JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING 16, 158–171 (1992)

A Solution of the Cache Ping-Pong Problem in Multiprocessor Systems

MI LU*

Department of Electrical Engineering, Texas A&M University, College Station, Texas 77843

AND

JESSE ZHIXI FANG

Hewlett–Packard Laboratories, Palo Alto, California 94303

The cache ping-pong problem arises often in parallel processing systems where each processor has its own local cache and employs a copy-back protocol for the cache coherence. To solve the problem of large amounts of data moving back and forth between the caches in different processors, techniques associated with parallel compilers need to be developed. Based on the concept in [Fang, J. Z., *Proc. International Conference on Parallel Processing*, Aug. 1990, pp. II-271–II-275] regarding the relations between array element accesses and enclosed loop indices in nested parallel loops, we present an algorithm in this paper to reduce the unnecessary data movement between the caches for parallel loops with multiple array subscript expressions. By analyzing the array subscript expressions in the nested parallel loop constructs, the compilers can use the algorithm to prepare information at compile time and let the processor execute the corresponding iterations of parallel loops in terms of the data in its cache. It benefits the parallel programs in which parallel loops are enclosed by a sequential loop and have multiple different subscript expressions for the same array, whose elements are repeatedly used in the different iterations of the outermost sequential loop. © 1992 Academic Press, Inc.

1. INTRODUCTION

An important task of a parallel compiler is to identify the parallel nature contained in a sequential program and to generate parallel code implemented on a parallel architecture. Recently, parallel compilers such as the Illinois PARAFRASE compiler [11, 13, 16] and the Rice Parallel Fortran Converter [2, 3, 5] incorporating a new theory and advanced technology have been developed.

It is assumed in most of the compiler systems that shared memory architectures are provided and a large memory block is directly addressable by all the processors in equal time intervals. However, the hierarchical memory system is widely applied in today's parallel systems. In many new products such as MIPS, IBM RIOS, SPARC, and Intel 860 which combine the semiconductor with the compiler back-end technique, a private cache is associated with each processor. To increase the memory bandwidth, more than one level of cache may be used, and the size of the cache may be very large. It should be noted that the time to access a private cache is much shorter than the time to access the global memory or the caches of other processors.

Poor cache hit ratios in such hierarchical memory multiprocessor systems are due to the following two reasons: the data requested by a processor are in the global memory and the data requested by a processor are in the caches of other processors. The first case can be handled by the traditional approach of cache utilization used in uniprocessor systems, in which it is desirable to keep frequently used data as much as possible in cache or local memory. One of the hardware solutions for obtaining high cache hit ratios is to provide large-size caches. However, when the parallel code is executing, the frequently used data may be shared by multiple processors which run the multiple threads of a parallel program at the same time. Therefore, increasing cache size cannot improve cache hit ratios in parallel program execution. A large cache size may even result in severe inefficiency when a parallel code requires that data move back and forth between the caches of different processors. This phenomena is called "cache thrashing" in shared memory multiprocessor systems [6]. The time needed for a processor to access the caches of other processors is close to that needed to access the global memory, since both of them must go through the data bus or interconnection network. Furthermore, it may be even longer than the global memory access time because of the increase in the traffic on the data bus or interconnection network and hence the degradation of the bus or network bandwidth.

* This research was supported by the Texas Advanced Technology Program under Grant 999903-165 and partially supported by the National Science Foundation under Grant MIP-8809328.

0743-7315/92 $5.00

Past research in this area has been focused on improving the data locality by restructuring the program, which may enhance the cache hit ratio on both uniprocessor and multiprocessor systems. Similar phenomena have been studied for virtual memory systems. Abu-Sufah *et al.* presented some source program transformation techniques to improve the paging behavior of the programs [1]. These transformations, referred to as "loop-block," include breaking iterative loops into smaller loops (strip-mining) and then recombining and reindexing these smaller loops (loop-fusing and loop-interchange). Since then, a number of loop-blocking algorithms have been developed for different computer architectures such as "loop-tilting" [17] and "loop-jam" [4]. These algorithms exploited and took advantage of the high degree of data reuse for the computation within a block. However, for most of the parallel code with complicated program structures, the benefit from blocking algorithms is very limited. For instance, when a parallel loop nest is enclosed by a serial loop and there is a loop-carried data dependence in the outermost serial loop, if the data are repeatedly used in the different iterations of the serial loop, the blocking technique cannot avoid the cache ping-pong for the data with dependence in the outer loop and independence in the enclosed parallel loop.

Recent research on the cache or local memory ping-pong problem by Fang [6] presented an overview of mathematical concepts for the problem. The concept defines the relation between the array element accesses and the enclosed loop indices in nested parallel constructs. The relation determined by an array subscript expression can be used to partition the iteration space into equivalence classes. All vectors in an equivalence class may access some common array elements at execution time. However, the method for calculating the next vector in an equivalence class from the previous vectors [6] is based on an assumption which allows only one subscript expression for an array variable in the nested loop. This assumption limits the results in [6] to be applied to real application programs. In this paper, we present an algorithm to solve the "cache ping-pong" problem for more general nested parallel constructs, in which an array variable may have more than one subscript expression in the same or different statements of the loop body. The algorithm is executed in an on-line fashion, finding for a linear integer system the next vector from a sequence of the stored vectors.

The rest of the paper is organized as follows. In Section 2, the cache or local memory ping-pong problem on our simple machine model is introduced. In addition, the preliminary concepts and the overall approach to solving the cache ping-pong problem on our simple program model are presented. We describe in Section 3 the main results in a simple case which has only a single array subscript expression. These results are then extended to the more complicated case which involves multiple subscript expressions or multiple array variables. Algorithms for eliminating the unnecessary data movement between the caches are presented. Section 4 shows the experimental results in a parallel compiler prototype. Parallel code is executed with or without the proposed compiler strategy and the results are compared. Finally, the paper concludes with Section 5.

2. BACKGROUND

2.1. Machine Model

In a shared memory multiprocessor system, a number of processors and global memory modules are connected by data-bus or interconnection network. Concurrent execution of multiple threads in parallel programming is ensured by a set of primitives, provided by the system, including fetch/increment or semaphore instructions.

In most of the processor design, the cache has the following characteristics: (1) local to a processor, (2) its size is large enough, (3) it uses copy-back and coherence strategy, (4) its line size is more than one word. In order to simplify the presentation, in this paper we assume that the cache memory is of one level and the line size is one word.

2.2. Cache Ping-Pong Problem

In a parallel program, a **thread** is referred to as the execution of a piece of code specified by parallel constructs [14]. It can be viewed as a unit of programmer-defined or parallel-compiler-specified work. As in a common parallel construct, a thread in a parallel loop is the execution of an iteration (or a chunk of iterations if we use strip-mining or other techniques) of the loop, and the threads are spawned when entering the parallel-loop merge at the end of the loop. The order in which the iterations of the loop are performed is arbitrary, and the processor on which the parallel loop is entered is not necessarily the same one on which the code following the parallel loop is executed.

In addition, parallel loops may be nested with sequential constructs when executed on multiprocessor systems, and some frequently used data may be repeatedly used and modified by different threads. If the threads accessing the same data are not assigned to the same processor, the set of data may be unnecessarily moved back and forth between the caches in the systems. This phenomena is called the **cache ping-pong** phenomena in shared memory multiprocessor systems.

The following example [6] shows the problem in a nested parallel construct:

```
      DIMENSION A(1000, 1000)
      I = 0
      WHILE_DO 10, 100
      I = I + 1
        ......
        PDO 20 J = 1, 100
          ......
          DO 30 K = 1, 100
            ......
S:          A(I + 2*J + 5*K, I + J + 3*K)
                = A(I + 2*J + 5*K, I + J + 3*K) + ...
30        CONTINUE
          ......
20      CONTINUE
        ......
10    CONTINUE
```

In this example the statement S does not have any data dependence in the DO J loop. If there is no other loop-carried dependence between the statements of the loop body, the J loop can be parallelized. There are a total of 10,000 threads $T_{I,J}$ in the execution of the parallel loop (both I and J range from 1 to 100). Each thread requests 100 elements of array A. Many of the array elements are repeatedly accessed in these threads.

For instance, thread $T_{1,1}$ requests data $A(8,5)$, $A(13,8)$, $A(18,11)$, $A(23,14)$, ..., $A(498,299)$, $A(503,302)$ for the innermost serial loop with index K from 1, ..., 100, respectively. Meanwhile, thread $T_{2,3}$ requests data $A(13,8)$, $A(18,11)$, $A(23,14)$, ..., $A(498,299)$, $A(503,302)$, $A(508,305)$ and thread $T_{3,5}$ requests data $A(18,11)$, $A(23,14)$, ..., $A(503,302)$, $A(508,305)$, $A(513,308)$. It can be observed that there exists a list of threads, $T_{1,1}$, $T_{2,3}$, $T_{3,5}$, $T_{4,7}$, ..., $T_{49,99}$, which reuse most of the array elements accessed in the previous thread. If the threads of the list are assigned to different processors, the data of array A are unnecessarily moved back and forth between caches in the system.

For instance, when $I = 1$, thread $T_{1,1}$ is assigned to processor 2. Note that loop I is serial. After the first iteration of loop I is completed, the processors need to be reassigned for the threads of the second iteration of loop I that contain a parallel loop J. If thread $T_{2,3}$ is assigned to processor 4, 99 array elements need to be moved from the cache in processor 2 to the cache in processor 4. This unnecessary data movement not only slows down the execution, but also degrades the bus or network bandwidth because it tremendously increases bus traffic. If thread $T_{3,5}$ is assigned to processor 2 in the third iteration of loop I, these data need to be moved back from the cache in processor 4 to the cache in processor 2.

In general, loops are the largest resource for parallelization in application programs. Parallel loops are the most common parallel program structures either defined by user directives or detected by automatic parallel compilers. The cache ping-pong phenomena shown in the above example are very common in the parallel code for scientific computations.

In order to gather evidence of the array access patterns in a wide range of applications, we studied a large number of benchmark programs. They include the Linpack benchmark, the Perfect Club benchmark, and application programs in mechanical CAE, computational chemistry, image and signal processing, and petroleum applications. These benchmarks were analyzed by PROFILE to recognize the most expensive routines and loop nests at execution time, which were chosen for future research. We found that almost all of the most time-consuming loop nests contain at least three level loops. About 60% of these loop nests contain at least one level parallel loop [7]. Ninety-five percent of the parallel loops can be moved from the innermost loop after using the loop-interchange technique. Only 6% of the parallelable nested loops have the parallel outermost loop. Ninety-four percent of parallel loops are enclosed by serial loops; that includes the loop nests in which a parallel loop appears in the outermost loop level in a subroutine, but the subroutine is called by a call-statement contained in a serial loop. Most of these loop nests are not perfect-nested. Fifty-three percent of the nested loops involve only one major array, which usually is two-dimensional or three-dimensional with a small size in the third dimension. Most of the loop bounds are passed by parameters, while there are a few simple triangular nested loops. Eighty-three percent of the array variables in the parallel program structures have more than one subscript expression either in the same parallel loop or in the different parallel loops enclosed by the same serial loop. This paper intends to solve the cache ping-pong problem in program models in which parallel loops are enclosed by a single serial loop and may contain other loops with multiple array subscript expressions in the loop body. This model can cover almost half the parallel loop nests in our study for a wide range of benchmarks in applications. If more than one serial loop encloses the parallel loops, only the inner serial loop needs to be concerned in the program model. If more than one level of loops, either parallel or serial, are enclosed by the parallel loops, only the outer loop needs to be concerned. The model allows multiple subscript expressions in the parallel loops, but only uses a single array variable. The multiple array variables with multiple subscript expressions are not included in the model discussed in this paper. More research is required to create a general rule to handle the cache ping-pong problem in more general program models.

2.3. Preliminaries

In this section, the preliminary concepts relevant to the iteration space and data dependence analysis are reviewed and the notations used in this paper are introduced.

Standard definitions are used to analyze the array accesses [2, 3, 5, 11, 13, 16]. Considering a nested parallel construct of k loops in the following form:

```
      DO I1 = L1, U1
        ......
        DO I2 = L2, U2
          ......
          DO Ik = Lk, Uk
            ......
S1:         A(h(I1, I2, ..., Ik) + a) = ...
            ......
S2:         ... = A(g(I1, I2, ..., Ik) + b) + ...
            ......
30        CONTINUE
          ......
20      CONTINUE
        ......
10    CONTINUE
```

where array A is of dimension d, and both **a** and **b** are offset vectors in Z^d. It is not necessary that this loop be perfect-nested. The loop bounds are not required to be constants. The functions **h** and **g** are linear:

$$\mathbf{h}, \mathbf{g} : Z^k \rightarrow Z^d.$$

The **iteration space** denoted as **C** is defined by the product $\prod_{j=1}^{k} N_j$, where N_j is the range of the jth index, $[L_j : U_j]$. The **domain space** denoted as **D** is defined by the product $\prod_{i=1}^{d} M_i$, where M_i is the size of array A in the ith dimension. Any array subscript expressions in the statements of a parallel nested loop can be more precisely defined by

$$\mathbf{h}, \mathbf{g} : \mathbf{C} \rightarrow \mathbf{D}$$

or we say that the array subscript expressions define the map

$$\mathbf{h}, \mathbf{g} : \prod_{j=1}^{k} N_j \rightarrow \prod_{i=1}^{d} M_i.$$

There exists a total order in the iteration space **C** defined by the point in time at which the element is executed. If we say a vector **t** is greater than a vector **s**, where

$$\mathbf{t} = (t_1, t_2, \ldots, t_k)$$

and

$$\mathbf{s} = (s_1, s_2, \ldots, s_k),$$

then there is a point m, which is in the range from 1 to k, such that $t_i = s_i$ for $i < m$ and $t_m > s_m$.

The standard data dependence definition [10, 12, 15] is given as follows. If two statements access the same memory location, we say that there is a data dependence between them.

In general, if a dependence is inside a loop, the dependence is called loop-independent dependence. If a dependence is across the iterations of a loop, it is called loop-carried dependence.

Loop-independent dependence does not cause the cache or local memory ping-pong problem if an iteration of a parallel loop must be performed by one thread. **Distance vector** [2, 3, 13] in dependence analysis shows the distance between two iterations that reference the same memory location. If the distance is t, a loop-carried flow dependence from S_1 to S_2 within a DO I loop has at least one variable which is computed in S_1 and referenced in S_2 after t iterations. The DO I loop should be executed sequentially or synchronized by some additional synchronizer to keep the execution order of the statements with the dependence.

Dependence analysis associated with a distance vector is a good approach to describing the data reference relationships between iterations in a loop. However, the cache or local memory ping-pong problem involved in multilevel loops in a nested parallel construct is more complicated. Furthermore, some loops do not have an explicit distance vector such as the example given in Section 2.2, but they can be parallelized by the Banerjee test. The dependence analysis approach is not enough to describe the nature of the cache ping-pong problem.

The overall nature of the cache or local memory ping-pong problem in the simple program model in Section 2.2 is described below. If the outermost loop is a parallel one in a nested parallel construct, there is no cache ping-pong problem, because the different parallel loop iterations never access the same memory location. If the outermost loop is serial, and encloses parallel loops, the dependences carried by the serial loop may cause the data to move back and forth between the threads that execute the parallel loops in the different iterations of the outer serial loop. Some array elements may be reused in the different iterations of the outermost serial loop due to the loop-carried dependences. Meanwhile, these array elements need to be moved between the caches of processors in each iteration of the outermost serial loop due to the parallel loops enclosed by the serial loop.

To help develop our algorithm to solve the cache ping-pong problem, we put two major constraints on our program model.

The first constraint is that this paper concentrates on parallel nested constructs in which only one-level loops are parallel or in which multilevel parallel loops may exist but only one-level loops are parallelized. This constraint is reasonable in order to match the simple machine model described in Section 2.1, which does not have the hardware processor cluster.

The second constraint is that we assume all dependences in the nested parallel construct use a unique iteration space. Usually, this assumption is not acceptable in application programs. If two parallel loops have different loop bounds, the iteration spaces must be different for the dependences carried by the loops. However, the mapping and transferring of iteration space will make the main results too complicated. It will also make the proofs tedious. The results presented in this paper can be extended to a general program model without the constraint by applying the space transformation technique onto the iteration spaces, which is similar to, but much more difficult than, the linear space transformation.

To simplify our discussion for the paper, we have the following assumptions in the program model,

1. All functions representing array subscript expressions are linear mapping: $\mathbf{C} \rightarrow \mathbf{D}$.
2. There are only one-level parallel loops in the nested parallel construct, which are enclosed by an outermost serial loop.
3. There may exist a one-level serial loop enclosed by the parallel loops.
4. All data dependences in the nested parallel construct use the same iteration space.

The program model on which we develop the main results may have three loop levels, the outermost serial loop, the middle parallel loop level, and the innermost loop level, either serial or parallel. They are not necessarily perfect-nested. There may exist more than one parallel loop in the middle level enclosed by the single outermost serial loop. Loop bounds can be any variables. These loops contain only one array variable with multiple subscript expressions. The program structures are not important as long as the data dependence uses a unique iteration space.

3. MAIN RESULTS

3.1. Mathematical Concepts

Most definitions and lemmas in this section are extended from the definitions and theorems given in [6] in the sense of considering multiple array subscript expressions rather than only a single expression.

DEFINITION 1. **Reduced Iteration Space** is a subspace of the iteration space for the parallel constructs described in Section 2.3 obtained by removing the dimension of the innermost loop index. For the example program in Section 2.2, the reduced iteration space is $[1 \cdots 100] \times [1 \cdots 100]$.

A linear function **h** defined in the program model, i.e., specified by an array subscript expression, is a map from the **reduced iteration space**, $N \times M$, to the set of subsets of the **domain space**

$$\mathbf{h} : N \times M \rightarrow 2^{D_1 \times D_2},$$

where the upper bounds of the outermost serial loop and the middle parallel loop are N and M, respectively. The dimensions of the array are $D_1 \times D_2$.

The following example illustrates the typical parallel program structures, which contain three level loops and r different array subscript expressions.

```
      DO i = 1, N
        ......
        PDO j = 1, M
          ......
          DO k = 1, L
            ......
            A(a1,1i + b1,1j + c1,1k + d1,1, a1,2i + b1,2j + c1,2k + d1,2) = ...
            ......
            ... = A(a2,1i + b2,1j + c2,1k + d2,1, a2,2i + b2,2j + c2,2k + d2,2) + ...
            ......
            A(ar,1i + br,1j + cr,1k + dr,1, ar,2i + br,2j + cr,2k + dr,2) = ...
            ......
30        CONTINUE
          ......
20      CONTINUE
        ......
10    CONTINUE
```

The linear function $\mathbf{h}_m$ is

$$f_m(\mathrm{i}, \mathrm{j}, \mathrm{k}) = a_{m,1}\mathrm{i} + b_{m,1}\mathrm{j} + c_{m,1}\mathrm{k} + d_{m,1}$$

and

$$g_m(\mathrm{i}, \mathrm{j}, \mathrm{k}) = a_{m,2}\mathrm{i} + b_{m,2}\mathrm{j} + c_{m,2}\mathrm{k} + d_{m,2},$$

where m is in the range from 1 to r, assuming there are r different array subscript expressions in the parallel construct.

To collect the sets of vectors in the reduced iteration space, which may access common memory locations within the corresponding threads, we define a set of elements of array A, which are accessed within thread T_{i_0,j_0} by linear function $\mathbf{h}_m(i, j, k)$ associated with the mth subscript expression as follows.

DEFINITION 2. For a given pair i_0 and j_0, the set of elements $A(f_m(i_0, j_0, k), g_m(i_0, j_0, k))$ of array A, which are accessed within thread T_{i_0,j_0} by a statement subscripted by the linear function $\mathbf{h}_m(i, j, k)$, is denoted by $A^{(m)}_{i_0,j_0}$, where $1 \le k \le L$ and $1 \le m \le r$.

$$A^{(m)}_{i_0,j_0} = \{A(f_m(i_0, j_0, k), g_m(i_0, j_0, k)) \mid \text{for a given } i_0 \text{ and } j_0, \text{ where } k \in [1, L]\}.$$

Since both f_m and g_m are linear in terms of i, j, and k, it is obvious to have the following lemma, which is useful in the rest of this section.

LEMMA 1. *In a program structure described above, if there exist two vectors in iteration space, (i, j, k) and (i', j', k'), such that*

$$f_m(i, j, k) = f_m(i', j', k')$$

and

$$g_m(i, j, k) = g_m(i', j', k'),$$

then for any constant n_0, we have a series of vectors in the space, $(i, j, k + n_0)$ and $(i', j', k' + n_0)$, satisfying the equations

$$f_m(i, j, k + n_0) = f_m(i', j', k' + n_0)$$

and

$$g_m(i, j, k + n_0) = g_m(i', j', k' + n_0),$$

where $1 \le k' + n_0 \le L$ and $1 \le k + n_0 \le L$.

It is clear from Lemma 1 that if

$$A^{(m)}_{i_1,j_1} \cap A^{(m)}_{i_2,j_2} \neq \varnothing,$$

then threads T_{i_1,j_1} and T_{i_2,j_2} should be assigned to the same processor, because they may access some common elements of array A subscripted by the linear function $\mathbf{h}_m$.

LEMMA 2. *In the program model described in Section 2.3, we have two vectors (i, j, k) and (i', j', k') holding the equations*

$$f_m(i, j, k) = f_m(i', j', k')$$

and

$$g_m(i, j, k) = g_m(i', j', k'),$$

if they satisfy the following conditions:

$$i' - i = \alpha_m = b_{m,1}\, c_{m,2} - b_{m,2}\, c_{m,1}$$
$$j' - j = \beta_m = a_{m,2}\, c_{m,1} - a_{m,1}\, c_{m,2}$$
$$k' - k = \gamma_m = a_{m,1}\, b_{m,2} - a_{m,2}\, b_{m,1}.$$

In the example program shown in Section 2.2, $m = 1$, $\alpha = 1$, $\beta = 2$, and $\gamma = -1$.

DEFINITION 3. For a given pair i_0 and j_0, the data set of elements of array A denoted by A_{i_0,j_0}, which may be accessed within thread T_{i_0,j_0} by the statements referring to the array variable A, is the union of the sets $A^{(m)}_{i_0,j_0}$, where m ranges from 1 to r.

It is clear from the above description that if

$$A_{i_1,j_1} \cap A_{i_2,j_2} \neq \varnothing,$$

then threads T_{i_1,j_1} and T_{i_2,j_2} should be assigned to the same processor, because they may access some common elements of array A at the execution of the parallel construct.

DEFINITION 4. In the program model described in Section 2.3, for a given pair i_0 and j_0, the common access set U_{i_0,j_0} denotes a set of vectors (i, j) in the reduced iteration space of size $N \times M$, satisfying the condition

$$U_{i_0,j_0} = \{(i, j) \mid A_{i_0,j_0} \cap A_{i,j} \neq \varnothing\}.$$

Using the program shown in Section 2.2 as an example, $U_{1,1} = \{(1,1), (2,3), (3,5), \ldots, (49,97), (50,99)\}$.

By Lemma 1, the definition can be described as

$$U_{i_0,j_0} = \{(i, j) \mid \exists\ k_0 \text{ and } k,\ m_0 \text{ and } m \text{ such that } f_{m_0}(i_0, j_0, k_0) = f_m(i, j, k) \text{ and } g_{m_0}(i_0, j_0, k_0) = g_m(i, j, k)\}.$$

U_{i_0,j_0} is the set of threads that may access some common elements of array A. Actually, Definition 4 defines a relation in the reduced iteration space. We call a relation

R in space **SP** satisfying the following condition an equivalence relation:

if a **R** b and b **R** c, then a **R** c, for a, b, c ∈ **SP**.

The equivalence relation can be used to partition the space into several equivalence classes. They are disjoint and cover the entire space. The relation defined in Definition 4 is an equivalence relation and U_{i_0,j_0} is an equivalence class in the reduced iteration space. The detailed proof can be found in [6] and [7]. The limitation of [6] is that it only concentrates on a simple case—array A has only a single subscript expression in the loop nest. This paper describes an approach to solve the cache ping-pong problem in a more complicated case—array A has more than one subscript expression in the program model. This approach provides an algorithm to calculate the next vector (i_2, j_2) from the current vector (i_1, j_1) in an equivalence class at execution time.

In order to develop an approach to compute the vector series in set U_{i_0,j_0}, we need to introduce some necessary notations. In the above example, for the linear function $\mathbf{h}_m$, where $1 \le m \le r$, let us denote:

$$\alpha_m = b_{m,1}\, c_{m,2} - b_{m,2}\, c_{m,1}$$

$$\beta_m = a_{m,1}\, c_{m,2} - a_{m,2}\, c_{m,1}$$

$$\gamma_m = a_{m,2}\, b_{m,1} - a_{m,1}\, b_{m,2}.$$

3.2. Partition of Iteration Space

As shown in Section 3.1, each linear function $\mathbf{h}_m$, where $1 \le m \le r$, gives a particular value of α_m, β_m, and γ_m.

Section 3.1 gave the definition of a set of vectors (i, j) in the reduced iteration space, U_{i_0,j_0}, in which each vector may access some elements of array A that are referenced in thread T_{i_0,j_0} for the given pair (i_0, j_0).

LEMMA 3. *The set of vectors* (i, j) *in Definition* 4, U_{i_0,j_0}, *is the same set shown below, if there is only one linear function in the parallel construct, indicated by the superscript* (1).

$$S^{(1)}_{i_0,j_0} = \{(i,j) \mid i = i_0 + p \times \alpha \text{ and } j = j_0 + p \times \beta \text{ for } p \in Z\}.$$

Now we extend the definition from one linear function to r linear functions in the parallel loop. As shown in the program model in Section 3.1, there are r pairs of subscript expressions for an array variable. Hence, we have a list of r triples: $(\alpha_1, \beta_1, \gamma_1)$, $(\alpha_2, \beta_2, \gamma_2)$, ..., $(\alpha_m, \beta_m, \gamma_m)$, ..., $(\alpha_r, \beta_r, \gamma_r)$.

If the relationship defined by the following definition can partition the reduced iteration space, then the corresponding threads may access some common elements of the array that are stored in the local cache by thread T_{i_0,j_0} for given (i_0, j_0).

DEFINITION 5. In reduced iteration space of loop i and loop j, if there are r different pairs of subscript expressions for an array variable, we define the set of the pairs of i and j so that the corresponding threads may access some common elements of the array subscripted by these linear functions, which are accessed in the cache or local memory by thread T_{i_0,j_0}.

$$S^{(r)}_{i_0,j_0} = \Big\{(i,j) \mid i = i_0 + \sum_{m=1}^{r} p_m \times \alpha_m \text{ and } j = j_0 + \sum_{m=1}^{r} p_m \times \beta_m \text{ for } p_1, \ldots, p_r \in Z\Big\}.$$

THEOREM 1. *The set of vectors in a reduced iteration space of loop i and loop j,* $S^{(r)}_{i_0,j_0}$, *defined in Definition* 5, *is a subset of the set of vectors* U_{i_0,j_0} *defined in Definition* 4.

The proof is straightforward following Lemma 2 and Lemma 3. All the threads corresponding to the vectors in $\mathbf{S}^{(r)}_{i_0,j_0}$ may access some common array elements at the execution of the parallel construct. Since Definition 5 is only a subset of Definition 4, the approach described in the paper is not the optimal solution for the cache ping-pong problem, but it can significantly reduce the cache ping-pong phenomena at execution time. The following theorem shows that the relation defined in the above definition can partition the reduced iteration space. Therefore, we can reduce the unnecessary data moving between processors and improve the system performance.

THEOREM 2. $S^{(r)}_{i_0,j_0}$ *is an equivalence class in the reduced iteration space of loop i and loop j.*

Proof. If we have

$$S^{(r)}_{i_0,j_0} \cap S^{(r)}_{i_1,j_1} \neq \varnothing,$$

then there exists (i, j) belonging to both $S^{(r)}_{i_0,j_0}$ and $S^{(r)}_{i_1,j_1}$.

Therefore, there exist $p_1, p_2, \ldots, p_r$ such that

$$i = i_0 + \sum_{m=1}^{r} p_m \times \alpha_m$$

$$j = j_0 + \sum_{m=1}^{r} p_m \times \beta_m.$$

Meanwhile, there exist $p'_1, p'_2, \ldots, p'_r$ such that

$$i = i_1 + \sum_{m=1}^{r} p'_m \times \alpha_m$$

$$j = j_1 + \sum_{m=1}^{r} p'_m \times \beta_m.$$

So, we have

$$i_1 = i - \sum_{m=1}^{r} p'_m \times \alpha_m = i_0 + \sum_{m=1}^{r} (p_m - p'_m) \times \alpha_m$$

$$j_1 = j - \sum_{m=1}^{r} p'_m \times \beta_m = j_0 + \sum_{m=1}^{r} (p_m - p'_m) \times \beta_m.$$

Therefore,

$$(i_1, j_1) \in S^{(r)}_{i_0,j_0}.$$

In the same way, we have

$$(i_0, j_0) \in S^{(r)}_{i_1,j_1}.$$

Finally, we have

$$S^{(r)}_{i_0,j_0} = S^{(r)}_{i_1,j_1}.$$

By Theorem 2 and Definition 5, it is obvious that there is less memory access from one processor to the caches or local memories of other processors, if all the threads in the same equivalence class are assigned to the same processor. We describe the fact in Theorem 3 and omit the proof.

THEOREM 3. *Every thread $T_{i,j} \in S^{(r)}_{i_0,j_0}$ may reuse some data in other threads belonging to the same equivalence class $S^{(r)}_{i_0,j_0}$, which can significantly reduce the access of the data referenced by the threads belonging to other equivalence classes.*

3.3. Computing Vectors in an Equivalence Class

To assign all the threads belonging to the same equivalence class to the same processor at execution time, an iterative algorithm is designed to calculate the next vector in $\mathbf{S}^{(r)}_{i_0,j_0}$ in terms of the current vector. The algorithm can be used to compute the current loop index of the middle parallel loop from the current outermost serial loop index and the previous serial and parallel loop indices.

The initial vector for each equivalence class, $\mathbf{S}^{(r)}_{i_0,j_0}$, or the initial value of $(p_1, \ldots, p_r)$ needs to be prepared at compilation time. The lower loop bound of the outermost serial loop is the initial value for i_0. All the other indices of the outermost loop, whose value is less than $min(\alpha_1, \ldots, \alpha_r)$, are initial too. For the given initial value of i_0, by Definition 5, it may be required to find all the possible $(p_1, \ldots, p_r)$ such that $\sum_{m=1}^{r} p_m \times \alpha_m = 0$ to calculate the initial value of j_0, but it is not necessary. In the program model shown in Section 2.2, Loop J is parallel. For any fixed i_0, there do not exist j_1 and j_2 such that

$$\mathbf{A}_{i_0,j_1} \cap \mathbf{A}_{i_0,j_2} \neq \varnothing.$$

By this assumption, a string $(p_1, \ldots, p_r)$ satisfying the above equation must satisfy another equation $\sum_{m=1}^{r} p_m \times \beta_m = 0$.

Therefore the algorithm to compute the initial vector for each equivalence class, $\mathbf{S}^{(r)}_{i_0,j_0}$, is straightforward. First let i_0 equal the lower bound of loop I; we have M initial vectors (i_0, j), where j ranges from 1 to M in our program model, for M equivalence classes. The list of $(p_1, \ldots, p_r)$ satisfying the equation $\sum_{m=1}^{r} p_m \times \alpha_m = 0$ is computed, which is useful in finding the initial value of j_0 as well as the next vector in the equivalence class at the execution time. Then increasing the value of i_0, we calculate the list of $(p_1, \ldots, p_r)$ in the same way as in the first step until the value of i_0 is equal to $min(\alpha_1, \ldots, \alpha_r)$. This computation needs to solve the integer linear equation $\sum_{m=1}^{r} p_m \times \beta_m = 0$. However, this is at the compilation time, so our approach does not affect the execution performance although the algorithm to solve the integer linear equation has high time complexity.

From Definition 5, we need to find the next vector in an equivalence class in the increasing order of component i. Assuming the initial vector is given, we are to compute component j step by step. Let an integer linear system L : $\{\mathbf{p}, \alpha, I\}$ be defined as follows:

$$\mathbf{p} = (p_1, p_2, \ldots, p_r)^T$$

$$\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_r)$$

$$I = \alpha \cdot \mathbf{p}$$

where $1 \leq I \leq N$ with N being a positive integer, and α, $\mathbf{p}$ are all r-dimensional positive integer vectors.

Consider the following problem: given initial $\mathbf{p}^0$, find $\mathbf{p}^1$ such that no $\mathbf{p}' \in Z^{+r}$ satisfies $\alpha \cdot \mathbf{p}^0 < \alpha \cdot \mathbf{p}' < \alpha \cdot \mathbf{p}^1$. Furthermore, given $\mathbf{p}^i$, find $\mathbf{p}^{i+1}$ such that there exists no $\mathbf{p}'$ satisfying $\alpha \cdot \mathbf{p}^i < \alpha \cdot \mathbf{p}' < \alpha \cdot \mathbf{p}^{i+1}$. We first give a graph representation of the system.

DEFINITION 6. A labeled digraph $G = (V, A, W)$ for the above integer system is defined as

- $V = \{I^1, I^2, \ldots, I^n\}$, where $I^1 < I^2 < \ldots I^n$
- $A = \{(I^l, I^{l'}) | 0 \leq l < l' \leq n$ and $\exists \alpha_m$ such that $I^{l'} = I^l + \alpha_m\}$
- $\{w(I^l, I^{l'}) = \alpha_m \mid I^{l'} = I^l + \alpha_m$ for $(I^l, I^{l'}) \in A\}$.

According to the definition, each arc $(I^l, I^{l'})$ in G is associated with a label, say α_m. If $\mathbf{u}(I^l, I^{l'})$ is an r-dimensional mth unit vector, then $\alpha_m = \alpha \cdot \mathbf{u}(I^l, I^{l'})$ for $(I^l, I^{l'}) \in A$.

The following lemma describes the path which can be found in the constructed graph in terms of unit vectors.

LEMMA 4. *Let I^s and I^t be the two nodes in the digraph defined by Definition 6, and $I^s = \alpha \cdot \mathbf{p}^s$ and $I^t = \alpha \cdot \mathbf{p}^t$. There is a path π from I^s to I^t such that $\pi = I^{l_1}, I^{l_2}, \ldots, I^{l_p}$, where $I^{l_1} = I^s$ and $I^{l_p} = I^t$, if and only if*

$$\mathbf{p}^t = \mathbf{p}^s + \sum_{k=1}^{p-1} \mathbf{u}(I^{l_k}, I^{l_{k+1}}).$$

Proof. From Definition 6, we have

$$I^t = I^{l_{p-1}} + w(I^{l_{p-1}}, I^t)$$
$$I^{l_{p-1}} = I^{l_{p-2}} + w(I^{l_{p-2}}, I^{l_{p-1}})$$
$$\cdots$$
$$I^{l_2} = I^s + w(I^{l_1}, I^{l_2})$$
$$I^t = I^s + \sum_{k=1}^{p-1} w(I^{l_k}, I^{l_{k+1}}).$$

Notice that $w(I^{l_k}, I^{l_{k+1}}) = \alpha \cdot \mathbf{u}(I^{l_k}, I^{l_{k+1}})$ and $I = \alpha \cdot \mathbf{p}$, we have

$$I^t = I^s + \sum_{k=1}^{p-1} \alpha \cdot \mathbf{u}(I^{l_k}, I^{l_{k+1}})$$

and

$$\mathbf{p}^t = \mathbf{p}^s + \sum_{k=1}^{p-1} \mathbf{u}(I^{l_k}, I^{l_{k+1}}). \quad \blacksquare$$

This exploited the relationship of $\mathbf{p}^s$ and $\mathbf{p}^t$ such that there exists a path from I^s to I^t. The definition below defines the concepts of accessible node and addressable digraph.

DEFINITION 7. Let G be a digraph described in Definition 6. A node $I^l \in V$ is *accessible* if and only if there exists a path from I^1 to I^l.

If all the nodes in the digraph are accessible, the digraph is referred to as an *addressable* digraph.

As an example, the addressable digraph for $I = 3p_1 + 7p_2$ is shown in Fig. 1.

COROLLARY 1. *In an addressable digraph, for any node $I^{l'}$ with $l' > 1$, there exists another node I^l and coefficient α_m such that the corresponding $\mathbf{p}^l$ and $\mathbf{p}^{l'}$ satisfy $\mathbf{p}^{l'} = \mathbf{p}^l + \mathbf{u}(\alpha_m)$.*

Proof. In fact, $\pi' = I^{l_1}, I^{l_2}, \ldots, I^{l_{p-1}}$ is a subpath of path $\pi = I^{l_1}, I^{l_2}, \ldots, I^{l_p}$ defined in Lemma 4. If we let $I^l = I^{l_{p-1}}$ and $I^{l'} = I^{l_p}$, then applying Lemma 4 to the subpath π', we have

$$\mathbf{p}^{l_{p-1}} = \mathbf{p}^{l_1} + \sum_{k=1}^{p-2} \mathbf{u}(I^{l_k}, I^{l_{k+1}}).$$

Comparing it with

$$\mathbf{p}^{l_p} = \mathbf{p}^{l_1} + \sum_{k=1}^{p-1} \mathbf{u}(I^{l_k}, I^{l_{k+1}}),$$

we have $\mathbf{p}^{l_p} = \mathbf{p}^{l_{p-1}} + \mathbf{u}(I^{l_{p-1}}, I^{l_p})$, that is, $\mathbf{p}^{l'} = \mathbf{p}^l + \mathbf{u}(\alpha_m)$. $\blacksquare$

This means, in an addressable digraph, $I^{l'}$ can be always found from I^l by tracing the arc $A(I^l, I^{l'})$. Correspondingly, given the vector $\mathbf{p}^{l'}$, we can always find $\mathbf{p}^l$ by applying the unit vector $\mathbf{u}(\alpha_m)$, and vice versa.

Let $I^1, I^2, \ldots, I^l, \ldots, I^q$ be a sorted sequence of positive integers for an integer linear system $I = \alpha \cdot \mathbf{p}$ satisfying the condition that there exists no $\mathbf{p} \in Z^{+r}$ such that $I^l < \alpha \cdot \mathbf{p} < I^{l+1}$ for $l = 1, 2, \ldots, q - 1$. We consider the following problem.

Given the initial vector $\mathbf{p}^0$, find $\mathbf{p}'$ such that there is no $\mathbf{p}'$ satisfying $\alpha \cdot \mathbf{p}^0 < \alpha \cdot \mathbf{p}' < \alpha \cdot \mathbf{p}^1$. Interactively, given $\mathbf{p}^l$, find $\mathbf{p}^{l+1}$ such that there is no $\mathbf{p}^{l'}$ satisfying $\alpha \cdot \mathbf{p}^l < \alpha \cdot \mathbf{p}^{l'} < \alpha \cdot \mathbf{p}^{l+1}$. This problem is to be solved in an on-line fashion.

According to Lemma 4 and the Corollary 1 described previously, all the I^{l+1} in the above sequence can be found by $I^k + \alpha_m$ for some k. Meanwhile, $\mathbf{p}^{l+1}$ can be found by $\mathbf{p}^k + \mathbf{u}(\alpha_m)$ correspondingly. Note that α_m may not exist for some $m = 1, 2, \ldots, r$.

Consider an addressable digraph representing the above system. We first define the relationship between the nodes I^l and I^{l+1}, and the relationship between the nodes I^{l+1} and I^k as described.

DEFINITION 8. Given an addressable digraph as defined in Definition 7 representing an integer linear system, node I^{l+1} is referred to as a *successive node of node* I^l if there is no $\mathbf{p} \in Z^{+r}$ such that $I^l < \alpha \cdot \mathbf{p} < L^{l+1}$, for $l = 1, 2, \ldots, q - 1$, and is denoted as *successive*(I^l). Node I^k and node $I^{k'}$ are *adjacent nodes* if they are connected by an arc pointed from I^k to $I^{k'}$. I^k is the *start node of* $I^{k'}$ *associated with* α_m denoted as *start-node*$_{\alpha_m}(I^k)$, and $I^{k'}$ is the *end node of* I^k *associated with* α_m, denoted as *end-node*$_{\alpha_m}(I^k)$.

If $I^l = \alpha \cdot \mathbf{p}^l$ and $I^l = \alpha \cdot \mathbf{p}^{l+1}$, then $\mathbf{p}^{l+1}$ is referred to as the *next vector of* $\mathbf{p}^l$.

Note that the out-degree for each node I^l, $l \le q - max\{\alpha_1, \ldots, \alpha_r\}$, is r where r is the dimension of the system. However, the in-degree of a node could be less than r in any case.

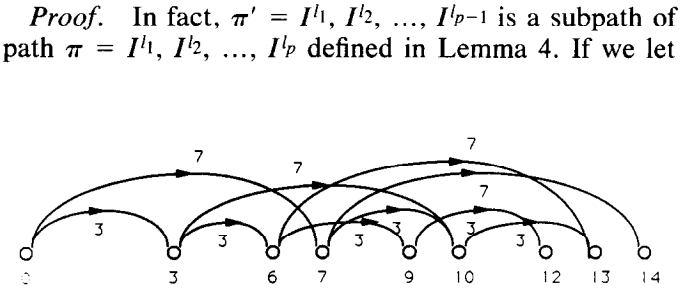

FIGURE 1

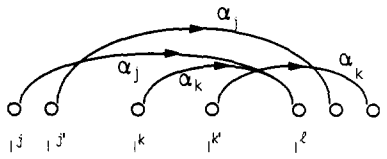

FIGURE 2

DEFINITION 9. If the in-degree of a node given in the digraph defined in Definition 8 is equal to r with r being the dimension of the vector $\mathbf{p}$, then the node is referred to as a *full* node.

LEMMA 5. *Let I^j and I^k be the start nodes of I^l associated with α_j and α_k, respectively. Assume the successive nodes of I^j and I^k are $I^{j'}$ and $I^{k'}$, respectively. In other words, $I^j = start\text{-}node_{\alpha_j}(I^l)$, $I^k = start\text{-}node_{\alpha_k}(I^l)$, $successive(I^j) = I^{j'}$, and $successive(I^k) = I^{k'}$ (see Fig. 2). If $I^{j'} - I^j < I^{k'} - I^k$, then $end\text{-}node_{\alpha_j} - I^l < end\text{-}node_{\alpha_k} - I^k$.*

Proof. Since

$$\begin{aligned} \text{end-node}_{\alpha_j}(I^{j'}) &= I^{j'} + \alpha_j \\ \text{end-node}_{\alpha_k}(I^{k'}) &= I^{k'} + \alpha_k, \\ \text{end-node}_{\alpha_j}(I^{j'}) - I^l &= I^{j'} + \alpha_j - I^l \\ &= I^{j'} - I^j \end{aligned}$$

and

$$\text{end-node}_{\alpha_k}(I^{k'}) - I^l = I^{k'} - I^k$$

for the same reason. Thus $I^{j'} - I^j < I^{k'} - I^k$ results in $\text{end-node}_{\alpha_j} - I^l < \text{end-node}_{\alpha_k} - I^k$. ■

THEOREM 4. *Suppose that node I^l is a full node and $\mathbf{p}^l$ corresponds to I^l. Consider all the start-nodes$_{\alpha_m}(I^l)$, for $m = 1, 2, \ldots, r$, and successive(start-nodes$_{\alpha_m}(I^l)$). If $m = m_0$ such that*

$$successive(start\text{-}nodes_{\alpha_{m_0}}(I^l)) - start\text{-}nodes_{\alpha_{m_0}}(I^l) \quad (1)$$

is the minimum, say Δ_{min}, then $successive(I^l) = I^l + \Delta_{min}$.

If $successive(start\text{-}nodes_{\alpha_{m_0}}(I^l)) = I^{j_0}$ and the corresponding input of the system is $\mathbf{p}^j$, then the next vector of I^l is $\mathbf{p}^{j_0} + \mathbf{u}(\alpha_{m_0})$.

This theorem can be proved by applying Lemma 5 inductively.

COROLLARY 2. *If I^l is not a full node, and start-nodes$_{\alpha_{m'}}(I^l)$ do not exist, then $I^l - \alpha_{m'}$ should be used to substitute it in (1), and successive(start-nodes$_{\alpha_m}(I^l)$) should be node I^j such that $I^j - (I^l - \alpha_{m'}) > 0$ and $I^j - (I^l - \alpha_{m'}) > 0$ is the minimum, for all $0 \le j \le q$.*

If we consider $I^l - \alpha_{m'}$ as a virtual node, similar proof as for Lemma 5 can be used for this corollary.

Given a vector $\mathbf{p}^l$ as described in Definition 7 and Definition 8, the next vector of $\mathbf{p}^l$, $\mathbf{p}^{l+1}$, can be found by the following algorithm.

ALGORITHM NEXT.

input: a sequence of vectors $\mathbf{p}^1, \mathbf{p}^2, \ldots, \mathbf{p}^l$ {to the integer linear system }

output: $\mathbf{p}^{l+1}$ { next vector of $\mathbf{p}^l$}

1. $I^l = \alpha \cdot \mathbf{p}^l$ { as is executed on line, I^{l-1}, I^{l-2}, ... should be already calculated and stored in a sorted order }
2. **for** $m = 1$ to r **do**
   begin
3. start-nodes$_{\alpha_m}(I^l) = I^l - \alpha_m$
4. **if** $P_m \neq 0$ **do**
   $I^j = \text{successive(start-nodes}_{\alpha_m}(I^l))$
5. **else**
   find node I^j such that $I^{j-1} < \text{start-nodes}_{\alpha_m}(I^l) < I^j$
6. $\Delta_m = I^j - \text{start-nodes}_{\alpha_m}(I^l)$
7. **if** $\Delta_m < \Delta_{m-1}$ **do**
   begin
8. $\Delta_{min} = \Delta_m$
9. $m_0 = m$
   end {of **if**}
10. $\mathbf{p}^{l+1} = \mathbf{p}^j + \mathbf{u}(\alpha_{m_0})$
11. **end** {of **for** }

THEOREM 5. *Algorithm NEXT finds the next vector $\mathbf{p}^{l+1}$ of vector $\mathbf{p}^l$ correctly.*

Proof. First, it is easy to see that I^i found in step 4 or step 5 is a node in the addressable digraph. $\mathbf{p}^j$ is a vector such that $\alpha \cdot \mathbf{p}^j = I^j$. According to the definition of the successive node and the statement in step 5, $I^j > \text{start-nodes}_{\alpha_m}(I^l)$. Since $\text{start-nodes}_{\alpha_m}(I^l) = I^l - \alpha_m$ and, as specified in step 2, $I^j > I^l - \alpha_m$ and $I^j + \alpha_m > I^l$ for any α_m. Here $I^j + \alpha_m$ is a node in the addressable digraph, and $\mathbf{p}^j + \mathbf{u}(\alpha_m)$ is the corresponding vector satisfying $\alpha \cdot (\mathbf{p}^j + \mathbf{u}(\alpha_m)) = I^j + \alpha_m$ according to Corollary 2.

Since $m_0 \in \{m | m = 1, 2, \ldots, r\}$ (step 9) and $\mathbf{p}^{l+1} = \mathbf{p}^j + \mathbf{u}(\alpha_{m_0})$ (step 10), we have proved that $I^j + \alpha_{m_0}$ is a node, $I^j + \alpha_{m_0} > I^l$, and $\mathbf{p}^{l+1} = \mathbf{p}^j + \mathbf{u}(\alpha_{m_0})$ satisfies $\alpha \cdot \mathbf{p}^{l+1} = I^j + \alpha_{m_0}$.

Now we prove that there is no vector $\mathbf{p}'$ existing such that $I^l < \alpha \cdot \mathbf{p}' < \alpha \cdot \mathbf{p}^{l+1}$.

(i) I' cannot be accessed by α_{m_0}, otherwise $I' - \alpha_{m_0}$ is a node. This is because $I^j < I' < I^{l+1}$, and $I^l + \alpha_{m_0} < I' - \alpha_{m_0} = I^j$, which means that there is a node existing between $I^l + \alpha_{m_0}$ and I^j, thus contradicting the generation of I^j stated in step 4 or step 5.

(ii) I' cannot be accessed by $\alpha_{m'}$, for $m = 1, 2, \ldots, r$ except for m_0. If I' can be accessed by $\alpha_{m'}$, for $m' \in \{m | m = 1, 2, \ldots, r \text{ and } m \neq m_0\}$, then $I' - \alpha_{m'}$ is a node. After executing the **for** loop in step 2 for $m = m'$, $(I' - \alpha_{m'}) - (I^l - \alpha_{m'})$ will be stored in Δ_m, which is equal to

$I' - I^l$. Now, if $I^l < I' < I^{l+1}$, then $I' - I^l < I^{l+1} - I^l$. $\Delta_m|_{m=m'} < \Delta_m|_{m=m_0}$, which contradicts the execution of step 7 to step 9. Thus I' can not possibly exist such that $I^l < I' < I^{l+1}$. ■

In fact, by executing Algorithm NEXT, we examined the possibility of accessing I^{l+1} from the found I's by different α_m's, for $m = 1, 2, \ldots, r$. For each specific α_m, we noticed that if I^l and I^{l+1} are closest, $I^l - \alpha_m$ and $I^{l+1} - \alpha_m$ are also closest. So, instead of finding the node closest to I^l, we search for the node closest to $I^l - \alpha_m$, for all the $m = 1, 2, \ldots, r$, utilizing the known information of the found vectors and found nodes.

The on-line feature of the algorithm limited the memory used and thus reduced the execution time.

Let the largest entry in α be α_{max}, for $m = 1, 2, \ldots, r$. To examine the range of start-nodes$_{\alpha_m}(I^l)$, note that the smallest start-nodes$_{\alpha_m}(I^l) = I^l - \alpha_{max}$. Therefore, we have $I^l - \alpha_{max} <$ start-nodes$_{\alpha_m}(I^l) < I^l$.

In the integer system, there are at most α_{max} existing in the interval between $I^l - \alpha_{max}$ and I^l. To compute a new output, say I^l, it is sufficient to store the previous results $I^j, \ldots, I^{l-2}, I^{l-1}$ and the corresponding vectors $\mathbf{p}^j, \ldots, \mathbf{p}^{l-2}$, $\mathbf{p}^{l-1}$ such that $j = I^l - \alpha_{max}$.

Step 2 shows that to find one next vector, the algorithm needs to execute r times for a vector with r dimensions. Step 6 involves a binary search on a sequence of α_{max} data items. $O(\log \alpha_{max})$ time is hence needed. Other steps take only constant time. So the time complexity of Algorithm NEXT is $O(r \log \alpha_{max})$.

The results of this paper including Algorithm NEXT have been implemented in our prototype compiler. The detailed implementation can be found in [7] and [8]. The prototype compiler employs a self-scheduling scheme [9]. The application programs just need to be transformed into a parallel form as generated by the regular parallel compiler. PDO represents a parallel loop. Neither special code transformation nor index variable reference modification is required by our approach. Our prototype compiler only produces necessary information, such as α, β, and the initial value of each equivalence class, and annotates the information with the corresponding PDO loop. For the example in Section 2.2, constants $\alpha = 1$, $\beta = 2$, and initial value of equivalence class $i_0 = 1, j_0 = 1, \ldots, 100$ are attached in the annotated list of PDO 20 J = 1, 100. At execution time, in the initial first iteration of the outermost DO I loop, when I = 1, the run-time library routine randomly assigns the initial value of the equivalence classes to the threads that are scheduled on the particular processor. For instance, a processor is assigned the value (1,1). It executes the thread $T_{1,1}$; i.e., the first iteration of PDO J loop in the first iteration of the outermost DO I loop. The processor saves $i_0 = 1, j_0 = 1$ and $\alpha = 1, \beta = 2$ into its local memory of the physical processor rather than to the logical thread. Then, when the second iteration of the outermost DO I loop is executed and $i = 2$, the thread assigned to the processor, uses the results in this paper to calculate the parallel loop index in terms of the constants saved in its local memory, then determines which iteration of the PDO J loop will be executed in the second iteration of the outermost DO I loop. The processor that executed $T_{1,1}$ uses Next Algorithm to obtain $j = 3$ because $i_0 = 1, j_0 = 1, i = 2, \alpha = 1$, and $\beta = 2$. It will execute $T_{2,3}$ and save the new value $i_0 = 2, j_0 = 3$ in its local memory. In the same way, the processor executes $T_{3,5}$ in the third iteration and $T_{4,7}$ in the fourth iteration of the outermost DO I loop. Gaussian Elimination, the code of which is shown in the next section, is another example, where $\alpha = 1$ and $\beta = 0$. If a processor is assigned to execute $T_{1,5}$, i.e., it executes the fifth iteration of PDO J loop in the first iteration of the outermost DO K loop, it stores the fifth column of array variable A in its cache memory. In the second iteration of the DO K loop, it calculates the corresponding PDO J loop index in terms of the constants in its local memory $i_0 = 1, j_0 = 5, \alpha = 1$, and $\beta = 0$. Then it executes $T_{2,5}$, the fifth iteration of the PDO J loop.

This approach doesn't need to examine the data stored in the local cache or local memory of a processor. It uses the relationship between the loop indices and the data accessed by the corresponding thread, which is shown in this paper, to calculate the right parallel loop index and then to execute the right iteration without unnecessary data movement among the processors. It doesn't require the transformation of the code generated by parallel compilers. The array reference, even the one in the innermost loop, doesn't need to be modified either.

Since the prototype compiler and the run-time library is so immature, we are only able to compile simple programs. We show the experimental results of a couple of simple benchmarks in the next section, although we examined a long list of benchmarks and applications in our study.

4. EXPERIMENTAL RESULTS

We have implemented the results in this paper in a parallel compiler prototype, which performs the dependence analysis and parallel transformations for FORTRAN programs. The prototype computes the α, β, γ in Theorem 1, and the initial value of the vectors at compiler time, then uses the information for dynamic scheduling [9] at the execution time. The parallel code generated by the prototype is run on a shared memory multiprocessor simulator based on a commercial system simulator. The system can simulate 4 to 16 processors, various sizes of cache memories, various cache coherence protocols, various cache line sizes and various memory bandwidths of data bus, crossbar, or interconnection network. The processor and the cache in the simulation system are based on MIPS R3000 and R4000. The

prototype simulates different scheduling strategies also. They include static scheduling, self-scheduling, and guided-self-scheduling. The experimental results show that these scheduling approaches are all slightly different on the execution performance. However, the technique presented in this paper to reduce cache ping-pong made a significant improvement on the execution performance, no matter which scheduling approach was used in the experiments. We employ the self-scheduling scheme [9] in our prototype and simulator, because it is the most simple scheme in the implementation.

In our experiment, we made the worst-case assumptions. The experimental results shown in this section were based on a machine model with a smaller cache size, 256 kB, compared to the 32 MB main memory size, and longer memory access latency, 30 machine cycles. The bus bandwidth and the cache missing penalty were supposed to have any hardware synchronization support. This may be why it only speeds up 1.6 times on a four-processor parallel processing system for $1K \times 1K$ Linpack. It should be noted that the purpose of these experiments was not to improve the hardware design. The purpose was to compare parallel performance on a poorly designed hardware system with more cache ping-pong to that of a system with less cache ping-pong using the results in this paper.

We can see from the table below that by adopting the proposed approach in our prototype compiler, a 60–70% performance improvement can be achieved on average without the enhancement of hardware support. Cache ping-pong phenomena can destroy loop-blocking, because data will unnecessarily move back and forth between cache memories. The results in this paper can help the blocking array in the multiprocessor, then significantly enhance the performance. The approach to integrating the two approaches is obviously beyond the scope of this paper.

In examining the experimental results, we compared the parallel code execution with or without the compiler strategy presented in this paper for the cache ping-pong problem and found significant enhancement by eliminating the unnecessary moving back and forth of data between processors. In the experiment, we assume that the memory and crossbar bandwidth are proportionally improved when the number of processors is increased.

Gaussian elimination. Gaussian elimination is a basic matrix operation that is used in many application programs. We use a $1K \times 1K$ array in the experimental benchmark. The following is the kernel code of the program:

```
DO k = 1, N
  .....
  PDO j = k + 1, N
    DO i = k + 1, N
      .....
      A(i, j) = A(i, j) - A(i, k)*A(k, j)/A(k, k)
      .....
    CONTINUE
  CONTINUE
CONTINUE
```

Number of processors	Original serial code	Parallel code with cache ping-pong	Speed-up vs uniprocessor	Parallel code with less cache ping-pong	Speed-up vs uniprocessor	Improvement by reducing cache ping-pong
4	285.0s	163.5s	1.7	102.2s	2.4	1.6
8	285.3s	109.8s	2.6	59.9s	4.7	1.7
16	286.2s	83.6s	3.4	39.8s	7.2	2.1

Linpack benchmark. Linpack benchmark consists of vectorized/parallelized code. We chose the loops containing SAXPY and SMXPY subroutine calls and inlined these routines in the loops, most of which have three level loops: serial, parallel, serial. As the table below shows, our approach achieved better performance than the original parallel code that doesn't have any consideration for the cache ping-pong problem. To make this measurement, we use the $1K \times 1K$ Linpack benchmark.

Number of processors	Original serial code	Parallel code with cache ping-pong	Speed-up vs uniprocessor	Parallel code with less cache ping-pong	Speed-up vs uniprocessor	Improvement by reducing cache ping-pong
4	514.7s	327.5s	1.6	234.0s	2.2	1.4
8	514.3s	226.9s	2.3	151.3s	3.4	1.5
16	515.2s	161.0s	3.2	94.7s	5.5	1.7

Number of processors	Original serial code	Parallel code with cache ping-pong	Speed-up vs uniprocessor	Parallel code with less cache ping-pong	Speed-up vs uniprocessor	Improvement by reducing cache ping-pong
4	1732.0s	1415.6s	1.3	832.7s	2.1	1.7
8	1734.1s	843.2s	2.1	481.8s	3.6	1.75
16	1735.7s	557.8s	3.1	309.9s	5.6	1.8

A complete application. We also performed the test on a complete application program benchmark, a computational chemistry application program. The kernel of the most frequently used routine has the following form:

```
      DO 100 K = 1, MAX_BOUND
      .....
      PARALLEL DO I = 1, M
        .....
        DO 200 J = 1, M
          X(I + K, J + I + K) = .....
            .....
            .... = X(I + K, J + I + K) * .....
200     CONTINUE
        .....
      END_PARALLEL_DO
      .....
100   CONTINUE
```

The dimension of array X is $3K \times 1K$. The table above shows the results when the modified approach is applied to eliminate the cache ping-pong.

5. CONCLUSIONS

A compiler technique to solve the "cache ping-pong" problem which is very interesting and very important in parallel processing has been developed. In particular, a very common parallel program structure has been studied in which parallel loops are enclosed by a serial loop and the array elements are reused in the parallel loops in different iterations of the enclosed serial loop. Efforts have been made to reduce the data movement between the caches for parallel programs. Methods of calculating the appropriate parallel loop indices for each processor in terms of the data stored in its cache have been used.

The results in this paper were developed under two constraints on machine models. We assumed that the cache memory is of one level and the cache line size is one word. The first constraint doesn't affect the results in our paper too much, because in general both the first level cache and the second level cache are local to one processor. If the system has the memory hierarchy and processor cluster hierarchy, i.e., the second level cache is shared by a cluster of processors, we have to study more complicated program structures and set up more general program models to solve the cache ping-pong problem. It is not trivial to explore some common patterns in complicated program structures. The second constraint can be taken off by adding more module calculations into our results. It will make the theorems and algorithms very complicated, however, and will tediously increase the length of the paper. Of course, some mathematical skill has to be introduced in the calculation, but the idea will remain the same as in the paper. The common access set of the threads will be modified, as the set of threads may access the same cache line. Although research in this area is not completed yet, our efforts are significant in the sense that they introduce the mathematical concepts, analyze the array subscript expressions, and provide the effective approach as a basic solution for further cache ping-pong elimination tasks in parallel processing systems.

REFERENCES

1. Abu-Sufah, W., Kuck, D. and Lawrie, D. On the performance enhancement of paging systems through program analysis and transformations. *IEEE Trans. Comput.* **C-30,** 5 (May 1981).
2. Allen, J. R., and Kennedy, K. PFC: A program to convert Fortran to parallel form. Rep. MASC-TR82-6, Rice University, Mar. 1982.
3. Allen, J. R. and Kennedy, K. Automatic loop interchange. *Proc. ACM SIGPLAN 84 Symposium on Compiler Construction.* June 1984, pp. 233–246.
4. Callahan, D., Carr, S., and Kennedy, K. Improving register allocation for subscripted variables. *Proc. ACM SIGPLAN'90 Conference on Programming Language Design and Implementation.* White Plains, NY, June 20–22, 1990.
5. Callahan, D., Cooper, K. D., Kennedy, K., and Torczon, L. Interprocedural constant propagation. *Proc. SIGPLAN 86 Symposium on Compiler Construction,* July 1986, pp. 59–67.
6. Fang, J. Z. Cache or local memory thrashing and the compiler strategy in parallel processing systems. *Proc International Conference on Parallel Processing.* Aug. 1990, pp. II-271–II-275.
7. Fang, J. Z., and Lu, M. An iteration partition approach for cache or local memory thrashing on parallel processing. Tech. Rep. TAMU-ECE 91-02, Department of Electrical Engineering, Texas A&M University.
8. Fang, J. Z., Lu, M. An iteration partition approach for cache or local memory thrashing on parallel processing, in preparation.
9. Fang, J. Z., Yew, C., Tang, T., and Zhu, C. Dynamic processor self-scheduling for general parallel nested loops. *IEEE. Trans. Comput.* **39,** 7 (July 1990), 919–929.
10. Kennedy, K. Automatic translation of Fortran programs to vector form. Tech. Rep. 476-029-4, Rice University, Houston, TX, Oct. 1980.

11. Kuck, D. J. *The Structure of Computers and Computations*, Vol. 1. Wiley, New York, 1978.
12. Kuck, D., Kuhn, R., Leasure, B., Padua, D., and Wolfe, M. Dependence graph and compiler optimizations. *Conf. Record of 8th ACM Symposium on Principles of Programming Languages*. Jan. 1981.
13. Kuck, D. J., Kuhn, R. H., Leasure, B., and Wolfe, M. The structure of an advanced vectorizer for pipeline processor. *Proc. IEEE Computer Society Fourth International Computer Software and Applications Conf.*, Oct. 1980.
14. Leasure, B., *et. al.* PCF Fortran: Language definition by the parallel computing forum. *Proc. International Conferences on Parallel Processing*, Aug. 1988.
15. Padua, D., and Kuck, D. High-speed multiprocessors and compilation techniques. *IEEE Trans. Comput.*, **C-29** (Sept. 1980), 763–776.
16. Wolfe, M. J. Techniques for improving the parallelism in programs. Rep. 78-929, Dept. of Computer Science, Univ. of Illinois at Urbana–Champaign, July 1978.
17. Wolfe, M. Iteration space tiling for memory hierarchies. *Proc. of the Third SIAM Conf. on Parallel Processing*, Los Angeles, CA, Dec. 1–4, 1987.

MI LU received the B.S.E.E. degree from the Shanghai Institute of Mechanical Engineering, China, in 1981. She received the M.S. and Ph.D. degrees in electrical and computer engineering from Rice University in 1984 and 1987, respectively. Mi Lu joined the Department of Electrical Engineering, Texas A&M University, as an assistant professor in 1987. Her research interests include parallel computing, distributed processing, parallel computer architectures and applications, computational geometry, and VLSI algorithms. Mi Lu is a member of the IEEE Computer Society. She has published over 40 technical papers. Her research was funded by NSF and by the Texas Advanced Technology Program.

JESSE FANG received the B.S. degree in mathematics from Fudan University, Shanghai, China, and the M.S. and Ph.D. degrees in computer science from the University of Nebraska, Lincoln, in 1982 and 1984, respectively. After graduation, he taught in the Computer Science Department at Wichita State University and was a visiting senior computer scientist in the Center for Supercomputing Research and Development at the University of Illinois, Urbana–Champaign. From 1986 to 1989, he was a consultant member of the technical staff at the Concurrent Computer Corporation. From 1989 to 1991, he worked on parallel/vectorized compiler and supercomputing system design for CONVEX Computer Corp. as a program manager in the Software Development Department. He is currently working as a project manager at the Hewlett–Packard Laboratories to develop compilers for the new generation of Hewlett–Packard RISC architecture. His research interests are instruction-level parallel compiler technologies on RISC architecture, superscalar and VLIW RISC architecture, parallel-processing systems, parallel/vectorized compilers, and scheduling and synchronization.

Received November 5, 1991; revised April 30, 1992; accepted May 20, 1992