M-hopping Method: An Effective Loop Scheduling Scheme for Non-uniform Dependence Loops

Huey-Ting Chua, Der-Lin Pean and Cheng Chen

Department of Computer Science and Information Engineering
1001 Ta Hsueh Road, Hsinchu, Taiwan, 30050, Republic of China
Tel:(886)35712121 EXT:54701~54704, Fax:(886)35724176

Abstract

In general, synchronization mechanism can be used to preserve dependence constraints of any nested loops, and it can be combined with loop scheduling scheme to form a uniform framework. Meanwhile, correct execution order and balance workload distribution will be achieved. In this paper, we propose a new scheduling scheme called M-hopping method to schedule non-uniform dependence doubly nested loop on multiprocessor systems. To initialize a set of hopping information, our approach is based on the concept of minimum dependence distance. During runtime, hopping information will be used to adjust number of parallelizable iterations. According to our experimental results, if loops carry sufficient parallelism, our proposed method will reliably exploit parallelism, and outperform most of the existing non-uniform dependence loop scheduling schemes by 20.29% in average.

Keyword: Scheduling, Parallelizing Compiler, Loop, Multiprocessor , Synchronization.

1 Introduction

Upon demanding of fast computation power, multiprocessor has been one of the popular architectural designs. To fully utilize the entire system, workload of each processor is expected to be as even as possible [10]. Ever since loops are major source of parallelism, loop scheduling problem has been studied to achieve equal and fair workload distribution; besides reducing synchronization, communication and threads management overhead [4]. In spite of generality of DOALL loop scheduling schemes [4, 8, 10, 12, 14, 16], non-uniform dependence loop scheduling schemes are rarely found.

Non-uniform dependence loops, which have irregular dependence on iteration level, are mainly due to coupled subscripts [1]. Fortran numerical packages such as Linpack [19], Eispack [20], Itpak [21] and Fishpak [22] are typical examples. They are library packages and can be called very frequently in users' programs for scientific and engineering computing. Henceforth, it is worthy and important to develop an efficient loop scheduling scheme for them.

Among the existing approaches, although Staggered Distribution [5] performs outstandingly in data-flow machine, the method is not adaptable for shared memory multiprocessor systems. On the other hand, most of the later systems acceptable scheduling schemes [2, 6, 7, 13, 15] introduce significant delay overhead in preserving dependence correctness. In this paper, we intend to develop a non-uniform dependence loop scheduling scheme, which is free from evident delay overhead, and capable of dynamically extracting parallelism. This newly developed method is called M-hopping method; the target platform is shared-memory or distributed shared-memory MP system.

```
for I = 1, U_I
  for J = 1, U_J
    S_d : A(f1(I,J), f2(I,J)) = ...
    Su:           ... = (f3(I,J), f4(I,J))
  endfor
endfor
```

Figure 1 Program Model

To simplify discussion, program model as shown in Figure 1 is used for description and preliminary evaluation. It has been widely discussed in several previous researches [2, 3, 6, 9, 11, 13, 15]. Only loops that are parallelizable along outermost loop level are considered. Each iteration will be given a unique identifier by the formula $(I-1)*U_I + J$; and iterations are scheduled in order.

Performance evaluations are carried out on CONVEX SPP 1000. According to our evaluation result, regardless with number of available processors, M-hopping method substantially eliminates delay overhead as well as multi barrier synchronization. When eight processors are used and real benchmarks are considered, M-hopping method is better than Index Synchronization Method [15] by 15.25% in average. If it is compared with loop partition techniques [9,11], it is superior by 25.33% in average. The amazing outcome has truly inspired the further extension.

Organization of this paper is as follows. Section 2 is related work; section 3 gives the basic concepts and principles of M-hopping method. Further generalization of the method is also done in this section. Performance evaluations will be presented in section 4 and section 5 is concluding remarks.

2 Related Work

Conventionally, performance of a loop scheduling mechanism is determined by five factors, workload balancing, scheduling overhead, communication overhead, threads management overhead and synchronization overhead [4]. Among them, synchronization, scheduling and threads management overheads are our major concerns. Execution time of loop body is assumed consistent for all iterations. Therefore, workload imbalance would be simply caused by scheduling mechanism.

Several previous works had been devoted to effectively schedule non-uniform dependence loops, including Index Synchronization Method [15], Group Synchronization Method [13], Static Strip Scheme [2,6] and so forth.

Index Synchronization Method (ISM) is proposed

to schedule uniform dependent two-way nested loop obtained from Dependence Uniformization. The basic idea is serially executing inner loop, and the outer loop is performed concurrently with insertion of synchronization, which is implemented through a globally shared array incorporated with delay operation. Performance of ISM will probably be constrained by Dependence Uniformization, because additional delay overhead is introduced. One of the variations of ISM is Group Synchronization Method; delay overhead is also inevitable, and it may restrict the performance gain.

Static Strip Scheme (SSS) is another approach that also associates with Dependence Uniformization. A strip is a group of iterations to be sequentially executed. Cross strips dependence are preserved through explicit synchronization primitives, for instance, post&wait. As the name implies, SSS is classified as static scheduling. Again, it is also constrained by Dependence Uniformization technique. Besides, different synchronization primitives result in distinct performance behavior.

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 with loop partition techniques during compile time. Any existing chunk size control functions can be applied to guide the scheduling manner such as Pure Self Scheduling, Chunk Self Scheduling and Guided Self Scheduling [10]. Barrier synchronization is inserted at the end of each tile. In spite of its intuitiveness and simplicity, it is closely restricted by both loop partition techniques and loop scheduling schemes. Multi barrier synchronization are unavoidable at the end of each tile, but poor choices of scheduling scheme can further incur apparent overhead.

Following, we will study a scheduling scheme, which is inessential to decompose iteration space into either parallelizable or sequential blocks. Meanwhile, delay instructions, synchronization primitives, and multi barrier synchronization will be eliminated.

3 Basic Concepts and Principles of M-hopping Method

M-hopping method is named after the feature that number of parallelizable iterations (M) hops across subsequent iterations as a result of relaxation of dependence constraint. The basic concept of M-hopping method is illustrated in Figure 2 and Figure 3.

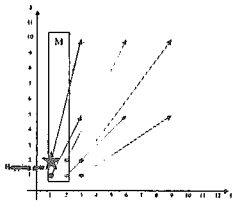

Figure 2 Before execution

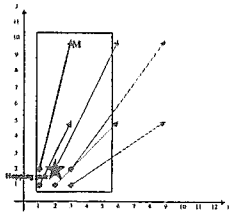

Figure 3 The first hop of M

To achieve the goal, we simply keep two global variables to track the dependence constraints, they are hopping gate and hopping distance. Hopping gate is set to monitor the occasion of M's hopping; while hopping distance is used to define distance to hop. In Figure 2, initially, hopping gate is set at iteration (1,2) and M covers the first 20 iterations range from (1,1) to (2,10). Once iterations (1,1) and (1,2) have been executed, 30 iterations from (3,1) to (5,10) are releasable. By the time, M will be incremented by 30 iterations and hopping gate shifted ahead by 10 as shown in Figure 3. These 30 iterations are so called hopping distance. If iterations before (2,2) have also been executed, another 30 iterations range from (6,1) to (8,10) will be added to M. Our approach to dependence analysis is based on Dependence Convex Hull Theory [15]. For program model presented in Figure 1, diophantine equation set and dependence vector functions can be expressed as below:

Diophantine Equations (1)

$s_1, s_2, s_3, t_1, t_2, t_3 \in R$

$$\begin{cases} i_1 = x \\ j_1 = y \\ i_2 = s_1 x + s_2 y + s_3 \\ j_2 = t_1 x + t_2 y + t_3 \end{cases}$$

Dependence vector functions (2)

$$\begin{cases} d_i(x, y) = (s_1 - 1)x + s_2 y + s_3 \\ d_j(x, y) = t_1 x + (t_2 - 1)y + t_3 \end{cases}$$

Following, a simple application of M-hopping method will be illustrated.

3.1 M-hopping Method for Backward Growing Pattern Loops

Dependence vectors of some loops may increase progressively along particular loop dimension. In [9], this kind of loop is said to have growing pattern on that loop dimension. If a loop has growing pattern on outermost loop level and dependence vectors are all flow dependent, it is called Backward Growing Pattern Loops, we give their formal definitions as below.

Definition 1 (Backward Growing Pattern Loop (BGPL)): Given a two way nested loop L as shown in Figure 1, and dependence vector function on outermost loop level V_I is positive real. If $V_I(i_1) \le V_I(i_2)$ for any x and y satisfy $1 \le i_1 < i_2 \le U_I$, then L is said to have backward growing pattern on loop dimension I, and L is called Backward Growing Pattern Loop (BGPL). □

If a loop is BGPL, then $V_I = \{d_i(x, y) \mid d_i(x,y) = (s_1-1)x + s_3$ and $[(s_1-1)x_1 + s_3] \le [(s_1-1)x_2 + s_3]$ for any $L_I \le x_1 < x_2 \le U_I\}$.

Example 1 (L_1) :

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

Figure 4 Array assignment pattern of L_1

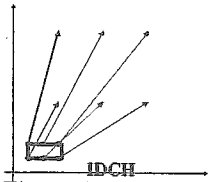

Figure 5 Dependence graph of L_1

Consider example 1, L_1 is an example of

non-uniform dependence loop, its dependence graph is shown in Figure 5. The diophantine equation set is $\{i_1 = x, j_1 = y, i_2 = 3x, j_2 = 5y\}$, and dependence vector set = $\{(3-1)x, (5-1)y\} = \{2x, 4y\}$. Dependence Convex Hull (DCH) is the intersection of eight half spaces [15], $\{(x, y) \mid 1 \le x \le 10\} \cap \{(x, y) \mid 1 \le y \le 10\} \cap \{(x, y) \mid 1 \le 3x \le 10\} \cap \{(x, y) \mid 1 \le 5y \le 10\}$, it forms a rectangular shape in dependence graph. For all $1 \le i_1 < i_2 \le 10$, $2i_1 < 2i_2$, therefore it is a BGPL.

From the definition of BGPL, a few properties of BGPL are summarized.

Property 1: Integer Dependence Convex Hull (IDCH) [11] of BGPL always situates at the side of $d_i(x, y) > 0$. Geometrically, it tells that dependence vectors within an outer loop instance I_i or anti-dependence 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 represents initial dependence offset along loop index I. Consider the dependence graph of L_1, dependence heads of (1,1) and (2,1) are (3,5) and (6,5). Their dependence distance on I dimension is 6-3 = 3.

Property (3) : s_2 is always 1 for BGPL; dependence heads of the two dependence tails (i, j) and (i, j+1) locate at the same outer loop index $i + d_i(x, y) = i + (s_1-1)x + s_3$. For example, dependence tails (1,1) and (1,2) on Figure 5 have dependence heads on (3,5) and (3,10). They both situates at the same column I_i=3.

The above features of BGPL and hopping information will be maintained as follows.

3.1.1 Determination of M

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

Definition 2 (M): Given a normalized two-level nested loop L, let lower and upper bound on outer loop index I be L_I and U_I respectively. U_J is upper bound on inner loop dimension J, and I_i to I_{i+j} are adjacent columns of iterations on loop dimension I, where $L_I \le I_i \le I_{i+j} \le U_I$. For any iterations $i \in [I_i, I_{i+j}]$, if it satisfies one of the following conditions :

(i) it is dependence free;

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

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

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

For DOALL loops, M is initially equal to the total number of iterations on iteration space (N), or $[I_i, I_{i+j}]=[L_I, U_I]$, and $M=[(U_I-L_I+1)*U_J]$. However, M always less than N for non-uniform dependence loop initially, due to existence of dependences. To initialize M, the number of parallelizable iterations before IDCH and the first group of parallelizable iterations on IDCH are computed. It is formally stated in Theorem 1.

Theorem 1 : Given a BGPL (L) as shown in Figure 1, M is initialized as $\lfloor(s_1*i_{left}) + s_3 - 1\rfloor * U_J$, where i_{left} is leftmost extreme of IDCH on loop dimension I, s_1 and s_3 are two coefficients in diophantine equations (1).

Proof: Parallelizable region before IDCH covers $[(i_{left}-1)*U_J]$ iterations. Subsequently, the first group of parallelizable iterations next to the region is certainly contributed by the leftmost extreme points of IDCH according to the definition of BGPL. Property (3) implies that dependence distance of the leftmost extreme is exactly $[(s_1-1)*i_{left}]+s_3$. So that, the initial value of M is :

$[(i_{left}-1)*U_J] + [(s_1-1)*i_{left}+s_3] * U_J$

$= \{i_{left}-1 + [(s_1-1)*i_{left}] +s_3\}* U_J$

$= [(s_1*i_{left}) +s_3 -1]* U_J$

$= \lfloor(s_1*i_{left}) +s_3 -L_I\rfloor * U_J$ (convert real to integer) □

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

3.1.2 Determination of Hopping Gate

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

Definition 3(Hopping Gate): Given a two-way nested loop L, let n be identifier of an iteration and M as defined in Definition 2. If iterations $i \le n$ are all parallelizable, and before their complete execution, any iterations $j > M$ cannot be executed; we call n as hopping gate. □

For BGPL, hopping gate is set at the last dependence tails of an outer loop instance I_i. However, the last dependence tail on each outer loop instance I_i may vary from each other, we will conservatively choose the largest one as hopping gate. Based on theorem stated in linear programming [11], the initial value of hopping gate can be defined as below directly.

Theorem 2: Given a BGPL (L) as shown in Figure 1, hopping gate is initialized as $[(i_{left}-1)* U_J] +j$, where j = max(j') and (i', j') are extreme points of IDCH, i_{left} is the leftmost extreme of IDCH on loop index I.

Proof: The proof is trivial. Hopping gate is invalid for parallelizable region; if it exists at left of IDCH, hopping gate will be shifted ahead by $(i_{left}-1)* U_J$. Since the maximum j appears at extreme points, hopping gate is initialized as $[(i_{left}-1)* U_J] +j$. □

Consider L_1 in example 1 again, among the extreme points (1,1), (3,1), (1,2) and (3,2), maximum j is 2 and i_{left} is 1, so that hopping gate will be initialized as (1-1)*10+2 = 2. It means that, once iterations (1,1) and (1,2) are completely executed, their corresponding dependence heads will be released. At the same time, hopping gate increments by 10 (U_J), the newly updated hopping gate situates at iteration 12 or (2,2).

3.1.3 Determination of Hopping Distance

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

Definition 4 (Hopping Distance): Given a two-way nested loop L, let U_I and U_J be upper bound on loop indices I and J respectively. M as defined in Definition 2, hopping gate (G) is defined in Definition 3,

and I_i to I_{i+j} are adjacent columns of parallelizable iterations on loop dimension I satisfy $(M / U_J) < I_i \leq I_{i+j} \leq U_I$. For all iterations $j \leq G$, if they have been completely executed, then iterations $i \in [I_i , I_{i+j}]$ are all releasable. The maximal length of $[I_i , I_{i+j}]$ is called the hopping distance. □

For BGPL, hopping distance can be determined by Theorem 3.

Theorem 3 : Given a BGPL (L) as shown in Figure 1, hopping distance of L is $(\lfloor s_1 \rfloor * U_J)$.

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

Unlike M and hopping gate, hopping distance is determined at compile time, but remains unchanged permanently. In example 1, 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 totally 20+30 = 50 parallelizable iterations range from iterations (1,1) to (5,10). Both the hopping gate and hopping distance will be active until iterations within IDCH have all been executed.

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

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

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

$$\lfloor (s_1 * i_{left}) + s_3 - 1 \rfloor * U_J$$
$$= \lfloor (s_1 - 1) * i_{left} + i_{left} + s_3 - 1 \rfloor * U_J$$
$$\geq \lfloor i_{left} - 1 \rfloor * U_J + \lfloor (s_1 - 1) * i_{left} + s_3 \rfloor * U_J$$
$$\geq [i_{left} - 1] * U_J + U_J$$
$(d_i(i_{left}, 0) = [(s_1 - 1) * i_{left} + s_3] \geq 1$, else it is meaningless)
$$\geq (i_{left} - 1) * U_J + j \qquad (1 \leq j \leq U_J)$$

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

Based on the corollary, when M-hopping method is applied, dynamic parallelism extraction is achieved by earlier relaxation of parallelizable iterations. Following, we will discuss the generalization of the M-hopping method.

3.3 Generalization of the M-hopping Method

If the loop does not belong to the BGPL, M-hopping method works similarly to BGPL, but slight modification on initialization of hopping information. Due to probable existence of coefficient s_2 in the dependence vector function $d_i(x,y)$, number of parallelizable iterations is determined through the concept of 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 on the extreme points.

Consequently, minimum dependence distance of a flow dependence loop is {md | md = min[$d_i(x,y)$] where (x, y) are extreme points of IDCH and md ∈ R}. Let U_J as defined in Figure 1, iterations within $\lfloor md \rfloor * U_J$ are parallelizable [14]. The hopping information can then be determined by Corollary 2.

Corollary 2 : Given a non-BGPL flow dependence loop (L) as shown in Figure 1, hopping information is 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 $= \lfloor md \rfloor * U_J$

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

Proof: (i) Similar to Theorem 1, M is initialized as:

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

(ii) Since iterations in $\lfloor md \rfloor * U_J$ are parallelizable, and so $\lfloor md - 1 \rfloor * U_J + j$. Therefore, 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 M and hopping gate as defined in (i) and (ii), assume that once iterations before hopping gate have all been executed, total number of releasable iterations is $\lfloor d \rfloor * U_J$. If d > md, there may exist dependence vectors locate within $\lfloor d \rfloor * U_J$, then $\lfloor d \rfloor * U_J$ will definitely not parallelizable. If d < md, iterations in $\lfloor d \rfloor * U_J$ are surely parallelizable, but d is not the maximal length of parallelizable iterations. As a result, d must be equal to md, and thus hopping distance $= \lfloor md \rfloor * U_J$. □

On the other hand, If $d_i(x, y) = 0$ passes through IDCH, iterations within IDCH can be flow-dependence tails or anti-dependence heads. By using array duplication and renaming, anti-dependences can be removed completely. Moreover, some dependence tails may have dependence heads locate at the same loop instance I_i. $D_j(x, y) = 0$ implies that there exists intra-iteration dependence for all iterations along the line segment $d_i(x, y) = 0$. As long as single iteration will be executed serially, intra-iteration dependence is preserved. M-hopping method works similarly, but determination of minimum dependence distance follows Theorem 4.

Theorem 4: If $d_i(x,y) = 0$ passes through IDCH, and $d_j(x,y) = 0$, then absolute minimum value of $d_i(x,y)$ appear at either extreme points or iterations next to intersection points of line segment $d_i(x,y) = 0$ with IDCH.

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

3.4 Algorithms of the M-hopping Method

Base on the above analysis, our M-hopping method

can be generalized easily. During compile phase, loop is examined whether interchangeable [15]. If so, M will hop across loop dimension which has more parallelism; or smaller value of i_{right}. Detail procedures are stated in Algorithm 1.

Algorithm 1: Compile phase of M-hopping method

```
begin
  hopping_gate := 0;
  hopping_distance := 0;
S := Banerjee(L); /* Determine diophantine equation
   sets */
D := Tzen_and_Ni(S); /* Determine IDCH */
if (Is_DCH_Empty() = TRUE)
 then   /* identify L as DOALL loop */
  M := N;
 else
  I := Transform_DCH_To_IDCH(D);
  switch (Determine_Position_Of_IDCH(S,V))
  begin
  case 1: the loop can be reconstructed as DOALL
loop
       M := N;
  case 2: (IS_BGPL(S) = TRUE)
       M := ⌊(s1*ileft) +s3 -1⌋ * UJ; /* Theorem 1*/
       hopping_gate := [(ileft-1)* UJ] +j;
          /* Theorem 2*/
       hopping_distance := ⌊s1⌋ * UJ;
          /* Threorem 3*/
  case 3: di(x,y) = 0 does not pass through the
   IDCH
       md := Determine_Minimum_ Depend-
   ence_Distance(V, D);
       M = ⌊(ileft-1) + md⌋ * UJ; /* Corollary 1*/
       hopping gate ⌊ileft + md -2⌋ * UJ + j;
       hopping distance = ⌊md⌋ * UJ;
  case 4: di(x,y) = 0 pass through the IDCH
       md := Determine_Overall_MDD(V, D);
       M = ⌊(ileft-1) + md⌋ * UJ; /* Corollary 1*/
       hopping gate ⌊ileft + md -2⌋ * UJ + j;
       hopping distance = ⌊md⌋ * UJ;
   endswitch
end
```

The complexity of this algorithm is bounded by forming DCH, transforming DCH to IDCH, determining position of IDCH and finding minimum dependence distance. The worst case complexity is $O(n^2) + 3O(m)$, where n is number of half space and m is number of integer points along parameter of DCH.

During execution phase, M-hopping method follows Algorithm 1. Notice that, M-hopping method defines the number of parallelizable iterations, but it does not tell the scheduling manner. So that M-hopping method must be incorporated with a predefined chunk size control function.

Algorithm 2: Execution phase of the M-hopping method

```
begin
 /* Count is used to record id of the currently
scheduled iteration and Executed is used to record id
of 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)
        lock(t);
        Executed = max(Executed,
            bound[processor_id].ub);
        /*processor triggers hopping*/
        if (bound[processor_id].lb ≤ hopping_gate
                    and
            bound[processor_id].ub ≥ hop-
                    ping_gate)
          /*check whether within IDCH region.*/
          if (hopping_gate ≥
[(iright-1)*UJ+hopping_gate])
                M := N;
             else
                hopping_count := [(Exe-
             cuted-hopping_gate)/UJ + 1];
                hopping_gate := hopping_gate +
             (hopping_count*UJ);
                M := M + (hopping_count *
             hopping_distance);
             unlock(t);
end
```

4 Preliminary Performance Evaluations

In this section, performance evaluations are studied to practically verify the effectiveness of M-hopping method. The experimental programs include program models used to be discussed in the previous section, and some practical code segments. The experimental evaluations are carried out on CONVEX SPP-1000 clustered multiprocessor system [17,18], which has eight PA-RISC processors, and memory is configured as distributed shared.

Our interesting performance metric is execution time. It is further divided into five components, they are busy time (average parallel execution time of each activated processor), waiting time (processors' average waiting time while others are busy executing), scheduling overhead (average latency of scheduling a chunk), fork and barrier overhead (average time spent at the end of each tile) as well as initialization overhead (the time consumed in initializing scheduling scheme and chunk to be executed).

The comparative mechanisms include Index Synchronization Method (ISM) [10], Minimum Dependence Distance Tiling (MDT) [14], Parallelization Part Splitting (PPS) [15] and Growing Pattern Detection (GPD) [15] will be taken into account.

As described in section 3, M-hopping method ought to be associated with one of the dynamic scheduling schemes. To determine which is best, we have applied Pure Self Scheduling (PSS), Chunk Self Scheduling (CSS), Guided Self Scheduling (GSS) [1] and Trapezoid Self Scheduling (TS) [3] on L_1. As you can see in Figure 7, association with GSS is superior most.

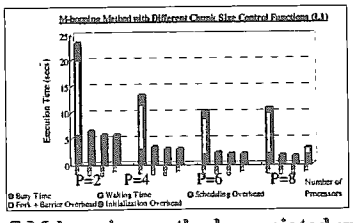

Figure7 M-hopping method associated with vari-

various chunk size control functions

Chunk size of PSS is constant 1; processors spend a lot of time in parallel executing single iteration, causing heavier initialization and scheduling overhead. These overheads can be reduced by increasing chunk size; by the time, subroutine is called to complete a chunk of iterations rather than single iteration. The faster the chunk size converges to 1, the fewer scheduling overhead and initialization overhead would be. CSS, GSS and TS are beneficial from the feature. Formula of TS may fail sometimes, and result in imbalance workload. In the following evaluations, M-hopping method will be incorporated with GSS to reliably extract parallelism, GSS will also be applied on PPS, MDT and GPD if they are available.

4.1 Performance Evaluation on Program Model

Table 1 shows the detail of program model L_1 with associations of different scheduling 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 totally.
M-hopping Method	M = 120; Hopping gate = 10; Hopping distance = 150; i_{right} = 6

Table 1 Scheduling related information of various approaches

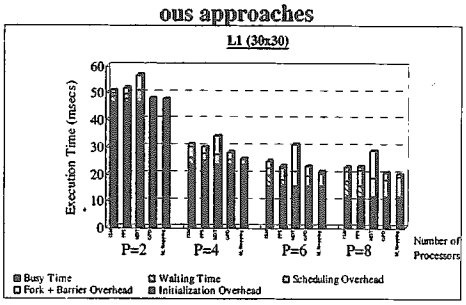

Figure 8 Execution time of L_1

Figure 8 is execution time of L_1 incorporated with different scheduling and partition mechanisms; number of processors (P) varies from 2 to 8. Among them, execution time of MDT is the longest no matter how many processors are used. It involves 15 times processes fork-join and so barrier synchronization; the overhead gets apparent when P becomes large. Besides, the limited tile size diminishes parallelizable iterations; consequently, delay the scheduling time. While L_1 is scheduled with ISM, system consumes visible time waiting for the completion of other iterations. Finally, the delay overhead causes low performance gain. Performance behavior of PPS is similar to ISM, because scheduling manner of iterations in IDCH region follows ISM. In case of IDCH region is large, performance of PPS will be bounded by the delay overhead inherited from ISM. Although GPD is specific to growing pattern loop, it is still restricted by multi barrier synchronization overhead. Its execution time is eventually worse than M-hopping method. In overall, M-hopping method takes fully advantage of removing system waiting time and multi barrier synchronization; as a result, it outperforms any others.

4.2 Performance Evaluation on Practical Code Segments

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

Figure 9a Code segment 1 **Figure 9b Code segment 2**

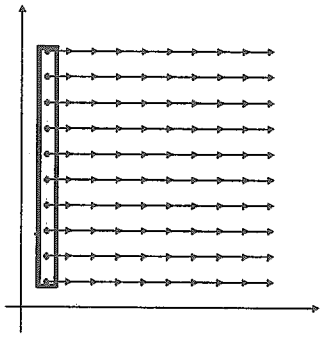

Figure 9c Dependence graph

In addition to the program models, practical code segments have also been taken into account. Evaluations on *Propogate* code segment are given at first. It is one of the dependence pattern widely found in Linpack and Eispack. The code segments are shown in Figure 9a and Figure 9b, both of them share a dependence graph as presented in Figure 9c. Result computed by iterations (1,j) will be propagated toward iterations (i,j) for all i ∈ [1, Q]. Detail scheduling information of distinct scheduling schemes are presented in Table 2.

Figure 10a and 10b are execution time and speedup of propagate code segment. Upper loop bound is set as 30 on either dimensions. For this particular code pattern, MDT tiles the iteration space with tile length equals to 1, or each column of iterations forms a parallelizable tile. The 30 times processes fork-join and barrier synchronization result in extremely significant overhead, therefore seriously degrade performance. ISM schedules the loop with delay factor equals to 1. Both the ISM and PPS are rather worse than M-hopping method because of their substantial waiting overhead. Again, M-hopping method retains its superiority. According to Figure 10b, we can further conclude that, if a given loop inherits adequate parallelism, M-hopping method affords to extract it with little hopping overhead.

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*
M-hoppin g Method	M = 30; Hop-ping gate = 30 Hopping dis-tance = 0	M = 30; Hop-ping gate = 30; Hopping dis-tance = 30

Table 2 Scheduling related information of various approaches

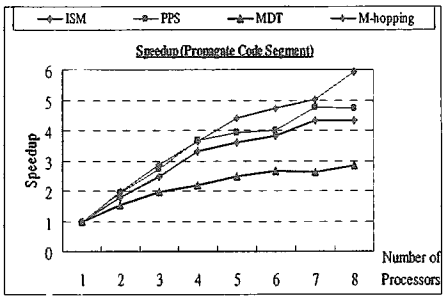

Figure 10a Execution time (Propagate)

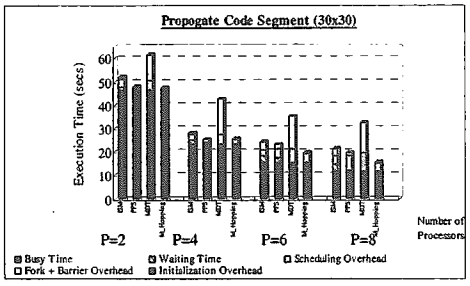

Figure 10b Speedup (Propagate)

Another evaluated code segment is called *Swap* code, which serves as kernel of Fishpak. Figure 11a and 11b are code pattern and their dependence graph. We find that IDCH occupies the whole iteration space, moreover $d_i(x,y) = 0$ goes through IDCH vertically and $d_j(x,y) = 0$. By the time, MDT fails and PPS works completely identical to ISM, because both the left and right tiles are empty. Right half of Table 2 lists the ISM and M-hopping method when they are applied on swap code.

Execution time and speedup graphs of swap code segment are presented in Figure 12a and Figure 12b respectively. Since there are at most 30 parallelizable iterations to be executed at any time, the limited parallelism causes significant scheduling overhead for M-hopping method. Except when more than 5 processors are used simultaneously, performance gain of M-hopping method is just as good as ISM.

In summary, performance of M-hopping method is exactly proportional to the underlying parallelism. If parallelism is sufficiently large, M-hopping method will reliably extract parallelism without introducing intolerable scheduling overhead. Consequently, M-hopping method is a well-encouraged approach for scheduling non-uniform dependence doubly nested loops.

```
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
```

Figure 11a Swap code segment

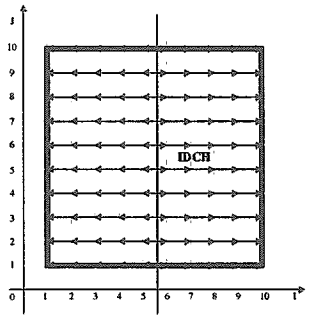

Figure 11b Dependence graph of swap code

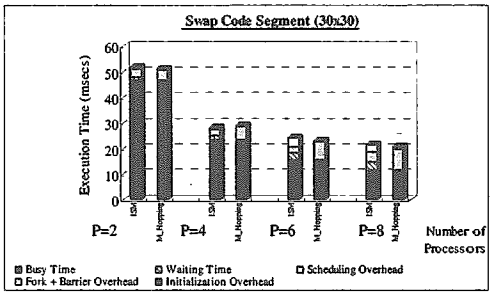

Figure 12a Execution time (Swap)

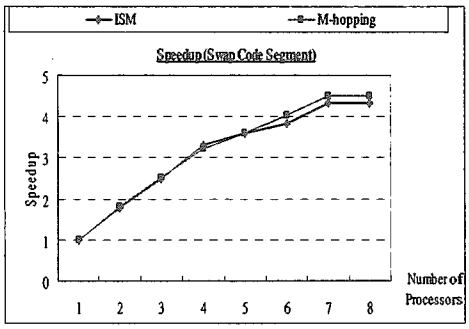

Figure 12b Speedup (Swap)

5 Concluding Remarks

Due to the rich parallelism, concurrent execution of loops is vital to the performance of shared-memory multiprocessors. An efficient non-uniform loop scheduling scheme, called M-hopping method, is studied in this paper; and its effectiveness is shown by practically executing program code on CONVEX SPP 1000 machine. Based on solving set of diophantine equations and idea of minimum dependence distance, we have presented the de-

termination procedures of hopping information, including number of parallelizable iterations, hopping gate and hopping distance. Ever since M-hopping method does not define the scheduling manner, it must be incorporated with one of the chunk size control function. The preliminary evaluation ensures that Guided Self Scheduling is more profitable to M-hopping method.

Our performance evaluation shows that the proposed method is significantly affected by the parallelism of target program. If parallelism is sufficiently large, M-hopping method will reliably extract parallelism without introducing serious synchronization overhead. Unlike any 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, it dynamically adjusts the hopping information to maintain dependence correctness.

In the future, we will further extend the M-hopping method to multiple-dimensional iteration space and establish an appropriate dynamic data allocation mechanism to reduce data conflict.

NSC 87-2213-E009-049.

References

[1] D. K. Chen and P. C. Yew, "An Empirical Study on DOACROSS Loops", Proc. Supercomputing, pp-620-632, 1991.

[2] D. K. Chen and P. C. Yew, "A Scheme For Effective Execution of Irregular DOACROSS Loops", Proc. of ICPADS, Vol. II, pp. 285-292, Aug. 1992.

[3] C. K. Cho and M. H. Lee, "A Loop Parallelization Method for Nested Loops with Non-uniform Dependences", Proc. of ICPADS, South Korea, pp. 314-321, Dec. 1997.

[4] Y. W. Fann, C. T. Yang and C. J. Tsai, "IPLS : An Intelligent Parallel Loop Scheduling for Multiprocessor System", Proc. of ICPADS, pp.775-782, Dec. 1998.

[5] A. R. Hurson, K. Kavi and J. T. Lim, "Cyclic Staggered Scheme : A Loop Allocation Policy for DOACROSS Loops", IEEE Trans. on Computers, Vol. 47, No. 2, pp.251-255, Feb. 1998.

[6] 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, No.2, pp.107-118, 1995.

[7] V. P. Krothapalli and P. Sadayappan, "Dynamic Scheduling of DOACROSS Loops for Multiprocessors", Proc. of Int'l Conf. on Database, Parallel Architectures and Their Applications, pp. 66-75, Apr. 1990.

[8] E. P. Markatos and T. J. LeBlanc, "Using Processor Affinity in Loop Scheduling on Shared-Memory Multiprocessor", IEEE Trans. on Parallel and Distributed Systems, Vol. 5, No. 4, pp. 379-400, 1994.

[9] D. L. Pean, C. C. Wu, H. T. Chua and C. Chen. " Effective Parallelization Techniques for Non-uniform Loops", Proc. of the 21st Australian Computer Science Conf., Perth, pp. 393-404, Feb. 1998.

[10] C. D. Polychronopoulos, "Guided Self Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers", IEEE Tran. Computers, Vol. C-36, No. 12, pp. 1425-1439, Dec. 1987.

[11] S. Punyamurtula and V. Chaudhary, "Minimum Dependence Distance Tiling of Nested Loops with Non-uniform Dependences", Proc. 6th IEEE Symposium on Parallel and Distributed Computer, pp. 74-81, May 1994.

[12] E. Rosti, E. Smirni, L. W. Dowdy, C. Serazzi and K. C. Sevcik, "Processor Saving Scheduling Policies for Multiprocessor Systems", IEEE Trans. on Computers, Vol. 47, No. 2, pp. 178-189, Feb. 1998.

[13] S.Y. Tseng, C.T. King, and C.Y. Tang, "Minimum Dependence Vector Set: A New Compiler Technique for Enhancing Loop Parallelism" Proc. 1992 Int'l Conf. on Parallel and Distributed Systems (ICPADS '92), pp.340-346, Dec. 1992.

[14] T. H. Tzen and L. M. Ni, "Trapezoid Self-Scheduling: A Practical Scheduling Scheme for Parallel Compilers", IEEE Trans. on Parallel and Distributed Systems, Vol. 4, No. 1, pp. 87-98, Jan. 1993.

[15] T. H. Tzen and L. M. Ni, "Dependence Uniformization : A Loop Parallelization Technique", IEEE Trans. On Parallel and Distributed Systems, Vol. 4, No. 5, pp. 547-558, May 1993.

[16] Y. Yan, C. Jin and X. Zhang, "Adaptively Scheduling Parallel Loops in Distributed Shared-Memory Systems", IEEE Trans. on Parallel and Distributed Systems, Vol. 8, No. 1, pp.70-81, Jan. 1997.

[17] *CONVEX Exemplar Programming Guide,* Convex Computer Corporation, Jun. 1994.

[18] *Exemplar Architecture,* Convex Computer Corporation, Jun. 1994.

[19] ftp://ftp.ucar.edu/ftp/dsl/lib/linpack/

[20] http://elib.zib.de/netlib/eispack/

[21] http://cm.bell-labs.com/netlib/itpack/index.html

[22] ftp://ftp.ucar.edu/ftp/dsl/lib/fishpak