An Efficient Solution to the Cache Thrashing Problem Caused by True Data Sharing

Guohua Jin[*] Zhiyuan Li[†] Fujie Chen [‡]

September 5, 1997

Abstract

When parallel programs are executed on multiprocessors with private caches, a set of data may be repeatedly used and modified by different threads. Such data sharing can often result in cache thrashing, which degrades memory performance. This paper presents and evaluates a loop restructuring method to reduce or even eliminate cache thrashing caused by true data sharing in nested parallel loops. This method uses a compiler analysis which applies linear algebra and the theory of numbers to the subscript expressions of array references. Due to this method's simplicity, it can be efficiently implemented in any parallel compiler. Experimental results show quite significant performance improvements over existing static and dynamic scheduling methods.

Keywords: Multiprocessors, cache thrashing, true data sharing, parallel threads, loop transformations, parallelizing compilers.

[*]Rice University, Department of Computer Science, 6100 Main Street, Houston, TX 77005-1892, USA, or Changsha Institute of Technology, Department of Computer Science, Changsha, Hunan 410073, CHINA. Work supported by a grant from the National Natural Science Foundation, China.

[†]Purdue University, Department of Computer Science, 1398 University Street, West Lafayette, IN 47907, USA. Work supported by a National Science Foundation CAREER Award, grant number CDA 9414015.

[‡]Changsha Institute of Technology, Department of Computer Science, Changsha, Hunan 410073, CHINA. Work supported by a grant from the National Natural Science Foundation, China.

1 Introduction

Parallel processing systems with memory hierarchies have become quite common today. Commonly, most multiprocessor systems have a local cache in each processor to bridge the speed gap between the processor and the main memory. Some systems use multi-level caches [5, 14]. Very often, a copy-back snoopy cache protocol is employed to maintain cache coherence in these multiprocessor systems. Certain supercomputers also use a local memory which can be viewed as a program-controlled cache. When programs with nested parallel loops are executed on such parallel processing systems, it is important to assign parallel loop iterations to the processors in such a way that unnecessary data movement between different caches is minimized. For convenience, in this paper, we call each iteration of a parallel loop a *thread*. The following loop nest is an example.

```
    DO I=1,100
       ......
       DOALL J=1,100
          .....
          DO K=1,100
             ......
S:             A(J+K,I+J) = A(J+K,I+J)+...
             ......
          ENDDO
          ......
       ENDDOALL
       ......
    ENDDO
```

In this example, loop J is a parallel loop because, with the value of I fixed, statement S has no loop-carried dependences, while J varies. Loop J is executed 100 times in the loop nest, creating 10,000 iterations, or 10,000 threads, in total. Each thread addresses 100 elements of array A. Many array elements are repeatedly accessed by these threads as shown in Table 1 and Figure 1, where $T_{i,j}$ denotes the thread corresponding to loop index values $I = i$ and $J = j$, while T_i denotes the set of threads created by the index value $I = i$. As shown in Figure 2, there exist lists of threads: ($T_{1,1}$), ($T_{1,2}$, $T_{2,1}$), ($T_{1,3}$, $T_{2,2}$, $T_{3,1}$), ($T_{1,4}$, $T_{2,3}$, $T_{3,2}$, $T_{4,1}$), ..., such that each thread modifies and reuses most of the array elements accessed by the neighboring threads in the same list. If the threads in the same list are assigned to different processors, the data of array A will unnecessarily move back and forth between different caches in the system, causing a cache thrashing problem due to true data sharing [12].

The nested loop construct shown in the above example is quite common in parallel code used for scientific computation. J. Fang and M. Lu studied a large number of programs including the LINPACK benchmarks, the PERFECT Club benchmarks, and programs for mechanical CAE, computational chemistry, image and signal processing, and petroleum applications [11]. They reported that almost all of the most time-consuming loop nests contain at least three loop levels, out of which 60% contain at least one parallel loop. Even after using loop interchange to move parallel loops outwards when it was legal, they still found 94% of the parallel loops enclosed by sequential loops. Such loop nests include the cases in which a parallel loop appears in the outermost loop level in a subroutine, but the subroutine is called by a call-statement which is contained by a sequential loop. Most of these loop nests are not perfectly nested, i.e. there exist statements right before or after an inner loop. Fang and Lu proposed a thread alignment algorithm to solve the cache thrashing problem which may be created by such multi-nested loops. Their algorithm, however, assigns threads to processors either by solving linear equations at run time or by storing the precomputed numerical solutions to the equations in the processor's memory. Since storing all the numerical solutions requires quite a large memory space, and since the exact number of threads often cannot be determined statically due to unknown loop bounds, they favor the on-line computation approach. In this paper, we present a method to reduce the run-time overhead in Fang and Lu's algorithm by using a thorough compiler analysis of array references to derive a closed-form formula that solves the thread assignment equations. The thread assignment then becomes highly efficient at run time. Previously, we presented preliminary algorithms [22, 23] to deal with a simple case in which data-dependent array references use the same linear function in the subscripts. No experimental data were given. In this paper, we extend the work by covering multiple linear functions and by clarifying the underlying theory. We report experimental results using a Silicon Graphics (SGI) multiprocessor.

Table 1: The elements of A accessed by each thread.

$$T_1 \left[\begin{array}{ll} T_{1,1} & A(2,2), A(3,2), A(4,2), A(5,2), ..., A(101,2) \\ T_{1,2} & A(3,3), A(4,3), A(5,3), A(6,3), ..., A(102,3) \\ T_{1,3} & A(4,4), A(5,4), A(6,4), A(7,4), ..., A(103,4) \\ \ldots & .... \\ T_{1,100} & A(101,101), A(102,101), A(103,101), A(104,101), ..., A(200,101) \end{array} \right.$$

$$T_2 \left[\begin{array}{ll} T_{2,1} & A(2,3), A(3,3), A(4,3), A(5,3), ..., A(101,3) \\ T_{2,2} & A(3,4), A(4,4), A(5,4), A(6,4), ..., A(102,4) \\ T_{2,3} & A(4,5), A(5,5), A(6,5), A(7,5), ..., A(103,5) \\ \ldots & .... \\ T_{2,100} & A(101,102), A(102,102), A(103,102), A(104,102), ..., A(200,102) \end{array} \right.$$

$$..................$$

$$T_{100} \left[\begin{array}{ll} T_{100,1} & A(2,101), A(3,101), A(4,101), A(5,101), ..., A(101,101) \\ T_{100,2} & A(3,102), A(4,102), A(5,102), A(6,102), ..., A(102,102) \\ T_{100,3} & A(4,103), A(5,103), A(6,103), A(7,103), ..., A(103,103) \\ \ldots & .... \\ T_{100,100} & A(101,200), A(102,200), A(103,200), A(104,200), ..., A(200,200) \end{array} \right.$$

With our method, the compiler analyzes the data dependences between the threads and uses that information to restructure the nested loop, perfectly nested or otherwise, in order to reduce or even eliminate true data sharing, which causes cache thrashing. Our method can be efficiently implemented in any parallel compiler, and our experimental results show quite significant improvement over existing static and dynamic scheduling methods.

In what follows, we first address related work. We then introduce basic concepts and assumptions. After that, we present solutions to the cache thrashing problem due to true data sharing, and lastly we show the experimental results conducted on an SGI multiprocessor system.

2 Related Work

Extensive research regarding efficient memory hierarchies has been reported in the literature.

Abu-Sufah, Kuck and Lawrie use loop blocking to improve paging performance by improving the locality of references [2]. Wolfe proposes iteration space tiling as a way to improve data reuse in a cache or a local memory [35]. Gallivan, Jalby and Gannon define a reference window for a dependence as the variables referenced by both the source and the sink of the dependence [15, 16]. After executing the source of the dependence, they save the associated reference window in the cache until the sink has also

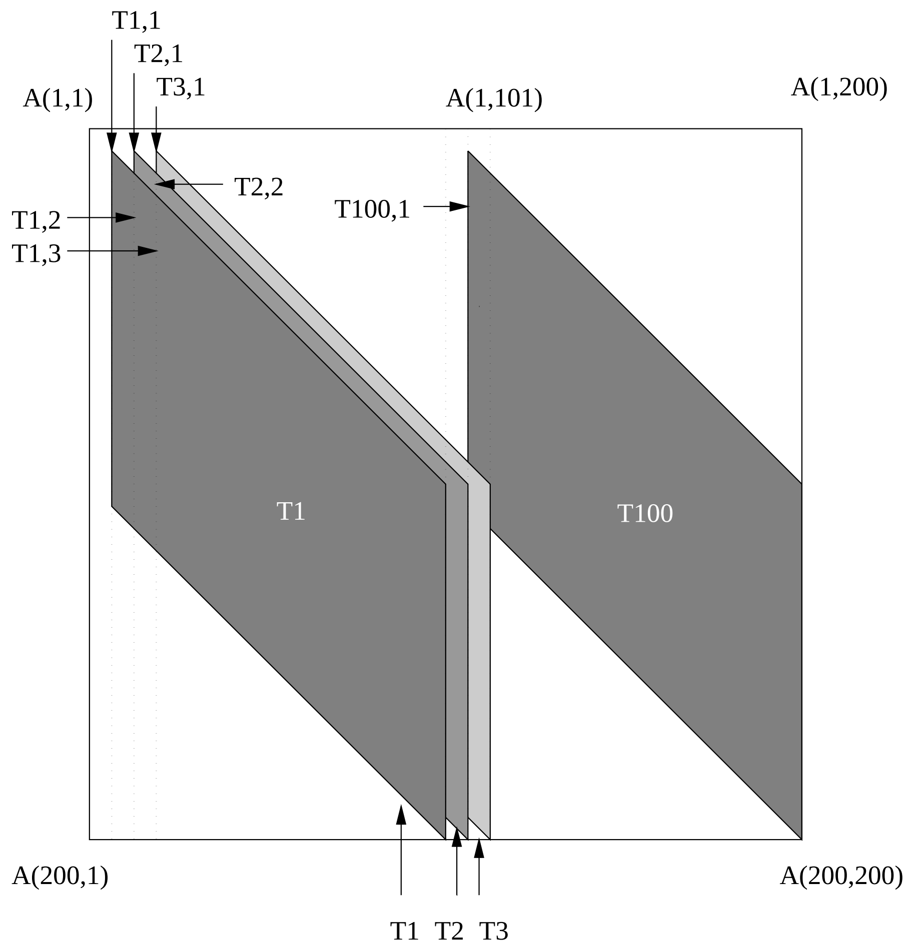


Figure 1: The elements of A accessed by different threads.

been executed, which may increase the number of cache hits. Carr, Callahan and Kennedy [7, 8] discuss options for compiler control of a uniprocessor's memory hierarchy. Wolf and Lam develop an algorithm that estimates all temporal and spatial reuse of a given loop permutation [34]. These optimizations all attempt to maximize the reuse of cached data on a single processor. They also have a secondary effect of improving multiprocessor performance by reducing the bandwidth requirement of each processor, thereby reducing contention in the memory system. In contrast, our work considers a multiprocessor environment where each processor has its own local cache or its own local memory, and where different processors may share data.

The work by Peir and Cytron [28], Shang and Fortes [30], and by D'Hollander [9] share the common goal of partitioning an index set into independent execution subsets such that the corresponding loop iterations can execute on different processors without interprocessor communication. Their methods apply to a specific type of loop nest called a *uniform recurrence* or a *uniform dependence algorithm*, in which the loops are perfectly nested, the loop bounds are constant, the *loop-carried dependences* have constant distances, and the array subscripts are of the form $i + c$, where i is a loop index and c an integer constant. Hudak and Abraham [1, 18] develop a static partitioning approach called *adaptive data partitioning* (ADP) to reduce interprocessor communication for iterative data-parallel loops. They also assume perfectly nested loops. The loop body is restricted to update a single data point $A(i, j)$ within a two-dimensional global matrix A. The subscript expressions of right-hand side array references are restricted to be the sum of a parallel loop index and a small constant, while the subscript expressions of left-hand array references are restricted to contain the parallel loop indices only. Tomko and Abraham

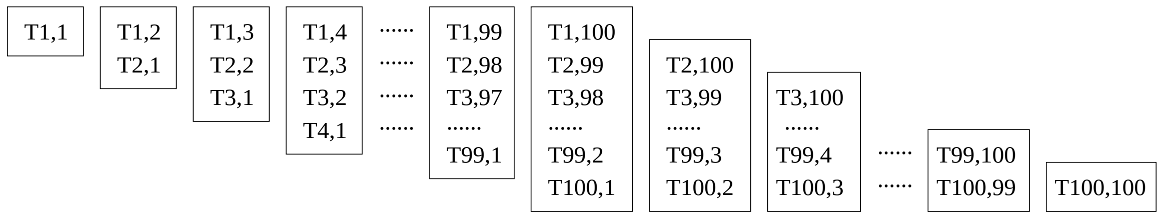


Figure 2: Lists of threads accessing similar array elements.

[32] develop iteration partitioning techniques for data-parallel application programs. They assume that there is only one pair of data access functions and that each loop index variable can appear in only one dimension of each array subscript expression. Agarwal, Kranz, and Natarajan [3] propose a framework for automatically partitioning parallel loops to minimize cache coherence traffic on shared-memory multiprocessors. They restrict their discussion to perfectly nested doall loops, and assume rectangular iteration spaces. Unlike these previous works, our work considers nested loops which are not necessarily perfectly nested. Loop bounds can be any variables, and array subscript expressions are much more general. Many researchers have studied the cache false sharing problem [10, 17, 19, 33] in which cache thrashing occurs when different processors share the same cache line of multiple words, although the processors do not share the same word. Many algorithms have been proposed to reduce false sharing by better memory allocation, better thread scheduling, or by program transformations. Our work considers cache thrashing which is due to the *true sharing* of data words.

Our work is most closely related to the research done by Fang and Lu [11, 12, 26, 13]. In their work, the iteration space is partitioned into a set of equivalence classes, and each processor uses a formula to determine which iterations belong to the same equivalence class at execution time. Each processor then executes the corresponding iterations so as to reduce or eliminate cache thrashing. These iterations are the solution vectors of a linear integer system. In Fang and Lu's work, these vectors may either be computed at run time or may be precomputed and later retrieved at run time when loop bounds are known before execution. Both approaches require additional execution time when a processor fetches the next iteration. Unlike Fang and Lu's approaches, we solve the thrashing problem at compile time to reduce run-time overhead, while we achieve the same effect of reducing cache thrashing. Our new method restructures the loops at compile time and is based on a thorough analysis of the relationship between the array element accesses and the loop indices in the nested loop. We have performed experiments on a commercial multiprocessor, namely a Silicon Graphics Challenge Cluster, thereby obtaining real data regarding cache thrashing and its reduction. In contrast, previous data were mainly from simulations [11, 12, 26].

3 Basic Concepts and Assumptions

Data dependences between statements are defined in [6, 24, 4, 25]. If a statement S_1 uses the result of another statement S_2, then S_1 is flow-dependent on S_2. If S_1 can safely store its result only after S_2 fetches the old data stored in that location, then S_1 is anti-dependent on S_2. If S_1 overwrites the result of S_2, then S_1 is output-dependent on S_2. A dependence within an iteration of a loop is called a loop-independent dependence. A dependence across the iterations of a loop is called a loop-carried dependence.

There can be no cache thrashing due to true data sharing if the outermost loop is parallel, because no data dependences will cross the threads. Therefore, in this paper, we consider only loop nests whose outermost loops are sequential. To simplify our discussion, we make the following assumptions about the program pattern: 1) All functions representing array subscript expressions are linear. 2) The loop construct considered here consists of a sequential loop which embraces one or several single-level parallel loops. If there exist multilevel parallel loops, only one level is parallelized, as on most commercial shared-memory multiprocessor systems. Hence, as shown below, a loop nest in our model has three levels: a parallel loop, its immediately enclosed sequential loop, and its immediately enclosing sequential loop:

```
DO I = 1, N1
    .....
    DOALL J = 1, N2
        ......
        DO K = 1, N3
            ......
            A(h(I, J, K)) =...
            ......
            ...= A(h'(I, J, K))+...
            ......
        ENDDO
        ......
    ENDDOALL
    ......
ENDDO
```

Figure 3: The loop nest model.

where $\vec{h}, \vec{h}^{'}$ are linear mappings from the iteration space $N_1 \times N_2 \times N_3$ to the domain space $M_1 \times M_2$ of A:

$$\vec{h}, \vec{h}^{'} : N_1 \times N_2 \times N_3 \rightarrow M_1 \times M_2,$$

and they can be expressed as:

$$\vec{h}(i, j, k) = (a_1 i + b_1 j + c_1 k + d_1, a_2 i + b_2 j + c_2 k + d_2)$$

$$\vec{h}^{'}(i, j, k) = (a_1^{'} i + b_1^{'} j + c_1^{'} k + d_1^{'}, a_2^{'} i + b_2^{'} j + c_2^{'} k + d_2^{'}).$$

So, if $\vec{h}(i, j, k) = (i+j, k)$ and $\vec{h}^{'}(i, j, k) = (i, k)$, for example, $A(\vec{h}(I, J, K))$ and $A(\vec{h}'(I, J, K))$ represent array accesses $A(i + j, k)$ and $A(i, k)$. The loops in the above example are not necessarily perfectly nested. Our restructuring techniques, to be presented later, assume arbitrary loop bounds, although we are showing lower bounds of 1 here for simplicity of notation. Multiple array variables and multiple linear subscript functions may exist in the nested loop.

Since we are considering cache thrashing due to true data sharing, i.e. due to data dependences between threads, we can also write the loop nest in Figure 3 as:

```
DO I = 1, N1
    .....
    DOALL J = 1, N2
        ......
        DO K = 1, N3
            LOOP BODY {(Aγ(hγ(I, J, K)), Aγ(h'γ(I, J, K)))|1 ≤ γ ≤ m}
        ENDDO
        ......
    ENDDOALL
    ......
ENDDO
```

where $A_\gamma(\gamma = 1, 2, ..., m)$ is an array name appearing in the loop body, $\vec{h}_\gamma, \vec{h}_\gamma^{'}(1 \leq \gamma \leq m)$ are linear mappings from iteration space $N_1 \times N_2 \times N_3$ to domain space $M_{1_\gamma} \times M_{2_\gamma}$ of A_γ, $(A_\gamma(\vec{h}_\gamma(I, J, K)),$

$A_\gamma(\vec{h}'_\gamma(I,\ J,\ K)))(1 \le \gamma \le m)$ are potentially dependent reference pairs, and m is the number of such pairs. Fang and Lu [11] reported that arrays involved in nested loops are usually two-dimensional or three-dimensional with a small-sized third dimension. The latter can be treated as a small number of two-dimensional arrays. Nested loops with the parallel loop at the innermost level are degenerate cases of the loop nest in Figure 3. Therefore, our loop nest model seems quite general. Our method can also be applied to a loop nest which contains several separate parallel loops at the middle level. Each of these parallel loops may be restructured according to its own reference patterns, such that the threads in different instances of the same parallel loop are aligned. We currently do not align the threads created by different parallel inner loops. For programs with more complicated loop nests, pattern-matching techniques can be used to identify a loop subnest that matches the nest shown in Figure 3. Other outer- or inner- loops that are not a part of the subnest can be ignored, as long as their loop indices do not appear in the array subscripts.

The compiler analysis is based on a simple multiprocessor model in which the cache memory has the following characteristics: 1) it is local to a processor; 2) it uses a copy-back snoopy coherence strategy; and 3) its line size is one word. The transformed code, however, will execute correctly on machines which have a multiword cache line and multilevel caches. Furthermore, as the experimental results will show, the performance of the transformed code is quite good on such realistic machines. Our analysis can also be extended to incorporate more machine parameters such as the cache line size.

4 Solutions

In this section, we analyze the relationship between linear functions in the array subscripts. Based on this analysis, we restructure a given loop nest to reduce or eliminate the cache thrashing due to true data sharing. We consider nested loops which are not necessarily perfectly nested and which may have variable loop bounds. For clarity of presentation, in Section 4.1 we first discuss how to deal with dependent reference pairs such that the same subscript function is used in both references in each pair (Different pairs may use different subscript functions). Later in Section 4.2, we will discuss how to deal with more general cases by using simple affine transforms to fit to this model.

4.1 The Basic Model

In this subsection, we assume that for each pair of dependent references, the same subscript function is used in both references. Under this assumption, if we extract the subscript function $\vec{h}_\gamma(I,J,K)$ from each pair of dependent references, then a model for a nested loop which has m pairs of dependent references can be illustrated by the following code segment.

```
DO I = 1, N_1
   .....
   DOALL J = 1, N_2
      ......
      DO K = 1, N_3
         LOOP BODY {h_γ(I, J, K) | 1 ≤ γ ≤ m}
      ENDDO
      ......
   ENDDOALL
   ......
ENDDO
```

Without loss of generality, suppose that all m linear subscript functions above are different. We assume that each function $\vec{h}_\gamma$, $1 \le \gamma \le m$, is of rank 2 and is in the form of $\vec{h}_\gamma(i,j,k) = (f_\gamma(i,j,k), g_\gamma(i,j,k))$, where

$$f_\gamma(i,j,k) = a_{\gamma,1}i + b_{\gamma,1}j + c_{\gamma,1}k + d_{\gamma,1}$$

$$g_\gamma(i,j,k) = a_{\gamma,2}i + b_{\gamma,2}j + c_{\gamma,2}k + d_{\gamma,2}.$$

Take the following example.

```
DO I = 1, N1
   ......
   DOALL J = 1, N2
      .....
      DO K = 1, N3
         ......
         A(J + 3K, J − 2I) = A(J + 3K, J − 2I)+...
         ......
         B(J + 3K + 1, I + J) = B(J + 3K + 1, I + J)+...
         ......
      ENDDO
      ......
   ENDDOALL
   ......
ENDDO
```

Figure 4: A nested loop with multiple linear subscript functions.

In this example, no data dependences exist within loop J. However, two data dependences exist in the whole loop nest, one between the references to A, and the other between those to B. We have two linear functions to consider, one for each dependence:

$$f_1(i,j,k) = j + 3k, \qquad g_1(i,j,k) = -2i + j$$

$$f_2(i,j,k) = j + 3k + 1, \qquad g_2(i,j,k) = i + j$$

The iteration subspace $N_1 \times N_2$ is called the *reduced* iteration space because it omits the K loop. In order to find the iterations in the reduced iteration space which may access common memory locations within the corresponding threads, we define a set of elements of array A_γ which are accessed within thread T_{i_0,j_0} by using subscript function $\vec{h}_\gamma$.

Definition 1: Given iteration (i_0, j_0) in the reduced iteration space, the elements $A_\gamma(f_\gamma(i_0, j_0, k)$, $g_\gamma(i_0, j_0, k))$ of array A_γ, where $1 \le k \le N_3, 1 \le \gamma \le m$, are accessed within thread T_{i_0,j_0}. They are denoted by $\mathcal{A}_\gamma^{i_0,j_0}$,

$$\mathcal{A}_\gamma^{i_0,j_0} = \{A_\gamma(f_\gamma(i_0,j_0,k), g_\gamma(i_0,j_0,k)) |\text{ for given } (i_0,j_0), \text{ where } 1 \le k \le N_3\}.$$

Definition 2: If we suppose that $T_{i,j}$ and $T_{i',j'}$ are two threads corresponding to (I,J)=(i,j) and (I,J)=(i',j') in the reduced iteration space of the given loop nest such that $\mathcal{A}_\gamma^{i,j} \bigcap \mathcal{A}_\gamma^{i',j'} \ne \emptyset (1 \le \gamma \le m)$, we say $T_{i,j}$ and $T_{i',j'}$ has a dependence because of $\vec{h}_\gamma$, denoted by $T_{i,j}\ \delta_\gamma\ T_{i',j'}$.

Definition 3: If there exists γ, $1 \le \gamma \le m$, such that $T_{i,j}\ \delta_\gamma\ T_{i',j'}$, then we write $T_{i,j}\ \delta\ T_{i',j'}$. If $T_{i,j}\ \delta\ T_{i',j'}$ and $T_{i',j'}\ \delta\ T_{i'',j''}$, then we also have $T_{i,j}\ \delta\ T_{i'',j''}$.

Since both f_γ and g_γ are linear in terms of i, j and k, the following lemma is obvious.

Lemma 1: In the program pattern described above, if there exist γ, $1 \le \gamma \le m$, and two iterations in the iteration space $N_1 \times N_2 \times N_3$, (i,j,k) and (i',j',k'), where $i < i'$, such that

$$f_\gamma(i,j,k) = f_\gamma(i',j',k')$$

$$g_\gamma(i,j,k) = g_\gamma(i^{'},j^{'},k^{'})$$

then for any constant n_0, we have a series of iterations in the space, $(i,j,k+n_0)$ and $(i^{'},j^{'},k^{'}+n_0)$ that satisfy the following equations:

$$f_\gamma(i,j,k+n_0) = f_\gamma(i^{'},j^{'},k^{'}+n_0)$$

$$g_\gamma(i,j,k+n_0) = g_\gamma(i^{'},j^{'},k^{'}+n_0)$$

where $1 \le k+n_0 \le N_3, 1 \le k^{'}+n_0 \le N_3$.

The following lemma and two theorems establish the relationship between the loop indexes corresponding to two inter-dependent threads. We will use this index relationship to *stagger* the loop iteration space such that inter-dependent threads can be assigned to the same processors.

Lemma 2: Let $T_{i,j}$, $T_{i^{'},j^{'}}$ be two threads, $i \ne i^{'}$. $T_{i,j}\ \delta_\gamma\ T_{i^{'},j^{'}}$ iff there exist k, $k^{'}$, $1 \le k, k^{'} \le N_3$, such that

$$a_{\gamma,1}i + b_{\gamma,1}j + c_{\gamma,1}k = a_{\gamma,1}i^{'} + b_{\gamma,1}j^{'} + c_{\gamma,1}k^{'}$$

$$a_{\gamma,2}i + b_{\gamma,2}j + c_{\gamma,2}k = a_{\gamma,2}i^{'} + b_{\gamma,2}j^{'} + c_{\gamma,2}k^{'}.$$

Theorem 1: Let $b_{\gamma,1}a_{\gamma,2} - a_{\gamma,1}b_{\gamma,2} \ne 0$. $T_{i,j}\ \delta_\gamma\ T_{i^{'},j^{'}}$ iff there exist k, $k^{'}$ $(1 \le k, k^{'} \le N_3)$ such that

$$i^{'} - i = \frac{b_{\gamma,1}c_{\gamma,2} - c_{\gamma,1}b_{\gamma,2}}{b_{\gamma,1}a_{\gamma,2} - a_{\gamma,1}b_{\gamma,2}}(k-k^{'})$$

$$j^{'} - j = \frac{a_{\gamma,2}c_{\gamma,1} - a_{\gamma,1}c_{\gamma,2}}{b_{\gamma,1}a_{\gamma,2} - a_{\gamma,1}b_{\gamma,2}}(k-k^{'}).$$

The proofs of Lemma 2 and Theorem 1 are obvious from Definition 3. We now consider the case of $b_{\gamma,1}a_{\gamma,2} - a_{\gamma,1}b_{\gamma,2} = 0$, assuming that the loop bounds, N_2 and N_3, are large enough to satisfy the following:

$$N_2 > max(|\frac{c_{\gamma,1}}{G.C.D.(b_{\gamma,1},c_{\gamma,1})}|, |\frac{c_{\gamma,2}}{G.C.D.(b_{\gamma,2},c_{\gamma,2})}|)$$

$$N_3 > max(|\frac{b_{\gamma,1}}{G.C.D.(b_{\gamma,1},c_{\gamma,1})}|, |\frac{b_{\gamma,2}}{G.C.D.(b_{\gamma,2},c_{\gamma,2})}|).$$

These assumptions are almost always true in practice [31]. When they are not true, the parallel loops will be too small to be important. With these assumptions, we have the following theorem.

Theorem 2 [21]: Let $b_{\gamma,1}a_{\gamma,2} - a_{\gamma,1}b_{\gamma,2} = 0$. The fact that the J loop at the middle level is a DOALL loop guarantees that $T_{i,j}\ \delta_\gamma\ T_{i^{'},j^{'}}$ iff we have
(1) $a_{\gamma,1}(i^{'}-i) + b_{\gamma,1}(j^{'}-j) = 0$, for $a_{\gamma,1} \ne 0$,
(2) $a_{\gamma,2}(i^{'}-i) + b_{\gamma,2}(j^{'}-j) = 0$, for $a_{\gamma,2} \ne 0$, and
(3) $j^{'} - j = 0$, for $a_{\gamma,1} = a_{\gamma,2} = 0$.

In order to find the threads which have data dependence relations with thread $T_{i,j}$, we make the following definition.

Definition 4: Given iteration (i,j) in the reduced iteration space, we let $S_{i,j}$ denote the following set of iterations in the space:

$$S_{i,j} = \{(i + \sum_{\gamma=1}^{m} r_\gamma L_{\gamma,1}, j + \sum_{\gamma=1}^{m} r_\gamma L_{\gamma,2}) | r_\gamma \in Z, 1 \le \gamma \le m\}$$

where $L_{\gamma,1}(L_{\gamma,1} \neq 0)$ and $L_{\gamma,2}(1 \leq \gamma \leq m)$ are defined as:
(1) $L_{\gamma,1} = \frac{b_{\gamma,1}c_{\gamma,2}-c_{\gamma,1}b_{\gamma,2}}{GCD_\gamma}$ and $L_{\gamma,2} = \frac{a_{\gamma,2}c_{\gamma,1}-a_{\gamma,1}c_{\gamma,2}}{GCD_\gamma}$, if $b_{\gamma,1}a_{\gamma,2} - a_{\gamma,1}b_{\gamma,2} \neq 0$,
with GCD_γ equal to $G.C.D.(b_{\gamma,1}c_{\gamma,2}-c_{\gamma,1}b_{\gamma,2}, a_{\gamma,2}c_{\gamma,1}-a_{\gamma,1}c_{\gamma,2}, b_{\gamma,1}a_{\gamma,2}-a_{\gamma,1}b_{\gamma,2})$ or equal to $-G.C.D.$ $(b_{\gamma,1}c_{\gamma,2}$- $c_{\gamma,1}b_{\gamma,2}$, $a_{\gamma,2}c_{\gamma,1}$-$a_{\gamma,1}c_{\gamma,2}$, $b_{\gamma,1}a_{\gamma,2}$-$a_{\gamma,1}b_{\gamma,2})$ to guarantee $L_{\gamma,1} > 0$;
(2) $L_{\gamma,1} = \frac{b_{\gamma,1}}{GCD_\gamma}$ and $L_{\gamma,2} = \frac{-a_{\gamma,1}}{GCD_\gamma}$, if $b_{\gamma,1}a_{\gamma,2} - a_{\gamma,1}b_{\gamma,2} = 0$ and $a_{\gamma,1} \neq 0$,
with GCD_γ equal to $G.C.D.(a_{\gamma,1}, b_{\gamma,1})$ or equal to $-G.C.D.(a_{\gamma,1}, b_{\gamma,1})$ to guarantee $L_{\gamma,1} > 0$;
(3) $L_{\gamma,1} = \frac{b_{\gamma,2}}{GCD_\gamma}$ and $L_{\gamma,2} = \frac{-a_{\gamma,2}}{GCD_\gamma}$, if $b_{\gamma,1}a_{\gamma,2} - a_{\gamma,1}b_{\gamma,2} = 0$ and $a_{\gamma,2} \neq 0$,
with GCD_γ equal to $G.C.D.(a_{\gamma,2}, b_{\gamma,2})$ or equal to $-G.C.D.(a_{\gamma,2}, b_{\gamma,2})$ to guarantee $L_{\gamma,1} > 0$;
(4) $L_{\gamma,1} = 1$ and $L_{\gamma,2} = 0$, if $a_{\gamma,1} = a_{\gamma,2} = 0$.
$(L_{\gamma,1}, L_{\gamma,2})$ is called the *staggering parameter* corresponding to linear function $\vec{h}_\gamma$. If there exist no data dependences between the given pair of references, we define the staggering parameter $(L_{\gamma,1}, L_{\gamma,2})$ as $(0, 0)$.

The staggering parameters for the example in Figure 4 can be calculated to be: $(L_{1,1}, L_{1,2}) = (3, 6)$ and $(L_{2,1}, L_{2,2}) = (3, -3)$, according to Definition 4(1).

The following theorem, derived from Theorems 1 and 2 and Definition 4, states that we can use the staggering parameters to uniquely partition the threads into independent sets.

Theorem 3: $S_{i,j}$ as defined above satisfies:
(1) if $(i, j) \in S_{i',j'}$ and $(i, j) \in S_{i'',j''}$ then $S_{i',j'} = S_{i'',j''}$,
(2) if $(i^{'}, j^{'}) \in S_{i,j}$ then $S_{i,j} = S_{i',j'}$, and
(3) if $(i^{'}, j^{'}) \in S_{i,j}$ and $T_{i',j'}\ \delta\ T_{i'',j''}$ then $(i^{''}, j^{''}) \in S_{i,j}$.

The theorem above indicates that $S_{i,j}$ includes all the iterations whose corresponding threads have a data dependence relation with $T_{i,j}$. We call $S_{i,j}$ an *equivalence class* of the reduced iteration space. In order to eliminate true data sharing, threads in the same equivalence class should be assigned to the same processor. We want to restructure the reduced iteration space such that threads in the same equivalence class will appear in the same column. Each staggering parameter (L_1, L_2) computed for a dependent reference pair tells us that if we stagger the $(i + L_1)$-th row in the reduced iteration space by $|L_2|$ columns to the right if $L_2 < 0$, or to the left if $L_2 > 0$, relative to the i-th, then the threads involved in the dependence pair will be aligned in the same column. Different staggering parameters may require staggering the iteration space in different ways. However, if these staggering parameters are in proportion, then staggering by the *unified staggering parameter* defined below will satisfy all the requirements simultaneously.

Definition 5: Given staggering parameters $(L_{1,1}, L_{1,2}), (L_{2,1}, L_{2,2}), ..., (L_{m,1}, L_{m,2})$ and $g = G.C.D.$ $(L_{1,1}, L_{2,1}, ..., L_{m,1})$, if $\forall (j, k)$ such that $1 \leq j, k \leq m$, we have $\frac{L_{k,1}}{L_{k,2}} = \frac{L_{j,1}}{L_{j,2}}$, and then we call $(g, \frac{L_{1,2}}{L_{1,1}}g)$ the *unified staggering parameter.*

Lemma 3 [21]: If the condition $\frac{L_{k,1}}{L_{k,2}} = \frac{L_{j,1}}{L_{j,2}}$ $(1 \leq j, k \leq m)$ in Definition 5 is true, then (a) the iterations (i, j) and (i, k), where $j \neq k$, belong to two different equivalent classes; and (b) the iterations (i, j) and $(i^{'}, k)$, where $0 < i^{'} - i < g$, belong to two different equivalence classes.

Theorem 4 [21]: If the condition $\frac{L_{k,1}}{L_{k,2}} = \frac{L_{j,1}}{L_{j,2}}$ $(1 \leq j, k \leq m)$ in Definition 5 is true, then the reduced iteration space must be staggered according to the unified staggering parameter $(g, \frac{L_{1,2}}{L_{1,1}}g)$ in order to reduce or eliminate data sharing among the threads, i.e. the $(i + g)$-th row in the reduced iteration space must be staggered by $|\frac{L_{1,2}}{L_{1,1}}g|$ columns to the right if $L_{1,2} < 0$, or to the left if $L_{1,2} > 0$, relative to the i-th row.

If a given loop nest satisfies the condition $\frac{L_{k,1}}{L_{k,2}} = \frac{L_{j,1}}{L_{j,2}}$ $(1 \le j, k \le m)$ in Definition 5, then, according to Theorem 4 above, the reduced iteration space can be transformed into a *staggered and reduced iteration space (SRIS)* by leaving the first g rows unchanged, $g = G.C.D.(L_{1,1}, L_{2,1}, ..., L_{m,1})$, and by staggering each of the remaining rows using the unified staggering parameter. There will be no data dependences between different columns in the SRIS.

However, if the staggering parameters are not in proportion, i.e, if there exist (j, k) such that $1 \le j, k \le m$ and $\frac{L_{k,1}}{L_{k,2}} \ne \frac{L_{j,1}}{L_{j,2}}$, then we can no longer obtain a unique unified staggering parameter. Moreover, staggering alone is no longer sufficient for eliminating data dependences between the different columns in the restructured iteration space. This is because some threads in the same equivalence class are still in different columns. We perform a procedure called *compacting* which stacks these columns onto each other. We will discuss staggering first.

Definition 6: Given staggering parameters $(L_{1,1}, L_{1,2}), (L_{2,1}, L_{2,2}), ..., (L_{m,1}, L_{m,2})$ and $g = G.C.D.$ $(L_{1,1},\ L_{2,1},\ ...,\ L_{m,1})$, suppose there exists (j, k) such that $1 \le j, k \le m$ and $\frac{L_{k,1}}{L_{k,2}} \ne \frac{L_{j,1}}{L_{j,2}}$. According to the theory of numbers [27], there exist integers $a_1,\ a_2,\ ...,\ a_m$ that satisfy

$$g = \sum_{\gamma=1}^{m} a_\gamma L_{\gamma,1}.$$

Let $g' = \sum\limits_{\gamma=1}^{m} a_\gamma L_{\gamma,2}$. We call (g, g') a *unified staggering parameter.*

Note that since the m-tuple $(a_1,\ a_2,\ ...,\ a_m)$ is not necessarily unique, the (g, g') pair may not be unique either. With Definition 6, a unified staggering parameter (g, g') of the example in Figure 4 is found to be $g = G.C.D.(3, 3) = 0 \times 3 + 1 \times 3 = 3$ and $g' = 0 \times 6 + 1 \times (-3) = -3$.

After staggering by using any unified staggering parameter (g, g'), the resulting SRIS has four possible shapes, as shown in Figure $5(b - e)$. Figure $5(a)$ shows details of one of these shapes. Figure $6(a)$ and $6(b)$ show the reduced iteration space for the example in Figure 4 before and after staggering with (3,-3) as the unified staggering parameter.

Next, we compute the *compacting parameter* d using Algorithm 1 and 2, to be presented shortly. We then partition the SRIS into n chunks, where $n = \left\lceil \frac{N_2 + (\lceil \frac{N_1}{g} \rceil - 1)g'}{d} \right\rceil$, which is the total number of columns in the SRIS devided by the compacting parameter d (Figure $5(d - e)$). These d-wide chunks are stacked onto each other to form a compacted iteration space of width d, as shown in Figure 7. As we will explain later, the threads in different columns after compacting the SRIS with d are independent. Moreover, the product of d and g equals the number of equivalence classes. The SRIS shown in Figure $6(b)$ for our example is transformed by being compacted with $d = 9$ into the form shown in Figure 8.

The following algorithm computes the compacting parameter d.

Algorithm 1:

Input: A set of staggering parameters $(L_{1,1}, L_{1,2}), (L_{2,1}, L_{2,2}), ..., (L_{m,1}, L_{m,2})$.
Output: The compacting parameter d.

Step 1: For each 2-element subset, $\{L_{i,1}, L_{j,1}\}$, of $\{L_{1,1}, L_{2,1}, ..., L_{m,1}\}$, compute

$$d_2\langle L_{i,1}, L_{j,1}\rangle = \frac{L_{j,1}L_{i,2} - L_{i,1}L_{j,2}}{G.C.D.(L_{i,1}, L_{j,1})}.$$

Let $d_2 = G.C.D.$ of all such $d_2\langle L_{i,1}, L_{j,1}\rangle$.

Step 2: For each j-element subset, $\{L_{i_1,1}, L_{i_2,1}, ..., L_{i_j,1}\}$, where $3 \le j \le m$, pick any element, say

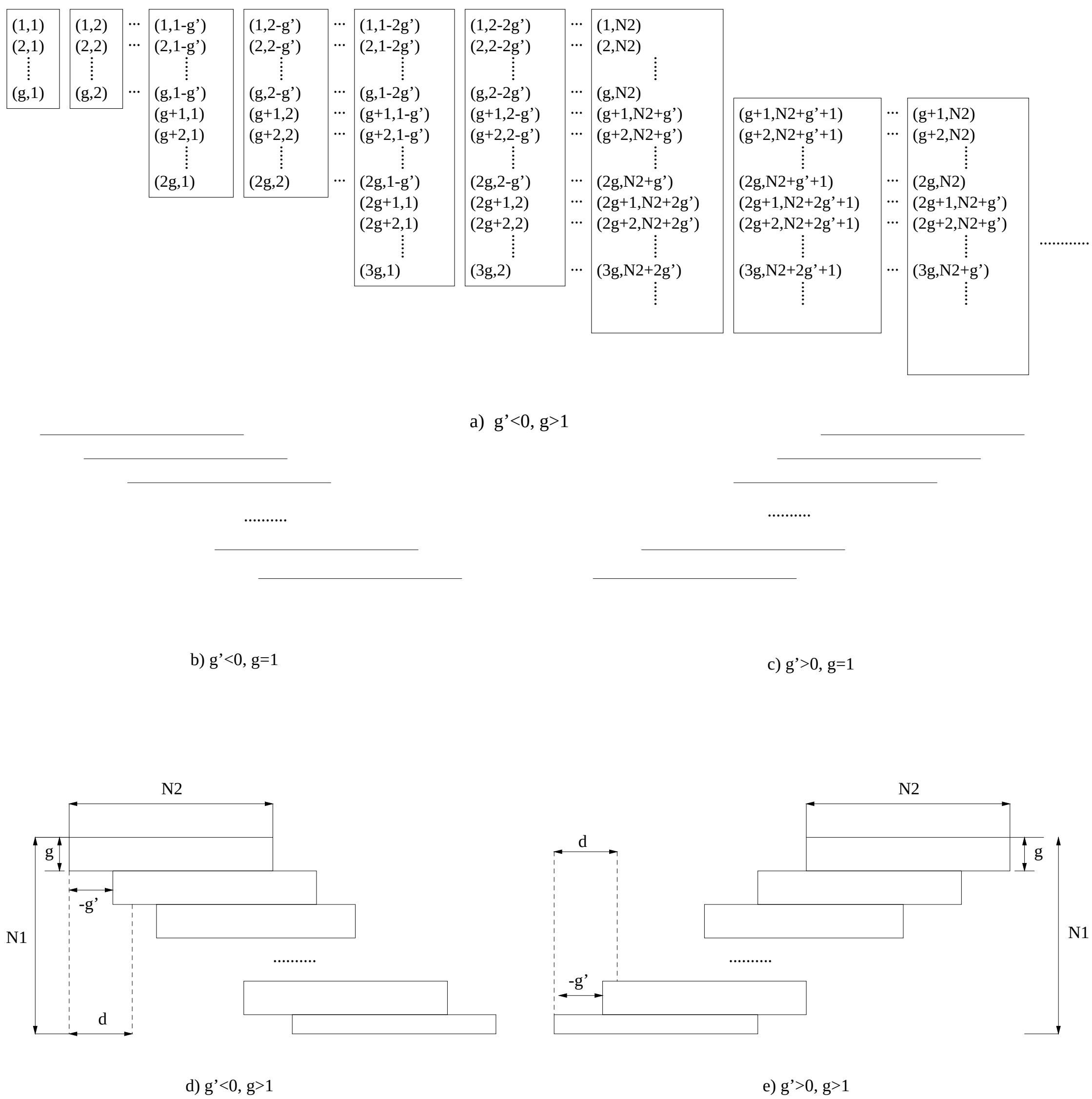


Figure 5: SRIS and outlines.

$L_{i_1,1}$, and compute

$$r_1 = \frac{L.C.D.(G.C.D.(L_{i_2,1}, ..., L_{i_j,1}), L_{i_1,1})}{L_{i_1,1}}.$$

Using the Euclidean Algorithm, compute integers $b_2, b_3, ..., b_j$ such that

$$G.C.D.(L_{i_2,1}, ..., L_{i_j,1}) = \sum_{\gamma=2}^{j} b_\gamma L_{i_\gamma,1}.$$

Apply Algorithm 2 below to find nonzero integers $r_2, ..., r_j$ such that

$$r_1 L_{i_1,1} = \sum_{\gamma=2}^{j} r_\gamma L_{i_\gamma,1}.$$

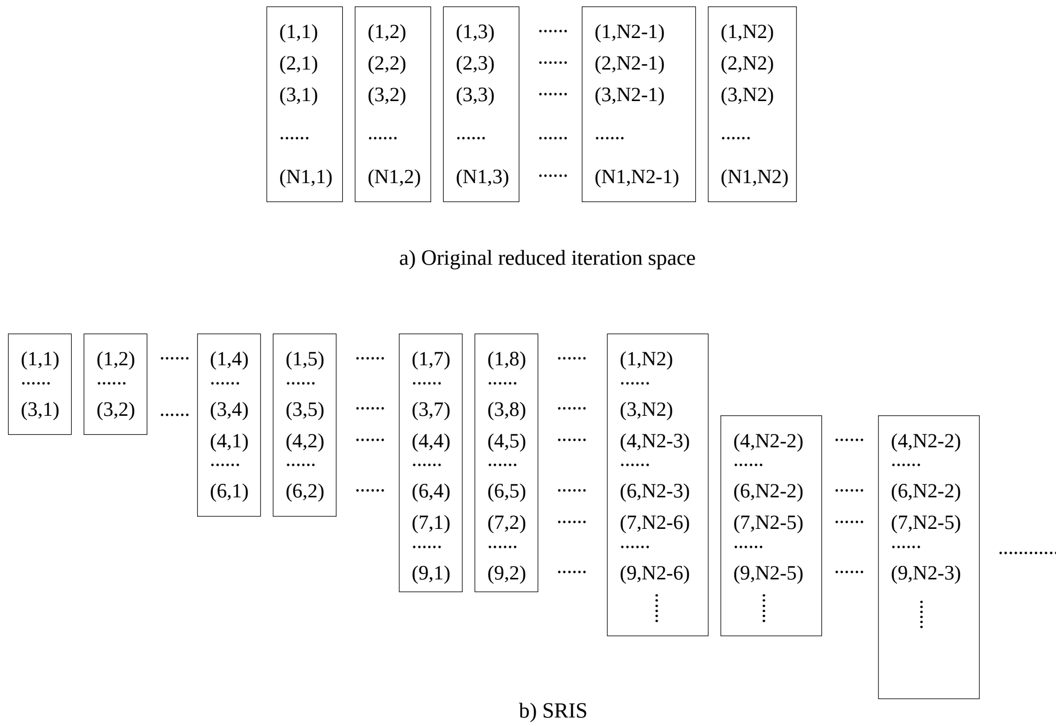


Figure 6: The reduced iteration space before and after rearrangement.

Let

$$d_j\langle L_{i_1,1}, L_{i_2,1}, ..., L_{i_j,1}\rangle = r_1 L_{i_1,2} - \sum_{\gamma=2}^{j} r_\gamma L_{i_\gamma,2}.$$

Step 3: For j from 3 to m, compute

$$d_j = G.C.D.(d_j\langle L_{i_1,1}, L_{i_2,1}, ..., L_{i_j,1}\rangle| \text{ for all distinct } \langle L_{i_1,1}, L_{i_2,1}, ..., L_{i_j,1}\rangle).$$

Step 4: $d = G.C.D.(d_2, d_3, ..., d_m)$.

As will be established later, d is unique regardless of the choice of $L_{i_1,1}$ in Step 2.

To calculate the compacting parameter d, non-zero integers $r_2, ..., r_j$ need to be found in Algorithm 1 from the integer coefficients $b_2, b_3, ..., b_j$ computed by the Euclidean Algorithm. Algorithm 2 is therefore invoked to derive a group of non-zero integer coefficients from a group of any integer coefficients of a linear expression.

Algorithm 2:

Input: Non-zero positive integers $p, L_1, L_2, ..., L_p$, integer $x \geq 0$, and integers $a_1, a_2, ..., a_p$ such that $x = a_1 L_1 + a_2 L_2 + ... + a_p L_p$.

Output: non-zero integers $b_1, b_2, ..., b_p$ such that $x = b_1 L_1 + b_2 L_2 + ... + b_p L_p$.

Step 1: If there are an even number of zero coefficients $a_{i_1}, a_{i_2}, ..., a_{i_{2k}} (0 \leq 2k \leq p)$ among $a_1, a_2, ..., a_p$, then let

$$b_j = a_j (a_j \neq 0)$$

$$b_{i_{2l-1}} = L_{i_{2l}} (1 \leq l \leq k)$$

$(1,1)$	$(1,2)$	$\cdots\cdots$	$(1,d)$
$(1,d+1)$	$(1,d+2)$	$\cdots\cdots$	$(1,2d)$
$\cdots$	$\cdots$		$\cdots$
$(g,1)$	$(g,2)$	$\cdots\cdots$	(g,d)
$(g,d+1)$	$(g,d+2)$	$\cdots\cdots$	$(g,2d)$
$\cdots$	$\cdots$		$\cdots$
$(g+1,a_1)$	$(g+1,a_2)$	$\cdots\cdots$	$(g+1,a_d)$
$(g+1,a_1+d)$	$(g+1,a_2+d)$	$\cdots\cdots$	$(g+1,a_d+d)$
$\cdots$	$\cdots$		$\cdots$
$(2g,a_1)$	$(2g,a_2)$	$\cdots\cdots$	$(2g,a_d)$
$(2g,a_1+d)$	$(2g,a_2+d)$	$\cdots\cdots$	$(2g,a_d+d)$
$\cdots$	$\cdots$		$\cdots$
$(2g+1,b_1)$	$(2g+1,b_2)$	$\cdots\cdots$	$(2g+1,b_d)$
$(2g+1,b_1+d)$	$(2g+1,b_2+d)$	$\cdots\cdots$	$(2g+1,b_d+d)$
$\cdots$	$\cdots$		$\cdots$

$$a_i = (i + (d - ((-g^{'})\ mod\ d)) - 1)\ mod\ d + 1\ (1 \leq i \leq d)$$

$$b_i = (i + (d - ((-2g^{'})\ mod\ d)) - 1)\ mod\ d + 1\ (1 \leq i \leq d)$$

Figure 7: Compacted SRIS.

$$b_{i_{2l}} = -L_{i_{2l-1}}(1 \leq l \leq k).$$

Step 2: If there are an odd number of zero coefficients $a_{i_1}, a_{i_2}, ..., a_{i_{2k+1}} (0 \leq 2k+1 \leq p)$ among $a_1, a_2, ..., a_p$, then let

$$b_j = a_j (a_j \neq 0)$$
$$b_{i_{2l-1}} = L_{i_{2l}}(1 \leq l \leq k-1)$$
$$b_{i_{2l}} = -L_{i_{2l-1}}(1 \leq l \leq k-1)$$
$$b_{i_{2k-1}} = L_{i_{2k}}$$
$$b_{i_{2k}} = -(L_{i_{2k-1}} + L_{i_{2k+1}})$$
$$b_{i_{2k+1}} = L_{i_{2k}}.$$

Obviously, the non-zero integers $b_1, b_2, ..., b_p$ computed by Algorithm 2 satisfy $x = b_1 L_1 + b_2 L_2 + ... + b_p L_p$.

For the example in Figure 4, since there are only two linear functions in the loop nest, only Step 1 and Step 4 of Algorithm 1 are used to calculate the compacting parameter d, that is,

$$d = d_2 \langle L_{1,1}, L_{2,1} \rangle = \frac{L_{2,1} L_{1,2} - L_{1,1} L_{2,2}}{G.C.D.(L_{1,1}, L_{2,1})} = \frac{3 \times 6 - 3 \times (-3)}{G.C.D.(3,3)} = 9.$$

Next, we need to establish two important facts. First, after compacting with d, the threads in different columns are independent. Second, the compacting parameter d computed by Algorithm 1 is the

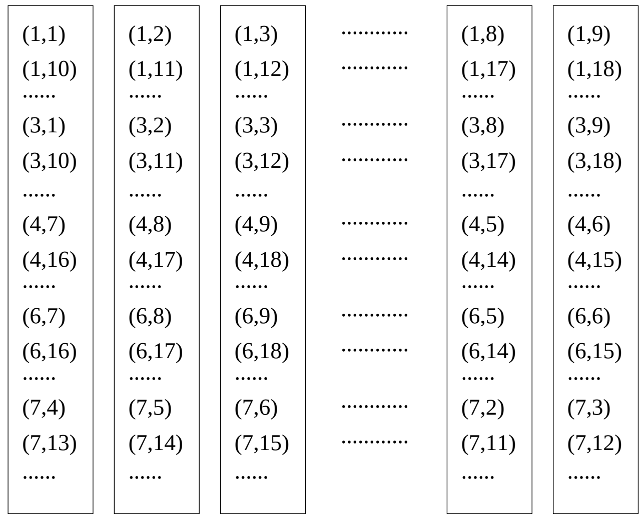


Figure 8: The reduced iteration space after compacting.

largest number of independent columns possible as the result of compacting the SRIS with a constant value. The first fact is established by Theorem 5, Theorem 6, and Corollary 1. To do so, we introduce the following definition.

Definition 7: Given an iteration (i, j) in the reduced iteration space, staggering parameters $(L_{1,1}, L_{1,2}), (L_{2,1}, L_{2,2}), ..., (L_{m,1}, L_{m,2})$, and the unified staggering parameter $(g, g^{'})$, and suppose a_1, a_2, ..., a_m are integers that satisfy

$$g = \sum_{\gamma=1}^{m} a_\gamma L_{\gamma,1},$$

a set of iterations $S^{'}_{i,j}$ is constructed as follows:

(1) For any integer r, iteration $(i + rg, j + rg^{'})$ in the space belongs to $S^{'}_{i,j}$, where $g^{'} = \sum_{\gamma=1}^{m} a_\gamma L_{\gamma,2}$;

(2) If there exist integers $r_1, r_2, ..., r_m$, not all zero, integer r, and iterations $(i^{'}, j^{'}), (i^{'}, k^{'})(k^{'} \neq j^{'})$ in the space, such that

$$k^{'} = j^{'} + \sum_{\gamma=1}^{m} r_\gamma L_{\gamma,2}, \sum_{\gamma=1}^{m} r_\gamma L_{\gamma,1} = 0$$

and

$$i = i^{'} + rg, j = j^{'} + rg^{'} (g^{'} = \sum_{\gamma=1}^{m} a_\gamma L_{\gamma,2}),$$

then $S^{'}_{i,j} = S^{'}_{i,j} \cup S^{'}_{i^{'},k^{'}}$.

The following three lemmas and Theorem 5 show that $S^{'}_{i,j}$ is the same as the equivalence class $S_{i,j}$. From the process of constructing $S^{'}_{i,j}$, we immediately have the following lemma.

Lemma 4: Given iterations (i, j) and $(i^{'}, j^{'})$ in the reduced iteration space, and a unified staggering parameter $(g, g^{'})$, if there exists integer r such that

$$i = i^{'} + rg$$

$$j = j^{'} + rg^{'}$$

then $S^{'}_{i,j} = S^{'}_{i^{'},j^{'}}$.

Lemma 5 [21]: Given iterations $(i^{'}, j^{'})$ and $(i^{'}, k^{'})(j^{'} \neq k^{'})$ in the reduced iteration space, if there exist integers $r_1, r_2, ..., r_m$, not all zero, such that

$$k^{'} = j^{'} + \sum_{\gamma=1}^{m} r_\gamma L_{\gamma,2}$$

$$\sum_{\gamma=1}^{m} r_\gamma L_{\gamma,1} = 0$$

then $S^{'}_{i^{'},j^{'}} = S^{'}_{i^{'},k^{'}}$.

Lemma 6 [21]: Given iterations (i, j) and $(i^{'}, j^{'})$ in the reduced iteration space, if $(i^{'}, j^{'}) \in S^{'}_{i,j}$ then $S^{'}_{i,j} = S^{'}_{i^{'},j^{'}}$.

Theorem 5 [21]: Given staggering parameters $(L_{1,1}, L_{1,2}), (L_{2,1}, L_{2,2}), ..., (L_{m,1}, L_{m,2})$, we have

$$S_{i,j} = S^{'}_{i,j}$$

for any iteration (i, j) in the reduced iteration space.

Next, we establish that $S^{'}_{i,j}$ is the result of staggering with $(g, g^{'})$ followed by compacting with d. This is stated by Corollary 1 below.

Lemma 7 [21]: Given staggering parameters $(L_{i_1,1}, L_{i_1,2}), (L_{i_2,1}, L_{i_2,2}), ..., (L_{i_j,1}, L_{i_j,2})$, suppose that $r_1, r_2, ..., r_j$ are integers that satisfy
(1) $\sum_{\gamma=1}^{j} r_\gamma L_{i_\gamma,1} = 0 (r_1 > 0)$; and
(2) if there exist integers $r^{'}_1, r^{'}_2, ..., r^{'}_j$ satisfying $\sum_{\gamma=1}^{j} r^{'}_\gamma L_{i_\gamma,1} = 0 (r^{'}_1 > 0)$, then $r_1 \leq r^{'}_1$.
For any integers $r^{''}_1, r^{''}_2, ..., r^{''}_j$ satisfying

$$\sum_{\gamma=1}^{j} r^{''}_\gamma L_{i_\gamma,1} = 0 (r^{''}_1 > 0)$$

there exists an integer $k \geq 1$ such that $r^{''}_1 = k r_1$.

Theorem 6 [21]: If d is the compacting parameter determined by Algorithm 1, and $d^{'} = \sum_{\gamma=1}^{m} r_\gamma L_{\gamma,2}$, where $r_1, r_2, ..., r_m$ are integers, not all zero, which satisfy

$$\sum_{\gamma=1}^{m} r_\gamma L_{\gamma,1} = 0$$

then there exists an integer k such that $d^{'} = kd$.

Corollary 1: The set $S^{'}_{i,j}$ in Definition 7 satisfies

$$S^{'}_{i,j} = \{(i^{'}, j^{'}) | i^{'} = i + rg, j^{'} = j + rg^{'} + kd\}$$

where k and r are integers, $(g, g^{'})$ is the unified staggering parameter in Definition 6, and d is the compacting parameter computed by Algorithm 1.

From the above result, the threads in different columns after compacting the SRIS with d are independent. Next, we establish with Theorem 7 that any two columns which are d columns apart, where d is computed by Algorithm 1, should be dependent and that, therefore, d is the largest possible number of independent columns as the result of compacting the SRIS with a constant number.

Theorem 7: Given (i, j), we have $(i, j + d) \in S^{'}_{i,j}$.

Proof: According to how d is computed in Algorithm 1, there exist integers $r_1, r_2, ..., r_m$ such that $\sum_{i=1}^{m} r_i L_{i,2} = d$ and $\sum_{i=1}^{m} r_i L_{i,1} = 0$. By the definition of $S^{'}_{i,j}$, $(i, j + d) \in S^{'}_{i,j}$. □

To further simplify the process of the staggering and the compacting of the reduced iteration space, the following theorem can be used to replace multiple staggering parameters, which are in proportion, with a single staggering parameter.

Theorem 8 [21]: Given staggering parameters $(L_{1,1}, L_{1,2}), (L_{2,1}, L_{2,2}), ..., (L_{m,1}, L_{m,2})$, where $(L_{i_1,1}$, $L_{i_1,2})$, $(L_{i_2,1}$, $L_{i_2,2})$, ..., $(L_{i_p,1}$, $L_{i_p,2})(2 \leq p < m)$ are the staggering parameters satisfying

$$\frac{L_{i_1,1}}{L_{i_1,2}} = \frac{L_{i_2,1}}{L_{i_2,2}} = ... = \frac{L_{i_p,1}}{L_{i_p,2}},$$

there exists an integer r satisfying

$$\sum_{\gamma=1}^{p} r_{i_\gamma} L_{i_\gamma,1} = rl$$

$$\sum_{\gamma=1}^{p} r_{i_\gamma} L_{i_\gamma,2} = rl \frac{L_{i_1,2}}{L_{i_1,1}}$$

where $l = G.C.D.(L_{i_1,1}, L_{i_2,1}, ..., L_{i_p,1})$. Therefore $(l, l\frac{L_{i_1,2}}{L_{i_1,1}})$ can replace $(L_{i_1,1}$, $L_{i_1,2})$, $(L_{i_2,1}$, $L_{i_2,2})$, ..., $(L_{i_p,1}, L_{i_p,2})$.

We now estimate the time needed by the compiler to compute the staggering parameters, a unified staggering parameter, and the compacting parameter. Suppose there are m reference pairs. The complexity of determining all the staggering parameters is $O(m)$. A unified staggering parameter $(g, g^{'})$ of these staggering parameters can be determined in $O(m)$ with the Euclidean Algorithm. Let $m^{'}$ be the number of groups of staggering parameters such that all parameters in the same group are in proportion. $m^{'}$ is very small in practice. According to Theorem 8, we only need to consider one representative from each group. The complexity of Algorithm 1 and 2 for computing the compacting parameter is $C^2_{m^{'}} O(2) + C^3_{m^{'}} O(3) + ... + C^{m^{'}}_{m^{'}} O(m^{'}) = O(m^{'} 2^{m^{'}-1})$.

Lastly, we show the result of restructuring the original loop nest based on staggering and compacting. Note that if all staggering parameters are in proportion, then compacting is unnecessary for data dependence elimination. However, to improve load balance, we compact the SRIS by a compacting factor d, which equals the number of the available processors. The restructured code is parameterized by the loop bounds and by the number of available processors, which can be obtained by a system call at runtime. There is no need to recompile for a different number of available processors.

If the given loop nest is perfectly nested, then the resulting code, after staggering and compacting, is shown in Code Segment 1 listed below.

Code Segment 1 (The result after restructuring the perfectly nested loop with multiple linear functions):

```
DOALL II = 1, d
    DO I = 1, N_1
        m = ⌊(I-1)/g⌋
        J_0 = (II + (d - ((-mg') mod d)) - 1) mod d + 1
        DO J = J_0, N_2, d
            DO K = 1, N_3
                LOOP BODY {h_γ(I, J, K) | 1 ≤ γ ≤ m}
            ENDDO
        ENDDO
    ENDDO
ENDDOALL
```

If the given loop nest is not perfectly nested, then the resulting code has two variants, one for $g^{'} \neq 0$ and the other for $g^{'}=0$. We show the code for $g^{'} \neq 0$ in Code Segment 2, listed below (the code for $g^{'}=0$ is similar [21]). In this code segment, LB_1 and LB_2 are the lower bounds of I and J, UB_1 and UB_2 are the upper bounds of these two loops, $(g, g^{'})$ and d are the unified staggering parameter and the compacting parameter, respectively, which have been determined above. PSI and PSJ are local variables that each processor uses to determine the first iteration J_0 of loop J to be executed on it. Variables J_0 and $OFFSET$ are also local to each processor. PSI and PSJ for each processor are modified every g iterations of the loop I, according to the staggering and the compacting parameters. The values of (PSI, PSJ) are initialized for the d different processors to $(LB_1, 0), (LB_1, 1), ..., (LB_1, d-1)$, respectively. We define a function *mod** such that x *mod** $d = x \ mod \ d$ if $x \ mod \ d \geq 0$, otherwise x *mod** $d = x \ mod \ d + d$.

Code Segment 2 (The restructured code for the case of $g' \neq 0$):

```
DO I = LB_1, UB_1
    ......
    DOALL JJ = 1, d
        if I - PSI = g then
            PSI = I
            PSJ = (PSJ + g') mod* d
        endif
        OFFSET = PSJ - (LB_2 mod* d)
        if OFFSET < 0 then
            OFFSET = OFFSET + d
        endif
        J_0 = LB_2 + OFFSET
        DO J = J_0, UB_2, d
            ......
            DO K = LB_3, UB_3
                LOOP BODY {h_γ(I, J, K) | 1 ≤ γ ≤ m}
            ENDDO
            ......
        ENDDO
        ......
    ENDDOALL
```

```
......
ENDDO
```

4.2 An Extended Model

The theory we developed in the previous subsection can be extended to more general cases in which the subscript functions in the same pair of references are not necessary the same. Suppose the following two linear functions

$$\vec{h_1}(i,j,k) = (f_1(i,j,k), g_1(i,j,k)) = (a_{1,1}i + b_{1,1}j + c_{1,1}k + d_{1,1}, a_{1,2}i + b_{1,2}j + c_{1,2}k + d_{1,2})$$

and

$$\vec{h_2}(i,j,k) = (f_2(i,j,k), g_2(i,j,k)) = (a_{2,1}i + b_{2,1}j + c_{2,1}k + d_{2,1}, a_{2,2}i + b_{2,2}j + c_{2,2}k + d_{2,2})$$

belong to the same pair of references. In order to determine which iterations in the reduced iteration space are dependent due to this reference pair, we consider an affine transformation

$$i = \alpha_1 i' + \beta_1$$

$$j = \alpha_2 j' + \beta_2$$

$$k = \alpha_3 k' + \beta_3,$$

such that the linear function $\vec{h_2}$ can be expressed as

$$\begin{aligned}\vec{h_2}(i,j,k) &= (f_2(\alpha_1 i' + \beta_1, \alpha_2 j' + \beta_2, \alpha_3 k' + \beta_3), g_2(\alpha_1 i' + \beta_1, \alpha_2 j' + \beta_2, \alpha_3 k' + \beta_3)) \\ &= (a_{2,1}\alpha_1 i' + b_{2,1}\alpha_2 j' + c_{2,1}\alpha_3 k' + a_{2,1}\beta_1 + b_{2,1}\beta_2 + c_{2,1}\beta_3 + d_{2,1}, \\ &\quad a_{2,2}\alpha_1 i' + b_{2,2}\alpha_2 j' + c_{2,2}\alpha_3 k' + a_{2,2}\beta_1 + b_{2,2}\beta_2 + c_{2,2}\beta_3 + d_{2,2})\end{aligned}$$

which we denote by $\vec{h'_2}(i',j',k')$. In order to use the previous results from Section 4.1, we let $\vec{h'_2}(i',j',k')$ be identical to $\vec{h_1}(i',j',k')$, which implies

$$\frac{a_{1,1}}{a_{2,1}} = \frac{a_{1,2}}{a_{2,2}} = \alpha_1, \quad \frac{b_{1,1}}{b_{2,1}} = \frac{b_{1,2}}{b_{2,2}} = \alpha_2, \quad \frac{c_{1,1}}{c_{2,1}} = \frac{c_{1,2}}{c_{2,2}} = \alpha_3,$$

and

$$a_{2,1}\beta_1 + b_{2,1}\beta_2 + c_{2,1}\beta_3 + d_{2,1} = d_{1,1} \tag{1}$$

$$a_{2,2}\beta_1 + b_{2,2}\beta_2 + c_{2,2}\beta_3 + d_{2,2} = d_{1,2}. \tag{2}$$

We can now apply the algorithms in Section 4.1 to $\vec{h'_2}$ and $\vec{h_1}$, which yield a staggering parameter, say (L_1, L_2). For a given iteration (i',j'), let $(i'',j'') = (L_1, L_2) + (i',j')$. The iteration (i,j) must have a dependence with (i'',j'') before the affine transformation if and only if the iteration (i',j') has a dependence with (i'',j'') after the transformation. We denote the distance between (i,j) and (i'',j'') as (L'_1, L'_2), which can be calculated as:

$$L'_1 = i'' - i = (1-\alpha_1)i' + i'' - i' - \beta_1 = (1-\alpha_1)i' + L_1 - \beta_1$$

$$L'_2 = j'' - j = (1-\alpha_2)j' + j'' - j' - \beta_2 = (1-\alpha_2)j' + L_2 - \beta_2.$$

or

$$L'_1 = i - i'' = (\alpha_1 - 1)i' + i' - i'' + \beta_1 = (\alpha_1 - 1)i' + \beta_1 - L_1$$

$$L_2^{'} = j - j^{''} = (\alpha_2 - 1)j^{'} + j^{'} - j^{''} + \beta_2 = (\alpha_2 - 1)j^{'} + \beta_2 - L_2.$$

such that $L_1^{'} > 0$.

If $\alpha_1 \neq 1$ or $\alpha_2 \neq 1$, then $(L_1^{'}, L_2^{'})$ will not be constant, meaning that the iterations cannot be aligned with a constant staggering parameter. In common practice, since loop J is DOALL in our loop nest model, the two linear functions $\vec{h_1}$ and $\vec{h_2}$ will have the same cofficients for loop index variables I and J, which implies that $\alpha_1 = \alpha_2 = 1$. In this paper, we will consider the case of $\alpha_1 = \alpha_2 = 1$ only. We now have

$$L_1^{'} \overset{\alpha_1=1}{=} L_1 - \beta_1$$

$$L_2^{'} \overset{\alpha_2=1}{=} L_2 - \beta_2$$

or

$$L_1^{'} \overset{\alpha_1=1}{=} \beta_1 - L_1$$

$$L_2^{'} \overset{\alpha_2=1}{=} \beta_2 - L_2.$$

We define $(L_1^{'}, L_2^{'})$, which are two constants given (β_1, β_2), as a staggering parameter in this case.

If Equations (1) and (2) have a unique solution for (β_1, β_2), then we have a unique staggering parameter $(L_1^{'}, L_2^{'})$.

On the other hand, if there exist multiple solutions for (β_1, β_2), then the following theorem shows that under certain conditions, $(L_1^{'}, L_2^{'})$ determined by different (β_1, β_2) should be in proportion.

Theorem 9: Assume $\alpha_1 = \alpha_2 = 1$. If the staggering parameter (L_1, L_2) of the subscript function $\vec{h_1}$ after the affine transformation is a solution for (β_1, β_2) in Equations (1) and (2), then $(L_1^{'}, L_2^{'})$, which is equal to $(\beta_1 - L_1, \beta_2 - L_2)$ if $\beta_1 - L_1 > 0$, or to $(L_1 - \beta_1, L_2 - \beta_2)$ if $L_1 - \beta_1 > 0$, is in proportion with (L_1, L_2), for any solution $(\beta_1, \beta_2, \beta_3)$ of Equations (1) and (2).

Proof: Since $(L_1 - \beta_1, L_2 - \beta_2)$ is in proportion with $(\beta_1 - L_1, \beta_2 - L_2)$, we only prove that $(\beta_1 - L_1, \beta_2 - L_2)$ is in proportion to (L_1, L_2), supposing that $\beta_1 - L_1 > 0$. Every solution to Equations (1) and (2) can be written as

$$\begin{pmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \end{pmatrix} = \begin{pmatrix} \xi_1 \\ \xi_2 \\ \xi_3 \end{pmatrix} + \begin{pmatrix} L_1 \\ L_2 \\ \Delta k \end{pmatrix}$$

where (ξ_1, ξ_2, ξ_3) is a solution of the homogeneous system associated with Equations (1) and (2), that is,

$$a_{2,1}\xi_1 + b_{2,1}\xi_2 + c_{2,1}\xi_3 = 0$$

$$a_{2,2}\xi_1 + b_{2,2}\xi_2 + c_{2,2}\xi_3 = 0.$$

So, if $a_{1,2}b_{1,1} - b_{1,2}a_{1,1} \neq 0$, we have

$$\xi_1 = \frac{b_{2,2}c_{2,1} - b_{2,1}c_{2,2}}{a_{2,2}b_{2,1} - b_{2,2}a_{2,1}}\xi_3 = \frac{1}{\alpha_3}\frac{b_{1,2}c_{1,1} - b_{1,1}c_{1,2}}{a_{1,2}b_{1,1} - b_{1,2}a_{1,1}}\xi_3$$

$$\xi_2 = \frac{c_{2,2}a_{2,1} - c_{2,1}a_{2,2}}{a_{2,2}b_{2,1} - b_{2,2}a_{2,1}}\xi_3 = \frac{1}{\alpha_3}\frac{c_{1,2}a_{1,1} - c_{1,1}a_{1,2}}{a_{1,2}b_{1,1} - b_{1,2}a_{1,1}}\xi_3.$$

Suppose $L_2 = L_1 = 0$. We have $c_{1,2}a_{1,1} - c_{1,1}a_{1,2} = 0$ and $b_{1,2}c_{1,1} - b_{1,1}c_{1,2} = 0$. Therefore $\beta_2 - L_2 = \xi_2 = 0$ and $\beta_1 - L_1 = \xi_1 = 0$. If $L_2 = 0$ and $L_1 \neq 0$, then we have $c_{1,2}a_{1,1} - c_{1,1}a_{1,2} = 0$, and hence $\beta_2 - L_2 = \xi_2 = 0$. If $L_2 \neq 0$, according to Definition 4, we have

$$\frac{\beta_1 - L_1}{\beta_2 - L_2} = \frac{\xi_1}{\xi_2} = \frac{b_{1,2}c_{1,1} - b_{1,1}c_{1,2}}{c_{1,2}a_{1,1} - c_{1,1}a_{1,2}} = \frac{L_1}{L_2}.$$

For the case of $a_{1,2}b_{1,1} - b_{1,2}a_{1,1} = 0$, as in Theorem 2, we have: (1) $a_{1,1}\xi_1 + b_{1,1}\xi_2 = 0$ for $a_{1,1} \neq 0$; (2) $a_{1,2}\xi_1 + b_{1,2}\xi_2 = 0$ for $a_{1,2} \neq 0$; (3) $\xi_2 = 0$ for $a_{1,1} = a_{1,2} = 0$. Therefore, according to Definition 4, we also have (ξ_1, ξ_2) in proportion with (L_1, L_2). So $(\beta_1 - L_1, \beta_2 - L_2)$ and $(L_1 - \beta_1, L_2 - \beta_2)$ are in proportion with (L_1, L_2). Therefore $(L_1^{'}, L_2^{'})$ is in proportion with (L_1, L_2). $\square$

If the condition in Theorem 9 is met, we choose $(L_1^{'}, L_2^{'}) = (\frac{L_1}{GCD(L_1,L_2)}, \frac{L_2}{GCD(L_1,L_2)})$ as the staggering parameter for the reference pair $\vec{h_1}$ and $\vec{h_2}$.

Table 2 shows examples of staggering parameters for different subscript functions appearing in the dependent reference pair, where the loop index variables are listed in the order from the outermost loop level to the innermost. If we simultaneously consider two reference pairs: $A(I, J)$ with $A(I-3, J-1)$, and $B(I, J)$ with $B(I-1, J-3)$, then the thread $T_{i,j}$ will share the same array element $A(i, j)$ with thread $T_{i+3,j+1}$ and the same array element $B(i, j)$ with thread $T_{i+1,j+3}$. Using Theorem 9, the staggering parameters $(L_1^{'}, L_2^{'})$ for these two pairs are (3,1) and (1,3) respectively. A unified staggering parameter and compacting parameter can be calculated as $(g, g^{'}) = (1, 3)$ and $d = 8$.

Table 2: Examples of different functions in the same dependent reference pair.

Loop nest	Dependent reference pair $(\vec{h_1}, \vec{h_2})$	(β_1, β_2)	(L_1, L_2)	$(L_1^{'}, L_2^{'})$
(I, J, K)	$A(I, J), A(I-3, J-1)$	(3,1)	(0,0)	(3,1)
(I, J, K)	$B(I, J), B(I-1, J-3)$	(1,3)	(0,0)	(1,3)
(I, J, K)	$A(J+K, I+J), A(J+K+c, I+J)$	$(d, -d)$, $d \in I$	(1,-1)	(1,-1)

5 Experimental Results

The thread alignment techniques described in this paper have been implemented as backend optimizations in KD-PARPRO [20], a knowledge-based parallelizing tool which can perform intra- and inter-procedural data dependence analysis and a large number of parallelizing transformations, including loop interchange, loop distribution, loop skewing, and strip mining for FORTRAN programs.

To evaluate the effect of the thread alignment techniques on the performance of multiprocessors with memory hierarchies, we experimented with three programs from the LINPACK benchmarks on a SGI Challenge cluster which can be configured to contain up to twenty MIPS 4400 processors. First, the programs were parallelized and optimized using KD-PARPRO. To reduce or eliminate cache thrashing due to true data sharing, our tool recognized the nested loops which may cause the thrashing. It applied the techniques described in the previous section to analyze and restructure the loop nests. The parallelized programs were then compiled using SGI's f77 compiler with the optimization option -O2. The sequential versions of the programs were compiled on the same machine using the same optimization option for f77. The output binary codes were then executed on various configurations with a different number of processors during dedicated time. Each MIPS 4400 processor has a 16K-byte primary data cache and a 4M-byte secondary cache. The cache block size is 32 bytes for the primary data cache and 128 bytes for the secondary cache. A fast and wide split transaction bus POWERpath-2 is used as its coherent interconnect. Cache coherence is maintained with a snoopy write-invalidate strategy.

We compared the results obtained by using our algorithm to align the threads with those obtained by using four different loop scheduling strategies provided by SGI system software, namely, *simple*, *interleave*, *dynamic*, and *gss*. The *simple* method divides the iterations by the number of processors and then assigns each chunk of consecutive iterations to one processor. The *interleave* scheduling method divides the iterations into chunks of the size specified by the CHUNK option, and execution of those chunks is statically interleaved among the processes. With *dynamic* scheduling, the iterations are also divided into CHUNK-sized chunks. As each process finishes a chunk, however, it enters a critical section to grab the next available chunk. With *gss* scheduling [29] , the chunk size is varied, depending on the number of iterations remaining. None of these SGI-provided methods consider task alignment.

The speedup of a parallel execution on shared memory machines just like the SGI cluster can be affected by many factors, including: program parallelism, data locality, scheduling overhead, and load balance. Usually *gss*, *dynamic* and *interleave* schedulings with a small chunk size are supposed to show better load balance than *simple* scheduling. On the other hand, they tend to incur more scheduling overhead than *simple*. Furthermore, *simple* captures more data locality in most cases than other schedulings do.

The programs we selected from LINPACK are SGEFA, SPODI, and SSIFA. SGEFA factors a double precision matrix by Gaussian elimination. The main loop structure in this program consists of three imperfectly nested loops. The innermost loop is inside subroutine SAXPY, which multiplies a vector by a constant and then adds the result to another vector. In order to show the array access pattern inside the loop body, we inlined the SAXPY in the code section given below. However, we kept the subroutine call when we applied our techniques to the program.

```
DO K = 1, n-1
    .....
    DOALL J = K+1, n
        ......
        DO I = K+1, n
            ......
            A(I,J) = A(I,J) + T * A(I,K)
            ......
        ENDDO
        ......
    ENDDOALL
    ......
ENDDO
```

The SRIS, after staggering the iterations in the reduced iteration space (K, J), is shown in Figure 9. For this program, we only consider the linear function (I, J). The staggering parameter is (1,0) according to Definition 4(4). The number of processors is used to determine the compacting factor.

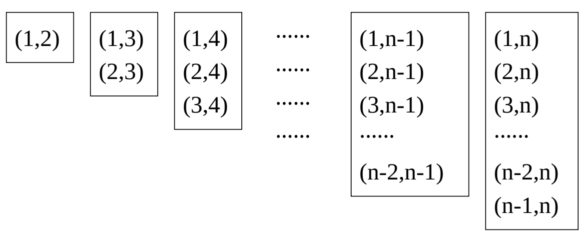


Figure 9: SRIS for SGEFA.

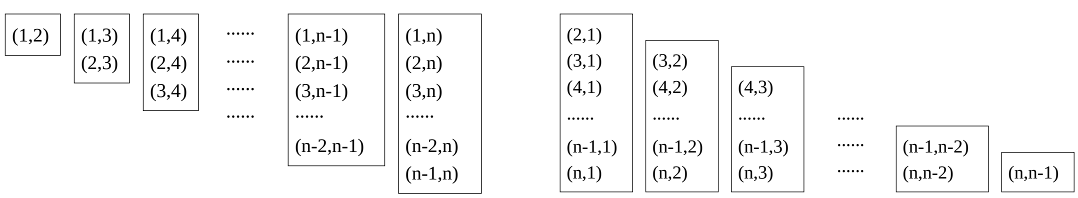
SPODI computes the determinant and inverse of a certain double precision symmetric positive-definite matrix. There are two main loop nests in this program, as shown below. We restructured both loop nests. The same as in SGEFA, the innermost loop is contained in the subroutine SAXPY.

```
/* the first loop nest */
DO K = 1, n
    .....
    DOALL J = K+1, n
        ......
        DO I = 1, K
            ......
            A(I,J) = A(I,J) + T * A(I,K)
```

```
            ......
        ENDDO
        ......
    ENDDOALL
    ......
ENDDO

/* the second loop nest */
DO J = 1, n
    .....
    DOALL K = 1, J - 1
        ......
        DO I = 1, K
            ......
            A(I, K) = A(I, K) + T * A(I, J)
            ......
        ENDDO
        ......
    ENDDOALL
    ......
ENDDO
```

The SRISs, after staggering the iterations in the reduced iteration spaces (K, J) and (J, K) for these two loop nests, respectively, are shown in Figure 10(a)(b). The linear functions we considered are: (I, J) and (I, K), respectively. Their staggering parameters are both (1,0), according to Definition 4(4).



(a) Loop nest 1 (b) Loop nest 2

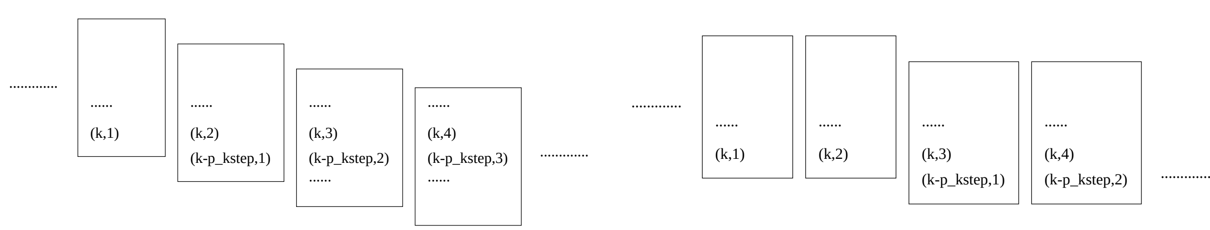
Figure 10: SRISs for SPODI.

SSIFA factors a double precision symmetric matrix by elimination. The main loop nest in this program is shown below. We view the backward GOTO loop as the outermost sequential loop, within which the value of $kstep$ may change between 1 and 2 in different iterations, based on the input matrix. Depending on the value of $kstep$, one of the two parallel loop nests inside the outermost sequential loop will be executed for each iteration of the outermost loop. The index step of the outermost loop equals $-kstep$, i.e. -1 or -2. The array access patterns for these two $kstep$ values are slightly different. The innermost loop is again inside the SAXPY subroutine.

```
    K = n
10: CONTINUE
        IF (K .EQ. 0) GOTO 20
        .....
        IF kstep = 1
            ......
            DOALL JJ = 1, K - 1
```

```
            ......
            DO I = 1, K − JJ
                ......
                A(I, K − JJ) = A(I, K − JJ) + T ∗ A(I, K)
                ......
            ENDDO
            ......
        ENDDOALL
        ......
    ELSE
        ......
        DOALL JJ = 1, K − 2
            ......
            DO I = 1, K − JJ − 1
                ......
                A(I, K − JJ − 1) = A(I, K − JJ − 1) + T ∗ A(I, K)
                ......
            ENDDO
            ......
            DO I = 1, K − JJ − 1
                ......
                A(I, K − JJ − 1) = A(I, K − JJ − 1) + T ∗ A(I, K − 1)
                ......
            ENDDO
            ......
        ENDDOALL
        ......
    ENDIF
    K = K − kstep
  GOTO 10
20: CONTINUE
```

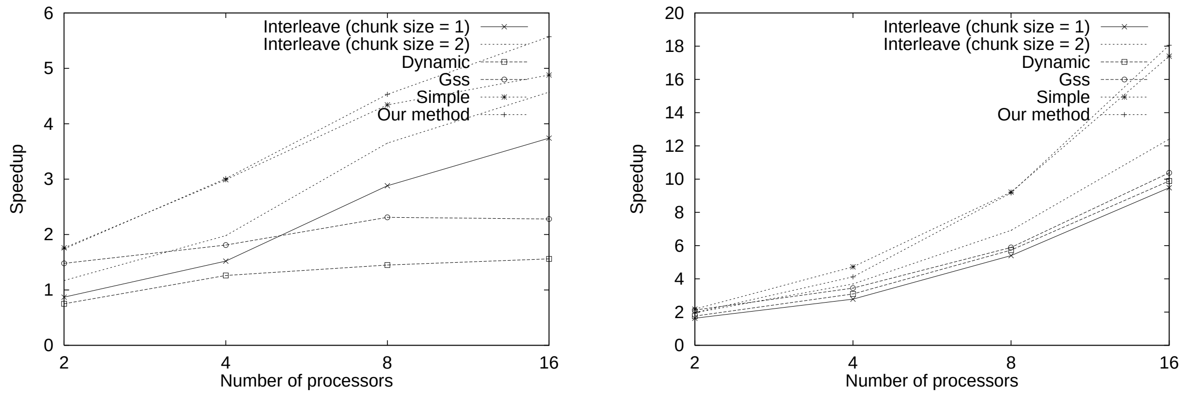
The SRISs, after staggering the iterations in the reduced iteration space (K, JJ) for two different $kstep$s, are shown in Figure 11. For clarity, we use c_kstep to denote the value of $kstep$ in the current K iteration, and we use p_kstep for its value in the previous K iteration. All threads will be aligned well if we properly align the threads created in the current K iteration with those in the previous K iteration. We need to consider four possible cases of dependences: one is between two references to $A(I, K - JJ)$; another is between two references to $A(I, K - JJ - 1)$; the third is from $A(I, K - JJ)$ to $A(I, K - JJ - 1)$; and the last is from $A(I, K - JJ - 1)$ to $A(I, K - JJ)$. For all of these cases, the staggering parameter is $(L_1, L_2) = (p\_kstep, -c\_kstep)$.



(a) c_kstep = 1 (b) c_kstep = 2

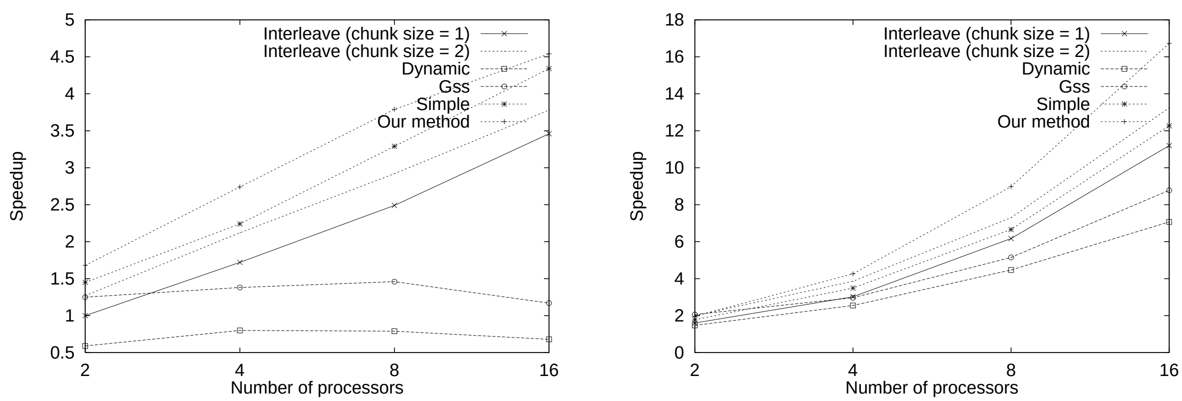
Figure 11: SRISs for SSIFA.

The problem sizes we used in our experiments are n=100 and 1000. The performance of the parallel codes transformed by our techniques, compared with the performance achieved by the scheduling methods provided by SGI, are shown in Figures 12-15. Both SGEFA and SSIFA may require pivoting for non-positive-definite symmetric matrices, but not for positive-definite symmetric matrices. We show data for SGEFA with pivoting, and we show data both with and without pivoting for SSIFA. Pivoting may potentially destroy the task alignment.



(a) n=100 (b) n=1000

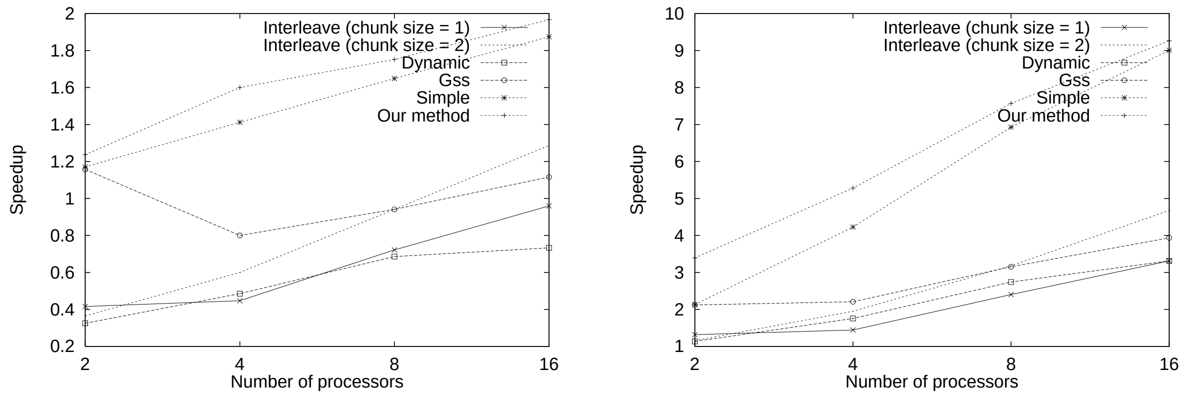
Figure 12: Gaussian elimination (SGEFA).



(a) n=100 (b) n=1000

Figure 13: Determinant and inverse of a symmetric positive matrix (SPODI).

As the figures show, our method always outperforms all of the SGI's scheduling methods, with the exception of program SGEFA. For this program, our method's performance is almost the same as that of *simple*, although our method outperforms *simple* by 14% on 16 processors with $n = 100$, which should be attributed to the reduction of cache thrashing due to true data sharing, a problem that tends to be more severe when more processors are running. The *simple* scheduling method tends to get better performance than the *dynamic*, *gss*, and *interleave* methods, because it results in better locality and less cache thrashing in most cases, and it also incurs less scheduling overhead. But when the programs



(a) n=100 (b) n=1000

Figure 14: Factorization of a symmetric matrix (SSIFA).

do not exhibit a good load balance, like SPODI, SSIFA, and other programs in LINPACK, which deal with symmetric matrices, *simple*'s performance results degrade substantially. Our method outperforms *simple* quite significantly in most cases, especially for SPODI (Figure 13), as well as for SSIFA without pivoting (Figure 15), where our method beats *simple* by as much as 105%. We are not able to get improvement over *simple* for program SGEFA when $n = 1000$, because pivoting is much more likely to destroy the locality we try to keep. For the rest of the programs, we attribute our performance gain over *simple* both to the reduction of cache thrashing due to true data sharing and to a better load balance, although, for SSIFA with pivoting, we believe our method benefits more from load balancing. We note that the SGI system software cannot pick the right scheduling method automatically to fit the particular program. On the other hand, our method seems more capable of delivering good performance for different loop shapes. As the thrashing problem becomes more serious on parallel systems with more processors and greater communication overhead, our method will likely be even more effective.

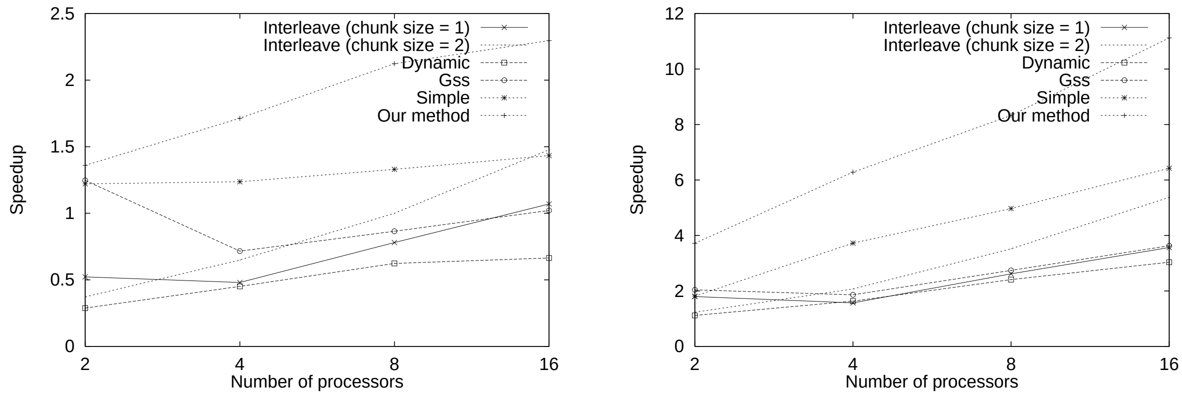
6 Conclusions

This paper presents a method in which the reduced iteration space is rearranged according to the staggering and the compacting parameters. The nested loop (either perfectly nested or imperfectly nested) is restructured to reduce or even eliminate cache thrashing due to true data sharing. This method can be efficiently implemented in any parallel compiler. Although the analysis per se is based on a simple machine model, the resulting code executes correctly on more complex models. Our experimental results show that the transformed code can perform quite well on a real machine. How to extend the techniques proposed in this paper to incorporate additional machine parameters is interesting future work.

References

[1] S. Abraham and D. Hudak. Compile-time partitioning of iterative parallel loops to reduce cache coherence traffic. *IEEE Transactions on Parallel Distributed Systems*, 2(3), July 1991.

[2] W. Abu-Sufah, D. Kuck, and D. Lawrie. On the performance enhancement of paging systems through program analysis and transformations. *IEEE Transactions on Computers*, C-30(5), May 1981.



(a) n=100

(b) n=1000

Figure 15: Factorization of a symmetric positive-definite matrix (SSIFA).

[3] A. Agarwal, D. Kranz, and V. Natarajan. Automatic partitioning of parallel loops and data arrays for distributed shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, Vol.6, No.9, pp.943-962, September 1995.

[4] J. R. Allen and K. Kennedy. Automatic loop interchange. In *Proceedings of the SIGPLAN'84 Symposium on Compiler Construction*, Montreal, Canada, June 1984.

[5] J. Baer and W.Wang. Multilevel cache hierarchies: organizations, protocols, and performance. *Journal of Parallel and Distributed Computing*, Vol.6, pp.451-476, 1989.

[6] U. Banerjee. *Dependence analysis for supercomputing*. Kluwer Academic Publishers, 1988.

[7] D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Languages Design and Implementation*, June 1990.

[8] S. Carr and K. Kennedy. Compiling scientific code for complex memory hierarchies. In *Proceedings of Hawaii International Conference on System Sciences*, pp.536-544, 1991.

[9] E. D'Hollander. Partitioning and labeling of loops by unimodular transformations. *IEEE Transactions on Parallel and Distributed Systems*, 3(4), July 1992.

[10] S. J. Eggers and T. E. Jeremiassen. Eliminating False Sharing. In *Proceedings of 1991 International Conference on Parallel Processing*, Aug. 1991.

[11] J. Fang and M. Lu. A solution of cache ping-pong problem in RISC based parallel processing systems. In *Proceedings of 1991 International Conference on Parallel Processing*, Aug. 1991.

[12] Z. Fang. Cache or local memory thrashing and compiler strategy in parallel processing systems. In *Proceedings of 1990 International Conference on Parallel Processing*, pp.271-275, Aug. 1990.

[13] J. Fang and M. Lu. An iteration partition approach for cache or local memory thrashing on parallel processing. *IEEE Transactions on Computers*, C-42, May 1993.

[14] M. Galles and E. Williams. Performance optimizations, implementation, and verification of the SGI Challenge multiprocessor. In *Proceedings of the 27th Annual Hawaii International Conference on System Sciences*, 1994.

[15] K. Gallivan, W. Jalby, and D. Gannon. On the problem of optimizing data transfers for complex memory systems. In *Proceedings of Supercomputing'88*, pp.238-253, 1988.

[16] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformation. *Journal of Parallel and Distributed Computing*, Vol.5, 1988.

[17] M. Gupta and D. Padua. Effects of program parallelization and stripmining transformation on cache performance in a multiprocessor. In *Proceedings of the 1991 International Conference on Parallel Processing*, Aug. 1991.

[18] D. Hudak and S. Abraham. Compiler techniques for data partitioning of sequentially iterated parallel loops. In *Proceedings of ACM International Conference on Supercomputing*, pp. 187-200, 1990.

[19] T. E. Jeremiassen and S. J. Eggers. Reducing false sharing on shared memory multiprocessors through compile-time data transformations. In *Proceedings of Fifth ACM SIGPLAN Symposium on Principals and Practice of Paralle Programming*, pp. 179-188, 1995.

[20] G. Jin and F. Chen. The design and the implementation of a knowledge-based parallelizing tool. In *Proceedings of the 2nd IES Information Technology Conference*, July 1991, Singapore.

[21] G. Jin, Z. Li, and F. Chen. An efficient solution to the cache thrashing problem (Extended Version). Technical Report TR 96-020, Department of Computer Science, University of Minnesota, 1996.

[22] G. Jin and F. Chen. Loop restructuring techniques for the thrashing problem. In *Proceedings of the 1992 International Conference on Parallel Architectures and Languages Europe*, 1992.

[23] G. Jin, X. Yang, and F. Chen. Loop staggering, loop staggering and loop compacting: restructuring techniques for the thrashing problem. In *Proceedings of 1991 International Conference on Parallel Processing*, Aug. 1991.

[24] D. Kuck. *The structure of computers and computations.* Vol.1, John Wiley and Sons, New York, 1978.

[25] D. Kuck, R. Kuhn, D. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the 8th ACM Symposium on Principle of Programming Languages (POPL)*, 1981.

[26] M. Lu and J. Fang. A solution of the cache ping-pong problem in multiprocessor systems. *Journal of Parallel and Distributed Computing* 16, Oct. 1992.

[27] I. Nivan, et al. *An introduction to the theory of numbers.* John Wiley & Sons, Fourth Edition New York, Chichester Brisbane Toronto, 1980.

[28] J. Peir and R. Cytron. Minimum distance: a method for partitioning recurrences for multiprocessors. *IEEE Transactions on Computers*, C-38, Aug. 1989.

[29] C. D. Polychronopoulos and D. Kuck. Guided self-scheduling: a practical scheduling scheme for parallel supercomputers. *IEEE Transactions on Computers*, C-36, Dec. 1987.

[30] W. Shang and J. Fortes. Time optimal linear schedules for algorithms with uniform dependencies. *IEEE Transactions on Computers*, C-40, June 1991.

[31] Z. Shen, Z. Li, and P. -C. Yew. An empirical study of Fortran programs for parallelizing compilers. *IEEE Transactions on Parallel and Distributed Systems*, 1(3), pp. 356-364, July 1990.

[32] K. Tomko and S. Abraham. Iteration partitioning for resolving stride conflicts on cache-coherent multiprocessors. In *Proceedings of 1993 International Conference on Parallel Processing*, Aug. 1993.

[33] J. Torrellas, M. S. Lam, and J. L. Hennessy. False sharing and spatial locality in multiprocessor caches. *IEEE Transactions on Computers*, C-43, June 1994.

[34] M. Wolf and M. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN'91 Conference on Program Language Design and Implementation*, June 1991.

[35] M. Wolfe. More iteration space tiling. In *Proceedings of Supercomputing'89*, 1989.