An Iteration Partition Approach for Cache or Local Memory Thrashing on Parallel Processing

J. Fang
Hewlett-Packard Laboratories
M. Lu
Texas A&M University

Abstract
Parallel processing systems with cache or local memory in the memory hierarchies have become very common. These systems have large-size cache or local memory in each processor and usually employ copy-back protocol for the cache coherence. In such systems, a problem called "cache or local memory thrashing" may arise in executions of parallel programs, when the data unnecessarily moves back and forth between the caches or local memories in different processors. The techniques associated with parallel compilers to solve the problem are not completely developed.

In this paper we present an approach to eliminate unnecessary data moving between the caches or local memories for nested parallel loops. This approach is based on relations between array element accesses and enclosed loop indexes in the nested parallel loops. The relations can be used to assign processors to execute the appropriate iterations for parallel loops in the loop nests with respect to the data in their caches or local memories. An algorithm to calculate the correct iteration of the parallel loop in terms of loop indexes of the previous iterations executed in the processor is presented in the paper, even though there is more than one subscript expression of the same array variable in the loop.

This method benefits parallel code with nested loop constructs in a wide range of applications, in which the array elements are repeatedly referenced in the parallel loops. The experimental results show that the technique is extremely effective—capable of achieving double speedups over application programs such as Linpack benchmarks.

1. Introduction
In recent years, shared memory parallel processing systems with complicated memory hierarchies have become very common. For instance, cache is usually introduced as a means to bridge the gap between fast processor cycles and slow main memory access time in hardware design. Most parallel processing systems have local cache in each processor in the memory hierarchies, and some of them use more than one level of cache to enhance the cache bandwidth. In general, a copy-back cache protocol is employed to maintain cache coherence in these parallel processing systems, and the size of the cache memory becomes larger and larger. Another example in
some supercomputing systems is local memory (or programmable cache), which stores copies of frequently used data and local variables. In order to avoid hot spot contention in the interconnection network, some supercomputer systems include secondary local memory in the memory hierarchy for a small cluster of processors. In such multiprocessor systems, the memory access time from a processor to its own cache or local memory is much faster than the time either to the global memory or to the caches or local memories in other processors. When executing parallel code, the frequently used data may be shared by multiple processors, which may run the multiple threads for a parallel program at the same time. The local cache may result in severe inefficiencies when a parallel code requires data moving back and forth between processors. This phenomena is called "cache or local memory thrashing" in shared memory parallel processing systems. The cache or local memory thrashing degrades the system performance.

Although great efforts have been devoted to developing compilers to take advantage of parallel processing systems, the techniques associated with parallel compiler to solve the "cache or local memory thrashing" problem are not completely developed yet. The parallel compiler concepts underlying Illinois PARAFAKE compiler [11] and Rice Parallel Fortran Converter [2] are based on the shared memory multiprocessor architecture, in which a large memory block is assumed to be directly addressable by all processors, and the memory access time from different processors is assumed to be the same [13].

Most of research work, which attempts to enhance the cache hit ratios, focuses on improving of the data localities by restructuring the original program at compiler time. These research results can be used for uniprocessor and multiprocessor systems. Early researchers in this area studied similar phenomena in virtual memory systems. In [1] W. Abu-Sufah, D. Kuck, and D. Lawrie presented a few source program transformation techniques that improve the paging behavior of programs. These transformations, called "loop-blocking", consist of breaking iterative loops into smaller loops (strip-mining) and then recombing and reindexing these smaller loops (loop-fusing and loop-interchange). Following the spirit of [1], a lot of different loop-blocking algorithms have been developed for different computer architectures. These modified blocking algorithms, such as "loop-tiling" [14] and "loop-jam" [6], are able to take advantage of cache or local memory, because there is a high degree of data reuse during computing within a block. But for most of the application code with complicated program constructs, the benefit of the blocking algorithms is very limited.

In this paper we present an approach to eliminate, at least to reduce, the "cache or local memory thrashing" problem. The technique benefits parallel programs with complicated parallel constructs in a wide range of applications, in which parallel loops are enclosed by a serial loop and the array elements are frequently reused in the parallel loops in different iterations of the enclosed serial loop. The technique calculates the appropriate parallel loop indexes for each processor in terms of the data stored in its cache or local memory, then reduces unnecessary data moving between caches or local memories in the systems.

To compute the appropriate loop index, a mathematical concept is introduced in this paper to define the relations between the array element accesses and the enclosed loop indexes in nested parallel constructs. The relations determined by the array subscript expressions can be used to partition an iteration space into equivalence classes. All vectors of the iteration space in an equivalence class may access some common array elements. The concept helps to develop a method to find the next vector in an equivalence class in terms of the previous vectors in the same equivalence class.
The rest of the paper is organized as follows. In section 2, the cache or local memory thrashing problem on our simple machine model is introduced. In section 3, the programming model that we focus on in the paper is discussed. In section 4, the results in a simple case, which has only single array subscript expression, are described first. Then these results are extended to the more complicated case with multiple subscript expressions. In section 5, the application of the results to parallel compiler is presented. The experimental results are shown that this technique is extremely effective for some application programs in section 6. In section 7, we conclude our discussion.

2. Background

2.1 Machine Model

A simple shared memory parallel processing system model is comprised of a number of processors and a global memory, which are connected by data-bus, crossbar or interconnection network. The system provides a set of synchronization primitives to support concurrent execution of multiple threads in parallel program constructs. These synchronization primitives can be fetch/increment or semaphore instructions.

To match fast processor speed, the system has caches or local memories incorporated in its memory hierarchy. To simplify our presentation, we ignore other complicated considerations in hardware cache design[3]. The cache design in the simple shared memory parallel processing system model has the following characteristics:

1. It is local to a processor.
2. Its size is large enough.
3. It uses copy-back coherence strategy.
4. Its line size is one word.
5. It has only one level.

A more complicated machine model can be viewed as an extension of the simple model with more levels of local memories in the memory hierarchy, such as the CEDAR supercomputer[10]. The approach presented in this paper can be extended to the complicated machine model by analyzing multi-level parallel loops in nested parallel constructs.

2.2 Cache or Local Memory Thrashing Problem

In a parallel program, the execution of a piece of code specified by parallel constructs is called a thread [12]. A thread can be viewed as a unit of work that is programmer-defined or parallel-compiler-specified in parallel program constructs. Parallel-loop is the most common parallel construct, which can be viewed as a straightforward parallel version of the conventional DO-loop: a thread 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. The order in which the iterations of the loop are performed is arbitrary. For example, consider the following program:

```
DIMENSION B(200), A(1000, 1000)
DO I = 1, 100
    B(I) = A(I,I) - B(I-1)
    X = 1.0
    PDO J = 1, 100
    X = A(I,J) * B(I)
    DO K = 1, 100
```

There are one hundred threads spawned by the parallel loop in the example if each iteration of the loop is a thread. In general, each thread in a parallel loop is determined by the body of the loop and the indexes of its enclosed loops, which can be either serial or parallel.

When programs with nested parallel loops are executed on multiprocessor systems, some frequently used data may be repeatedly used and modified by different threads for the nested loops. If the threads, which may access the same data, are not assigned to the same processor, the 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 parallel processing systems.

In the above example the statement S doesn't have data dependence in the DO J loop by increment-Banerjee test [4]. If there is not other loop-carried dependence between statements of the loop body, the J loop can be parallelized. There are a total of 10,000 threads T_{I,J} in the execution of the parallel loop. Each thread requests 100 elements of array A. Many of the array elements are repeatedly referenced in these threads.

For instance, thread T_{1,1} requests data
\[ A(8,5), A(13,8), A(18,11), A(23,14), ..., A(498,299), A(503,302) \]
for the innermost serial loop index K from 1, ..., 100 respectively.

Meanwhile, thread T_{2,3} requests data
\[ A(13,8), A(18,11), A(23,14), ..., A(498,299), A(503,302), A(508,305) \]
and thread T_{3,5} requests data
\[ A(18,11), A(23,14), ..., A(503,302), A(508,305), A(513,308). \]

It can be observed that there exists a set of threads: T_{1,1}, T_{2,3}, T_{3,5}, T_{4,7}, ..., T_{49,99}, which reuse most of the array elements referenced in the previous thread. If the threads of the set are assigned to different processors, the data of array A are unnecessarily moved back and forth between caches or local memories in the system.

3. Programming Model For Cache or Local Memory Thrashing Solution

3.1 Preliminaries

In this section, the preliminary concepts relevant to the iteration space and data dependence analysis are reviewed and the notations used in this paper are introduced. Standard definitions are used to analyze the array accesses [2, 13, 6]. Considering a nested parallel construct of k loops with an array A of dimension 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 index in loop j, [ \( L_j : U_j \) ], where \( L_j \) and \( U_j \) are the low bound and up bound in the loop j respectively. The domain space denoted as \( D \) is defined by the product \( \prod_{i=1}^{d} M_i \), where \( M_i \) is the size of array A in the i-th dimension. Any array subscript expressions in statements of a parallel nested loop can be more precisely defined by \( h, g: C \rightarrow D \).

There exists a total order in the iteration space \( C \) that is defined by the point in time at which the element is executed. We say that a vector \( t \) is greater than a vector \( s \), where
\[ t = (t_1, t_2, \ldots, t_k) \] and
\[ s = (s_1, s_2, \ldots, s_k), \]
if there is a point \( m \), which is from 1 to \( k \), such that \( t_i = s_i \) for \( i < m \) and \( t_m > s_m \).

The standard data dependence definition in [11, 13, 5] is given as follows. If two statements access the same memory location, we say that there is a data dependence between them. A flow dependence from a statement \( S_1 \) to a statement \( S_2 \) exists if a variable is computed in \( S_1 \) and is later referenced and used in \( S_2 \). An antidependence from \( S_1 \) to \( S_2 \) exists if a variable is referenced by \( S_1 \) before it is rewritten by \( S_2 \). An output dependence or input dependence from \( S_1 \) to \( S_2 \) exists if both statements write or read the same memory location respectively.

3.2 Study of Program Structures in Applications
We have studied a broad range of application programs including well-known Linpack and Perfect benchmarks. A profile software tool, which is able to obtain statistic information at execution time such as the percentage of runtime for each subroutine and the percentage of each loop in a subroutine for a benchmark, was very helpful in our study. A parallel compiler with interprocedural analysis helps us also to explore program constructs containing subroutine calls. In the study, we assume that only one-level loops are parallel or there are multi-level parallel loops but only one-level loops are parallelized in the parallel loop nests. It is reasonable to match our simple machine model in section 2, which doesn’t introduce the hardware processor cluster concept.

In our study, the nested loop constructs may contain subroutine calls in the loop body. The subroutine may have loop nests containing other subroutine calls. Since both Linpack and Perfect benchmarks are written in standard Fortran, there are no recursive calls in our benchmarks. Table 1 lists the number of loop nests and the most time-consuming loop nests in 11 perfect benchmarks.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Number of Loop Nests</th>
<th>Time-Consuming Loop Nests</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total</td>
<td>Parallel</td>
</tr>
<tr>
<td>ADM (APS.f)</td>
<td>186</td>
<td>146</td>
</tr>
<tr>
<td>ARC2d (SRS.f)</td>
<td>156</td>
<td>141</td>
</tr>
<tr>
<td>BDNA (NAS.f)</td>
<td>183</td>
<td>103</td>
</tr>
<tr>
<td>DYFESM (SDS.f)</td>
<td>156</td>
<td>91</td>
</tr>
<tr>
<td>FLO52 (TFS.f)</td>
<td>115</td>
<td>96</td>
</tr>
<tr>
<td>MDG (LWS.f)</td>
<td>36</td>
<td>30</td>
</tr>
<tr>
<td>OCEAN (OCS.f)</td>
<td>89</td>
<td>65</td>
</tr>
<tr>
<td>QCD (LGS.f)</td>
<td>124</td>
<td>88</td>
</tr>
<tr>
<td>SPICE (CSS.f)</td>
<td>341</td>
<td>41</td>
</tr>
<tr>
<td>TRACKER (MTS.f)</td>
<td>65</td>
<td>47</td>
</tr>
<tr>
<td>TRFD (TIS.f)</td>
<td>32</td>
<td>29</td>
</tr>
</tbody>
</table>

Table 1: Nested Loops in Perfect Benchmarks

In the study, we focus on the most time-consuming nested loops, each of which requires at least 3% of the execution time. By using parallel/vectorized compiler with interprocedural analysis, the parallel loops are interchanged to the outer levels even cross the subroutine call boundary, if it is possible. The serial loop, which immediately encloses the parallel loop after loop-interchange, is called wrap loop. The serial loop, which is immediately enclosed by the
interchanged parallel loop, is called encl loop. Table 2 illustrates the loop structure of the time-consuming parallel loop nests in Perfect benchmarks. It shows the number of time-consuming parallel loops, the wrap loop, the number of loop levels, the level of parallel loops, the encl serial loops and the percentage of execution time required by the parallel loops.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Total</th>
<th>WrapLp</th>
<th>LvlS</th>
<th>ParLp</th>
<th>ParLvl</th>
<th>EncILp</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linpack</td>
<td>1</td>
<td>#134</td>
<td>3</td>
<td>#161</td>
<td>2</td>
<td>#325</td>
<td>49%</td>
</tr>
<tr>
<td>ADM (APS.f)</td>
<td>2</td>
<td>#4320</td>
<td>6</td>
<td>#453</td>
<td>5</td>
<td>#454</td>
<td>16%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#4729</td>
<td>6</td>
<td>#4858</td>
<td>5</td>
<td>#4859</td>
<td>11%</td>
</tr>
<tr>
<td>ARC2d (SRS.f)</td>
<td>8</td>
<td>#2155</td>
<td>4</td>
<td>#2165</td>
<td>3</td>
<td>#2165</td>
<td>7%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>#2215</td>
<td>3</td>
<td>#2218</td>
<td>7%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>#3144</td>
<td>3</td>
<td>#3145</td>
<td>5%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>#2348</td>
<td>3</td>
<td>#2350</td>
<td>5%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>#2366</td>
<td>3</td>
<td>#2367</td>
<td>5%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>#3722</td>
<td>3</td>
<td>#3723</td>
<td>4%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>#313</td>
<td>3</td>
<td>#2732</td>
<td>5%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>#2783</td>
<td>2</td>
<td>#2784</td>
<td>4%</td>
</tr>
<tr>
<td>BDNA (NAS.f)</td>
<td>3</td>
<td>#238</td>
<td>3</td>
<td>#3546</td>
<td>2</td>
<td>#3548</td>
<td>36%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>#3061</td>
<td>3</td>
<td>-</td>
<td>10%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>#3088</td>
<td>3</td>
<td>-</td>
<td>10%</td>
</tr>
<tr>
<td>DYFESM (SDS.f)</td>
<td>4</td>
<td>#6845</td>
<td>5</td>
<td>#875</td>
<td>4</td>
<td>#877</td>
<td>10%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>#891</td>
<td>4</td>
<td>#893</td>
<td>10%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>#698</td>
<td>7</td>
<td>#4358</td>
<td>35%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>#796</td>
<td>7</td>
<td>#4513</td>
<td>11%</td>
</tr>
<tr>
<td>FLO52 (TFS.f)</td>
<td>8</td>
<td>#874</td>
<td>4</td>
<td>#880</td>
<td>3</td>
<td>#879</td>
<td>5%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>#904</td>
<td>3</td>
<td>#903</td>
<td>5%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>#1014</td>
<td>3</td>
<td>#1013</td>
<td>10%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>#1027</td>
<td>4</td>
<td>#1025</td>
<td>4%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>#1040</td>
<td>3</td>
<td>#1039</td>
<td>10%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>#1053</td>
<td>4</td>
<td>#1051</td>
<td>5%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>#1269</td>
<td>4</td>
<td>#1264</td>
<td>7%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>#1278</td>
<td>4</td>
<td>#1265</td>
<td>7%</td>
</tr>
<tr>
<td>QCD (LGS.f)</td>
<td>1</td>
<td>#1138</td>
<td>14</td>
<td>#1144</td>
<td>14</td>
<td>-</td>
<td>22%</td>
</tr>
<tr>
<td>TRFD (TIS.f)</td>
<td>3</td>
<td>#279</td>
<td>6</td>
<td>#285</td>
<td>5</td>
<td>#289</td>
<td>35%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#338</td>
<td>7</td>
<td>#346</td>
<td>6</td>
<td>#350</td>
<td>23%</td>
</tr>
</tbody>
</table>

Table 2: Parallel Loops in Nested Constructs in Perfect Benchmarks

From the table, it can be seen that most of the time-consuming parallel loops in the benchmarks are in the middle level of the loop nests (or can be interchanged to the middle level). They enclose at least one level serial loop. The serial wrap loop usually encloses more than one parallel loop. For instance, in FLO52 a serial loop #874 encloses all time-consuming parallel loops of the benchmark. In our study, we noticed that most of the non-time-consuming parallel loops have such constructs also, which are contained by some serial wrap loop and may enclose other serial loops. The array access patterns in the parallel loops are studied for such nested parallel nested constructs. We concentrated on the major array variables with largest sizes, which have
to be bigger than the size of cache memory.

In the study, we noticed that most of the major arrays are two-dimensional. Some of them have three dimensions, but one dimension is always a constant in the time-consuming loop nests. These three-dimension arrays can be considered to be two-dimension arrays also.

A loop-alignment technique [5] in parallel/vectorized compiler was used to transform the array expressions into the same form for the different parallel loops enclosed by the same wrap loop in the same format. Especially, we pay attentions on a phenomena called inherent thrashing in our study, which is similar to the alignment conflict condition in [5]. The inherent thrashing phenomena may happen in the parallel loop nests with multiple major arrays. Let's check a parallel loop nest with arrays A and B. Suppose that A is assigned and B is used in the first parallel loop, B is assigned and A is used in the second parallel loop, and these parallel loops are enclosed by the same wrap loop. If the expressions of A and B are different or they cannot be aligned due to the alignment conflict, then there always exists cache or local memory thrashing for array A or array B at the execution of the common serial wrap loop. Actually, it is easy for parallel compilers to detect the inherent thrashing condition between two major arrays by checking the dependence graph. Fortunately, we did not find any inherent thrashing loop in the time-consuming parallel nests in the Perfect benchmarks.

**Lemma 1.** For a pair of array variables with loop-independent flow dependence in two parallel loops, which are enclosed by the same wrap loop, if there exists a cycle of flow dependence edges carried by the wrap loop between the variables in dependence graph, and the array subscript expressions are not the same or cannot be aligned to the same, then the array variables is called inherent thrashing.

### 3.3 Programming Model

In this section, we define a programming model, which represents certain types of parallel loop nests that are encountered very often in a wide range of applications and easy to be identified by parallel/vectorized compilers.

Based on the results of study shown in the previous section, if the cache or local memory thrashing is eliminated at the execution of the wrap loop, the speed up of the benchmarks will be significantly enhanced. In this paper, we present a solution to eliminate, or at least to reduce the cache or local memory thrashing on a common program construct in benchmark analysis: a serial wrap loop encloses parallel loops and each of the parallel loops may contain one or more serial loops. The loop constructs are not necessary to be perfect nested. Scalar code can be contained in the wrap loop or in the parallel loop. To simplify our presentation, we assume that there exists only one major array variable A. The results in the paper can be easily extended to the same program model with multiple major arrays, if they do not satisfy the inherent thrashing condition.

The programming model for the solution of the cache or local memory thrashing has the following characteristics:

1. It has a serial wrap-loop in the outermost level, denoted as DO Loop \( i \), which cannot be distributed in that level.
2. There are only one level loops parallel (or parallelized) denoted as PDO Loop \( j \) in the loop nest, which are immediately enclosed by the wrap-loop.
3. The parallel loops may contain serial loops after loop-interchange. The immediately enclosed
serial loop, named encl-loop, is denoted as DO Loop k. The encl-loop k may be an empty loop.

4. It may be a non-perfectly nested loop construct and have arbitrary IF-THEN-ELSE structure. Loop bounds are not necessary to be constants.

5. Only one two-dimensional major array variable A is in the loop nest, which has linear subscript expressions in terms of the indexes of wrap-loop, parallel loop and encl-loop after induction variable replacement.

6. The array A may have more than one expression in a parallel loop, but it has to have the same expressions in the different parallel loops after loop-alignment.

We assume that there are r different subscript expressions on the two dimension array A[1..D1, 1..D2]. These expressions are linear functions $f_m$ and $g_m$ respectively in column and row:

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

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

These linear functions $f, g: \mathbb{Z}^3 \rightarrow \mathbb{Z}^2$ are mapping from $N \times M \times L$ to $D_1 \times D_2$, where $N = U_{wrap} - L_{wrap}$ is the loop bound of the serial wrap loop, $M = \max(U_{par1} - L_{par1}, U_{par2} - L_{par2})$ is the maximal number of the parallel loop bounds if there are t parallel loops contained by the wrap loop, and $L = \max(U_{enc1} - L_{enc1}, U_{enc2} - L_{enc2}, U_{enc3} - L_{enc3})$ is the maximal number of the serial enclosed loops if there are s enclosed loops contained by the parallel loops.

We studied lots of scientific computation benchmarks such as mechanical CAE, structural analysis, fluid dynamics, heat transfer, computational chemistry, petroleum, and geographic applications. Most of the time-consuming parallel loop nests in the benchmarks have very similar characteristics with the programming model.

To compare with other approaches to eliminate or reduce the cache or local memory thrashing problem, we must mention blocking algorithms. Several research groups demonstrated the effectiveness of blocking algorithms for shared memory multiprocessors with memory hierarchies [7]. In general, a blocking algorithm needs to partition the array variables and broadcast the blocks to several processors if they reference data in the blocks for different iterations. In our program model, the wrap loop in the model is serial and cannot be distributed. The entire array is referenced in each iteration of the outermost serial loop. We have to rewrite the programs to use blocking algorithms. The approach presented in this paper can be applied in automatic parallel compilers and runtime library support. It doesn't require programs to rewrite application programs like most of the blocking algorithms.

The nature of the cache or local memory thrashing can be described in the following way. When a serial outermost wrap-loop encloses several parallel loops, the dependences carried by the serial loop may cause the data moving back and forth between threads that execute the iterations of the parallel loops in the different iterations of the outer serial loops. Some array elements may be reused in the different iterations of the serial loop due to the loop-carried dependences in the loop. Meanwhile, these array elements need to be moved in the caches or local memories between processors in each iteration of the serial loops due to the parallel loops.

4. Main Results

4.1 An Important Lemma

In the program model described in Section 3.3, the linear array subscript expressions can be used to partition the set of the pairs of wrap-loop index and the parallel loop index. The linear
functions \( f_m \) and \( g_m \), where \( m \) is from 1 to \( r \), specify a map from the reduced iteration space, \( N \times M \), to the set of subsets of the domain space.

\[
f, g: N \times M \rightarrow 2^{D_1 \times D_2}.
\]

In section 4.1 and 4.2, we discuss the simple case, where only one array subscript expression exists in the parallel construct. The subscript \( m \) for functions and coefficients can be omitted in the simple case. Then the results will be extended to the multiple expressions in section 4.3.

In the program model shown in section 3.3 with only one expression in the parallel construct, we define a set of elements of array \( A \), which are accessed within thread \( T_{i_0,j_0} \) as follows.

**Definition 1.** For a given pair \( i_0 \) and \( j_0 \), the set of elements \( A(f(i_0, j_0, k), g(i_0, j_0, k)) \) of array \( A \), which are accessed within thread \( T_{i_0,j_0} \), is denoted by \( A_{i_0,j_0} \), where \( 1 \leq k \leq L \).

\[
A_{i_0,j_0} = \{ A(f(i_0, j_0, k), g(i_0, j_0, k)) \mid \text{for given } i_0 \text{ and } j_0, \text{ where } k \in [1, L] \}.
\]

Since both \( f \) and \( g \) are linear in terms of \( i, j, \) and \( k \), the following lemma is obvious and useful in the rest of this section.

**Lemma 2.:** In a loop construct in the programming model described above, if there exist two points in iteration space, \( (i, j, k) \) and \( (i', j', k') \), such that

\[
f(i, j, k) = f(i', j', k')
\]

and

\[
g(i, j, k) = g(i', j', k'),
\]

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

\[
f(i, j, k + n_0) = f(i', j', k' + n_0)
\]

and

\[
g(i, j, k + n_0) = g(i', j', k' + n_0)
\]

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

It is clear from Lemma 2 that if

\[
A_{i_0,j_0}^{(m)} \cap A_{i_2,j_2}^{(m)} \neq \emptyset,
\]

then threads \( S_{i_1,j_1} \) and \( S_{i_2,j_2} \) should be assigned to the same processor, because they may access some common elements of array \( A \).

**4.2 Result in Single Expression**

Lemma 2 in the previous section can be used to collect a set of loop index pairs \( (i,j) \), whose corresponding threads may reuse some elements of array variable \( A \). The following theorem provides an efficient method to compute the parallel loop index from the current outer serial loop index and the previous parallel loop index in such a way that the thread to execute the current parallel loop iteration may access some array elements that were accessed in the thread of the previous parallel loop iteration within the previous serial loop iterations.

**Theorem 1.** In the program model described in Section 3.3, we have two points \( (i,j,k) \) and \( (i',j',k') \) in iteration space such that

\[
f(i,j,k) = f(i',j',k')\quad \text{and} \quad g(i,j,k) = g(i',j',k'),
\]
if they satisfy the following condition:

\[ i' - i = b_1c_2 - b_2c_1 \]
\[ j - j' = a_1c_2 - a_2c_1 \]
\[ k - k' = a_2b_1 - a_1b_2. \]

Let us denote:

\[ \alpha = b_1c_2 - b_2c_1 \]
\[ \beta = a_1c_2 - a_2c_1 \]
\[ \gamma = a_2b_1 - a_1b_2. \]

From \( \alpha, \beta, \gamma \), we can compute a set of points in the reduced iteration space for a given pair \((i_0, j_0)\). The threads corresponding to these points may access some common array elements at execution time. By \( \alpha, \beta, \gamma \) and the loop low bounds, it is easy to calculate the initial points for each set from which the following points can be computed from Theorem 1.

To prove that the sets of points in the reduced iteration space specified by Theorem 1 can partition the space, we give the following definition.

**Definition 2.** In the programming model described in Section 3.3, for a given pair \( i_0 \) and \( j_0 \), \( S_{i_0,j_0} \) denotes a set of points \((i,j)\) in the reduced iteration space of size \( N \times M \), which satisfy the following condition:

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

By Lemma 1 given in the previous section, the definition can be described as:

\[ S_{i_0,j_0} = \{ (i,j) \mid \text{there is } k_0 \text{ and } k \text{ such that } f(i_0,j_0,k_0) = f(i,j,k) \text{ and } g(i_0,j_0,k_0) = g(i,j,k) \}. \]

**Theorem 2.** In the programming model in Section 3.3, \( S_{i,j} \) is an equivalence class in the reduced iteration space \( N \times M \). Therefore the relation defined in Theorem 1 partitions the reduced iteration space.

By Theorem 2, the thread corresponding to a point of an equivalence class never accesses the element of array A, which is accessed in the threads corresponding to the points of the different equivalence classes. This means that if the threads corresponding to the points of an equivalence class are not assigned to the other processor, there is no unnecessary data moving between processors for array A. Of course, additional execution time is required to compute the current parallel loop index in terms of the current serial wrap-loop index, the previous wrap-loop and parallel loop indexes.

### 4.3 Results in Multiple Expression Case

In this section we discuss complicated case in which there are multiple expressions in the parallel loops of the programming model described in Section 3.3. As shown in Section 4.2, each linear function gives a particular value of \( \alpha, \beta, \gamma \) by Theorem 1.

Section 4.2 gave the definition of a set of points \((i,j)\) in the reduced iteration space, \( S_{i_0,j_0} \), in which each point may access some elements of array A that are referenced in thread \( T_{i_0,j_0} \) for given \((i_0, j_0)\).

This definition can be rewritten below, where the superscript \( (1) \) indicates that only one linear subscript expression is in the parallel loops.

\[ S_{i_0,j_0}^{(1)} = \{ (i,j) \mid i = i_0 + p \times \alpha \text{ and } j = j_0 + p \times \beta \text{ for } p \in \mathbb{Z} \}. \]
Now we extend the definition from one linear expression to multiple linear expressions in the parallel loop. As shown in the program model in Section 3.3, there is more than one subscript expression for a major array variable in the parallel loops. Assume the number of the different array subscript expressions is \( r \). By Theorem 1, we have a list of \( m \) triples: \( (\alpha_1, \beta_1, \gamma_1), (\alpha_2, \beta_2, \gamma_2), \ldots, (\alpha_r, \beta_r, \gamma_r) \).

The set of points in the reduced iteration space defined as follows can be viewed as an extension of the equivalence class defined in section 4.2. If the relationship defined by Definition 3 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 3.** In reduced iteration space of wrap-loop \( i \) and parallel loop \( j \), there are \( r \) different subscript expressions for a major array variable. The set of the pairs of \( i \) and \( j \), whose corresponding threads may access some common elements of the array or arrays that are stored in the local cache by thread \( T_{i_0, j_0} \), is defined as \( S^{(r)}_{i_0, j_0} \).

\[
S^{(r)}_{i_0, j_0} = \{(i, j) \mid i = i_0 + \sum p_m \times \alpha_m \text{ and } j = j_0 + \sum p_m \times \beta_m \text{ for } p_1, \ldots, p_r \in \mathbb{Z}\}.
\]

The following Theorem shows that the relation defined in the above definition can partition the reduced iteration space. If all threads corresponding to the points \( S_{i, j} \) are assigned into the same processor, the processor never accesses any data in the caches or local memories of the other processors. Therefore, we can reduce the unnecessary data moving between processors and improve the system performance.

**Theorem 3.** \( S^{(r)}_{i_0, j_0} \) is an equivalence class in the reduced iteration space of loop \( i \) and loop \( j \).

By Theorem 1 and Definition 3, it is obvious that there is no memory access from one processor to cache or local memory of other processor, if all threads in the same equivalence class are assigned to one processor.

**Theorem 4:** Every thread \( T_{i, j} \notin S^{(r)}_{i_0, j_0} \) may reuse some data in the other threads belonging to the other equivalence classes \( S^{(r)}_{i_0, j_0} \), but never access the data referenced by the threads belonging to the other equivalence classes.

A similar but more complicated calculation for the current parallel loop index in terms of the current serial wrap-loop index and the previous wrap-loop and parallel loop indexes as one in Section 4.2 can be developed from Theorem 1 and Theorem 3 in [8].

5. Application of the Results in Parallel Compilers

In this section, we briefly describe a way to apply the above results in parallel compilers to eliminate the cache or local memory thrashing. Here we only show the essential idea of the approach, and leave the detailed implementation to readers.

In general, parallel/vectorized compilers hold the information of nested loop structure. After dependence analysis and loop transformation, the serial wrap-loop that immediately encloses parallel loops is easy to be determined, assuming only one level loops are parallel or parallelized. A heuristic algorithm may be required to estimate the execution time for the loop nest. If the loop nest might be time-consuming, the results in this paper are applied.

The algorithm to determine whether the time-consuming loop nest satisfies the programming model in section 3.3 can be described as follows:
1. An heuristic algorithm is required to choose the major array variables by estimating the amount of elements referenced in the loop nest and checking the linear expressions.

2. Check the expression in each dimension for these major arrays. If an array has more than two dimensions and more than two subscript expressions using the wrap-loop and parallel loop indexes, move the array from the major array set.

3. If an array appears in more than one parallel loops in the loop nest and the loop-alignment cannot make its expression in the same form for the parallel loops, move the array from the major array set.

4. Applying the approach presented in Lemma 1 for each pair of arrays, if there is inherent thrashing condition between them, move one of them from the major array set.

5. If the major array set is not empty, calculate the initial points for each equivalence class by using the approach in this paper, and save all information in a shared data structure.

6. Create run-time library for the parallel loops in the loop nest, which will compute the current parallel loop index in terms of the previous wrap-loop and parallel loop indexes and the current wrap loop index as shown in Theorem 1, 2 and 3. The information in the shared data structure can be used to schedule processor dynamically at the execution time.

We want to emphasize here that the approach presented in the paper may eliminate some unnecessary data moving then reduce the cache or local memory thrashing. In worst case, the approach cannot predicate the right major arrays, but it never increases cache missing and degrade the system performance.

If the parallel nested constructs are found in the time-consuming loops satisfying our program model, the implementation will be straightforward. In general, sequential optimizers create template control variables such as "loop-increment", "loop-bound" and "control counter" to perform loop control. The loop index variable is recomputed in the beginning of the loop body at the execution time. The same idea can be employed in parallel loops to compute the right iteration to be executed for the processor.

In static scheduling, all the equivalence classes can be computed and assigned to the processors at the compiler time. The initial points of each equivalence class can be determined by calculating the loop lower bounds and the $\alpha, \beta, \gamma$ in Theorem 1. Each processor requires some local memory locations to keep the $\alpha, \beta, \gamma$, and other information.

In self-scheduling [9], the initial points of the equivalence classes are prepared at compile time, and are stored in a shared data structure. In the first iteration of the outermost serial loop, the processors get the initial points from the shared data structure and execute the corresponding equivalence classes. After that, each processor can calculate the current vectors from the previous vectors which are stored in the local memory in the processor. If all iterations in an equivalence class have completed, the processor can check the shared data structure to find whether any other equivalence classes are available.

The approach presented in the paper may be against the idea of load balancing in self-scheduling, because restricting each processor to specific set of array elements to minimize cache or local memory thrashing may interfere with the load balancing mechanism. Some heuristics needs to be developed for a processor, which have completed all iterations in the equivalence
classes with data in its own cache or local memory, to move data in other equivalence class from the other processors if these processors have heavy working load. The heuristic development will depend on results of experiment and is beyond the scope of the paper.

6. Experimental Results

We have implemented the results of this paper in a parallel compiler prototype, which performs the dependence analysis and parallel transformations for FORTRAN programs. The prototype computes the $\alpha$, $\beta$, $\gamma$ in Theorem 1, and the initial points at compiler time, then uses the information for dynamic scheduling as in [9] at execution time. The parallel code generated by the prototype is running on a shared memory multiprocessor simulator has MIPS-based system simulator. The system can simulate 4 to 16 processors, sizes of cache memories, cache coherence protocols, cache line sizes and memory bandwidth of data bus, crossbar or interconnection network. The processor and the cache in the simulation system are based on MIPS R3000 and R4000. The prototype, simulates different scheduling strategies also. They include static scheduling, self-scheduling, and guided-self-scheduling. The experimental results show that these scheduling approaches are slightly different on the execution performance. But the technique presented in this paper to reduce the cache thrashing problem made a significant improvement for the execution performance, no matter which scheduling approach was used in the experiments.

In examining the experimental results, the reader should be aware that some of the improvements cited may have been achieved because the huge cost of the cache is missing in RISC architecture. We compared the parallel code execution with or without the compiler strategy presented in this paper for the cache thrashing problem and found significant enhancement by eliminating the unnecessary data moving back and forth between processors. In the experiment, we assume that the memory and crossbar bandwidth is proportionally improved when the number of processors gets increased.

Gaussian Elimination. Gaussian Elimination is a basic matrix operation that is used in many application programs. We use a 1K by 1K array in the experimental benchmark.

<table>
<thead>
<tr>
<th>Number Processor</th>
<th>Original Serial Code</th>
<th>Parallel Code with Cache Thrashing</th>
<th>Parallel Code without Cache Thrashing</th>
<th>Speed Up</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>285.0s</td>
<td>163.5s</td>
<td>102.2s</td>
<td>1.6</td>
</tr>
<tr>
<td>8</td>
<td>285.3s</td>
<td>109.8s</td>
<td>59.9s</td>
<td>1.7</td>
</tr>
<tr>
<td>16</td>
<td>286.2s</td>
<td>83.6s</td>
<td>39.8s</td>
<td>2.1</td>
</tr>
</tbody>
</table>

Linpack Benchmark. Linpack benchmark is vectorized/parallelized code. We chose the loops containing SAXPY and SMXYPY subroutine calls, and inlined these routines in the loops, most of which have three level loops: Serial, parallel, serial. As the table below shows, our approach achieved better performance from the original parallel code that doesn’t have any consideration for the cache thrashing problem. To make this measurement, we use 1K by 1K Linpack benchmark.

<table>
<thead>
<tr>
<th>Number processor</th>
<th>Original Serial Code</th>
<th>Parallel Code with Cache Thrashing</th>
<th>Parallel Code without Cache Thrashing</th>
<th>Speed Up</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>514.7s</td>
<td>327.5s</td>
<td>234.0s</td>
<td>1.4</td>
</tr>
<tr>
<td>8</td>
<td>514.3s</td>
<td>226.9s</td>
<td>151.3s</td>
<td>1.5</td>
</tr>
<tr>
<td>16</td>
<td>515.2s</td>
<td>161.0s</td>
<td>94.7s</td>
<td>1.7</td>
</tr>
</tbody>
</table>
A Complete Application. We also performed the test on a complete application program benchmark, a computational chemistry application program. The kernel of the most frequently used routine has the following form:

\[ K = 0 \]
\[ 100 \quad K = K + 1 \]

\[ \text{PARALLEL DO } I = 1, M \]
\[ \text{DO } 200 \quad J = 1, M \]
\[ X(I+K,J+I+K) = \ldots \]
\[ \ldots = X(I+K, J+I+K) \ast \ldots \]
\[ 200 \quad \text{CONTINUE} \]
\[ \text{END PARALLEL DO} \]
\[ \text{IF (K .LT. MAX_BOUND) GO TO 100} \]

The dimension of the array \( X \) is 3K by 1K. The approach presented in the paper works perfectly for the program. The table below shows the results when our approach was applied to eliminate cache thrashing.

<table>
<thead>
<tr>
<th>Number Processor</th>
<th>Original Serial Code</th>
<th>Parallel Code with Cache Thrashing</th>
<th>Parallel Code without Cache Thrashing</th>
<th>Speed Up</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1732.0s</td>
<td>1415.6s</td>
<td>832.7s</td>
<td>1.7</td>
</tr>
<tr>
<td>8</td>
<td>1734.1s</td>
<td>843.2s</td>
<td>481.8s</td>
<td>1.75</td>
</tr>
<tr>
<td>16</td>
<td>1735.7s</td>
<td>557.8s</td>
<td>309.9s</td>
<td>1.8</td>
</tr>
</tbody>
</table>

The compilation time was measured also on our experience. Since the loops and array subscript expressions in the loops are well represented in vectorized/parallel compiler, the implementation of the approach presented in the paper only costs less than 7000 line C code. The compilation time only increases less than 5% for all benchmarks with thousands line FORTRAN code. In dynamic scheduling, the approach presented in the paper requires more complicated shared data structure than one described [9]. It usually takes couple thousands bytes for a 1K x 1K array variable to save the initial points for the equivalence classes.

7. Conclusions

Most of the compiler techniques used to enhance the cache hit ratio focus on data locality by blocking the original programs. This paper describes another phenomena called "cache or local memory thrashing", which degrades the cache hit ratio and increases unnecessary data-bus, crossbar or interconnection network traffic. A mathematical concept is presented in this paper, which partitions the iteration space so that the threads in the same equivalence class never access the caches or local memories in other processors if they are assigned to the same processor. The approach in this paper is good for some program constructs that are very common in application programs. For these application programs, our approach can significantly reduce the cache or local memory thrashing phenomena.

We only discuss a simple machine model and a simple program model in this paper. Even with that, the multiple linear expression case complicates the algorithm to compute the appropriate parallel loop index at the execution time. Future research work on this topic includes
mathematical concepts for more complicated machine models and more complex program models, and simpler algorithms for the multiple linear expression cases. Studying new algorithms for the other features of cache hardware design, such as different cache coherence strategies, multiple word cache lines, or multiple level cache memory, is also important in developing parallel compilers.

Reference


