An Approach to Solve the Cache Thrashing Problem*

Zhixi Fang

CONVEX Computer Corp.
Richardson, TX 75083-3851

Mi Lu AND Hua Lin

TEXAS A & M University
College Station, TX 77843

Abstract
Cache or local memory thrashing problem arises very often in parallel processing architectures where each processor has its local cache or memory and a write-back protocol is employed for cache coherence. To solve the problem that large amount of data moving back and forth between the caches or local memories in different processors, techniques associated with parallel compiler need to be developed. Based on the relations between array element accesses and enclosed loop indices in a nested parallel construct, we present in this paper some approach to reduce the data movement between the caches or local memories for parallel programs. By analyzing the array subscript expressions, the compilers let the processor execute the corresponding iterations of parallel loops in terms of the data in its cache or local memory at execution time. It benefits, particularly, the parallel programs in which a parallel loop is enclosed by a sequential loop and array elements are repeatedly used in different iterations of the parallel loop.

1. Introduction
An major assumption in most of the parallel compilers is that shared memory architectures are provided, and a large memory block is directly addressable by all the processors in equal time intervals. However, hierarchical memory system is widely applied in today's parallel systems. Fast processor cycles and slow main memory access time are balanced by the use of fast memory components, for instance, cache and local memory. Most of the multiprocessor systems have a private cache associated with each processor. It should be noted that the access time to a private cache or local memory is much faster than to the global memory or to the cache or memories of other processors. Poor cache hit ratio 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 time needed for a processor to access the caches of other processors is the same as the access time needed to the global memory, since both of them have to go through the data bus or interconnection network.

Past research in this area has been focused on improving the data locality by the program restructuring, which may enhance the cache hit ratio on both uniprocessor and multiprocessor systems. Similar phenomena has been studied for virtual memory systems. W. Abu-Sufah, D. Kuck, and D. Lawrie 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-tiling" [2] and "loop-jam" [3]. 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 constructs, the benefit of the blocking algorithms is very limited.

In this paper we present a compiler technique to solve the "cache or local memory thrashing" problem. The technique benefits the parallel code with complicated parallel constructs 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. The technique calculates the appropriate parallel loop indices for each processor in terms of the data stored in its cache or local memory.

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. The concurrent execution of multiple threads in parallel programming are ensured by a set of primitives, provided by the system, including fetch/increment or semaphore instructions.

For simplicity, some considerations in hardware design to enhance the memory hierarchy performance are ignored in our machine model. It is assumed that the cache memory local to a processor is of one level, the size of it is large enough, the line size of it is one word, and the coherence strategy is write-back. The results can be extended to more complicated machine models considering more levels of local memories can be included in the memory hierarchy.

2.2 Cache or local memory thrashing problem
In a parallel program, a thread is referred to as the
execution of a piece of code specified by parallel constructs [4]. 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 spawned on entering the parallel-loop merge at the end of the loop.

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 cache or local memory thrashing in shared memory multiprocessor systems[5].

In general, loops are the largest resource for parallelization and vectorization in application programs. Parallel loops are the most common parallel program constructs either defined by user directives or detected by automatic parallel/vectorized compilers. These parallel loops usually are encased by other serial loops in scientific computations. A number of application programs such as Gaussian Elimination, Linpack, Linear Equations, Finite Elements and Finite Different Equations contain many nested parallel constructs in which serial loops encase a parallel loop with linear array subscript expressions. Some array subscript expressions in the applications only contain one or two enclosed loops. This paper intends to solve the cache or local memory thrashing problem in a general case with linear array subscript expressions.

2.3 Preliminaries

Standard definitions are used in this paper to analyze the array accesses [3, 6, 7]. Considering a nested parallel construct of $k$ loops of the form

\[
\text{DO } i_1 = L_1, U_1 \\
\text{\hspace{1cm} DO } i_2 = L_2, U_2 \\
\text{\hspace{2cm} DO } i_3 = L_3, U_3 \\
\text{\hspace{3cm} S_1: } A(h(i_1,i_2,...,i_k) + a) = ... \\
\text{\hspace{3cm} S_2: } = A(g(i_1,i_2,...,i_k) + b) + ... \\
\text{CONTINUE}
\]

20 CONTINUE

where the array $A$ is of dimension $d$, both $a$ and $b$ are offset vectors in $Z^d$. The function $h$ and $g$ are linear:

\[
h, g : Z^k \rightarrow Z^d.
\]

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 $j$-th index, $[L_{j} : U_{j}]$. The domain space denoted as $D$ is defined by the product $\prod_{j=1}^{k} M_{j}$, where $M_{j}$ is the size of array $A$ in the $j$-th dimension. Any array subscript expressions in the statements of a parallel nested loop can be more precisely defined by:

\[
h, g : C \rightarrow D.
\]

Loop-independence dependence does not cause the cache or local memory thrashing problem if an iteration of a parallel loop must be performed by one thread. The overall nature of the cache or local memory thrashing problem is described below. If the outermost loop is parallel in a nested parallel construct, there is no cache thrashing problem, because the different parallel loop iterations never access the same memory locations. If the outermost loop is serial, and encloses a parallel loop, the dependences carried by the serial loop may cause the data moving back and forth between the threads that execute the iterations of the parallel loop in different iterations of the outer serial loop. Some array elements may be reordered in the different iterations of the outermost serial loop due to the loop-carried dependences in the loop. Meanwhile, these array elements need to be moved in the caches between processors in each iteration of the outermost serial loop due to the parallel loop enclosed by the serial loop.

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

1. All functions representing array subscript expressions are linear mapping: $C \rightarrow D$.
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 one-level serial loop enclosed by the parallel loops.
4. All data dependences in the nested parallel construct use the same iteration space.

3. Main Results

3.1 Mathematical concepts

Definition 1. Reduced Iteration Space is a subspace in the iteration space for the parallel constructs described in Section 2.2 by removing the dimension of the innermost serial loop index.

A linear function $h$ defined in the program model is a map from the reduced iteration space, $N \times M$, to the set of subsets of the domain space.

\[
h : N \times M \rightarrow 2^{D_1 \times D_2},
\]

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

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

\[
\text{DO } i = 1, N \\
\text{\hspace{1cm} DO } j = 1, M \\
\text{\hspace{2cm} DO } k = 1, L \\
\text{\hspace{3cm} A(s_1,i + b_1,i + c_1,i + d_1,i, e_1,i + b_2,i + c_2,i + d_2,i) = ... } \\
\text{\hspace{3cm} A(s_2,i + b_1,i + c_2,i + d_2,i, e_2,i + b_2,i + c_2,i + d_2,i) = ... } \\
\text{\hspace{3cm} A(s_3,i + b_3,i + c_3,i + d_3,i, e_3,i + b_3,i + c_3,i + d_3,i) = ... } \\
\text{\hspace{3cm} CONTINUE} \\
20 \text{ CONTINUE} \\
10 \text{ CONTINUE}
\]

The linear function $h_{m}$ is

\[
f_{m}(i,j,k) = a_{m,1,i} + b_{m,1,i} + c_{m,1,k} + d_{m,1}
\]

and
\[ g_m(i,j,k) = a_m z_i + b_m z_j + c_m z_k + d_m z_k, \]
where \( m \) is 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 \( T_{i,j} \) by linear function \( h_m(i,j,k) \) defined by the \( m \)th subscript expression as follows.

**Definition 2.** For a given pair \( i_0 \) and \( j_0 \), the set of elements \( A(f_m(i_0, j_0, k)) \) of array \( A \), which are accessed within thread \( T_{i_0,j_0} \) by statement subscripted by the linear function \( h_m(i,j,k) \), is denoted by \( A^{(m)}_{i_0,j_0} \), where \( 1 \leq k \leq L \) and \( 1 \leq m \), and

\[ A^{(m)}_{i_0,j_0} = \{ A(f_m(i_0, j_0, k)) \mid k \in [1, L] \}. \]

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 construct 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 following 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 \( 1 \leq k' + n_0 \leq L \) and \( 1 \leq k + n_0 \leq L \).

It is clear from Lemma 1 that if

\[ A^{(m)}_{i_0,j_1} \cap A^{(m)}_{i_0,j_2} \neq \emptyset, \]

then threads \( T_{i_0,j_1} \) and \( T_{i_0,j_2} \) should be assigned to the same processor, because they may access some common elements of array \( A \) subscribed by the linear function \( h_m \).

**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

\[ g_m(i, j, k) = g_m(i', j', k'), \]
then the vectors \( (i, j, k) \) and \( (i', j', k') \) if they satisfy the following conditions:

\[ i' - i = c_m = b_m z_i + c_m z_j - b_m z_i + c_m z_j, \]

\[ j' - j = b_m = a_m z_i + c_m z_j - a_m z_i + c_m z_j, \]

\[ k' - k = -c_m = -a_m z_i + b_m z_j - b_m z_i + b_m z_j, \]

**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 statements referencing to the array variable \( A \), is the union of the sets \( A^{(m)}_{i_0,j_0} \), where \( m \) is from 1 to \( r \).

It is clear from the above description, \( A_{i_0,j_1} \cap A_{i_0,j_2} \neq \emptyset \), then threads \( T_{i_0,j_1} \) and \( T_{i_0,j_2} \) 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 a program model described in Section 2.3, for a given pair \( i_0 \) and \( j_0 \), \( U_{i_0,j_0} \) denotes a set of vectors \( (i, j) \) in the reduced iteration space of size \( N \times M \), which satisfies the following condition:

\[ U_{i_0,j_0} = \{ (i, j) \mid A_{i_0,j_0} \cap A_{i_0,j_0} \neq \emptyset \}. \]

By Lemma 1, the definition can be described as:

\[ S^{(i_0,j_0)} = \{ (i, j) \mid \exists k_0, k_1, k_2, k_3 \text{ and } k \text{ such that} \]

\[ f_m(i_0, j_0, k_0) = f_m(i, j, k) \text{ and} \]

\[ g_m(i_0, j_0, k_0) = g_m(i, j, k) \].

What we need to do is to prove that the set \( S^{(i_0,j_0)} \) is an equivalence class, then to find a way to calculate \( j_2 \) from the current \( i_0 \) and the previous vector \( (i_1, j_1) \) in the equivalence class.

In order to develop an approach to compute the vector series in set \( S^{(i_0,j_0)} \), we need to introduce some necessary notations. In the above example, for the linear function \( h_m \), where \( 1 \leq m \leq r \), let us denote:

\[ \alpha_m = b_m z_i + c_m z_j - b_m z_i + c_m z_j, \]

\[ \beta_m = a_m z_i + c_m z_j - a_m z_i + c_m z_j, \]

\[ \gamma_m = -a_m z_i + b_m z_j - b_m z_i + b_m z_j, \]

**3.2 Partition of iteration space**

As shown in Section 2.3, each linear function \( h_m \), where \( 1 \leq m \leq r \), gives a particular value to \( \alpha_m, \beta_m, \) and \( \gamma_m \).

Section 2.3 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).

\[ S^{(i_0,j_0)} = \{ (i, j) \mid i = i_0 + p x \alpha \text{ and} \]

\[ j = j_0 + p x \beta, \text{ for } p \in Z \}. \]

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 2.3, there are \( r \) pairs of subscript expressions for an array variable or for multiple array variables. We have a list of \( r \) triples: \((a_1, \beta_1, \gamma_1), (a_2, \beta_2, \gamma_2), \ldots, (a_m, \beta_m, \gamma_m), \ldots, (a_r, \beta_r, \gamma_r)\). The set of vectors in the reduced iteration space defined as follows can be proved to be equal to the Definition 4 in the previous section. 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 or arrays that are stored in the local cache by thread \( T_{i_0,j_0} \) for given \((i_0, j_0)\).

**Definition 5.** In the reduced iteration space of loop \( i \) and loop \( j \), if there are \( r \) different pairs of subscript expressions for an array variable or multiple array variables, we define the set of the pairs of \( i \) and \( j \) so that the corresponding threads may access some common elements of the array or arrays subscripted by these linear functions, which are accessed in the cache or local memory by thread \( T_{i_0,j_0} \).
$S_{i, j, s}^r = \{(i, j) \mid i = i_0 + \sum_{m=1}^r p_m \times \alpha_m$ and 

$j = j_0 + \sum_{m=1}^r p_m \times \beta_m$ for $p_1, \ldots, p_r \in Z$, \}.

Theorem 1. The set of vectors in a reduced iteration space of loop $i$ and loop $j$, $S_{i, j, s}^r$, defined in Definition 5, is a subset of the set of vectors, $U_{i, j, s}^r$, defined in Definition 4.

The proof is straightforward following Lemma 2 and Lemma 3. All threads corresponding to the vectors in $S_{i, j, s}^r$ may access some common array elements in the execution of the parallel construct. Since Definition 5 is only a subset of Definition 4, the approach described in the paper is not an optimal solution for the cache thrashing problem, but it can significantly reduce the cache thrashing at the 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. $S_{i, j, s}^r$ is an equivalence class in the reduced iteration space of loop $i$ and loop $j$.

Theorem 3. Every thread $T_{i, j} \in S_{i, j, s}^r$ may reuse some data in the other threads belonging to the same equivalence class $S_{i, j, s}^r$, which can significantly reduce the access of the data referenced by the threads belonging to the other equivalence classes.

3.3 Computing vectors

To assign all threads belonging to the same equivalence class to the same processor at the execution time, an iterative algorithm is designed to calculate the next vector in $S_{i, j, s}^r$ 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 $S_{i, j, s}^r$, or the initial value of $(p_1, ..., p_r)$ needs to be prepared at compilation time. The lower 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 $\min(\alpha_1, ..., \alpha_r)$, are initial too. For the given initial value of $i_0$, by Definition 5, it seems to be required to find all possible $(p_1, ..., p_r)$ such that

$\sum_{m=1}^r p_m \times \alpha_m = 0$ to calculate the initial value of $j_0$. However, it is not necessary. In the program model in section 2.2, Loop $j$ is parallel. For any fixed $i_0$, there do not exist $j_1$ and $j_2$ such that

$A_{i, j_1} \cap A_{i, j_2} \neq \Phi$.

By this assumption, a string $(p_1, ..., p_r)$ 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 $S_{i, j, s}^r$ is straightforward. First let $i_0$ equal to the lower bound of Loop $i$, we have $M$ initial vectors $(i_0, j)$, where $j$ is from 1 to $M$ in our program model, for $M$ equivalence classes. The list of $(p_1, ..., p_r)$ satisfying the equation

$\sum_{m=1}^r p_m \times \alpha_m = 0$ is computed, which is useful to find 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 $i_0$, calculate the list of $(p_1, ..., p_r)$ in the same way as in the first step, until the value of $i_0$ is equal to $\min(\alpha_1, ..., \alpha_r)$. This computation needs to solve the integer linear equation

$\sum_{m=1}^r p_m \times \beta_m = 0$. It is done at the compilation time without affecting the execution performance, although the algorithm to solve the linear integer 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 $L = \{p, \alpha, \lambda\}$ be defined as follows:

$p = (p_1, p_2, \cdots, p_r)^T$

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

$\lambda = \alpha \cdot p$

where $1 \leq \lambda \leq N$ with $N$ being positive integer, and $\alpha$ and $p$ are all $r$-dimensional positive integer vectors.

Consider the following problem: given initial $p^0$, find $p^1$ such that no $p^0 \in Z^+$ satisfying $\alpha \cdot p^0 < \alpha \cdot p^1$. Furthermore, given $p^0$, find $p^{i+1}$ such that there exists no $p^i$ satisfying $\alpha \cdot p^0 < \alpha \cdot p^i < \alpha \cdot p^{i+1}$.

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 follows:

$V = \{l^{i_1}, l^{i_2}, \cdots, l^m\}$

where $l^{i_1} < l^{i_2} < \cdots < l^m$.

$A = \{(l^{i_1}, l^{i_2}) \mid 0 \leq i_1 < i_2 \leq n \text{ and } \exists \alpha_m \text{ such that } l^{i_2} = l^{i_1} + \alpha_m\}$

$W = \{u(l^{i_1}, l^{i_2}) = \alpha_m \mid l^{i_2} = l^{i_1} + \alpha_m \text{ for } (l^{i_1}, l^{i_2}) \in A\}$.

According to the definition, each arc $(l^{i_1}, l^{i_2})$ in $G$ is associated with a label, say $\alpha_m$. If $u(l^{i_1}, l^{i_2})$ is an $r$-dimensional $m$th unit vector, then $\alpha_m = \alpha \cdot u(l^{i_1}, l^{i_2})$.

For $(l^{i_1}, l^{i_2}) \in A$.

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

Lemma 4. Let $l^{i_1}$ and $l^{i_2}$ be the two nodes in the digraph defined by Definition 6, and

$l^i = \alpha \cdot p^i$ and

$l^{i_1} = \alpha \cdot p^{i_1}$. There is a path $\pi$ from $l^i$ to $l^{i_1}$ such that $\pi = l^{i_1}, l^{i_2}, \cdots, l^r$, where $l^{i_1} = l^i$ and $l^{i_2} = l^i$, if and only if

$p^{i_1} = p^i + \sum_{k=1}^{p-1} u(l^{i_k}, l^{i_{k+1}})$.

Proof: From Definition 6, we have

$l^{i_1} = l^i + \sum_{k=1}^{p-1} u(l^{i_k}, l^{i_{k+1}})$.

Notice that $u(l^{i_k}, l^{i_{k+1}}) = \alpha \cdot u(l^{i_k}, l^{i_{k+1}})$ and $l^i = \alpha \cdot p^i$, we have

$l^{i_1} = l^i + \sum_{k=1}^{p-1} \alpha \cdot u(l^{i_k}, l^{i_{k+1}})$

and

$p^{i_1} = p^i + \sum_{k=1}^{p-1} u(l^{i_k}, l^{i_{k+1}})$.
This exploited the relationship of \( p' \) and \( p' \) such that there exists a path from \( I^* \) to \( I^* \). 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 \( I^* \in V' \) is accessible if and only if there exists a path from \( I^* \) to \( I^* \).

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

As an example, the addressable digraph for \( I = 2p + 1 \) is shown in Figure 9.

**Corollary 1.** In an addressable digraph, for any node \( I^* \) with \( I > 1 \), there exists another node \( I^* \) and coefficient \( \alpha_m \) such that the corresponding \( p^* \) and \( p^* \) satisfy \( p^* = p^* + u(\alpha_m) \).

**Proof:** In fact, \( \pi = I^*, I^*, \ldots, I^* \) is a subpath of \( \pi = I^*, I^*, \ldots, I^* \) defined in Lemma 4. If we let \( I^* = I^* + 1 \) and \( I^* = I^* + 1 \), then apply Lemma 4 to the subpath \( \pi \), we have

\[
p^* = p^* + u(\alpha_m) + u(\alpha_m).
\]

Comparing it with \( p = p + \sum_{k=1}^{m-1} u(\alpha_m), \) we have \( p^* = p^* + \sum_{k=1}^{m-1} u(\alpha_m) \), that is, \( p^* = p^* + u(\alpha_m) \).

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

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

Given the initial vector \( p^0 \), find \( p^* \) such that there is no \( p \) satisfying \( \alpha \cdot p < \alpha \cdot p^* < \alpha \cdot p^* \). Intermittently, give \( p^* \), find \( p^*+1 \) such that there is no \( p \) satisfying \( \alpha \cdot p^* < \alpha \cdot p^* < \alpha \cdot p^*+1 \). This problem is to be solved in an on-line fashion. According to Lemma 4 and Corollary 1 described previously, all the \( I^*+1 \) in the same sequence can be found by \( I^* + \alpha_m \) for some \( k \). Meanwhile, \( p^*+1 \) can be found by \( p^* + u(\alpha_m) \) correspondingly. Note that \( \alpha_m \) may not exist for some \( m = 1, 2, \ldots, r \).

Consider an addressable digraph representing the above system. We first define the relationship between nodes \( I^* \) and \( I^*+1 \), and then the relationship between the nodes \( I^*+1 \) and \( I^* \).

**Definition 8.** Given an addressable digraph as defined in Definition 7 representing an integer linear system. Node \( I^*+1 \) is referred to as successive node of node \( I^* \) if there is no \( p \in Z^+ \) such that \( I^* < \alpha \cdot p < I^*+1 \), for \( l = 1, 2, \ldots, q - 1 \), and denoted as successive \( I^* \). Node \( I^*+1 \) and \( I^*+1 \) are adjacent node if they are connected by an arc pointed from \( I^* \) to \( I^* \). \( I^*+1 \) is the start node of \( I^*+1 \) associated with \( \alpha_m \) denoted as start-node-\( \alpha_m \)(\( I^* \)), and \( I^* \) is the end node of \( I^* \) associated with \( \alpha_m \) denoted as end-node-\( \alpha_m \)(\( I^* \)). If \( I^* = \alpha \cdot p^* \) and \( I^* = \alpha \cdot p^*+1 \) then \( p^*+1 \) is referred to as the next vector of \( p^* \).

Note that the out-degree for each node \( I^* \), \( l \leq q - \max \{ \alpha_1, \ldots, \alpha_r \} \), 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.

**Definition 9.** If the in-degree of a node given in the digraph is defined in Definition 6 is equal to \( r \) with being the dimension of the vector \( p \), then the node is referred to as a full node.

**Lemma 5.** Let \( \bar{I} \) and \( I^* \) be the start nodes of \( I^* \) associated with \( \alpha_j \) and \( \alpha_j \) respectively. Assume the successive node of \( I^* \) and \( I^* \) are \( I^* \) and \( I^* \) respectively. In other words, \( \bar{I} = \text{start-node}_{\alpha_j}(I^*) \), \( I^* = \text{start-node}_{\alpha_j}(I^*) \), successive\( I^* = I^* \) and successive\( I^* = I^* \) (see Figure 2). If \( I^* < I^* < I^* \), then end-node-\( \alpha_j \) \( I^* \) < end-node-\( \alpha_j \) \( I^* \).

**Proof:** Since end-node-\( \alpha_j \)(\( I^* \)) = \( I^* + \alpha_j \) and end-node-\( \alpha_j \)(\( I^* \)) = \( I^* + \alpha_j \) and end-node-\( \alpha_j \)(\( I^* \)) = \( I^* + \alpha_j \) and end-node-\( \alpha_j \)(\( I^* \)) = \( I^* + \alpha_j \), we have end-node-\( \alpha_j \)(\( I^* \)) = \( I^* + \alpha_j \) and end-node-\( \alpha_j \)(\( I^* \)) = \( I^* + \alpha_j \). Thus \( I^* < I^* < I^* \) results in end-node-\( \alpha_j \) \( I^* < I^* < I^* \).

**Theorem 4.** Suppose that node \( I^* \) is a full node and \( p^0 \) is corresponding to \( I^* \). Consider all the start-node-\( \alpha_j \)(\( I^* \)) for \( m = 1, 2, \ldots, r \), and successive\( (\text{start-node}_{\alpha_j}(I^*)) \) if \( m = m_0 \) such that censive\( (\text{start-node}_{\alpha_j}(I^*)) \) - censive\( (\text{start-node}_{\alpha_j}(I^*)) \) (1) is minimum, say \( \Delta_{\text{min}} \) then successive\( (\text{start-node}_{\alpha_j}(I^*)) \) = \( I^* + \Delta_{\text{min}} \).

If successive\( (\text{start-node}_{\alpha_j}(I^*)) \) = \( I^* + \Delta_{\text{min}} \) and the corresponding input of the system is \( p^0 \), then the next vector of \( I^* \) is \( p^0 + u(\alpha_m) \).

This theorem can be proved by applying Lemma 5 inductively.

**Corollary 2:** If \( I^* \) is not a full node, and start-node-\( \alpha_j \)(\( I^* \)) does not exist, then \( I^* - \alpha_m \) should be used to substitute it in (1), and successive\( (\text{start-node}_{\alpha_j}(I^*)) \) should be node \( I^* \) such that \( I^* - (\alpha_m - \alpha_m) > 0 \) and \( I^* - (\alpha_m - \alpha_m) > 0 \) is the minimum, for all \( 0 \leq j \leq q \).

Given a vector \( p^0 \) as described in Definition 7 and Definition 8, the next vector of \( p^0 \) can be found by the following algorithm.

**Algorithm NEXT**

```plaintext```
input: a sequence of vectors \( p^0, p^2, \ldots, p^k \) (to the integer linear system)
output: \( p^k \) (next vector of \( p^k \))
```
1. \( I' = \alpha \cdot p^1 \) (as is executed on line 1, \( I' = \alpha \cdot p^1 \) should be already calculated and stored in a sorted order)
2. for \( m = 1 \) to \( r \) do
3. \( \text{start-nodes}_{\alpha_m}(I') = I' - \alpha_m \)
4. if \( I' = \text{successive}(\text{start-nodes}_{\alpha_m}(I')) \) then
   \( \Delta_m = I' - \text{start-nodes}_{\alpha_m}(I') \)
5. else
   find node \( I' \) such that \( I' \prec \Delta_m \)
6. if \( \Delta_m < \Delta_{m-1} \) do
   \( \Delta_m = \Delta_m \)
7. end of if
8. \( m_0 = m \)
9. \( p^{i+1} = p^i + u(\sigma_{m_0}) \)
10. end of for

Theorem 5. Algorithm NEXT finds the next vector \( p^{i+1} \) of vector \( p^i \) correctly.

Proof: First, it is easy to see that \( I' \) found in step 4 or step 5 is a node in the addressable digraph. \( p^j \)

is a vector such that \( \alpha \cdot p^j = I' \). According to the definition of successive node and the statement in step 5, \( P > \text{start-nodes}_{\alpha_m}(I') \).

Since \( \text{start-nodes}_{\alpha_m}(I') = I' - \alpha_m \), as specified in step 2, \( P > I' - \alpha_m \) and \( P + \alpha_m > I' \) for any \( \alpha_m \). Here \( P + \alpha_m \) is a node in the addressable digraph, and \( p^j + u(\alpha_m) \) is the corresponding vector satisfying \( \alpha \cdot (p^j + u(\alpha_m)) = (P + \alpha_m) \), according to Corollary 2.

Now we prove that there is no vector \( p^i \) existing such that \( I' < \alpha \cdot p^i < I' + u(\alpha_m) \).

i) \( I' \) can not be accessed by \( \alpha_{m_0} \), otherwise \( I' - \alpha_{m_0} \) is a node. This is because \( I' \prec I' + u(\alpha_{m_0}) \), and \( I' + \alpha_{m_0} \prec I' + \alpha_{m_0} \), which means that there is a node existing between \( I' + \alpha_{m_0} \) and \( I' \), thus contradicts the generating of \( I' \) stated in step 4 or 5.

ii) \( I' \) can not be accessed by \( \alpha_{m_0} \) for \( m = 1 \), \( 2 \), \( \ldots \), \( r \) other than \( m_0 \). If \( I' \) can be accessed by \( \alpha_{m_0} \), for \( m \in \{m|m = 1, 2, \ldots, r \} \), then \( I' - \alpha_{m_0} \) is a node. After executing the for loop in step 2 for \( m = m_0 \), \( (I' - \alpha_{m_0}) - (I' - \alpha_{m_0}) \) will be stored in \( \Delta_{m_0} \), and is equal to \( I' \).

After executing the for loop in step 2 for \( m = m_0 \), \( (I' - \alpha_{m_0}) - (I' - \alpha_{m_0}) \) will be stored in \( \Delta_{m_0} \) which is equal to \( I' + u(\alpha_{m_0}) \). Now, if \( I' \prec I' + u(\alpha_{m_0}) \), then \( I' - I' \prec I' + I' - I' \). \( \Delta_m |_{m = m_0} \prec \Delta_m |_{m = m_0} \), which contradicts the execution of step 7 to step 9. Thus \( I' \) is not possibly existing such that \( I' \prec I' \).

Let the largest entry in \( \alpha \) be \( \alpha_{m_0} \), for \( m = 1, 2, \ldots, r \). To examine the range of \( \text{start-nodes}_{\alpha_m}(I') \), notice that the smallest \( \text{start-nodes}_{\alpha_m}(I') \) is \( I' - \alpha_{m_0} \). Therefore, we have \( I' - \alpha_{m_0} < \text{start-nodes}_{\alpha_m}(I') \leq I' \).

The time complexity of Algorithm NEXT is \( O(r \log \alpha_{m_0}) \).

4. Conclusions

A compiler technique to solve the "cache or local memory thrashing" problem which is very interesting and very important in parallel processing has been developed. In particular, a very common parallel program construct 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 or local memories for parallel programs. Methods of calculating the appropriate parallel loop indices for each processor in terms of the data stored in its cache or local memory have been used. Although researches in this area are not completed yet, our efforts are significant in the sense that it introduced the mathematical concepts, analyzed the array subscript expressions, and provided the effective approach as a base solution for further cache or local memory thrashing-eliminating tasks in parallel processing systems.

Reference