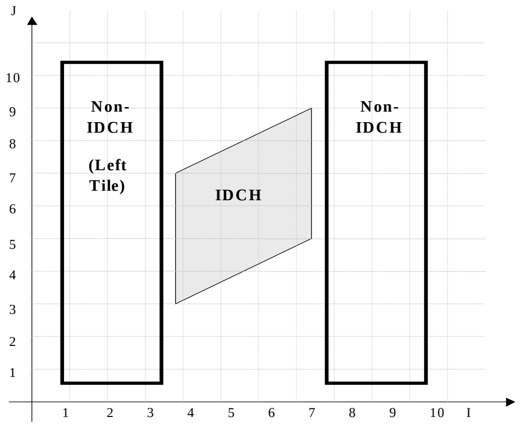

Fig. 6. IDCH and Non-IDCH regions of an iteration space.

Lemma 3.3 [15]: For a nested loop L, let Γ be the iteration space with iteration i_k, $i_m \in \Gamma$(non-IDCH). Then, i_k and i_m are cross-iteration independent and can be executed in parallel.

Therefore, the two non-IDCH regions shown in Fig. 6 are parallelizable. Theorem 3.2 defines the size of the non-IDCH region on the left-hand side of the IDCH.

Theorem 3.2 [15]: Let L be a doubly nested loop, as shown in Fig. 1, and let L_I(IDCH) be the lower bound of the IDCH in loop index I. If $L_I < L_I$(IDCH), then the loop iteration can be partitioned from L_I to (L_I(IDCH)-1) to form a parallel execution tile called the left tile, with tile size $T_L = (L_I(\text{IDCH}) - L_I)*(U_J-L_J+1)$.

Using the size of the left tile, Theorem 3.3 aims to initialize M.

Theorem 3.3: Given a GPL (L) as shown in Fig. 1, M is initialized as $\lfloor(s_1 * i_{left}) + s_3 - 1\rfloor * U_J$, where i_{left} is the leftmost extreme point of the IDCH in loop dimension I, and s_1 and s_3 are the two coefficients in the Diophantine equations (3).

Proof: The parallelizable region before the IDCH covers $[(i_{left}-1)*U_J]$ iterations. As a dependence will only exist in the IDCH region, the first group of parallelizable iterations next to the region is certainly contributed by the leftmost extreme points of the IDCH, according to the definition of a GPL. Property (3) implies that the dependence distance of the leftmost extreme is exactly $[(s_1 - 1)*i_{left}] + s_3$. Therefore, the initial value of M is

$$\begin{aligned}&[(i_{left} - 1)*U_J] + [(s_1 - 1)*i_{left} + s_3]*U_J\\ &= \{i_{left} - 1 + [(s_1 - 1)*i_{left}] + s_3\}*U_J\\ &= [(s_1*i_{left}) + s_3 - 1]*U_J\\ &= \lfloor(s_1*i_{left}) + s_3 - L_1\rfloor*U_J \quad \text{(convert real to integer).}\end{aligned}$$ ❑

Even though the iteration space is an integer space, the partitioning parameters are composed of those coefficients of the Diophantine equations that may be positive and real. On the other hand, because we have partitioned the integer iteration space, the partition parameters must be changed into integer space. Thus, a floor operation is used based on the concept of conservation of partitioning.

Consider Example 1, in which a set of extreme points is {(1, 1), (1, 2), (3, 1), (3, 2)}. Among these, the leftmost extreme point in loop dimension I is 1; therefore, M is initialized as $\lfloor(3*1) + 0 - 1\rfloor*10 = 20$. Geometrically, this indicates that there are 20 parallelizable iterations at first, and that any idle processor can schedule iterations from them.

We propose a strategy to analyse static dependent information for non-uniform dependence loops. We then dynamically use this information to synchronize the parallel-executed iterations (chunks), such that better parallel execution performance for non-uniform dependence loops can be achieved. In a general scheduling mechanism, the iterations inside a current partitioning tile are being scheduled, and iterations in the following tile will be scheduled until all the iterations in the current tile have been scheduled and completed. If all the iterations in the current tile have been scheduled and completed, then the synchronization instruction (in this case a barrier instruction) will be released, and the iterations in the next tile can be scheduled. This continues until all the iterations in the loop have been scheduled.

In our scheduling mechanism, we obtain the scheduling information, called a "hopping gate" and a "hopping distance," in the compilation phase. The hopping gate is the iteration that the iteration in the following tile depends on, as shown in Fig. 4. The hopping distance contains information about how many iterations can be released if all the dependences in the current tile can be resolved. Note that the tile is pseudo, and we that do not really partition the loop in the compilation phase. After the compilation phase of the RCS method, this dependency information between two tiles is remembered as the hopping gate, and the boundary of the next tile is remembered as the hopping distance. In the execution phase, if we have the iteration named hopping gate has been scheduled and completed, then the iterations as the next tile remembered in the hopping distance can be released and scheduled.

3.1.2 Determination of the hopping gate

When the iterations before the hopping gate have all been executed, the corresponding dependence heads will be released. Therefore, the hopping gate specifies the relaxation time of the dependence heads, and is defined below.

Definition 3.5 (Hopping Gate): Given a two-way nested loop L, let n be an identifier of an iteration, and let M be defined as in Definition 3.3. If iterations $i \leq n$ are all parallelizable, then before they are completely executed, no iterations $j > M$ can be executed. We call n a **hopping gate**. ❑

When the iterations before a hopping gate have all been executed, there may exist a certain number of releasable iterations, so that M will probably be incremented. If a hopping gate always carries single parallelizable iterations, M may be updated each time the iteration is executed. Since M and the hopping gate are globally shared variables, frequent adjustments can lead to a synchronization overhead. Therefore, we adjust M only when the dependence tails within an outer loop instance I_i have been completely executed, and the hopping gate then moves ahead by U_J. In other words, the hopping gate will be defined by the last dependence tail of an outer loop instance I_i. However, as the total number of dependence tails in each outer loop instance I_i may vary, we will conservatively choose the highest number as the hopping gate. Based on linear programming theorems stated in [14], the initial value of the hopping gate can be defined as follows.

Theorem 3.4: Given a BGPL (L) as shown in Fig. 1, the hopping gate is initialized as $[(i_{left} - 1)*U_J] + j$, where $j = \max(j')$ and (i', j') are the extreme points of the IDCH, and i_{left} is the leftmost extreme of the IDCH in loop index I.

Proof: The hopping gate is invalid for a parallelizable region. If it exists on the left side of an IDCH, then the hopping gate will be shifted ahead by $(i_{left} - 1)*U_J$. Since the maximum j appears at the extreme points, the hopping gate is initialized as $[(i_{left}-1)*U_J] + j$. ❑

Consider L_1 shown in Fig. 2 again. Among the extreme points (1,1), (3,1), (1,2) and (3,2), the maximum j is 2, and i_{left} is 1, so that the hopping gate will be initialized as $(1-1)*10+2 = 2$. This means that once iterations (1,1) and (1,2) have been completely executed, their corresponding dependence heads will be released. At the same time, the hopping gate will increment by 10 (U_J), and the newly updated hopping gate will be situated at (2,2).

3.1.3 Determination of the hopping distance (HD)

When the hopping gate is reached, the hopping distance defines the total number of releasable iterations. A formal definition is given below.

Definition 3.6 (Hopping Distance, HD): Given a two-way nested loop L, let U_I and U_J be the upper bounds in loop indices I and J, respectively. For M as defined in Definition 3.3 and the hopping gate (G) as defined in Definition 3.5, I_i to I_{i+j} are adjacent columns of parallelizable iterations in the loop dimension I and satisfy $(M/U_J) < I_i \le I_{i+j} \le U_I$. For all iterations, $j \le G$. If these have been completely executed, then iterations $i \in [I_i, I_{i+j}]$ are all releasable. The maximum length of $[I_i, I_{i+j}]*U_J$ is called the hopping distance. ❑

For a GPL, the hopping distance can be determined by the following theorems.

Theorem 3.5: Given a BGPL (L) as shown in Fig. 1, the hopping distance of L is $(\lfloor s_1 \rfloor * U_J)$.

Proof: The proof is straightforward. For a BGPL, each time the hopping gate is reached, iterations between two adjacent columns of the dependence heads will be freed. According to Property 2, the dependence distance between two adjacent columns of dependence heads is s_1; therefore, the hopping distance is assigned to be $(\lfloor s_1 \rfloor * U_J)$. ❑

Unlike M and the hopping gate, the hopping distance is determined at compilation time but remains constant. In Example 1, the hopping distance of L_1 is $(3*10) = 30$. When iterations (3, 5) and (3, 10) are released, M is incremented by 30, or M has a total of $20 + 30 = 50$ parallelizable iterations, ranging from iterations (1, 1) to (5, 10). Both the hopping gate and the hopping distance will be active until iterations within the IDCH have all been executed.

The following corollary guarantees that the hopping gate is less than, or equal to, M.

Corollary 3.1: Given a BGPL (L) as shown in Fig. 1, the hopping gate is always less than, or equal to, M.

Proof: Initially, the hopping gate is less than, or equal to, M:

$$\begin{aligned}
&\lfloor (s_1 * i_{left}) + s_3 - 1 \rfloor * U_J \\
&= \lfloor (s_1 - 1) * i_{left} + i_{left} + s_3 - 1 \rfloor * U_J \\
&\ge \lfloor i_{left} - 1 \rfloor * U_J + \lfloor (s_1 - 1) * i_{left} + s_3 \rfloor * U_J \\
&\ge [i_{left} - 1] * U_J + U_J \qquad (d_i(i_{left}, 0) = [(s_1 - 1) * i_{left} + s_3] \ge 1, \text{ else it is meaningless}) \\
&\ge (i_{left} - 1) * U_J + j \qquad (1 \le j \le U_J)\ .
\end{aligned}$$

When execution commences, M, together with the hopping gate, will be updated by the same processor. Each time hopping takes place, M increments by at least $\lfloor s_1 \rfloor * U_J$ (the hopping count), and the hopping gate increments by U_J. Since $s_1 > 1$ for a BGPL, this implies that $(\lfloor s_1 \rfloor * U_J) \ge U_J$. Therefore at any time instance, the hopping gate is always less than, or equal to, M. ❑

Based on Corollary 3.1, when the RCS method is applied, dynamic parallelism extraction is achieved through the earlier relaxation of parallelizable iterations. However, the hopping information can only be obtained and implemented in the outer loop dimension. We will propose a new method to exploit parallelism in another loop dimension.

3.2 Combination With the Irregular Loop Interchange Mechanism

In this section, we will present an effective method to check whether a loop interchange is allowable in a non-uniform dependence loop. Thus, after the Irregular Loop

Interchange (ILI), the parallelism of the loop is dependent on the maximum parallelism before, and after, the ILI mechanism. In the following paragraphs, we discuss some dependence lemmas and then introduce our ILI mechanism.

Definition 3.7 [25]: The **standard execution order** is the execution order defined by the Fortran standard. It is denoted by the symbol '«'.

Definition 3.8 [25]: Let S and S' be arbitrary statements. Then S bef S': <==> S occurs textually before S'.

The iteration vector i(1:m) will be abbreviated using the notation \i for the sake of ease of representation.

Lemma 3.4: Let S_h and $S_{h'}$ be statements at level m, with associated iteration vectors j and j'. Then

$S_h(j)$ « $S_{h'}(j')$ <==> $((j < j') \vee (j = j') \wedge h < h')$. ❑

Lemma 3.5 [15]: Assuming that S and S' are two statements with associated iteration vectors i and i', respectively, then

$S(i)$ « $S'(i')$ <==> $(\backslash i < \backslash i') \vee (\backslash i = \backslash\, i' \wedge S$ bef $S')$. ❑

Proof: Let the indices of S and S' in the statement sequence of the body of L_m be h and h', respectively, and let $i \in [S]$ and $i' \in [S']$ be arbitrarily selected. Then $\backslash i \in [S_h]$ and $\backslash i' \in [S_{h'}]$, and $S_h(\backslash i)$ « $S_{h'}(\backslash i')$ iff $S(i)$ « $S'(i')$. Application of Lemma 3.4 (with $j = \backslash i$, $j' = \backslash i'$) yields

$S(i)$ « $S'(i')$ <==> $(\backslash i < \backslash i') \vee (\backslash i = \backslash i' \wedge h < h')$. ❑

Definition 3.9 [25]: A **direction vecto**r is a vector $\theta \in \{<, =, >, *\}$ for some $k \geq 1$.

Definition 3.10 [25]: Let S and S' be statements in a loop. Then

(1) S « S': <==> $\exists i, i'$: $S(i)$ « S'(i'),
(2) $S\ \delta\ S'$: <==> $\exists i, i'$: $S(i)\ \delta\ S'(i')$.

$S\ \delta\ S'$ is expressed as S' and is (**data**) **dependent** on S. ❑

In the following, we will use «/c to denote the standard execution order of L/c, and use δ/c for its dependence relation. For an arbitrary vector x of length n, x/c is the vector obtained from x by swapping components x_c and x_{c+1}. Note that (L/c)/c = L.

Lemma 3.6 [15]: Assume that S and S' are statements in L, and that i and i' are iteration vectors associated with S and S', respectively. Then θ = dir (i, i'), c is an interchange level, and $S(i)$ « $S'(i')$. Then

$S'(i'/c)$ «/c $S(i/c)$ iff $\theta = (=^{c-1}, <, >, *\ldots*)$. ❑

Proof: Let the assumptions of the lemma hold. We apply Lemma 3.5 twice to obtain

(1) $S(i)$ « $S'(i')$ <==> $(\backslash i < \backslash i') \vee (\backslash i = \backslash i' \wedge S$ bef $S')$,
(2) $S'(i'/c)$ «/c $S(i/c)$ <==> $(\backslash i' </c\ \backslash i) \vee (\backslash i'/c = \backslash i/c \wedge S'$ bef $S)$.

Now suppose $S'(i'/c)$ «/c $S(i/c)$. Then $\backslash i'/c < \backslash i/c$, $i = i'$ cannot hold, and there is a $k \in [1: n]$, such that $i <_k i'$. Clearly, the condition cannot be satisfied for any $k \neq c$. For $k = c$, we have

$$i = (i_1, \ldots, i_{c-1}, i_c, i_{c+1}, \ldots, i_n),$$
$$i' = (i_1, \ldots, i_{c-1}, i'_c, i'_{c+1}, \ldots, i_n),$$

with $i_c < i'_c$. In i/c and i'/c, the components c and c+1 are switched. From $S'(i'/c)$ «/c S (i/c), we immediately obtain $i_{c+1} > i'_{c+1}$. This is the direction vector and has the form $\theta = (=^{c-1}, <, >, *\ldots*)$.

Conversely, $\theta = (=^{c-1}, <, >, *\ldots*)$, which implies that $S'(i'/c)$ «/c S (i/c). This concludes the proof of the lemma. ❑

Definition 3.11 [15]: A dependence $S\ \delta_\theta\ S'$ in loop **L** is c-**interchange preventing** iff $S(i/c)\ \delta/c\ S'(i'/c)$ does not hold. ❑

Lemma 3.7 [15]: A dependence $S\ \delta_\theta\ S'$ on L is c-**interchange preventing** iff $\theta = (=^{c-1}, <, >, *\ldots*)$. ❑

Lemma 3.8 [15]: Loop interchange at level c is valid iff there exists no c-**interchange preventing** dependence. ❑

It is, thus, most important to distinguish whether a non-uniform loop interchange is valid or not. Below, we will first present some theorems to distinguish if an interchange is legal or not and then apply an appropriate non-uniform loop interchange technique to it.

Theorem 3.6: Assume that the general solutions for interchanged dependence vectors are $V_c(X, Y)$ and $V_{c+1}(X, Y)$, respectively. A dependence $S\ \delta_\theta\ S'$ exists in L, where $\boldsymbol{\theta} = (=^{c-1}, V_c(X, Y), V_{c+1}(X, Y), *\ldots*)$. The loop interchange at level c is valid iff $V_c(X, Y)*V_{c+1}(X, Y) \geq 0$.

Proof: Let the assumptions of the above theorem hold. Applying Lemma 3.6, the loop interchange at level c is valid; thus, there exists no c-**interchange preventing** dependence. From Lemma 3.6, a dependence $S\ \delta_\theta\ S'$ exists in L; thus, $\boldsymbol{\theta} \neq (=^{c-1}, <, >, *\ldots*)$. Clearly, $V_c(X, Y)*V_{c+1}(X, Y) < 0$ cannot be satisfied. We have $V_c(X, Y)*V_{c+1}(X, Y) \geq 0$.

Conversely, $V_c(X, Y)*V_{c+1}(X, Y) \geq 0$, implies that a dependence $S\ \delta_\theta\ S'$ exists in L, thus $\boldsymbol{\theta} \neq (=^{c-1}, <, >, *\ldots*)$, which concludes the proof of the theorem. ❑

Theorem 3.7: Assume that applying the loop interchange at level c is valid. The loop bounds of the normalized loop indices c and c+1, are U_c and U_{c+1}, respectively. The minimum dependence distances for the loop indices c and c+1 are $Min(d_c)$ and $Min(d_{c+1})$, respectively. The maximum parallelism for loop indices c and c+1 is $Max(Min(d_c)*U_{c+1}, Min(d_{c+1})*U_c)$.

Proof: Let the assumption of the above theorem hold. There is no dependency between any two iterations inside $Min(d_c)$ and $Min(d_{c+1})$, respectively. The iterations that can be parallelized are $Min(d_c)*U_{c+1}$ and $Min(d_{c+1})*U_c$. The maximum parallelism of loop indices c and c+1 are the maximum dependence distance of the loop indices c and c+1. Thus, the maximum parallelism is $Max(Min(d_c)*U_{c+1}, Min(d_{c+1})*U_c)$. ❑

Theorem 3.6 detects whether certain specific kinds of non-uniform loops can be interchanged. Any non-uniform loops can be interchanged immediately after validation has been determined. Theorem 3.7 tells us how to calculate the hopping distance before, and after, the ILI method is performed and how to select the largest minimum dependence distance as the hopping distance.

Example 3: In Fig. 2, loop L_1 is an example of a non-uniform dependence loop, and its dependence graph is shown in Fig. 3. The Diophantine equation set is $\{i_1 = x, j_1 = y, i_2 = 3x, j_2 = 5y\}$, and the dependence vector set is $\{2x, 4y\}$, where x and y are two integer variables. According to Theorem 3.6, the production of two dependence vectors is $V_1*V_2 = 2x*4y > 0$, and $Min(d_1) = 2$, $Min(d_2) = 4$; therefore, loops I and J are interchangeable. Thus, a degree of parallelism can be exploited through the ILI mechanism.

In the following section, we will discuss the generalities of the RCS method.

4. GENERALITIES OF THE RCS METHOD

If a loop does not belong to the GPL, then the RCS method works in a similar manner to a GPL, with a slight modification in initialization of the hopping information. Because of the probable existence of the coefficient s_2 in the dependence vector function $d_i(x, y)$, the number of parallelizable iterations is determined based on the concept of the minimum dependence distance [14]. If $d_i(x, y) = 0$ does not pass through the IDCH, then the absolute minimum and maximum values of $d_i(x, y)$ appear as the extreme points.

Consequently, the minimum dependence distance of a flow dependence loop is $\{md \mid md = \min[d_i(x, y)]$, where (x, y) are the extreme points of the IDCH, and $md \in R\}$. Let U_J, be as defined as shown in Fig. 1, so that iterations within $\lfloor md \rfloor * U_J$ are parallelizable [14]. Here, the hopping information can be determined by means of the following corollary.

Corollary 4.1: Given a non-GPL flow dependence loop (L) as shown in Fig. 1, the hopping information can be initialized as follows:

(i) $M = \lfloor (i_{left} - 1) + md \rfloor * U_J$,
(ii) hopping gate $= \lfloor i_{left} + md - 2 \rfloor * U_J + j$,
(iii) hopping distance $= \lfloor md \rfloor * U_J$,

where md is the minimum dependence distance of L, and j = max(j'), where (i', j') are the extreme points of the IDCH.

Proof: (i) Similar to Theorem 3.3, M is initialized as

$[(i_{left}-1)*U_J] + \lfloor md \rfloor * U_J$

$= \lfloor (i_{left}-1) + md \rfloor * U_J$.

(ii) Iterations in $\lfloor md \rfloor * U_J$ are parallelizable, so $\lfloor md-1 \rfloor * U_J + j$. Therefore, the hopping gate can be initialized as

$[(i_{left}-1)*U_J] + \lfloor md - 1 \rfloor * U_J + j$

$= \lfloor i_{left} + md - 2 \rfloor * U_J + j$.

(iii) Given that M and the hopping gate are defined as in (i) and (ii), if we assume that the iterations before the hopping gate have all been executed, then the total number of releasable iterations is $\lfloor d \rfloor * U_J$. If d > md, then there may exist dependence vectors located within $\lfloor d \rfloor * U_J$, so $\lfloor d \rfloor * U_J$ will definitely not be parallelizable. If d < md, then iterations in $\lfloor d \rfloor * U_J$ are parallelizable, but d is not the maximal length of a parallelizable iteration. As a result, d must be equal to md; thus, the hopping distance = $\lfloor md \rfloor * U_J$. ❑

On the other hand, if $d_i(x, y) = 0$ passes through the IDCH, then the iterations within the IDCH can be flow-dependent tails or anti-dependent heads. By using array duplication and renaming, the anti-dependences can be removed completely. Moreover, some dependence tails may have dependence heads located at the same loop instance I_i. $D_j(x, y) = 0$, which implies that there exists intra-iteration dependence for all iterations along the line segment $d_i(x, y) = 0$. As long as a single iteration is executed serially, the intra-iteration dependence is preserved. The RCS method works similarly, but determination of the minimum dependence distance follows Theorem 3.7.

Theorem 4.1: If $d_i(x, y) = 0$ passes through the IDCH and $d_j(x, y) = 0$, then the absolute minimum value of $d_i(x, y)$ appears at either the extreme points or at the iterations next to the intersection points of the line segment $d_i(x, y) = 0$ with the IDCH.

Proof: Let E represent a set of extreme points, let E' be a subset of E, and let (x_1, y_1) and (x_2, y_2) be two points of intersection of the line segment $d_i(x, y) = 0$ with the IDCH. If $d_i(x,y) = 0$ passes through the IDCH, then it divides the IDCH into a unique tail set and a unique head set [16]. The two unique sets are subsets of the IDCH. If they are denoted as S_f and S_a, then the extreme points around their parameter will be union of E' and {(x, y) | (x, y) | a coordinate of the IDCH, which is closest to the intersection points (x_1, y_1) or (x_2, y_2)}. Their respective absolute minimum dependence distances can now be determined. Assuming that they are md_f and md_a, the overall minimum dependence distance is $\min[md_f, md_a]$. ❑

Finally, we can check the hopping information of both axes. If the loop is found to be interchangeable, then we can choose the one with the larger hopping distance. Thus, the degree of exploitation of parallelism will be maximized.

Based on the above analysis, our RCS method can be easily generalized. During the compilation phase, the loop is examined to determine whether or not it is interchangeable according to Theorem 3.6. If it is, then, an irregular loop interchange is applied and more parallelism in another loop dimension will be exploited. The detailed procedure is shown in detail in Algorithm 1.

Algorithm 1: Compilation phase of the RCS method

```
Input: A two-way nested loop L, as shown in Fig. 1, which has a total of N=$U_I$ * $U_J$ iterations.
Output: A set of initialized variables {M, hopping_gate, hopping_distance, $i_{right}$}
Method
    function Banerjee (L): General solutions of the Diophantine equation set (S)
    /* This is used to create general solutions of the Diophantine equation set (S). The
       output is a list of eight half spaces and the dependence vector set (V).*/
    function Tzen_and_Ni (S) : DCH (D)
    /* This aims to form the DCH from the given Diophantine equation set (S) and is
       expressed as a list of nodes. */
    function Is_DCH_Empty (D) : Boolean
    /* If the DCH is empty, then it returns TRUE, else it returns FALSE*/
    function Transform_DCH_To_IDCH : IDCH(I)
    /* This is used to convert the DCH into an IDCH and to record the maximum j and
       the leftmost and rightmost extreme points as j, $i_{left}$ and $i_{right}$.*/
    function Determine_Position_Of_IDCH(S,V) : integer
    /* If $d_i$(x', y') > 0 for all the extreme points of the IDCH (x', y'), then the IDCH is
       located at $d_i$(x, y) > 0 and returns 1. Else, if $d_i$(x', y') < 0 for all the extreme points
       of the IDCH (x', y'), then the IDCH is located at $d_i$(x, y) < 0 and returns -1. Oth-
       erwise, it returns 0.*/
    function Is_BGPL(S) : Boolean
    /* If $s_1$ > 1, and $s_2$ = 0, then it returns TRUE, else it returns FALSE. */
    function Determine_Minimum_Dependence_Distance(V, D) : integer
    /* This determines the minimum dependence distance of the IDCH using Theorem
       4.1.*/
    function Determine_Overall_MDD(V, D) : integer
    /* This determines the minimum dependence distance of the IDCH.*/
    function J_Dependence_Free(V) : Boolean
    /* If $d_j$(x, y) = 0, or $j_2$ = y, for Diophantine Equation (1), then it returns TRUE, else
       it returns FALSE.*/
begin
    hopping_gate : = 0;
    hopping_distance : = 0;
    S : = Banerjee(L);
    D : = Tzen_and_Ni(S);
    if (Is_DCH_Empty = TRUE)
        then /* identify L as DOALL loop */
                M : = $N^2$;
        else
            I : = Transform_DCH_To_IDCH(D);
            switch (Determine_Position_Of_IDCH(S, V))
    begin
        case -1 : /*Reconstruct the loop as DOALL loop.*/
                  $M := N^2$;
        case 1 : if (IS_BGPL(S))
            then
                $M := \lfloor (s_1 * i_{left}) + s_3 - 1 \rfloor * U_J$;    /* Theorem 3-3*/
                hopping_gate : = $[(i_{left} - 1) * U_J] + j$;    /* Theorem 3-4*/
                hopping_distance : = $\lfloor s_1 \rfloor * U_J$;    /* Theorem 3-5*/
            else
                md : = Determine_Minimum_Dependence_Distance(V, D);
                $M = \lfloor (i_{left} - 1) + md \rfloor * U_J$;    /* Corollary 4-1*/
                hopping gate $\lfloor i_{left} + md - 2 \rfloor * U_J + j$;
                hopping distance = $\lfloor md \rfloor * U_J$;
          fi
        case 0 : if (J_Dependence_Free(V) = TRUE)
            then
                md : = Determine_Overall_MDD(V, D);
                $M = \lfloor (i_{left} - 1) + md \rfloor * U_J$;    /* Corollary 4-1*/
                hopping gate $\lfloor i_{left} + md - 2 \rfloor * U_J + j$;
                hopping distance = $\lfloor md \rfloor * U_J$;
                fi
        endswitch
    fi
end
```

The complexity of this algorithm is bound by the formation of the DCH, by the transformation of the DCH into the IDCH, and by the determination of the position of the IDCH and of the minimum dependence distance. During the execution phase, the RCS method follows Algorithm 2. Notice that the RCS method defines the number of parallelizable iterations and can also incorporate a predefined chunk size control function.

Algorithm 2: Execution phase of the RCS method
Input: Number of processors P, and output of Algorithm 1, {M, hopping_gate, hopping_distance and i_{right}}.
Output: Scheduling code of the RCS method on BGPL.
Method

```
    /* Define a structure to record the lower and upper bounds of the current chunk for
       each processor. The following example is written in C:
    typedef struct {
        int lb; /*lower bound*/
        int ub; /*upper bound*/
    } bound_type;
    bound_type bound[P]; */
    procedure Get_A_Chunk_Of_Iterations(Count);
    /* Chunk size control function is defined during compilation time. The following
        example assumes that the GSS is applied.
        lock(s);
            bound[processor_id].lb = Count + 1;
            bound[processor_id].ub = Count + $\lceil$(N-Count)/P$\rceil$;
          unlock(s); */
        procedure Execute_A_Chunk();
        /* Executing loop body*/
begin
  /* Count is used to record the id of the currently scheduled iteration, and Executed
    is used to record the id of the currently executed iteration. */
    Count : = 0;
    Executed : = 0;
    /* Start parallel execution */
    while (Count < N)
        Get_A_Chunk_Of_Iterations(Count);
        Execute_A_Chunk();
        if (M < N)
          then
            lock(t);
              Executed = max(Executed, bound[processor_id].ub);
                /*processor triggers hopping*/
              if (bound[processor_id].lb $\le$ hopping_gate    and
                  bound[processor_id].ub $\ge$ hopping_gate)
                then
                  if (hopping_gate $\ge$ [($i_{right}$-1)*$U_J$+hopping_gate])
                    then
                      M : = $N^2$;
                    else
                      hopping_count : = [(Executed-hopping_gate)/$U_J$ + 1];
                      hopping_gate : = hopping_gate + (hopping_count*$U_J$);
                      M : = M + (hopping_count * hopping_distance);
                  fi
              fi
            unlock(t);
     fi
        endwhile
end
```

Consider L_1 in Example 1. If we assume that there are two available processors, P_1 and P_2, and that the GSS is chosen for scheduling, then one of the possible execution orders is that shown in Table 1. Note that only the activities that are protected by pairs of lock-unlock operations are included. The ranges shown in parentheses in Table 1 represent the lower and upper bounds of that chunk.

If we suppose that P_1 acquires a lock at the beginning, then $chunk_2$ finishes its execution before $chunk_1$ does, but is not allowed to trigger hopping as shown in step 4 because the hopping gate is included in $chunk_1$ and P_1 is the unique candidate for triggering the hopping gate. When $chunk_1$ finishes, the hopping gate ought to be adjusted twice as

Table 1. One possible execution order of L_1.

Step	P_1	P_2
1	$chunk_1 = \lceil(20-0)/2\rceil = 10$ (1~10)	
2		$Chunk_2 = \lceil(20-10)/2\rceil = 5$ (11~15)
3		Executed = max (0, 15) = 15 $\because$ (11>2) and (15>2) $\therefore$ hopping fails.
4	Executed = max(15, 10) = 15 $\because$ (1<2) and (10>2), $\therefore$ hopping count = (15-2)/10+1 = 2 hopping gate = 2 + (2x10) = 22 M = 20 + (2x30) = 80	$Chunk_3 = \lceil(20-15)/2\rceil = 3$ (15~17)
5	$chunk_4 = \lceil(80-17)/2\rceil = 32$ (18~49)	
6		Executed = max(15, 17) = 17 $\because$ (15<22) and (17<22) $\therefore$ hopping fails.
7		$Chunk_5 = \lceil(80-49)/2\rceil = 16$ (49~64)
8		Executed = max(17, 64) = 64 $\because$ (49<22) and (64<22) $\therefore$ hopping fails.
9	Executed = max(64, 49) = 64 $\because$ (18<22) and (49>22) and 22 = [(3-1)*10+2] $\therefore$ M = 100	$chunk_6 = \lceil(80-64)/2\rceil = 8$ (64~71)
10	$chunk_7 = \lceil(100-71)/2\rceil = 15$ (72~85)	
11		$chunk_8 = \lceil(100-85)/2\rceil = 8$ (85~92)
12	$chunk_9 = \lceil(100-92)/2\rceil = 4$ (93~96)	
13		$chunk_{10} = \lceil(100-96)/2\rceil = 2$ (97~98)
14	$chunk_{11} = \lceil(100-98)/2\rceil = 1$ (99)	
15		$chunk_{12} = \lceil(100-99)/2\rceil = 1$ (100)

long as $chunk_2$ has already passed over the subsequent hopping gate. Once P_1 appears to schedule $chunk_4$, it schedules iterations from the newly updated M (step 5). Once P_1 detects that iterations within the IDCH have all been executed, M will reach N, as shown in step 9. Scheduling continues until M is completely executed.

5. PRELIMINARY PERFORMANCE EVALUATIONS

In this section, performance evaluations will be presented to verify the practical effectiveness of the RCS method. The experimental programs include program models discussed in the previous section and some practical code segments. The experimental evaluations were carried out on a CONVEX SPP-1000 clustered multiprocessor system [17, 18], which had eight PA-RISC processors and a shared distributed memory configuration.

To find out whether our RCS technique would perform better than the 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 called Simulation and Evaluation Environment for Shared-Memory Multiprocessor Architecture (SEESMA) [23], which was an enhanced version of MINT [24]. This system had a highly parallel shared memory multiprocessor system environment and was similar to the Exemplar system architecture with a large number of processors.

Our performance merit was the execution time. This was divided into five components: (i) busy time (the average parallel execution time of each activated processor); (ii) waiting time (the processor's average waiting time while others were busy executing); (iii) the scheduling overhead (average latency of scheduling a chunk); (iv) the fork and barrier overhead (the average time spent at the end of each tile); and (v) the initialization overhead (the time consumed in initializing a scheduling scheme and chunk to be executed).

The methods that were compared consisted of the ISM [10], the MDT [14], the PPS [15], and the GPD [15].

As discussed in section 3, the RCS method ought to be associated with one of the dynamic scheduling schemes. To determine which was best, we applied the Pure Self Scheduling (PSS), the Chunk Self Scheduling (CSS), the Guided Self Scheduling (GSS) [1], and the Trapezoid Self Scheduling (TS) [3] schemes on L_1. As shown in Fig. 7, the association with GSS was the most superior scheme.

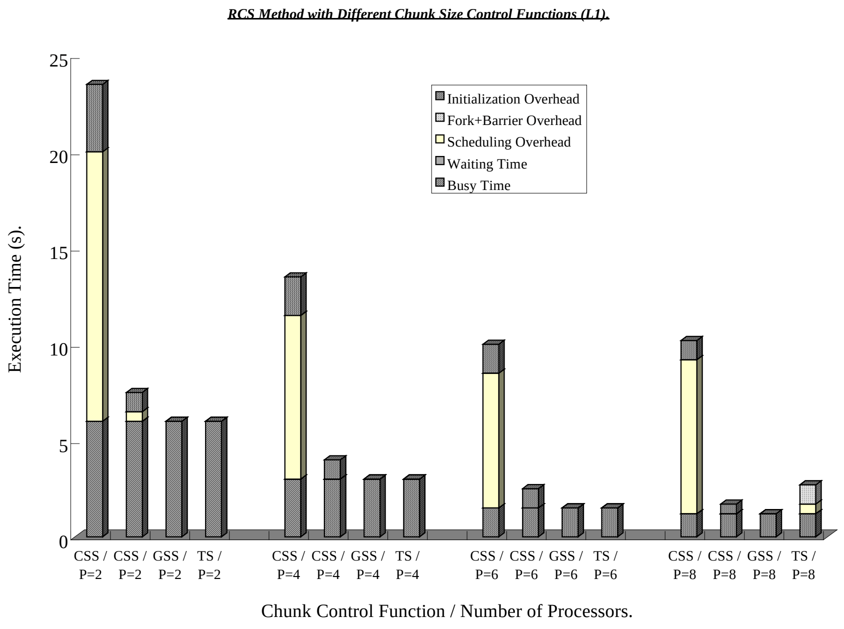

Fig. 7. The RCS method associated with various chunk size control functions.

The chunk size of the PSS was kept constant at 1. The processors spent a lot of time performing parallel execution of a single iteration, leading to heavier initialization and scheduling overhead. These overheads can be reduced by increasing the chunk size. In the same time, a subroutine is called on to complete a chunk of iterations rather than a single iteration. The faster the chunk size converges to 1, the lower the scheduling and initialization overheads will be. The CSS, GSS, and TS schemes benefit from this ability. The formulae in the TS scheme failed sometimes, resulting in imbalanced workload. In the following evaluations, the RCS method was incorporated with the GSS scheme to reliably extract parallelism, and the GSS was also incorporated with the PPS, MDT and GPD methods.

5.1 Performance Evaluation of the Program Models

Table 2 detailed listing of the results for the program model L_1 when different scheduling approaches were used.

Table 2. Scheduling-related information about the various approaches.

Scheduling Schemes	L_1
ISM	BDVS = {(0, 1), (1, –1)}
PPS	IDCH covers from I = 1 to 10; BDVS = {(0, 1), (1, –1)}
MDT	Tile length = 2; Tile size = 60
GPD	Tile lengths = 2, 4, 12, 12; there are 4 tiles in total.
RCS Method	M = 120; Hopping gate = 10; Hopping distance = 150; i_{right} = 6

Fig. 8 shows the execution time of L_1 when different scheduling and partition mechanisms were incorporated. The number of processors (P) varied from two to eight. In the above condition, the MDT execution time was the longest, no matter how many processors were used. It required 15 times the number of process fork-joins than other mechanisms, and thus required barrier synchronization, so the overhead became significant when P became large. In addition, its limited tile size caused the number of parallelizable iterations to be reduced, with a consequent delay in the scheduling time. When L_1 was scheduled with the ISM, the system spent a significant amount of time waiting for other iterations to complete. Finally, the delay overhead resulted in low performance gain. The performance behaviour of the PPS was similar to that of the ISM. This is because the method of scheduling iterations in the IDCH region followed that of the ISM. When the IDCH region was large, the performance of the PPS was bound by the delay overhead inherited from the ISM. Although the GPD was specific to the growing pattern loop, it was still restricted by the multi-barrier synchronization overhead. Its execution time eventually became worse than that of the RCS method. Overall, the RCS method achieves success by removing the system waiting time and multi-barrier synchronization time. As a result, it outperformed all the other methods.

5.2 Performance Evaluation on Practical Code Segments

In addition to the program models, practical code segments were also taken into account. Evaluations of the *Propagate* code segment will be presented first. This is one

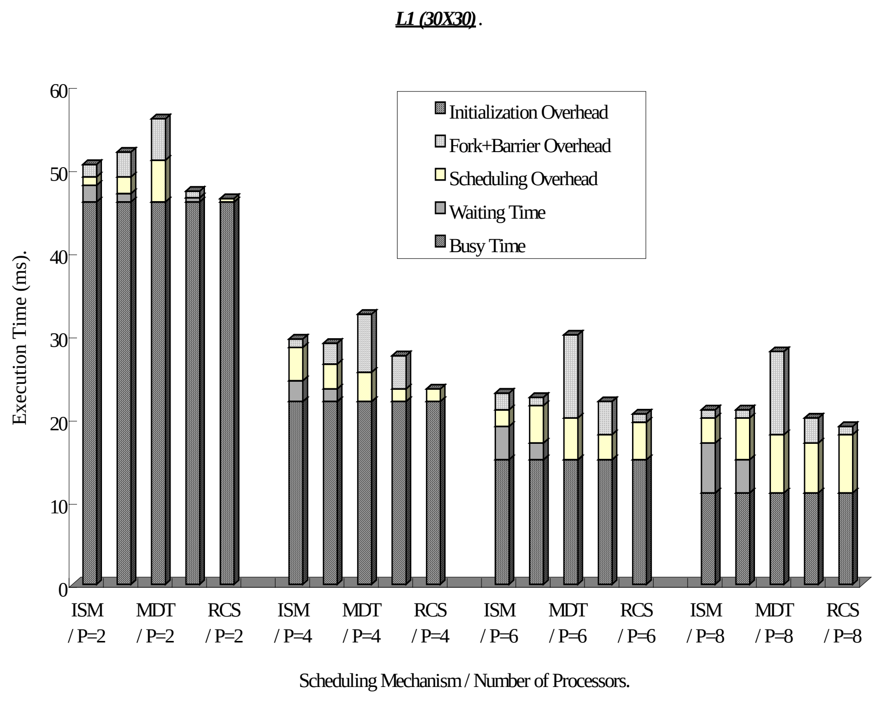

Fig. 8. Execution time of L_1.

dependence pattern widely found in the Linpack and Eispack packages. The code segments are shown in Figs. 9a and 9b. Both of these share a dependence graph as shown in Fig. 9c. The results, computed based on iterations (1, j), are propagated toward iterations (i,j) for all i ∈ [1, Q]. The dscheduling information for the distinct scheduling schemes is presented in detail in Table 3.

```
DO I = 1, Q
    DO J = 1, R
        AR(I, J) = AR(1, J)
    CONTINUE
CONTINUE
```

Fig. 9a. Code segment 1.

```
DO I = 1, Q
    DO J = 1, R
        B(J, I) = B(J, 1)
    CONTINUE
CONTINUE
```

Fig. 9b. Code segment 2.

Figs. 10a and 10b show the execution time and speedup of the propagate code segment. The upper loop bound was set at 30 on each dimension. For this particular code pattern, the MDT tiles the iteration space with a tile length equal to 1 or each column of iterations forms a parallelizable tile. The 30 times fork-join processes and barrier synchronization result in an extremely large overhead that seriously degrades performance. The ISM schedules the loop with a delay factor equal to 1. Both the ISM and PPS methods are rather worse than the RCS method because of their substantial waiting over head. Again, the RCS method retains its superiority. From Fig. 10b, we can further conclude

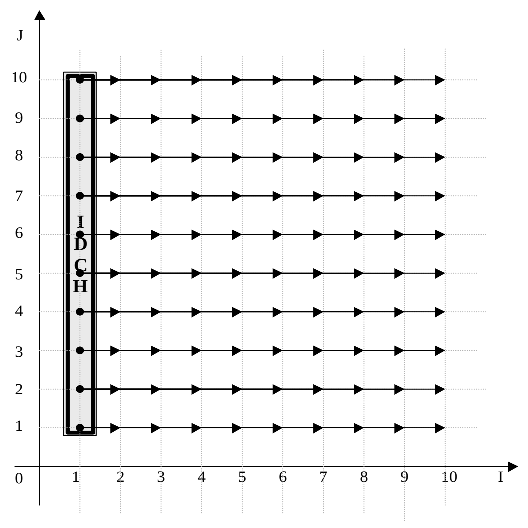

Fig. 9c. Dependence graph.

Table 3. Scheduling-related information for various approaches.

Scheduling Schemes	Propagate Code Segment (30x30)	Swap Code Segment (30x30)
ISM	BDVS = {(0, 1), (1, 0)}	BDVS = {(0, 1), (1, 0)}

PPS	IDCH at I = 1;
	BDVS = {(0, 1), (1, 0)}
		IDCH covers from I = 1 to 30;
		BDVS = {(0, 1), (1, 0)}

MDT	Tile length = 1 Tile size = 30	***Non***
RCS Method	M = 30; Hopping gate = 30 Hopping distance = 0	M = 30; Hopping gate = 30; Hopping distance = 30

that if a given loop inherits adequate parallelism, the RCS method can extract it with little hopping overhead.

Another code segment evaluated is called the *Swap* code, which serves as the kernel of the Fishpak package. Fig. 11a and 11b show the code patterns and their dependence graph. We found that the IDCH occupied the whole iteration space; moreover, $d_i(x, y) = 0$ went through the IDCH vertically, and $d_j(x, y) = 0$. At this point, the MDT failed, and the performance of the PPS was identical to that of the ISM, because both the left and right tiles were empty. The right half of Table 3 tabulates the performance of the ISM and RCS methods when they were applied to the swap code.

The execution time and speedup graph of the swap code segment are presented in Figs. 12a and 12b, respectively. Since there were at most, 30 parallelizable iterations to be executed at any given time, the limited parallelism caused a significant scheduling overhead when the RCS method was employed. Except for the case when five or more processors were used simultaneously, the performance gain of the RCS method was just as good as that of the ISM method.

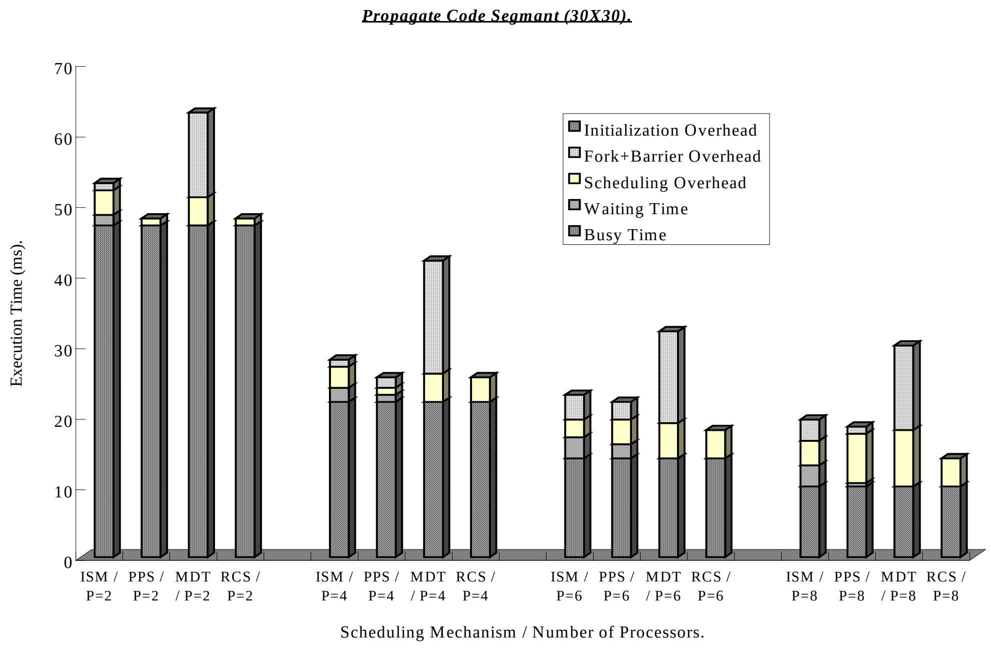

Fig. 10a. Execution time of the propagate code segment.

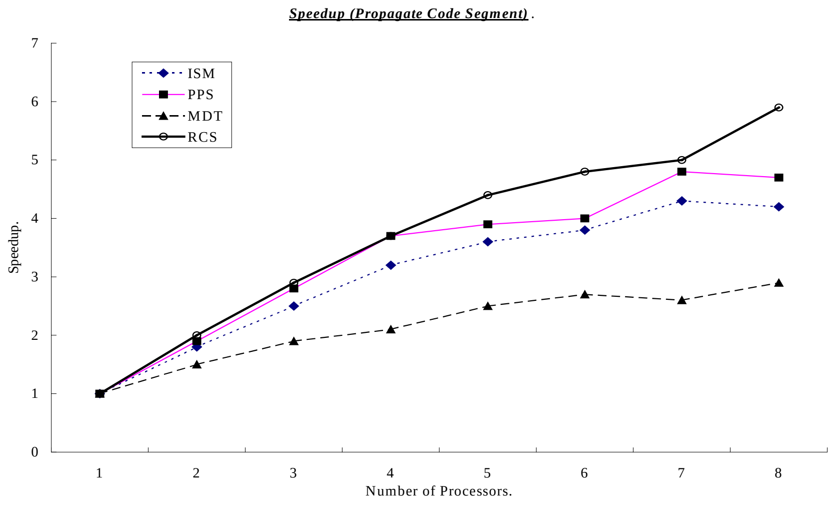

Fig. 10b. Speedup of the propagate code segment.

On the other hand, to show the performance of our mechanism when the number of processors is larger than eight, we implemented different mechanisms in the SEESMA environment, using 128 processors. The speedup of the Propagate and Swap code seg ments is shown in Figs. 13 and 14, respectively. The loop bounds of the benchmarks

```
DO I = 1, 10
   DO J = 1, 10
      A1 = Y(J, I)
      Y(J, I) = Y(J, N+1-I)
      Y(J,N+1-I) = A1
   CONTINUE
CONTINUE
```

Fig. 11a. Swap code segment.

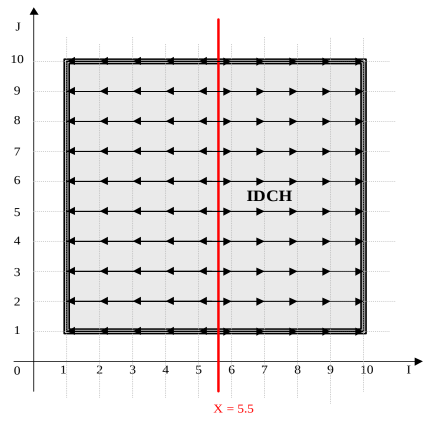

Fig. 11b. Dependence graph of the swap code.

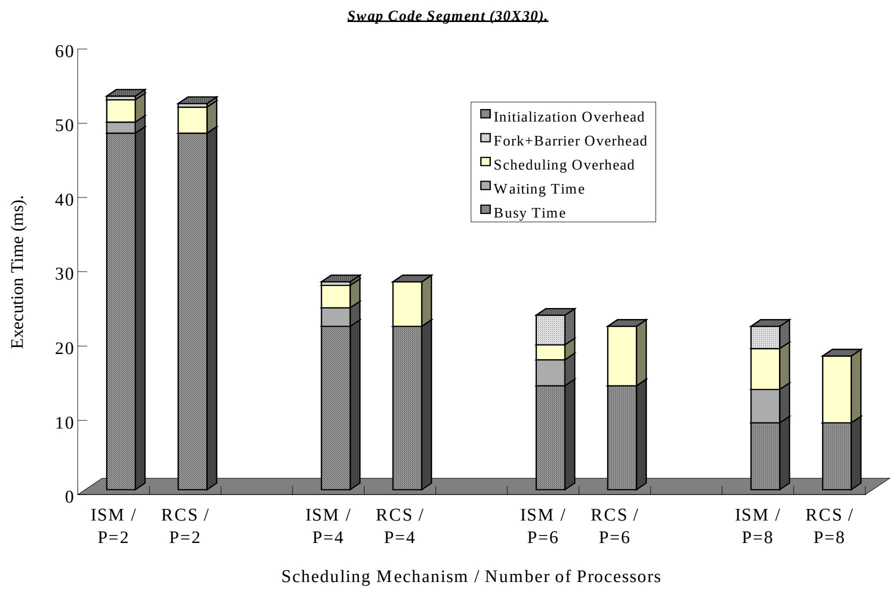

Fig. 12a. Execution time of the swap code segment.

were set to 100. We can see that our RCS mechanism performed better than the other scheduling mechanisms when a large number of processors was used.

In summary, the performance of the RCS method is exactly proportional to the underlying parallelism. If the parallelism is sufficiently large, the RCS method can reliably extract parallelism without introducing an intolerable scheduling overhead. Consequently, the RCS method is an encouraging approach to scheduling non-uniform dependence nested loops.

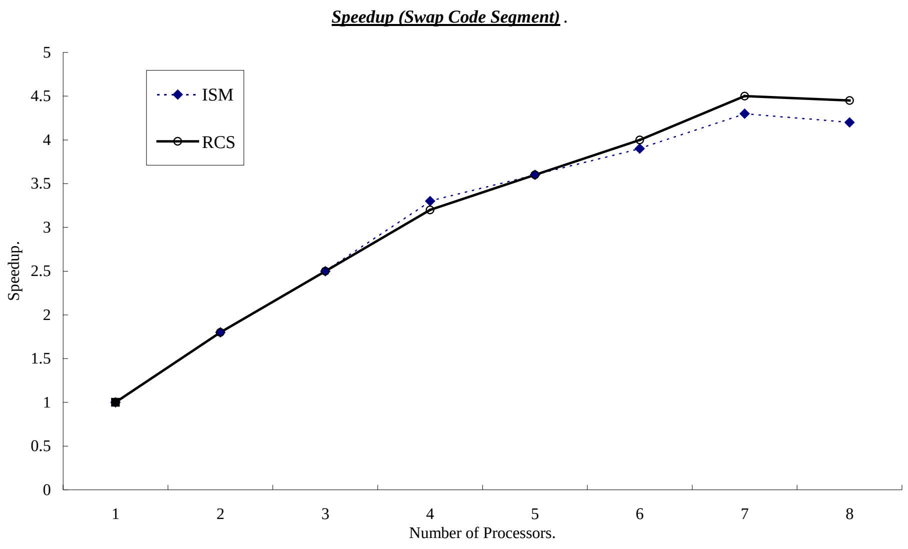

Fig. 12b. Speedup of the swap code segment.

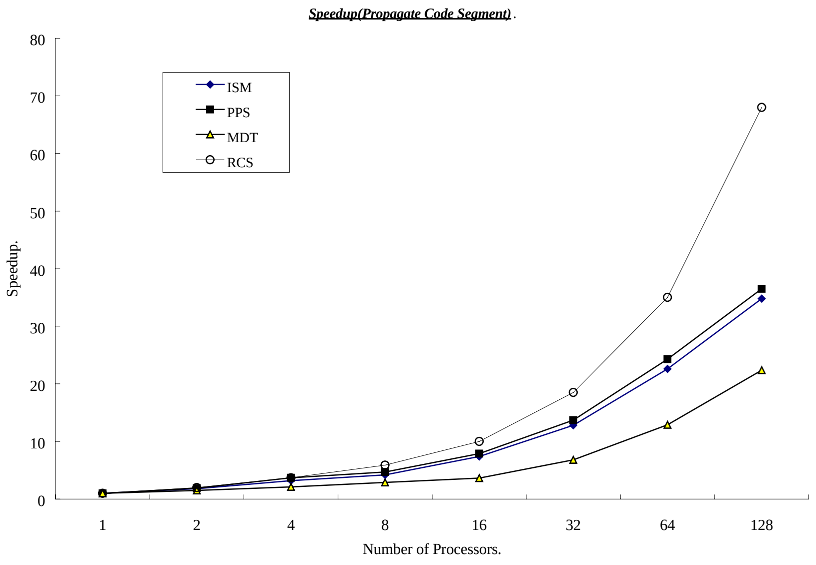

Fig. 13. The speedup of the propagate code segment in the SEESMA environment.

6. CONCLUDING REMARKS

Due to the available parallelism, concurrent execution of loops is vital to the performance of shared-memory multiprocessors. An efficient non-uniform loop scheduling scheme, called the RCS method, has been developed, and its effectiveness has been

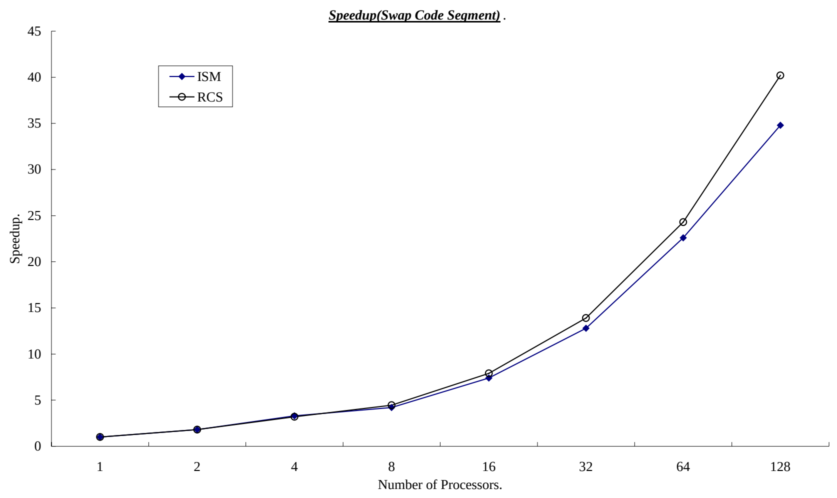

Fig. 14. Speedup of the swap code segment in the SEESMA environment.

shown by executing program codes on a CONVEX SPP 1000 machine. Based on solving a set of Diophantine equations, and using the idea of the minimum dependence distance, we have presented the procedures for determining scheduling information, including the number of parallelizable iterations, the hopping gate and the hopping distance. As the RCS method does not define the scheduling approach, it must be incorporated with a chunk size control function. Preliminary evaluation has shown that GSS is the best scheme to use with the RCS method.

Our performance evaluation shows that the RCS method is significantly affected by the parallelism of the target program. If the parallelism is sufficiently large, the RCS method will reliably extract any available parallelism without introducing any serious synchronization overhead. Unlike other loop partition techniques, it can successfully eliminate multi-barrier synchronization and release parallelizable iterations earlier. Instead of relying on cross-block synchronization primitives and delay instructions, the RCS method dynamically adjusts the hopping information to maintain dependence correctness.

In the future, we plan to further extend the RCS method to a multi-dimensional iteration space and establish an appropriate dynamic data allocation mechanism to reduce data conflict.

REFERENCES

1. C. D. Polychronopoulos, "Guided self scheduling: A practical scheduling scheme for parallel supercomputers," *IEEE Transactions on Computers*, Vol. C-36, 1987, pp. 1425-1439.
2. Y. W. Fann, C. T. Yang, and C. J. Tsai, "IPLS: An intelligent parallel loop scheduling for multiprocessor system," in *Proceedings of the 1998 International Conference on Parallel and Distributed Systems*, 1998, pp. 775-782.
3. T. H. Tzen and L. M. Ni, "Trapezoid self-scheduling: A practical scheduling scheme

for parallel compilers," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 4, 1993, pp. 87-98.

4. E. Rosti, E. Smirni, L. W. Dowdy, C. Serazzi, and K. C. Sevcik, "Processor saving scheduling policies for multiprocessor systems," *IEEE Transactions on Computers*, Vol. 47, 1998, pp. 178-189.
5. E. P. Markatos and T. J. LeBlanc, "Using processor affinity in loop scheduling on shared-memory multiprocessor," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 5, 1994, pp. 379-400.
6. Y. Yan, C. Jin, and X. Zhang, "Adaptively scheduling parallel loops in distributed shared-memory systems," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 8, 1997, pp. 70-81.
7. D. K. Chen and P. C. Yew, "An empirical study on DOACROSS loops," in *Proceedings of Supercomputing*, 1991, pp. 620-632.
8. A. R. Hurson, K. Kavi, and J. T. Lim, "Cyclic staggered scheme: A loop allocation policy for DOACROSS loops," *IEEE Transactions on Computers*, Vol. 47, 1998, pp. 251-255.
9. V. P. Krothapalli and P. Sadayappan, "Dynamic scheduling of DOACROSS loops for multiprocessors," in *Proceedings of the International Conference on Database, Parallel Architectures and Their Applications*, 1990, pp. 66-75.
10. T. H. Tzen and L. M. Ni, "Dependence uniformization: A loop parallelization technique," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 4, 1993, pp. 547-558.
11. S. Y. Tseng, C. T. King, and C. Y. Tang, "Minimum dependence vector set: A new compiler technique for enhancing loop parallelism," in *Proceedings of the 1992 International Conference on Parallel and Distributed Systems*, 1992, pp. 340-346.
12. D. K. Chen and P. C. Yew, "A scheme for effective execution of irregular DOACROSS loops," in *Proceedings of the 1992 International Conference on Parallel and Distributed Systems*, Vol. II, 1992, pp. 285-292.
13. M. J. Hwu and D. J. Buehrer, "An improved scheme for effective execution of nested loops with irregular dependence constraints," *Journal of the Chinese Institute of Electrical Engineering*, Vol. 2, 1995, pp. 107-118.
14. S. Punyamurtula and V. Chaudhary, "Minimum dependence distance tiling of nested loops with non-uniform dependences," in *Proceedings of the 6th IEEE Symposium on Parallel and Distributed Computer*, 1994, pp. 74-81.
15. D. L. Pean, C. C. Wu, H. T. Chua, and C. Chen. "Effective parallelization techniques for non-uniform loops," in *Proceedings of the 21st Australian Computer Science Conference*, 1998, pp. 393-404.
16. C. K. Cho and M. H. Lee, "A loop parallelization method for nested loops with non-uniform dependences," in *Proceedings of the 1997 International Conference on Parallel and Distributed Systems*, 1997, pp. 314-321.
17. Richardson, *Exemplar Architecture*, Convex Computer Corporation, United States of America, Texas, 1994.
18. Richardson, *CONVEX Exemplar Programming Guide*, Convex Computer Corporation, United States of America, Texas, 1994.
19. J. J. Dongarra, C. B. Moler, J. R. Bunch, and G. W. Stewart, *LINPACK Users' Guide by SIAM*, Philadelphia, 1979, ftp://ftp.ucar.edu/ftp/dsl/lib/linpack/.

20. A. A. Dubrulle, *A Version of EISPACK for the IBM 3090VF*, TR G320-3510, IBM Scientific Center, Palo Alto CA, 1988, http://elib.zib.de/netlib/eispack/.
21. E. Grosse, "http://cm.bell-labs.com/netlib/itpack/index.html," 1996.
22. J. Adams, P. Swarztrauber, and R. Sweet, *A Package of FORTRAN Subprograms for the Solution of Separable Elliptic Partial Differential Equations*, The National Center for Atmospheric Research Boulder, Colorado, U.S.A., Version 3.2, November, 1988, ftp://ftp.ucar.edu/ftp/dsl/lib/fishpak.
23. J. P. Su, C. C. Wu, and C. Chen, "Reducing the overhead of migratory-shared access for the linked-based directory coherent protocols in shared memory uultiprocessor systems, " in *Proceedings of the 1996 International Computer Symposium on Computer Architecture*, 1996, pp. 160-167.
24. J. E. Veenstra and R. J. Fowler, MINT Tutorial and User Manual, Technical Report, 452, University of Rochester, New York, 1994.
25. H. Zima, *Supercompilers for Parallel and Vector Computers*, ACM Press, New York, 1990.
26. U. Banerjee, *Dependence Analysis for Supercomputing*, Kluwer Academic Publishers, 1988.

Der-Lin Pean (平德林) is a Ph.D. candidate in Computer Science and Information Engineering at the National Chiao Tung University, Taiwan R.O.C. He received his B.S. degree in Information and Computer Engineering at Chung Yuan Christian University, Taiwan, R.O.C. He served as a lecturer in the Department of Computer and Information Engineering, as well as Employment and Vocational Training Administration, Council of Labor Ministry, Taiwan R.O.C. His current research interests include computer architecture, personal computer system architecture design, parallel processing system design, parallelizing compiler techniques, and microprocessor system design.

Huey-Ting Chua (蔡慧婷) received the B.S. and M.S. degree in 1997 and 1999 respectively, both in Computer Science and Information Engineering from National Chiao Tung University, Taiwan. Her major research interest is parallel compiler.

Cheng Chen (陳正) is a professor in the Department of Computer Science and Information Engineering at National Chiao Tung University, Taiwan, R.O.C. He received his B.S. degree from the Tatung Institute of Technology, Taiwan, R.O.C. in 1969 and M.S. degree from the National Chiao Tung University, Taiwan, R.O.C. in 1971, both in electrical engineering. Since 1972, he has been on the faculty of National Chiao Tung University, Taiwan, R.O.C. From 1980 to 1987, he was a visiting scholar at the University of Illinois at Urbana–Champaign. During 1987 and 1988, he served as the chairman of the Department of Computer Science and Information Engineering at the National Chiao Tung University. From 1988 to 1989, he was a visiting scholar of the Carnegie Mellon University (CMU). Between 1990 and 1994, he served as the deputy director of the Microelectronics and Information Systems Research Center (MIRC) in National Chiao Tung University. His current research interests include computer architecture, parallel processing system design, parallelizing compiler techniques, and high performance video server design.