Journal of Parallel and Distributed Computing **60**, 924–965 (2000)
doi:10.1006/jpdc.2000.1639, available online at http://www.idealibrary.com on IDEAL®

Compiler Algorithms for Optimizing Locality and Parallelism on Shared and Distributed-Memory Machines

M. Kandemir

Department of Computer Science and Engineering, The Pennsylvania State University, University Park, Pennsylvania 16802
E-mail: kandemir@cse.psu.edu

J. Ramanujam

Department of Electrical and Computer Engineering, Louisiana State University, Baton Rouge, Louisiana 70803
E-mail: jxr@ee.lsu.edu

and

A. Choudhary

Department of Electrical and Computer Engineering, Northwestern University, Evanston, Illinois 60208
E-mail: choudhar@ece.nwu.edu

Received July 31, 1998; revised June 20, 1999; accepted February 21, 2000

Distributed-memory message-passing machines deliver scalable performance but are difficult to program. Shared-memory machines, on the other hand, are easier to program but obtaining scalable performance with large number of processors is difficult. Recently, scalable machines based on logically shared physically distributed memory have been designed and implemented. While some of the performance issues like parallelism and locality are common to different parallel architectures, issues such as data distribution are unique to specific architectures. One of the most important challenges compiler writers face is the design of compilation techniques that can work well on a variety of architectures. In this paper, we propose an algorithm that can be employed by optimizing compilers for different types of parallel architectures. Our optimization algorithm does the following: (1) transforms loop nests such that, where possible, the iterations of the outermost loops can be run in parallel across processors; (2) optimizes memory locality by carefully distributing each array across processors; (3) optimizes interprocessor communication using message vectorization whenever possible; and (4) optimizes cache locality by assigning appropriate storage layout for each array and by transforming the iteration space. Depending on the

0743-7315/00 $35.00

machine architecture, some or all of these steps can be applied in a unified framework. We report empirical results on an SGI Origin 2000 distributed-shared-memory multiprocessor and an IBM SP-2 distributed-memory message-passing machine to validate the effectiveness of our approach.

Key Words: data reuse; locality optimizations; temporal locality; spatial locality; memory performance; parallelism; array restructuring; loop transformations.

1. INTRODUCTION

Optimizing locality and parallelism together is important for UMA (uniform memory access) architectures (e.g., SGI Power Challenge XL, Dec Alpha-server 8400 5/440, and Sun Ultra Enterprise 6000), shared-memory NUMA (nonuniform memory access) architectures (e.g., SGI Origin [30], Convex Exemplar [9], Stanford DASH [31], and MIT Alewife [1]), and distributed-memory multicomputers (e.g., the IBM SP-2 and Intel Paragon). Optimizing *parallelism* leads to tasks of larger granularity with lower synchronization and communication costs and is beneficial for these parallel machines. Since individual nodes of contemporary parallel machines have some form of memory hierarchy, optimizing *cache locality* of scientific codes on these machines results in better performance. Additionally, since the distribution of data is an important issue for distributed-memory multicomputers and some shared-memory NUMA machines, optimizing *memory locality* through data distribution has a large impact on the overall performance of programs on these machines.

We have witnessed a tremendous increase in processor speeds in recent years. In fact, during the past decade, processor speeds have been improving at a rate of at least 60% each year [22]. Since the per-year improvement in memory access times is only around 7% [22], the performance gap between processors and memory has widened. Although caches are capable of reducing the average memory access time and optimizing compilers are able to detect significant parallelism, the performance of scientific programs on both uniprocessors and parallel machines can be rather poor if locality is not exploited [46]. Some issues that challenge compiler writers are maximizing parallelism, minimizing communication via loop level optimizations and block transfers, and optimizing memory and cache locality. Since these issues are interrelated, we believe that they should be handled in a *unified* framework. For example, given a loop nest that has accesses to a number of arrays, locality optimizations may imply a preferred order for the loops whereas the parallelism optimizations may suggest another. In this paper, we present a unified method by which an optimizing compiler can enhance performance of regular scientific codes for both locality and parallelism. Our method is implemented using the Parafrase-2 compilation framework [37]. Specifically, our optimizations perform the following:

(1) Maximizing the granularity of parallelism by transforming the loop nest such that the iterations of the outermost loops can run in parallel on a number of processors; this reduces communication and synchronization costs.

(2) Vectorizing communication, i.e., performing communication in large chunks of data in order to amortize the high startup cost.

(3) Reorganizing data layouts in memory: we believe that matching the loop order with individual array layouts in memory is important for obtaining high levels of performance in regular scientific codes.

A recent study shows that a group of highly parallelized benchmark programs spend as much as 39% of their cycles stalled waiting for memory accesses [44]. In order to eliminate the memory bottleneck, cache locality should be exploited as much as possible. One way of achieving this is to transform loops such that the innermost loop exhibits unit-stride accesses for as many array references as possible. While this approach produces satisfactory results for several cases, we show in this paper that there is still room for significant improvement if the compiler is allowed to choose memory layouts for large multidimensional arrays. We believe that compiler optimizations that exploit spatial locality by changing the conventional memory layouts (which are language-specific) will be very useful in both uniprocessor and multiprocessor–multicomputer environments. Of course, any approach aimed at improving locality on parallel machines should also maintain high granularity of parallelism. This is not only because increasing the granularity of parallelism reduces communication and synchronization costs, but also because larger granularity of parallelism often correlates with good memory locality and high levels of performance [46].

The remainder of this paper is organized as follows. In Section 2, we outline basic concepts related to locality, loop transformations, and data transformations. In Section 3, related work is summarized, emphasizing the difference between our work and the previous work on locality. A locality optimization algorithm is introduced in Section 4. An algorithm that maximizes granularity of parallelism and improves memory locality is discussed in Section 5. The algorithms in Sections 4 and 5 prepare the reader for the main algorithm of the paper in Section 6, in which a unified compiler algorithm that can be used on different architectures is presented. The unified algorithm is generalized to handle multiple nests in Section 7. Experimental results are given in Section 8, and finally the conclusions are presented in Section 9.

2. PRELIMINARIES

2.1. Definitions

The *iteration space* $\mathscr{I}$ of a loop nest contains one point for each iteration. An *iteration vector* can be used to label each such point in $\mathscr{I}$. We use $\vec{I}=(i_1, ..., i_n)^T$ to represent the iteration vector for a loop nest of depth n, where each i_k corresponds to a loop index, starting with i_1 for the outermost loop index. In fact, an iteration space $\mathscr{I}$ can be viewed as a polyhedron bounded by the loop limits. Such a polyhedron can be represented by an integer system $\mathscr{C}\vec{I} \leqslant \vec{c}$.

The subscript function for a reference to an array X is a mapping from the iteration space $\mathscr{I}$ to the data space $\mathscr{D}$; this can also be viewed as a polyhedron bounded

by array bounds. A subscript function defined this way maps an iteration vector to an array element. In this paper, we assume that subscript mappings are affine functions of the enclosing loop indices and symbolic constants. Many references found in regular scientific codes fall into this category. Under this assumption, a reference to an array X can be represented as $X(\mathscr{L}^X\vec{I}+\vec{b}^X)$ where $\mathscr{L}^X$ is a linear transformation matrix called the *array access* (*reference*) *matrix*, $\vec{b}^X$ is the *offset* (*constant*) *vector*, and $\vec{I}$ is the iteration vector [49, 32, 38, 50]. The kth row of $\mathscr{L}^X$ is denoted by $\vec{\ell}_k^X$. For a reference to an m-dimensional array inside an n-dimensional loop nest, the array access matrix is of size $m \times n$ and the offset vector is of size m. Let us now consider the loop nest given below to illustrate these concepts.

```
DO i = 1, N
  DO j = 1, N
    A(i, j) = B(i + j − 1, j + 2)
  END DO
END DO
```

For this loop nest, $\vec{I}=\binom{i}{j}$, $\mathscr{L}^A=\left(\begin{smallmatrix}1 & 0\\ 0 & 1\end{smallmatrix}\right)$, $\vec{b}^A=\binom{0}{0}$; and $\mathscr{L}^B=\left(\begin{smallmatrix}1 & 1\\ 0 & 1\end{smallmatrix}\right)$, $\vec{b}^B=\binom{-1}{2}$.

2.2. Iteration Space (Loop) Transformations

A linear one-to-one mapping between two iteration spaces can be represented by a nonsingular matrix T. Let $\mathscr{I}$ denote the original iteration space and $\mathscr{I}'$ the transformed iteration space. A linear iteration space transformation of $\mathscr{I}$ by a matrix T means that $\forall\vec{\imath}\ (\vec{\imath}\in\mathscr{I}\rightarrow T\vec{\imath}\in\mathscr{I}')$. That is, each element $\vec{\imath}$ of the original iteration space $\mathscr{I}$ is mapped to $T\vec{\imath}$ on the *transformed* iteration space $\mathscr{I}'$. Converting the original loop nest to the transformed nest is a two-step process [50]:

(1) *Loop bounds are transformed.* If we assume that $\mathscr{I}$ can be represented by $\mathscr{C}\vec{I}\geqslant\vec{c}$, since $I'=TI$, the new system defining the transformed polyhedron will be $\mathscr{C}T^{-1}\vec{I}'\leqslant\vec{c}$. In general, finding the new loop bounds from $\mathscr{C}T^{-1}\vec{I}'\leqslant\vec{c}$ may require use of Fourier–Motzkin elimination [42]. In this paper, we denote T^{-1} by Q. An important characteristic of our approach is that (using the array access matrices) the entries of Q are derived in a systematic manner.

(2) *Subscript expressions are rewritten in terms of new loop indices.* If an original subscript expression for a reference is $\mathscr{L}I+\vec{b}$, since $I'=TI$, the new subscript expression is $\mathscr{L}T^{-1}I'+\vec{b}$.

2.3. Reuse and Locality

We refer to the unit of data transfer between main memory and cache as a *block* or a *line*. In order to obtain good performance from programs running on a machine which contains some sort of cache memory, *cache locality* should be exploited. That is, a data item brought into cache should be reused as much as possible before it is replaced. The reuse of the same data while it is still in the cache is termed *temporal locality*, whereas the use of the nearby data in a cache line is called *spatial locality* [50].

We have to stress that a program may reuse the data, but if that data has been replaced between reuses we say that it does *not* exhibit locality. In other words, a program may have data reuse, but due to the replacement of the data, it might not be able to exploit cache locality [32]. Consider the example shown below.

```
DO i = 1, N
  DO j = 1, N
    A(j) = B(i) + C(j, i)
  END DO
END DO
```

Assuming a column-major memory layout as the default, array B has temporal reuse in the j loop and spatial reuse in the i loop in this loop nest. Array A has temporal reuse in the i loop and spatial reuse in the j loop. Array C, on the other hand, has only spatial reuse in the j loop. Assuming that N is very large, it is reasonable to expect that, during execution of this nest, only the reuses associated with the *innermost loop* (the j loop) will exhibit locality. Similar assumptions have also been made by Wolf and Lam [49]. As a result, the exploitable reuses for this nest are the temporal reuse for B and the spatial reuses for A and C.

A large body of previous compiler research has focused on optimizing locality through loop transformations. Some of the previous research along that direction will be discussed in Section 3.

2.4. *Data Space (Array Layout) Transformations*

Locality can be improved by transforming the iteration and/or data spaces. Consider the loop nest shown in Fig. 1a. In this loop nest, assuming default *column-major* layouts and that N is very large, the only locality that can be exploited is the spatial locality due to array B. Exploiting spatial locality for A, on the other hand, requires loop interchange [50]. While loop interchange will improve the locality for A, it will affect the locality of B negatively. If, on the other hand, without applying any loop transformation, array A is stored as *row-major* in memory, then we can exploit the spatial locality for both of the references. This simple example shows that data layout transformations may result in better performance in some cases compared to loop (iteration space) transformations.

```
DO i = 1, N
  DO j = 1, N
    A(i,j) = B(j,i)
  END DO
END DO
```

(a)

```
DO i = 2, N
  DO j = 1, N-1
    A(i,j) = A(i-1,j+1)
  END DO
END DO
```

(b)

FIG. 1. Two loop nests for which data layout transformations are useful: (a) loop transformation is not effective, (b) loop transformation is not legal. Notice that in both cases a simple data layout transformation (column-major to row-major conversion) may help.

Consider the code in Fig. 1b. In this example, again assuming a column-major memory layout as the default, locality is poor for both of the references to array A. Loop interchange is *illegal* due to data dependence between references. On the other hand, if we change the layout of array A to row-major, locality will be very good for both of the references. This example reveals an important fact: data transformations can be applicable where loop transformations are not, since, unlike loop transformations, they are not constrained by data dependences in the program [12].

In addition, data transformations can work for *imperfectly nested loops* and can be used to optimize *explicitly parallelized programs* as well [12]. But since they *cannot* improve temporal locality directly, and their effect on program locality is *global* (i.e., program-wide), in principle, the best results should be obtained if data space and iteration space transformations are applied in concert [12]. In this paper, we attempt to achieve this by determining data layouts for each array and loop transformations for each nest in a unified framework. As observed by Chandra *et al.* [11], due to some certain conditions related to storage and sequence assumptions about the arrays, and to passing arrays as subroutine arguments, data transformations may sometimes not be legal. We assume no such situation occurs for the programs studied in this paper. Chandra *et al.* [11] have proposed methods to deal with storage sequence and parameter passing problems when data transformations are to be applied. An investigation of such issues is beyond the scope of this paper.

2.5. *Scope of Our Work*

The scope of our work is regular dense array programs. As mentioned earlier, we assume that array subscript functions and loop bounds are affine functions of the enclosing loop indices and symbolic constants. We also assume that the memory layout of an m-dimensional array can be in any of the $m!$ forms, each corresponding to layout of data in memory linearly by a nested traversal of the axes in some predetermined order. In other words, the data storage schemes we consider can be expressed as permutations of the dimensions of the array. For a two-dimensional array, these are only row-major and column-major layouts. For a three-dimensional array, there are six possible storages, and so on. Blocked layouts, on the other hand, are useful for some matrix codes and automatic detection of blocked layouts is in our future research. Each layout that we consider in this paper has an associated *fastest changing dimension* (FCD), that is, the innermost dimension in the traversal of array in memory. For instance, in row-major layouts the last dimension is the fastest changing dimension. Our layout detection algorithm only determines the FCD, since under the assumption of large array bounds the relative order of the other dimensions may not be important. We also assume that on distributed-memory machines the arrays will be distributed across processors along only a *single* dimension. We refer to this dimension as the *distributed dimension* (DD). The general flavor of our approach is to determine the appropriate FCD and DD for a given multidimensional array. Finally, we assume that unless specified otherwise the default layout is column-major for all arrays.

3. RELATED WORK

3.1. Related Work on Iteration Space Transformations

Loop transformations have been used for optimizing cache locality in several papers [32, 49, 10, 34]. Results have shown that on several architectures the speedups achieved by loop transformations can be quite large. McKinley *et al.* [34] offer a unified optimization technique consisting of loop permutation, loop fusion, and loop distribution. By considering iteration space transformations only, they obtain significant speedups for several scientific codes.

Wolf and Lam [49] propose a data locality optimizing algorithm based on a mathematical description of *reuse*. They identify and quantify reuse within an iteration space. They use vector spaces to represent the directions where reuse occurs and define different types of reuses found in dense matrix programs. Their approach is sort of exhaustive in the sense that they try all possible subsets of the loops in the nest, and (if necessary) by applying unimodular transformations they bring the subset with the best potential reuse into the innermost positions. They report performance numbers on some common kernels such as LU decomposition and SOR (successive-over-relaxation). The main problem with their algorithm is that they base all decisions depending on whether or not a loop carries reuse. They do not take the loop bounds into account and sometimes may end up in suboptimal solutions due to inaccuracies in their representation of reuse vector spaces.

In contrast, Li [32] describes a data reuse model and a compiler algorithm called *height reduction* to improve cache locality. He discusses the concept of a *data reuse vector* and defines its *height* as the number of dimensions from the first nonzero entry to the last entry. The nonzero entries of a reuse vector indicate that there are reuses carried by the corresponding loops. The individual reuse vectors constitute reuse matrices which in turn constitute the *global reuse matrix*. His algorithm assigns priorities to reuse vectors depending on the number of times they occur and tries to reduce the height of the global reuse matrix starting from the reuse vector of highest priority. His algorithm gives an *implicit preference* to *spatial reuses*. The height reduction algorithm both reduces the sensitivity of tiling to the tile size and places the loops carrying reuse into innermost positions.

Kennedy and McKinley [27] present a compiler algorithm for optimizing parallelism and data locality using loop transformations alone. They show that an appropriate combination of tiling and loop permutations can be used to obtain both outer loop parallelism and inner loop locality.

None of [49], [32], and [27] considers data space transformations. In this paper, we show that data space transformations can also make a difference on the locality properties of the programs. Moreover, by unifying data space transformations with iteration space transformations, locality and parallelism can be exploited in a better way which is not possible by the pure loop or pure data transformations alone.

Intuitively, the more spatial reuse is exploited, the lower the miss ratio will be and a lower amount of false sharing will occur [2]. Our unified algorithm tries aggressively to exploit the spatial locality by considering different memory layouts for different arrays. Since Li's approach is representative of a class of algorithms that use loop transformations alone to exploit locality [49, 34, 32], in the course of this paper, we compare our algorithm to Li's algorithm (denoted l-opt).

3.2. Related Work on Data Space Transformations

Data transformations, on the other hand, deal with data layout and array accesses rather than reordering of loop iterations. Only a few papers have considered data transformations to optimize locality. In fact, just like iteration space, the data space can also be transformed using linear nonsingular transformation matrices. O'Boyle and Knijnenburg [35] have applied this idea to improve the cache locality of programs. Let $\mathscr{T}$ be a linear nonsingular data transformation matrix. Omitting the shift-type transformations, a data transformation denoted by $\mathscr{T}$ is applied in two steps, (1) the original reference matrix $\mathscr{L}$ is transformed to $\mathscr{T}\mathscr{L}$ and (2) the layout of the array in memory is also transformed using $\mathscr{T}$, and the array declaration statements are changed accordingly. Notice that determining the bounds of a transformed array may require the use of Fourier–Motzkin elimination [42]. In fact, rather than trying to determine desired data transformation matrices, their work focuses more on restructuring the code given a data transformation matrix. In comparison, we apply *both* data and iteration space transformations and concentrate on the problem of determining suitable transformation matrices with locality *and* parallelism in mind.

Anderson *et al.* [2] propose a data transformation technique for distributed-shared-memory machines. By using two types of data transformations (strip-mining and permutation), they attempt to make the data accessed by the same processor contiguous in the shared address space. Their algorithm inherits parallelism decisions made by a previous phase of the SUIF compiler [47]; so in a sense it is not directly comparable to ours which attempts to come up with a transformation matrix suitable for both locality and parallelism. It is also not clear to us how useful their approach is on a uniprocessor environment whereas our approach is applicable to uniprocessors and distributed memory and shared memory machines.

3.3. Related Work on Unified Loop and Data Transformations

Ju and Dietz [25] present a systematic approach that integrates data layout optimizations and loop transformations to reduce cache coherence overhead. Cierniak and Li [12] present a unified approach like ours to optimize locality that employs both data and control transformations. The notion of a *stride vector* is introduced and an optimization strategy is developed for obtaining the desired mapping vectors representing layouts and a loop transformation matrix. At the end, the following equality is obtained: $T^T v = \mathscr{L}^T x$. In this formulation only $\mathscr{L}$, the array access (reference) matrix, is known. The algorithm tries to find T, the iteration-space transformation matrix; x, a *mapping vector* which can assume $m!$

different forms for an m-dimensional array; and v, the desired stride vector for consecutive array accesses. Since this optimization problem is difficult to solve, the following heuristic is used: first, it is assumed that the transformation matrix contains only zeroes and ones. Second, the value of the stride vector v is assumed to be known beforehand. Then the algorithm constructs the matrix T row-by-row by considering a restricted set of legal mappings. In comparison, our approach is more accurate, as it does not restrict the search space of possible loop transformations. We can employ any nonsingular legal loop transformation as long as it is suitable for our purpose. Also, our approach is simpler to be embedded in a compilation system, since it it does not depend on any new reuse abstraction such as stride vector whose value should be guessed by a compiler. We simply exploit the available parallelizing compiler technology to detect optimal array layouts as well as the iteration space transformation. Our extension to multiple nests is also different from the one offered by Cierniak and Li [12] for global optimization.

3.4. Related Work on Parallelism

Previous work on parallelism has concentrated, among other topics, on compilation techniques for multicomputers [5, 8, 51, 24], for multiprocessors [47, 7], and for automatic discovery of parallelism [21, 48, 39, 18, 36, 26]. Since neither data layout transformations nor cache locality was the central issue in any of these papers, we do not discuss them here any further.

4. ALGORITHM FOR ENHANCING CACHE LOCALITY

Since accessing data on memory is usually an order of magnitude slower than accessing data in cache, optimizing compilers must reduce the number of memory accesses as well as the volume of data transferred. In this section, we present an algorithm which automatically

(1) transforms a given loop nest to exploit cache locality, and

(2) assigns appropriate memory layouts for arrays referenced in the nest.

This algorithm can be used for optimizing locality in uniprocessors and shared-memory multiprocessors. Moreover, it can also be employed as part of a unified technique for optimizing locality and parallelism in distributed-memory multicomputers.

4.1. Explanation

The overall algorithm is shown in Fig. 2. In the algorithm, C is the array reference on the left-hand side whereas A_r represents the rth right-hand side array ($1 \leqslant r \leqslant R$). The symbol $\vec{q_i}$ refers to the ith column of $Q = T^{-1}$ from left. Let $j_1, ..., j_n$ be the loop indices of the *transformed* nest, starting from outermost position.

- **INPUT:** $\mathcal{L}^C$ for the left hand side (LHS) array reference
  $\mathcal{L}^{A_r}$ for the right hand side (RHS) array references where $1 \leq r \leq R$
- **OUTPUT:** A loop transformation matrix T and a memory layout (in fact, only a FCD) for each array
- **PROCEDURE:**

(1) // check if the LHS array can be optimized for temporal locality
if there is a $\vec{q_n}$ such that for each i $(1 \leq i \leq m)$, $\vec{q_n} \in Ker\{\vec{\ell_i}^C\}$ then goto Step (3).

(2) // optimize the LHS for spatial locality in the innermost loop
flag $\leftarrow$ TRUE; $i \leftarrow 1$
while $(i \leq m)$ and (flag == TRUE) {
- check if there is a q_n and a nonzero constant c such that $\vec{\ell_i}^C \times \vec{q_n} = c$, and $\vec{\ell_k}^C \times \vec{q_n} = 0$ for each $k \neq i$
- if a solution exists for i, mark i as the FCD for array C, and set flag to FALSE; if no solution exists for i, increment i

}

(3) // optimize each RHS array reference in turn
$r \leftarrow 1$; initialize done[r] to FALSE for all $1 \leq r \leq R$;
while $(r \leq R)$ {

(3.1) // check for temporal locality
if $\vec{q_n} \in Ker\{\vec{\ell_j}^{A_r}\}$ holds for each $1 \leq j \leq m$ then go to Step (3.3)

(3.2) // check for spatial locality in the innermost loop
- try to find a j and a nonzero constant c such that $\vec{\ell_j}^{A_r} \times \vec{q_n} = c$, and $\vec{\ell_k}^{A_r} \times \vec{q_n} = 0$ for each $k \neq j$
- if a solution exists, mark j as the FCD for array A_r and goto Step (3.3)

}
(3.3) set done[r] to TRUE; increment r

(3.4) // check for spatial locality in the other loops
$r \leftarrow 1$; initialize FEASIBLE to $\{\}$
while $((r \leq R)$ and done[r] == FALSE) {
- check if there is a j for which there exists a vector $\vec{x}$ (the gcd of whose components is 1) and a non-zero constant c such that $\vec{\ell_j}^{A_r} \times \vec{x} = c$, and for each $k \neq j$, $\vec{\ell_k}^{A_r} \times \vec{x} = 0$.
- if there is such a vector $\vec{x}$, then mark j as the FCD for array A_r, set done[r] to TRUE.
  If $\vec{x} \notin$ FEASIBLE then (set frequency[$\vec{x}$] = 1 and add $\vec{x}$ to FEASIBLE); else (increment frequency[$\vec{x}$])
- increment r

}
Sort the elements of the set FEASIBLE in decreasing order of their frequencies; let s be the size of the set FEASIBLE. Set $\vec{q_n}_v$ to be the v^{th} element of the sorted FEASIBLE set for $1 \leq v \leq s$.

(4) Complete Q as needed using a dependence-sensitive matrix completion algorithm and compute $T = Q^{-1}$

(5) Assign arbitrary memory layouts for arrays with temporal locality and arrays for which no optimal layout has been found

FIG. 2. Compiler algorithm for optimizing cache locality. This algorithm first tries to optimize references for temporal locality. If this is not possible, spatial locality in the innermost loop is attempted. When it fails, spatial locality in the outer loops is tried. The $Ker\{.\}$ notation denotes the null set of a vector or matrix; $\times$ denotes a matrix–vector or vector–vector multiply.

In Step (1), the algorithm attempts to find a $\vec{q_n}$ such that the left-hand side reference will have a temporal locality in the new innermost loop; that is, j_n will not appear in any subscript position (dimension) of this reference. If such a $\vec{q_n}$ is found we leave the memory layout of this array unspecified for now, and later in Step (5) we assign an arbitrary layout for that array. Since it exhibits temporal locality in the *innermost* loop, the spatial locality (and therefore memory layout) for it is of secondary importance. If there is more than one $\vec{q_n} = (q_1, q_2, ..., q_n)$, we select the one with the smallest $\gcd\{q_1, q_2, ..., q_n\}$.

If it is not possible to find a $\vec{q_n}$ to exploit temporal locality, the algorithm obtains spatial locality for this reference in Step (2). It tries to transform this

reference into a form $C(f_1, f_2, ..., f_m)$ where there exists an $1 \leqslant i \leqslant m$ such that f_i is an affine function of all loop indices with the coefficient of j_n being c (a nonzero constant), and each f_j $(i \neq j)$ is an affine function of $j_1, j_2, ..., j_{n-1}$ but not of j_n. In other words, j_n appears only in the ith subscript position In such a case the dimension i is the FCD for that array. Note that it is always possible to find such a $\vec{q_n}$. To see this, suppose that we would like to ensure $\vec{\ell_i^C} \times \vec{q_n} = c$ is satisfied. We can delete the ith row of $\mathscr{L}^C$ and select a $\vec{q_n}$ from the *Ker* set (i.e., the null set) of the reduced matrix. Then, this $\vec{q_n}$ gives us a constant c value when it is multiplied by $\vec{\ell_i^C}$. In practice, it is possible most of the time to find a $\vec{q_n}$ such that $c = 1$.

Having optimized the left-hand side reference and *having fixed* a $\vec{q_n}$, the algorithm then focuses on the right-hand side references and in Step (3) optimizes each of them in turn. For each right-hand side reference, it first checks (Step (3.1)) whether temporal locality exists for that reference under the $\vec{q_n}$ found in the previous step. If so, the algorithm proceeds to the next right-hand side array, otherwise it checks the reference for spatial locality and determines the memory layout (the FCD) of the associated array (Step (3.2)) using a method similar to the one used for the left-hand side reference. It is possible that the spatial locality cannot be exploited for this reference in the innermost loop. After this, in Step (3.4), the algorithm tries to exploit spatial locality in outer loops for all those right-hand side arrays for which neither spatial nor temporal locality was exploited in the innermost loop in Steps (3.1) thru (3.3). It computes a set of feasible vectors with associated frequencies, and the set is then sorted in decreasing order of frequency. Let s be the size of this set. The element v of this sorted set is assigned as column $n - v$ of the matrix Q, for v from 1 to s.

In Step (4) the inverse of the loop transformation matrix is completed using a *dependence-sensitive* matrix completion method (e.g., [32] or [6]), and in Step (5) the algorithm assigns arbitrary memory layouts for the arrays with temporal locality in the innermost loop and the arrays for which no optimal layout has been found.

The following points should be noted. First, our algorithm considers all m possible subscript positions (dimensions) as the potential FCD for a given array. Second, it should be noted that the algorithm first optimizes the left-hand side array. Although this is not strictly necessary, we found it useful in practice as the left-hand side array might be both read and written, whereas the other arrays are only read. If there are more than one left-hand side reference inside the loop nest, we start with the one that would be more frequently accessed at run-time. Such information can be obtained through profiling. Third, an important case occurs when an array is referenced more than once. If all of its references belong to the same *uniformly generated set* [17] (i.e., have the same array access matrix), then it is sufficient to consider only one of them. If, on the other hand, there are references to the same array with different access matrices, then a reasonable heuristic might be to select a *representative reference* for that array and use it in the optimization process. Our representative reference selection scheme is based on the *weight* of references. Essentially, the weight of a reference is the number of times it is accessed in a typical execution. This number can be estimated by multiplying the trip counts (the number of iterations) of the loops which enclose that reference. If the trip

counts are not available at compile-time, we use profiling to get this information. In practice, average trip count estimations are sufficient for a majority of applications. After the weight of each reference is obtained, the references are divided into (conformity) groups such that two references belong to the same group if they are conformant in the sense that they require the same FCD. It is easy to see that two references are conformant if *canonical forms* of the last columns of their array access matrices are equal. The canonical form of a column vector $\bar{x} = (x_1, x_2, ..., x_e)^T$ is $Canonical(\bar{x}) = (x_1/g, x_2/g, ..., x_e/g)^T$ where $g = \gcd\{x_1, x_2, ..., x_e\}$.

Thus, two references R_1 and R_2 belong to the same conformity group if the last columns of their access matrices, $\bar{x}_{R_1}$ and $\bar{x}_{R_2}$, satisfy

$$Canonical(\bar{x}_{R_1}) = Canonical(\bar{x}_{R_2}).$$

After determining the conformity groups, the compiler computes the *weight* of each conformity group. The weight of a conformity group is the sum of the weights of the references it contains. Assuming that a conformity group C_i contains references R_1, R_2, ..., and R_k, the weight of C_i can be computed as

$$weight(C_i) = \sum_{j=1}^{k} weight(R_j) = \sum_{j=1}^{k} \prod_{l} trip\ count(l),$$

where l iterates over the loops that enclose R_j.

Once the weights of the conformity groups are computed, our approach chooses a reference from a conformity group with the largest weight as *representative reference*. Then the compiler proceeds to optimize locality for this reference.

We assume that such a representative reference selection scheme is used in this paper whenever necessary.

4.2. Complexity

For the left-hand side reference, in the worst case $m+1$ equation systems are solved (one for temporal and m for spatial locality corresponding to each subscript). Assuming $n \geqslant m$ (where n is the depth of the loop nest), each system requires maximum $\Theta(n^3)$ time, giving a total cost of $\Theta(n^4)$ for the left-hand side reference. The cost of Step (3.3) is $\Theta(n^5 R)$ at the worst case since there are R references. The remaining steps do not change this complexity. It should be mentioned that in practice most of the steps of the algorithm are not executed at all.

4.3. Example

In this subsection, we illustrate how the locality optimization algorithm works by giving an example. Figure 3a shows the *ijk* matrix multiply routine. The array access matrices are as follows.

```
// A, B, C: column-major
DO i = 1, N
 DO j = 1, N
  DO k = 1, N
   C(i,j)+=A(i,k)*B(k,j)
  END DO
 END DO
END DO
```

(a)

```
// A, C: row-major
// B: column-major
DO u = 1, N
  DO v = 1, N
    DO w = 1, N
     C(u,v)+=A(u,w)*B(w,v)
    END DO
  END DO
END DO
```

(b)

```
//A, C: distributed by rows
//B: distributed by columns
DO u = 1, N
  DO v = 1, N
   transfer B(*,v)
   DO w = 1, N
    C(u,v)+=A(u,w)*B(w,v)
   END DO
  END DO
END DO
```

(c)

FIG. 3. (a) Original matrix multiply nest. The default layout is column-major. (b) Locality optimized version. Boldfaced references indicate that the associated arrays are row-major. Note that array C has temporal locality in the innermost loop whereas arrays A and B have spatial locality. (c) Parallelism optimized version. Arrays A and C are distributed by rows and array B by columns. Note that the communication due to array B is vectorized.

$$\mathscr{L}^C = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}, \qquad \mathscr{L}^A = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \qquad \text{and} \qquad \mathscr{L}^B = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}.$$

In the following we show only the successful trials.

In Step (1), from $\vec{q_n} \in Ker\{(1, 0, 0)\}$ and $\vec{q_n} \in Ker\{(0, 1, 0)\}$, we determine $\vec{q_n} = (0, 0, 1)^T$ as the last column of the inverse of the loop transformation matrix.

Then we move to Step (3) to optimize the right-hand side references. In Step (3.1) we see that $\vec{q_n} = (0, 0, 1)^T$ is in the *Ker* set of the rows of neither $\mathscr{L}^A$ nor $\mathscr{L}^B$.

In Step (3.2) we check for spatial locality of A and B in the innermost loop. Since

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \vec{q_n} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \qquad \text{and} \qquad \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \vec{q_n} = \begin{pmatrix} 1 \\ 0 \end{pmatrix},$$

we determine the FCDs for references to A and B as the second and first dimensions, respectively. That is, for good spatial locality, arrays A and B should have *row-major* and *column-major* memory layouts, respectively.

Afterward, in Step (4), we complete Q to identity matrix, taking into account the last column of it ($\vec{q_n}$) found above. Finally, in Step (5), we arbitrarily assign *row-major* layout for array C.

The transformed program is shown in Fig. 3b. We note that array C exhibits temporal locality whereas arrays A and B enjoy spatial locality, all in the innermost loop. To indicate the layout transformations, the references to arrays C and A are **boldfaced**.

As stated earlier, after finding FCDs, we determine a suitable (complete) memory layout. Note that determining an FCD in the two-dimensional case specifies a complete memory layout. For the higher-dimensional cases, we order the remaining layouts arbitrarily, though our approach can be extended to determine a second fastest changing dimension and so on. Once a suitable memory layout is determined, we implement this layout by representing it with a data transformation matrix [35] and transforming array references as well as array declarations accordingly. For example, in Fig. 3b the layout of array A is determined as row-major.

In order to implement this layout in a language whose default array layout is column-major (e.g., Fortran), we need to find a two-by-two data transformation matrix M such that

$$M\begin{pmatrix}1 & 0 & 0\\ 0 & 0 & 1\end{pmatrix}=\begin{pmatrix}x & x & 1\\ x & x & 0\end{pmatrix},$$

where x denotes a *don't-care* value. The process of detecting such a transformation matrix and code generation after that is quite mechanical and beyond the scope of this paper. We refer the interested reader to [35] for details. In this example, an *array transpose matrix* [35] will suffice: $M=\left(\begin{smallmatrix}0 & 1\\ 1 & 0\end{smallmatrix}\right)$.

Our algorithm can also optimize loop nests with very complex subscript expressions. Consider the loop nest shown in Fig. 4a. The array access matrices are as follows.

$$\mathscr{L}^A=\begin{pmatrix}1 & -1 & 0\\ -1 & 1 & 1\end{pmatrix},\qquad \mathscr{L}^B=\begin{pmatrix}-1 & 1 & 1\\ 0 & 0 & 1\end{pmatrix},\qquad \text{and}\qquad \mathscr{L}^C=\begin{pmatrix}1 & -1 & 0\\ 1 & -1 & 1\\ 1 & 0 & 1\end{pmatrix}.$$

In Step (1), from $\vec{q_n} \in Ker\{(1, -1, 0)\}$ and $\vec{q_n} \in Ker\{(-1, 1, 1)\}$, we determine $\vec{q_n} = (1, 1, 0)^T$ as the last column of the inverse of the loop transformation matrix.

Then we move to Step (3) to optimize the right-hand side references. In Step (3.1) we see that

$$\begin{pmatrix}-1 & 1 & 1\\ 0 & 0 & 1\end{pmatrix}\vec{q_n}=\begin{pmatrix}0\\ 0\end{pmatrix},$$

meaning that it is possible to exploit temporal reuse for array B in the innermost loop.

Unfortunately, for array C, exploiting temporal locality is not possible as a row of C, namely (1, 0, 1), is not in the null set of $\vec{q_n}$.

```
DO i = li, ui
  DO j = lj, uj
    DO k = lk, uk
      A(i-j,j+k-i) = B(j+k-i,k) + C(i-j,i-j+k,i+k)
    END DO
  END DO
END DO
```

(a)

```
//C: row-major; A,B: column-major
DO u = lu, uu
  DO v = lv, uv
    DO w = lw, uw
      A(u,v-u) = B(v-u,v) + C(u,u+v,u+w)
    END DO
  END DO
END DO
```

(b)

FIG. 4. (a) A loop nest with complex access pattern. (b) Locality optimized code. Note that in this optimized code for two out of three references we have temporal reuse, and for the third reference we have spatial reuse, all in the innermost loop. For a loop i, li and ui denote the lower and upper bounds, respectively.

However, in Step (3.2) we check for spatial locality of C in the innermost loop. Since

$$\begin{pmatrix} 1 & -1 & 0 \\ 1 & -1 & 1 \\ 1 & 0 & 1 \end{pmatrix} \vec{q_n} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix},$$

we determine the last dimension of array C as the FCD. A layout that matches this FCD is row-major.

Afterward, in Step (4), we complete Q as

$$\begin{pmatrix} 1 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix},$$

and in Step (5), we arbitrarily assign *column-major* layouts for arrays A and B.

The transformed code is shown in Fig. 4b. We note that for two out of three references we have temporal reuse and for the third reference we have spatial reuse, all in the innermost loop.

5. ALGORITHM FOR ENHANCING MEMORY LOCALITY AND PARALLELISM

In this section, we present a technique which considers loop transformations to optimize parallelism and communication in message-passing machines. Specifically, the algorithm presented here transforms a loop nest such that

(1) the outermost transformed loop is distributed over the processors,

(2) the memory locality is optimized by distributing arrays across memories of the processors, and

(3) communication is performed in large chunks, and it is optimized such that all nonlocal data are transferred to respective local memories before the execution of the innermost loop.

We note that the algorithm can also be used for the shared-memory UMA and NUMA architectures. For the NUMA case (depending on the specific architecture), the algorithm performs Steps (1) and (2) listed above, whereas for the UMA case it performs only Step (1).

5.1. Explanation

The overall algorithm is presented in Fig. 5. As before, let $j_1, j_2, ..., j_n$ be the loop indices of the *transformed* loops from outermost position.

In Step (1) we try to determine a distributed dimension i for the left-hand side array such that no communication will incur and the outermost.loop parallelism is

- **INPUT:** $\mathcal{L}^C$ for the left hand side (LHS) array reference. $\mathcal{L}^{A_r}$ for the right hand side (RHS) array references where $1 \leq r \leq R$
- **OUTPUT:** A loop transformation matrix T and a distributed dimension (DD) for each array
- **PROCEDURE:**

(1) // distribute LHS array to obtain outermost parallelism

- try to find a $Q = [\vec{q_1}, \vec{q_2}, \cdots, \vec{q_n}]$ ($\vec{q_j}$ is the j^{th} column of matrix Q) and an i such that
  - $\vec{\ell_i^C} \times \vec{q_1} = c$ where c is a non-zero integer constant, and for each j $(2 \leq j \leq n)$ $\vec{\ell_i^C} \times \vec{q_j} = 0$. Note that in doing this check only those elements of Q that are essential in satisfying the constraints will have defined values; the rest are don't-care denoted by x.
  - if a solution exists for i, mark i as the DD for array C; save the computed matrix Q (along with the don't-care entries).

(2) // optimize each RHS array reference in turn; the matrix Q (including don't-care entries) is used here.

$r \leftarrow 1$

while $(r \leq R)$ {

(2.0) Initialize $Q^r = [\vec{q_1^r}, \vec{q_2^r}, \cdots, \vec{q_n^r}]$ ($\vec{q_j^r}$ is the j^{th} column of matrix Q^r) to the matrix of zeros and initialize the distributed dimension DD of r to -1.

(2.1) // check whether communication free distribution is possible

if there is an l such that $\vec{\ell_l^{A_r}} = \vec{\ell_i^C}$ distribute A_r along l^{th} dimension, set all the elements of Q^r to don't-care, and goto Step (2.3).

(2.2) // try to vectorize communication out of the innermost loop

- check if there is an s for which there is a non-zero constant c such that $\vec{\ell_s^{A_r}} \times \vec{q_n^r} = c$, $\vec{\ell_s^{A_r}} \times \vec{q_j^r} = 0$ for $1 \leq j \leq (n-1)$, and for each $k \neq s$, $\vec{\ell_k^{A_r}} \times \vec{q_n^r} = 0$.
- if a solution exists and if for all i, j either $Q[i,j] = Q^r[i,j]$ or at least one of the two entries $Q[i,j]$ and $Q^r[i,j]$ is a dont-care, then mark s as the *non-DD* for array A_r, and select a $k \neq s$ as DD. If either there is no solution or if some non-don't-care entries of Q and Q^r differ, then set Q^r to the zero-matrix and set DD of r to -1. Goto Step (2.3).

(2.3) increment r

}

(3) // simple conflict resolution

Define two matrices (whose entries could be don't-cares) Q^1 and Q^2 to be compatible if for all i, j at least one of the two entries $Q^1[i,j]$ and $Q^2[i,j]$ is a don't-care or $Q^1[i,j] = Q^2[i,j]$. From among all the matrices Q^r which are not zero matrices from Step (2.2), find the sets of mutually compatible matrices. Let there be f such sets $P_1, \cdots, P_f$. Note that there is a single Q matrix associated with each set.

Arrange the sets $P_1, \cdots, P_f$ in decreasing order of the size of the set, i.e., the number of compatible matrices in the sets. Let the matrix associated with the first set be Q^*.

For each matrix $Q^r (1 \leq r \leq R)$ that is not the zero matrix, if Q^* and Q^r are compatible, then store the DD of this reference that was computed in either Step (2.1) or Step (2.2) (these are mutually exclusive).

(4) For all i, j assign $Q[i,j]$ as follows: If $Q[i,j]$ is not dont-care, leave it unchanged; if $Q[i,j]$ is a dont-care, then copy $Q^*[i,j]$ into $Q[i,j]$. Complete Q using a completion algorithm and compute $T = Q^{-1}$

FIG. 5. Compiler algorithm for optimizing memory locality and parallelism and minimizing communication. For each reference, the algorithm first attempts to obtain communication-free outermost loop parallelism. If this is not possible, it tries to vectorize the communication out of the innermost loop.

obtained. This can be done if the reference for the left-hand side array can be transformed into the form $C(f_1, f_2, ..., f_m)$ where there exists an $1 \leqslant i \leqslant m$ such that f_i is an affine function of only j_1 with the coefficient of c, and each f_j $(i \neq j)$ is an affine function of $j_1, j_2, ..., j_n$. In other words, in the ith dimension only j_1 will appear. When this gets satisfied, array C can be distributed along the ith dimension (i.e., i will be the DD), and at the same time the outermost loop is parallelized.

In Step (2) we try to optimize communication for each right-hand side array in turn. For a given right-hand side array A_r if we can find a row l such that $\vec{\ell_l^{A_r}} = \vec{\ell_i^C}$, we can distribute A_r along the lth dimension without incurring any

communication due to this array reference. This possibility is checked in Step (2.1) in the algorithm. If all the right-hand side references have a row in the array access matrix identical to that of the left-hand side reference, then the entire loop can be distributed along that dimension and there is no communication [39, 40].

If a communication-free distribution is *not* possible, in Step (2.2) we attempt to vectorize the communication out of the innermost loop. It is possible that the transformation matrix required in order to move the communication due to one reference may conflict with that required for another. Therefore, in this step, we compute the required transformation matrix Q^r for each right-hand side reference r independently, i.e., as if reference r were the only right-hand side reference. This can be done if we can transform the right-hand side reference to array A_r to the form $A_r(f_1, f_2, ..., f_m)$ where there exists a dimension s such that f_s is an affine function of only j_n and for each $k \neq s$, f_k is an affine function of $j_1, j_2, ..., j_{n-1}$ but not of j_n. If this is satisfied, it is easy to see that the communication due to this reference can be vectorized out of the innermost loop j_n, and the array can be distributed along any dimension other than s. It is possible that the loop transformation that is needed to vectorize communication for this reference may conflict with that needed for another reference. Therefore, all the loop transformation matrices computed in this step for each reference are saved. The algorithm in Fig. 5 only shows vectorization of communication from the innermost loop. Note that it is possible to repeat Step (2.2) if desired, this time attempting to move the communication out of the second innermost loop (j_{n-1}). This process terminates when a loop is encountered outside of which the communication cannot be moved.

Step (3) shows a simple conflict resolution scheme. Other approaches such as those based on profiling can be used here. Recall that in Step (2), for each reference r, the inverse of the associated transformation matrix Q^r is either the zero matrix (that is a matrix in which all entries are zeros, i.e., there was no solution in Steps (2.1) and (2.2)) or some entries of Q^r have been computed. We define two matrices (whose entries could be don't-cares) Q^1 and Q^2 to be *compatible* if for all i, j either $Q^1[i, j] = Q^2[i, j]$ or at least one of the two entries $Q^1[i, j]$ and $Q^2[i, j]$ is a don't-care. In this step, we find the sets of mutually compatible matrices from among all the matrices Q^r which were defined in Step (2.2). We use a simple greedy heuristic to compute the sets. Let there be f such sets $P_1, ..., P_f$. Note that there is a single Q matrix associated with each set. We assume that the sets $P_1, ..., P_f$ are arranged in decreasing order of their sizes, i.e., the number of compatible matrices in the sets. The algorithm in Fig. 5 uses the matrix Q^* associated with the set that has the maximum number of mutually compatible references, copies this matrix to Q, and completes this matrix as needed (Step (4)) and then uses its inverse as the desired transformation matrix.

Of course, the transformation matrix should be nonsingular and must satisfy the data dependences. It is possible that the matrix Q^* associated with the first set (of the sorted sets $P_1, ..., P_f$) violates dependence constraints or is singular. In such a case, we scan the matrices associated with the P sets in order and use the first matrix that satisfies the dependence constraints and is nonsingular. Notice that the algorithm adheres to the *owner-computes rule* [4, 20, 51] by performing a left-hand

side based data distribution across the processors. The details of the algorithm can be found in Refs. [39] and [40].

5.2. Complexity

For the left-hand side reference, in the worst case m equation systems are solved (one or each candidate DD). Assuming $n \geqslant m$ each system requires maximum $\Theta(n^3)$, giving a total cost of $\Theta(n^4)$ for the left-hand side reference. The cost of a right-hand side reference is at the worst case $\Theta(mn^3 + mn^3)$, where first term is for checking communication-free distribution and the second term is for vectorization. Thus, the overall cost of the algorithm is $\Theta(n^4R)$. Notice that if we attempt to vectorize communication above the outer loops as well, the overall complexity becomes $\Theta(n^5R)$.

5.3. Example

To illustrate the technique, we consider again the *ijk* matrix-multiply nest shown in Fig. 3a.

The array access matrices are as follows.

$$\mathscr{L}^C = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}, \qquad \mathscr{L}^A = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \qquad \text{and} \qquad \mathscr{L}^B = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}.$$

As before, we only show the successful trials.

In Step (1) $\mathscr{L}^C \times Q = \left(\begin{smallmatrix} 1 & 0 & 0 \\ x & x & x \end{smallmatrix}\right)$. Therefore $q_{11} = 1$, $q_{12} = 0$, and $q_{13} = 0$.

Since $\vec{\ell}^A_1 = \vec{\ell}^C_1$, A can be distributed along the first dimension as well.

Array B, however, has no common row with array C; therefore we try to vectorize communication for it. $\mathscr{L}^B \times Q = \left(\begin{smallmatrix} 0 & 0 & 1 \\ x & x & 0 \end{smallmatrix}\right)$. Therefore $q_{31} = q_{32} = q_{23} = 0$ and $q_{33} = 1$. Thus, array B can be distributed along the second dimension as the new innermost loop index appears only in the first dimension.

In these equations, x denotes to a *don't-care* entry.

At this point, we have

$$Q = \begin{pmatrix} 1 & 0 & 0 \\ q_{21} & q_{22} & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

The remaining entries should be selected such that Q should be of full-rank and no data dependence is violated. In this case the compiler can set $q_{21} = 0$ and $q_{22} = 1$ using a completion algorithm.

This results in the identity matrix meaning that no transformation is needed. Arrays A and C are distributed by *rows* and array B by *columns*. The resulting program with the data transfer call is shown in Fig. 3c. Note that the communication for array B is performed *outside* the innermost loop; that is, it is vectorized. The notation $B(*, v)$ in Fig. 3c denotes the set of elements $\{(w, v) : 1 \leqslant w \leqslant N\}$.

```
DO i = 1, N
  DO j = i, MIN(i+2*b-2,N)
    DO k = MAX(i-b+1,j-b+1,1), MIN(i+b-1,j+b-1,N)
      C(i,j-i+1) = C(i,j-i+1) + A(k,i-k+b) * B(k,j-k+b) + A(k,j-k+b) * B(k,i-k+b)
    END DO
  END DO
END DO
```

(a)

```
//A,B,C: column-major
DO u = 1, N
  DO v = u, MIN(u+2*b-2,N)
    transfer A(*,v+b)
    transfer A(*,u+v+b)
    transfer B(*,v+b)
    transfer B(*,u+v+b)
    DO w = MAX(u-b+1,v-b+1,1), MIN(u+b-1,v+b-1,N)
      C(u+w+1,u) = C(u+v+1,u) + A(w,v+b) * B(w,u+v+b) + A(w,u+v+b) * B(w,v+b)
    END DO
  END DO
END DO
```

(b)

FIG. 6. (a) SYR2K loop nest from BLAS. Notice that the subscript expressions are quite complex. (b) Optimized version of (a) obtained by the algorithm given in Fig. 5. All arrays are distributed along columns and all communications are vectorized above the innermost loop.

This algorithm can optimize loop nests with quite complex subscript functions as well. As an example, consider the SYR2K nest shown in Fig. 6a from BLAS [16]. Figure 6b shows the optimized version of this loop nest. The resulting loop transformation matrix is

$$T = \begin{pmatrix} 1 & -1 & 0 \\ -1 & 0 & 1 \\ 0 & 0 & -1 \end{pmatrix}.$$

The arrays A, B, and C are all distributed by *columns*; and all communications are vectorized before the innermost transformed loop.

6. UNIFIED ALGORITHM

This section presents a unified algorithm which combines the characteristics of the two algorithms presented in the previous two sections (Sections 4 and 5). There are many factors involved in this case and the algorithm presented here is a *greedy*

heuristic in the sense that it handles the array references one by one and uses the decisions taken during optimization of a reference in optimization of the others.

Specifically, given a loop nest, our unified algorithm

(1) transforms the nest such that the outermost transformed loop can be run in parallel across the processors,

(2) optimizes memory locality by distributing each array across the memories of the processor,

(3) optimizes interprocessor communication by vectorizing it whenever possible, and

(4) optimizes cache locality by assigning an appropriate memory layout for each array and by transforming the iteration space.

For distributed-memory multicomputers, all four steps can be applied. For the shared-memory NUMA case, depending on the specific machine, Steps (1), (2), and (4) are attempted; and for the UMA case, only Steps (1) and (4) can be performed.

6.1. Explanation

In this subsection, we give the rationale behind the unified algorithm. As before, for simplicity, we assume there is only one uniformly generated set per array. If this is not the case, the representative reference selection scheme discussed earlier should be used.

At the heart of our unified optimization algorithm is determining for each array an array access matrix suitable for *both* parallelism and locality. One might think of an *ideal* access matrix as the one which helps to exploit *temporal* locality in the innermost loop *and* produces outermost loop parallelism without incurring interprocessor communication. Such an ideal access matrix is of the form

$$\begin{pmatrix} x & x & \cdots & \cdots & x & 0 \\ x & x & \cdots & \cdots & x & 0 \\ 1 & 0 & \cdots & \cdots & 0 & 0 \\ x & x & \cdots & \cdots & x & 0 \\ x & x & \cdots & \cdots & x & 0 \\ x & x & \cdots & \cdots & x & 0 \end{pmatrix}_{m \times n}.$$

In this matrix x denotes to a *don't-care* entry. Notice that the matrix has a zero last column which guarantees temporal locality in the innermost loop and a row of the form $(1, 0, ..., 0, 0)$ which implies outermost loop parallelism. Supposing that this row is the ith row where $1 \leqslant i \leqslant m$, the array in question can be distributed across the memories of the processors along the ith dimension (i.e., i is the DD). In addition, during the entire execution of the innermost loop the elements of this array can be held in a register. For such an array, the memory layout is of secondary importance as it exhibits temporal locality in the innermost loop.

Another ideal access matrix is the one which helps to exploit *spatial* locality in the innermost loop and ensures outermost loop parallelism without communication. Such an access matrix is of the following form:

$$\begin{pmatrix} x & x & \cdots & \cdots & x & 0 \\ x & x & \cdots & \cdots & x & 0 \\ 1 & 0 & \cdots & \cdots & 0 & 0 \\ x & x & \cdots & \cdots & x & 0 \\ x & x & \cdots & \cdots & x & 1 \\ x & x & \cdots & \cdots & x & 0 \end{pmatrix}_{m \times n}.$$

Assuming that (1, 0, ..., 0, 0) is the *i*th row and the nonzero entry in the last column occurs in the *j*th row, we have *i* as the DD and *j* as the FCD for the array in question. In other words, we can distribute the array along the *i*th dimension and store it in memory such that the *j*th dimension will be the fastest changing dimension. It should be emphasized that the compiler should try all possible *m* values for *i* and *j*. It should also be noted that $i \neq j$ as the same dimension cannot be a DD and FCD at the same time.

Our algorithm tries to achieve one of these ideal access matrices for as many references as possible. For the remaining references, the algorithm considers the next most desirable access matrices. The selection of those matrices depends on whether parallelism is favored over locality or vice versa. For example, we can insist on exploiting spatial locality in the innermost loop, but can accept some communication vectorized out of the innermost loop. Obviously, such an approach favors locality over parallelism. Alternatively, we can insist on communication-free outermost loop parallelism, but can sacrifice some locality by, let's say, exploiting spatial locality in the second innermost loop instead of the innermost loop. Such an approach favors parallelism over cache locality.

In Table 1 we show the alternative access matrices that our current implementation tries for a reference to a two-dimensional array enclosed by a three-dimensional loop nest.

The upper-left quadrant (UL) of the table presents the desired access matrices for obtaining outermost loop parallelism as well as exploiting temporal locality in the innermost loop. The upper-right (UR) quadrant, on the other hand, shows the access matrices for obtaining outermost loop parallelism and exploiting spatial locality in the innermost loop. The lower-left (LL) corner of the table gives the access matrices that will be used when the locality is favored over parallelism. In that case, the spatial locality is exploited in the innermost loop, but the communication incurred by the reference is vectorized out of the innermost loop. Finally, the lower-right (LR) quadrant shows the access matrices when parallelism is favored over locality. In this case the outermost loop parallelism is achieved but the spatial locality is exploited in the second innermost loop. Depending on the target architecture these four quadrants can be tried in different orders. In our current implementation, for the distributed-memory message-passing machines we

TABLE 1

Desired Array Access Matrices for a Two-Dimensional Array Enclosed in a Three-Deep Loop Nest

group (1)	group (2)
$\begin{pmatrix} 1 & 0 & 0 \\ x & x & 0 \end{pmatrix}, \begin{pmatrix} x & x & 0 \\ 1 & 0 & 0 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 & 0 \\ x & x & 1 \end{pmatrix}, \begin{pmatrix} x & x & 1 \\ 1 & 0 & 0 \end{pmatrix}$
group (3)	**group 4**
$\begin{pmatrix} x & x & 0 \\ 0 & 0 & 1 \end{pmatrix}, \begin{pmatrix} 0 & 0 & 1 \\ x & x & 0 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 & 0 \\ x & 1 & x \end{pmatrix}, \begin{pmatrix} x & 1 & x \\ 1 & 0 & 0 \end{pmatrix}$

Note. An x denotes a *don't-care* entry. The four quadrants in this table correspond to the groups: upper-left quadrant (group (1)) temporal locality in the innermost loop, no communication; upper-right quadrant (group (2)) spatial locality in the innermost loop, no communication; lower-left quadrant (group (3)) spatial locality in the innermost loop, vectorized communication; and lower-right quadrant (group (4)) spatial locality in the *second* innermost loop, no communication.

adopt UL, UR, LR, and LL in that order. For the distributed-shared-memory machines, on the other hand, we try UL, UR, LL, and LR in that order. Given accurate cost measurements, it is possible to select an appropriate order for the trials.

Essentially, starting from the left-hand side reference, the algorithm handles each array reference in turn and tries to find a suitable access matrix for satisfying locality and parallelism. In doing so, it fills *some* elements of Q as well. Then it proceeds with the next reference and tries to fill out the remaining elements of Q and so on. After all the references have been handled, if there are still unspecified elements of Q, they are filled out using a modified version of the completion algorithms proposed by Li [32] and Bik and Wijshoff [6].

An important question is how many access matrices should be tried for a given array reference. In theory, an approach can perform an exhaustive search over all possible access matrices (not just the ones belonging to one of the four sets mentioned above). Such an approach will also try access matrices such as those, let's say, which exploit the spatial locality in the third innermost loop and vectorize the communication in the second innermost loop, etc. Our experience, albeit limited, shows that for most regular scientific codes encountered in practice examining four sets of access matrices mentioned above is sufficient. These four sets correspond to four quadrants shown in Table 1. In the following we will call each quadrant a *group*. Notice that given an array dimensionality m, and a loop depth n, it is always possible to generate these four groups which contain desired access matrices corresponding to (1) temporal locality, no communication; (2) spatial locality, no communication; (3) spatial locality, vectorized communication; and (4) spatial locality (in the second innermost loop), no communication, respectively.

- **INPUT:**

  $\mathcal{L}^{A_0}$ for the left hand side (LHS) array reference

  $\mathcal{L}^{A_r}$ for the right hand side (RHS) array references where $1 \leq r \leq R$

- **OUTPUT:**

  A loop transformation matrix T and a distributed dimension (DD) and a fastest changing dimension (FCD) for each array

- **PROCEDURE:**

(1) // optimize each reference in turn
for $r = 0$ to R { // iterate over the references starting with the LHS
done $\leftarrow$ FALSE
$i \leftarrow 1$
while (($i \leq 4$) and done == FALSE) { // iterate over the four groups
$j \leftarrow 1$
while (($j \leq m(m-1)$ and done == FALSE) { // iterate over the access matrices
initialize $Q^r = [\vec{q_1^r}, \vec{q_2^r}, \cdots, \vec{q_n^r}]$ to the zero matrix, DD of r to -1, and the FCD of r to -1.
Check if there is a solution Q^r such that $\mathcal{L}^{A_r} \times Q^r = R_{ij}$;
If a solution exists, mark i as the DD for reference r, j as the FCD for reference r,
save Q^r, and set done = TRUE.
increment j
}
increment i
}
}

(2) // simple conflict resolution

- We use a simple conflict resolution scheme similar to that used in Step (3) of the algorithm in Figure 5. See Figure 5 for the definitions. Each matrix Q^r for $0 \leq r \leq R$ is either the zero matrix or has a value computed in Step (1).
- Compute the sets of compatible matrices using a greedy strategy. Let there be g such compatible sets which are sorted in decreasing order of their sizes (i.e., the number of mutually compatible matrices that belong to each set) as $P_1, \cdots, P_g$.
- Let Q^* be the matrix associated with the set P_1. Copy Q^* into Q.
- For each $Q^r (1 \leq r \leq R)$ computed in Step (1) (i.e., for each Q^r that is not the zero matrix), if Q^* and Q^r are compatible, then store the DD and FCD of this reference that was computed.

(3) complete Q using a completion algorithm and compute $T = Q^{-1}$

(4) assign arbitrary memory layouts for arrays with temporal locality

FIG. 7. Sketch of the unified compiler algorithm for optimizing cache locality, memory locality, parallelism, and minimizing communication. Note that for each reference we check all possible access matrices from all four groups.

The sketch of our unified algorithm is given in Fig. 7. We assume that four groups are denoted by $i = 1, 2, 3, 4$, respectively. Also assume that R_{ij} refers to the jth access matrix from group i. For example in Table 1,

$$R_{11} = \begin{pmatrix} 1 & 0 & 0 \\ x & x & 0 \end{pmatrix} \quad \text{and} \quad R_{32} = \begin{pmatrix} 0 & 0 & 1 \\ x & x & 0 \end{pmatrix}.$$

Notice that $1 \leqslant j \leqslant m(m-1)$ for an m-dimensional array. The reason is that we need to select a dimension for DD and another dimension for FCD.

In Step (1) we have a three-deep loop nest. For each reference we try each possible access matrix from four groups. When an acceptable access matrix is found, we store the matrix associated with this reference and move on to the next array. The reason we do this is to be be able to detect and resolve conflicts. In this step, the associated matrix Q^r could either be undefined or has some values set.

Since there may be conflicting requirements on the transformation matrix from different references, we use a conflict resolution scheme in Step (2). This scheme is similar in spirit to the conflict resolution step used in Fig. 5. A key difference between the two algorithms is that the left-hand side array gets a special treatment in the algorithm in Fig. 5, whereas that is not the case in the combined algorithm in Fig. 7. Accordingly, the conflict resolution step takes all the computed Q^r matrices (both left- and right-hand side) into account. (See the discussion in Section 4.1 for a description of the conflict resolution scheme.)

Finally, in Step (3) we complete Q, and in Step (4) we assign arbitrary layouts for the references with temporal locality in the innermost loop.

Our current approach precomputes the desired access matrices of all four groups of access matrices for commonly used array dimensions (e.g., $m = 2, 3, 4$) and loop depths (e.g., $n = 2, 3, 4$) and stores them in tables for the algorithm shown in Fig. 7.

6.2. Complexity

The complexity of the approach is the product of the trip counts of the loops shown in Fig. 7 and $\Theta(n^3)$; that is, $\Theta(n^4 R)$. Notice that if we consider all levels of spatial reuse and vectorization above the outer loops, the complexity would be higher.

6.3. Example

We consider the original matrix-multiply nest of Fig. 3a once more. The array access matrices are

$$\mathscr{L}^C = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}, \qquad \mathscr{L}^A = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \qquad \text{and} \qquad \mathscr{L}^B = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}.$$

Again, we show only the successful trials.

For the left-hand side reference, we can use

$$R_{11} = \begin{pmatrix} 1 & 0 & 0 \\ x & x & 0 \end{pmatrix} \cdot \mathscr{L}^C \times Q = \begin{pmatrix} 1 & 0 & 0 \\ x & x & 0 \end{pmatrix} \Rightarrow Q = \begin{pmatrix} 1 & 0 & 0 \\ x & x & 0 \\ x & x & x \end{pmatrix}.$$

This means we can exploit temporal locality in the innermost loop and can have outermost loop parallelism by distributing the array row-wise across the memories of the processors.

For array A, we can use

$$R_{21} = \begin{pmatrix} 1 & 0 & 0 \\ x & x & 1 \end{pmatrix} \cdot \mathscr{L}^A \times Q = \begin{pmatrix} 1 & 0 & 0 \\ x & x & 1 \end{pmatrix} \Rightarrow Q = \begin{pmatrix} 1 & 0 & 0 \\ x & x & 0 \\ x & x & 1 \end{pmatrix}.$$

In this case, the FCD for array A is two; that is, the array should be stored as row-major. In addition, due to row (1, 0, 0) in the transformed access matrix, there will be no communication for that reference with the outermost loop parallelism.

For array B, we can use

$$R_{21} = \begin{pmatrix} 0 & 0 & 1 \\ x & x & 0 \end{pmatrix} \cdot \mathscr{L}^A \times Q = \begin{pmatrix} 0 & 0 & 1 \\ x & x & 0 \end{pmatrix} \Rightarrow Q = \begin{pmatrix} 1 & 0 & 0 \\ x & x & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

Thus, array B will not have outermost loop parallelism, but its communication will be vectorized out of the innermost loop. In addition, it will have a column-major memory layout in memory.

In Step (3), we set $q_{21} = 0$ and $q_{22} = 1$ reaching the identity matrix. Finally, in Step (4), we assign row-major memory layout to array C. The resulting program is the same as shown in Fig. 3c with appropriate memory layouts.

7. GLOBAL OPTIMIZATION PROBLEM

The impact of data layout optimizations is global in the sense that when memory layout for an array is modified this new layout should be applied for all references to that array in the entire program. In some cases, this global effect can cause problems. These problems can be handled to a certain extent by applying iteration space transformations for the loop nests that are suffering from negative impact of the layout transformation. In other words, global impact of data layout transformations can be resolved locally by iteration space transformations.

In this section, we concentrate on the global locality optimization problem; that is, optimizing a number of consecutive loop nests simultaneously. In fact, we will handle a subproblem, namely optimizing cache locality across a number of loop nests. The other part of the global problem, optimizing data distribution across processors in multiple nests, has been handled in the literature (see, for example, [43, 21, 18, 36, 26, or 3]) and we do not discuss that problem here. Although the algorithm to be presented in this section can be modified to incorporate optimal global data distribution detection as well, for the sake of clarity we assume in this section that a suitable data distribution will be always available after the global algorithm discussed here has run. In a future work, we plan on integrating the global algorithm to be presented here with one of the techniques proposed by previous authors on automatic data distribution such as Anderson *et al.* [2], Kremer [28], and Gupta and Banerjee [21].

7.1. Local Candidates

In previous sections we tried to determine locally optimal memory layouts given a loop nest. When working on a global setting which comprises a number of loop nests, however, a local suboptimal solution may be globally optimal or vice versa. This implies that it may *not* be a good idea to consider only locally optimal memory layouts during global optimization process. In other words, we need to

consider a number of local alternatives per nest in the global optimization (which includes the best local alternative, of course). The important question now is *how* to select those local alternatives. As the reader would recall, the algorithm in Fig. 2 returns only a single solution which consists of memory layouts for all arrays. We call such a solution a *local combination* or a *local alternative*. The number of local alternatives can be increased in at least two ways:

(1) In Step (5) of the algorithm in Fig. 2, instead of assigning a single arbitrary layout for the references with temporal locality in the innermost loop, we can consider all possible memory layouts for such references; that is, for an m-dimensional array with temporal locality in the innermost loop we can consider all m possible dimensions as the FCD; or

(2) We can adopt an exhaustive search to find local alternatives based on the layout of the left-hand side array. In other words, for the left-hand side array we can try all $m+1$ alternatives which exploit either temporal locality in the innermost loop (one alternative) or exploit spatial locality in the innermost loop (m alternatives). After that, for each such alternative we attempt to optimize all right-hand side references. This approach requires a minor modification to the algorithm given in Fig. 2. Specifically, Step (3) in Fig. 2 should be embedded within Steps (1) and (2) so that for each alternative solution for the left-hand side reference we can consider all the right-hand side references.

We have chosen to implement the second approach, because it puts an upper bound to the number of local alternatives: We can have at most $m+1$ local alternatives (one for temporal locality and m for spatial locality with the FCD ranging from 1 to m). Notice that each possible local alternative is also associated with an accompanying loop transformation. Henceforth, we refer to this modified loop-level locality optimization approach as *local*. Throughout this section, we assume that *local* is run for each individual loop nest in the program, and all local alternatives and associated loop transformations are determined.

It is important to emphasize that although we can have $m+1$ local alternatives at most, this number can be reduced in general using an aggressive approach. For example, given a local alternative and associated loop transformation the cache miss rates can be estimated using the approach proposed by Sarkar *et al.* [41]. The local alternatives whose estimated miss rates are higher than a threshold can be eliminated from further consideration.

Alternatively, the number of local alternatives can be further increased if we consider a number of alternative loop transformations to obtain the same FCD for a given array.

In the explanation which follows we assume that an arbitrary number of alternatives can be returned by *local* (less than, equal to, or greater than $m+1$). In the run-time experiments, however, we considered exactly $m+1$ local alternatives per nest.

7.2. General Problem

Let $\{\mathcal{N}_1, \mathcal{N}_2, ..., \mathcal{N}_n\}$ denote the different loop nests in the program and $\{\mathcal{A}_1, \mathcal{A}_2, ..., \mathcal{A}_m\}$ denote the different arrays. In general each nest can access a

subset of these arrays. We first present a sketch of a proof which shows that the problem of finding global memory layouts (for all arrays) which satisfy all the (transformed) nests is NP-complete [19] even for the restricted case where only row-major (r-m) and column-major (c-m) memory layouts are considered.

When *local* is run for each nest in the program, we obtain local alternatives similar to those shown in Table 2a. For example, nest $\mathcal{N}_1$ accesses three arrays and *local* returns two local alternatives for that nest. We define the number of entries in the table as the size of the problem.

First, the problem belongs to the NP class [19]; this is because a nondeterministic algorithm need only guess a solution and check in polynomial time whether or not it satisfies all the nests. Next, we reduce the *satisfiability* problem [19] to our problem as follows: A given formulation is transformed to multiplications of sums (a polynomial-time operation). After that, each multiplicative term is associated with a nest, and each subterm (clause) in a multiplicative term is associated with a layout combination. With each logical variable x we associate an

TABLE 2

Local Layout Assignments for Different Examples

	$\mathcal{A}_1$	$\mathcal{A}_2$	$\mathcal{A}_3$	$\mathcal{A}_4$	$\mathcal{A}_5$
			(a)		
$\mathcal{N}_1$	r-m	r-m	r-m		
	r-m	c-m	c-m		
$\mathcal{N}_2$		c-m			c-m
		r-m			c-m
$\mathcal{N}_3$	c-m		c-m	r-m	
			(b)		
$\mathcal{N}_1$	r-m	c-m	c-m		
$\mathcal{N}_2$		c-m	c-m	r-m	r-m
$\mathcal{N}_3$		r-m	r-m	r-m	
			(c)		
$\mathcal{N}_1$	r-m	c-m	c-m		
	r-m	r-m	c-m		
$\mathcal{N}_2$		c-m	c-m	r-m	r-m
$\mathcal{N}_3$		r-m	r-m	r-m	
		r-m	c-m	r-m	
		c-m	c-m	c-m	

Note. A nonempty entry in $(\mathcal{N}_i, \mathcal{A}_j)$ means that the nest $\mathcal{N}_i$ accesses the array $\mathcal{A}_j$. For a given nest $\mathcal{N}_i$, each row represents a local alternative. r-m denotes row-major and c-m denotes column-major.

array X. If the logical variable appears itself, we assign c-m layout for X; if $\bar{x}$ complement of x) appears, we assign r-m layout for X.

For example, the layout assignments shown in Table 2a correspond to the following formulation where a_i and $\overline{a_i}$ are the logical variables associated with array $\mathscr{A}_i$.

$$(\overline{a_1}\,\overline{a_2}\,\overline{a_3} + \overline{a_1}a_2a_3) \cdot (a_2a_5 + \overline{a_2}\,a_5) \cdot (a_1a_3\overline{a_4})$$

There might be some special cases to handle, though. For example, after obtaining the desired form, a multiplicative term can contain expressions such as in $(ac + b\bar{c})$ which does not have all the variables. This expression should be transformed to $(a(b + \bar{b})\, c + (a + \bar{a})\, b\bar{c}) = abc + a\bar{b}c + ab\bar{c} + \bar{a}b\bar{c}$ so that each subterm contains logical variables a, b, and c or complements of them.

It is easy to see that the formulation is satisfied *if and only if* there is a layout assignment that satisfies all the nests. Since the reduction can be achieved in polynomial time, the problem is NP-hard; and since it belongs to the class NP as well, it is NP-complete. Since this problem is a restricted version of the most general problem of finding suitable layout assignments such that the value of a cost function will be $\leqslant k$, we argue that the general problem is also NP-complete. Note that in our restricted version, $k = 0$.

7.3. A Heuristic for Detecting Memory Layouts

Given that even the restricted form of the global layout problem is NP-complete, we search for a near-optimal solution with polynomial time which is good enough in practice. Let $LL^{\ell}_{\mathscr{N}}(\mathscr{A})$ be a local layout for an array $\mathscr{A}$ in a local alternative ℓ for nest $\mathscr{N}$ and $GL(\mathscr{A})$ be the global layout (to be determined) for array $\mathscr{A}$. For example, in Table 2a, $LL^1_{\mathscr{N}_1}(\mathscr{A}_1) =$ r-m and $LL^2_{\mathscr{N}_2}(\mathscr{A}_5) =$ c-m. We define the following parameter:

$$\mu(\mathscr{A}, \mathscr{N}, \ell) = \begin{cases} 0 & \text{if} \quad LL^{\ell}_{\mathscr{N}}(\mathscr{A}) = GL(\mathscr{A}) \text{ or } \mathscr{A} \text{ is not referred in } \mathscr{N} \\ 1 & \text{otherwise.} \end{cases}$$

Essentially, $\mu(\mathscr{A}, \mathscr{N}, \ell)$ represents the cost of the layout of array $\mathscr{A}$ for a specific local alternative ℓ of a specific loop nest $\mathscr{N}$. Given this definition of μ, the cost of nest $\mathscr{N}$ under a local alternative ℓ is $LCost(\mathscr{N}, \ell) = \sum_{\mathscr{A}} \mu(\mathscr{A}, \mathscr{N}, \ell)$. Similarly, $ACost(\mathscr{A}, \ell) = \sum_{\mathscr{N}} \mu(\mathscr{A}, \mathscr{N}, \ell)$ is the cost of array $\mathscr{A}$ considering all the loop nests, again under a specific local alternative ℓ. An important relation between $LCost$ and $ACost$ is

$$\sum_{\mathscr{A}} ACost(\mathscr{A}, \ell) = \sum_{\mathscr{N}} LCost(\mathscr{N}, \ell) = \sum_{\mathscr{N}} \sum_{\mathscr{A}} \mu(\mathscr{A}, \mathscr{N}, \ell).$$

We can now formulate the global layout determination problem as a problem of finding a global memory layout for each array (that is, determining $GL(\mathscr{A})$ for each $\mathscr{A}$) and a corresponding local alternative for each nest (that is, determining ℓ for each $\mathscr{N}$) such that $\sum_{\mathscr{N}} \sum_{\mathscr{A}} \mu(\mathscr{A}, \mathscr{N}, \ell)$ is minimized.

Let us now consider the case shown in Table 2b. In this simple example there is only one alternative per loop nest. In this *special case*, we can apply the following heuristic: Consider each column in turn and pick up the layout that occurs most frequently. In the case of a tie, choose a layout arbitrarily. For this example, the heuristic leads to r-m, c-m, c-m, r-m, and r-m layouts for $\mathscr{A}_1$, $\mathscr{A}_2$, $\mathscr{A}_3$, $\mathscr{A}_4$, and $\mathscr{A}_5$, respectively. The complexity of this heuristic is $\Theta(s \times R)$ where s is the number of nests and R is the number of arrays. However, *local* can return multiple local alternatives for a single nest. Assuming p local alternatives per nest, a simple extension of the above heuristic results in $\Theta(p^s \times s \times R)$ complexity which is not acceptable unless s is very small. In the following, we formulate the problem on a DAG (*directed acyclic graph*) and solve it using a *shortest path* algorithm.

To demonstrate our approach, we consider the example shown in Table 2c. Let *alternatives*($\mathscr{N}$) be a function that gives the number of alternative layout combinations for nest $\mathscr{N}$. For our example (Table 2c), *alternatives*($\mathscr{N}_1$) = 2, *alternatives*($\mathscr{N}_2$) = 1, and *alternatives*($\mathscr{N}_3$) = 3. Similarly let *arrays*($\mathscr{N}$) be a function that gives the number of arrays referenced in nest $\mathscr{N}$. Again, for our example, *arrays*($\mathscr{N}_1$) = 3, *arrays*($\mathscr{N}_2$) = 4, and *arrays*($\mathscr{N}_3$) = 3. Our approach consists of *four* steps:

Step (1): We first construct a bipartite graph where one group of nodes corresponds to loop nests while the other group corresponds to the arrays. There is an edge between an array node and a nest node if and only if the array is referenced in the nest. Such a bipartite graph is called an *interference graph* by Anderson *et al.* [3], and they use it to solve the global data distribution problem. Then an algorithm to find *connected components* is run on this graph. Each connected component corresponds to a group of loop nests that access a subset of the arrays declared in the program. For the example given in Table 2c, the connected component algorithm returns only one component: the graph itself. The complexity of the connected components algorithm on a bipartite graph is $\Theta(s + R)$ where s is the number of nests and r is the number of arrays [14]. The following steps operate on a *single* connected component at a time.

Step (2): In this step, an appropriate order of the loop nests is determined. This order will be used *only* for constructing a DAG on which a *shortest path* algorithm is run and is *not* used to change the textual order of the nests in the program by any means. We present two *heuristics* to determine an order for the loop nests: one which tries to minimize the number of edges of the DAG (`min-edge`), and another with a higher accuracy (`high-accuracy`). It will be shown that there is a tradeoff between complexity of the DAG and accuracy of the solution.

`Min-edge` **heuristic**. This heuristic attempts to find an order which will minimize the number of edges in the DAG. It proceeds as follows:

(1) The nests are ordered such that *alternatives*($\mathscr{N}_i$) $\leqslant$ *alternatives*($\mathscr{N}_{i+1}$), $1 \leqslant i \leqslant n-1$ (i.e., according to a nondecreasing number of alternatives).

(2) The place of $\mathscr{N}_1$ is fixed in the middle.

(3) The nests $\mathcal{N}_n$ and $\mathcal{N}_{n-1}$ are placed on the left and right sides of $\mathcal{N}_1$, respectively. Then the nests $\mathcal{N}_2$ and $\mathcal{N}_3$ are placed before $\mathcal{N}_n$ and after $\mathcal{N}_{n-1}$, respectively, and so on.

The aim of this heuristic is to minimize the sum of the subproducts $alternatives(\mathcal{N}_j) \times alternatives(\mathcal{N}_k)$ where $\mathcal{N}_j$ and $\mathcal{N}_k$ are neighboring nests in the final order. For the example given in Table 2c, $\{\mathcal{N}_3, \mathcal{N}_2, \mathcal{N}_1\}$ is the nest order returned by the `min-edge`. The rationale behind this heuristic is that the complexity of the single-source shortest path algorithm on a DAG is $\Theta(V+E)$ where V and E correspond to the number of vertices and edges of the DAG, respectively [14]. In our case, the number of vertices is fixed for a given program and is equal to the sum of the number of alternatives for all the nests plus two extra nodes for source and target. The number of edges, on the other hand, can be minimized by changing the order of the nests. The practical significance of this heuristic is that it strives to reduce the compilation time, as the shortest path algorithm will run at compile time. To appreciate the reduction in the complexity of the DAG when `Min-edge` is used, suppose that we have three nests with the number of local alternatives 1, 4, and 9. If the nests are ordered as 1, 4, 9, the total number of edges will be $1+1\times4+4\times9+9=1+4+36+9=50$. The first term (1) and the last term (9) are due to source and target nodes, respectively (see below). If we use the `Min-edge` heuristic, however, we order the nests as 9, 1, 4 and the total number of edges will be $9+9\times1+1\times4+4=9+9+4+4=26$.

`Max-accuracy` **heuristic**. This heuristic tries to increase the accuracy of the solution at the expense of a more complex DAG. The shortest path algorithm to be explained in the next step minimizes the cost originating from conflicting layout requirements between the adjacent nests in the final order and is based on the idea that this cost minimization process between each adjacent pair, *hopefully*, leads to a near-optimal overall cost. This may not be true if the two neighbors (left and right) of a loop nest have conflicting layouts for some arrays, but those arrays are not referenced in that loop nest itself. In this case, the cost of the shortest path between these two neighbors may be zero, but in reality a global layout assignment will lead to a nonzero cost. In order to *reduce the possibility* of the occurrence of this unfortunate case, the `max-accuracy` heuristic orders the loop nests as $\{\mathcal{N}_1, \mathcal{N}_2, ..., \mathcal{N}_n\}$ where $arrays(\mathcal{N}_1) \geqslant arrays(\mathcal{N}_2) \geqslant \cdots \geqslant arrays(\mathcal{N}_{n-1}) \geqslant arrays(\mathcal{N}_n)$. For the example given in Table 2c, $\{\mathcal{N}_2, \mathcal{N}_1, \mathcal{N}_3\}$ and $\{\mathcal{N}_2, \mathcal{N}_3, \mathcal{N}_1\}$ are the nest orders returned by the `max-accuracy`. It should be noted that this heuristic in general may increase the compile time (as it does not care about the number of edges) but (hopefully) reduces the run time by resulting in better global layouts. Alternatively, an approach can apply both heuristics and make a decision based on all returned nest orders.

It should be emphasized that the rest of the global locality optimization algorithm is *independent* from how the nests are ordered.

Step (3): Suppose that, without loss of generality, $\{\mathcal{N}_1, \mathcal{N}_2, ..., \mathcal{N}_n\}$ is the order obtained by the previous step. We construct a DAG as follows: For each alternative layout combination of each nest we create a node. This node is given the name $N_{i,j}$ where i is the nest number and j is the number of the local alternative.

There is a directed edge from $N_{i,j}$ to $N_{i+1,k}$ for all $1 \leqslant j \leqslant alternatives(\mathcal{N}_i)$ and $1 \leqslant k \leqslant alternatives(\mathcal{N}_{i+1})$. This edge is annotated by a set of arrays whose local memory layouts differ in $\mathcal{N}_i$ and $\mathcal{N}_j$. The *cost* of this edge is defined as the number of those arrays. Notice that such a cost definition is coarse, but the algorithm can be fed more accurate cost models where available. For example, a more aggressive approach can use the estimated number of potential misses [41] as the cost function. Alternatively, we can weight each edge with the probability of execution of the nest times the total array sizes whose layouts differ. If necessary, profiling the sequential code can provide us with information about loop trip counts, array sizes, and branch probabilities. A source node (S) and a target node (T) (both with zero cost) are also added onto DAG such that there is an edge from S to $\mathcal{N}_{1,j}$ for all $1 \leqslant j \leqslant alternatives(\mathcal{N}_1)$ and an edge from $\mathcal{N}_{n,k}$ to T for all $1 \leqslant k \leqslant alternatives(\mathcal{N}_n)$. Then a shortest path algorithm for this DAG is run from S to T. The path with the minimum cost gives a *suggested local alternative* for each nest. Figure 8a shows the DAG obtained by the order $\{\mathcal{N}_1, \mathcal{N}_2, \mathcal{N}_3\}$ for the example given in Table 2c. The edges are annotated with names of the arrays whose layouts are different in the connected alternatives. The shortest path algorithm on this DAG returns with two *near-optimal* solutions, $\mathcal{N}_{1,1}, \mathcal{N}_{2,1}, \mathcal{N}_{3,2}$ and $\mathcal{N}_{1,1}, \mathcal{N}_{2,1}, \mathcal{N}_{3,3}$, omitting the source and target nodes. In the first solution a cost occurs due to array $\mathscr{A}_2$ whereas in the second solution the cost is due to array $\mathscr{A}_4$. Now let us concentrate on the DAG in Fig. 8b which is obtained from the order $\{\mathcal{N}_2, \mathcal{N}_1, \mathcal{N}_3\}$. In this DAG, the shortest path algorithm returns the solution $\mathcal{N}_{2,1}, \mathcal{N}_{1,1}, \mathcal{N}_{3,3}$ with a cost of zero. But, if these local layouts are assigned there will be a cost originating from the conflicting requirements on $\mathscr{A}_4$ by $\mathcal{N}_{2,1}$ and $\mathcal{N}_{3,3}$. Since our approach considers only adjacent nest pairs, this cost does not reflect on the cost of the shortest path and the solution is still suboptimal. This problem occurs because of the fact that $\mathscr{A}_4$ is not referenced in $\mathcal{N}_1$ but is referenced in its left and right neighbors ($\mathcal{N}_2$ and $\mathcal{N}_3$, respectively). In fact, this is the case that the `max-accuracy` heuristic is designed to prevent from happening as much as possible.

Step (4): The final phase of the heuristic determines the global memory layouts for all arrays using the suggested local alternatives obtained in the previous step. We refer to the shortest path obtained in the previous step by δ; and the ith

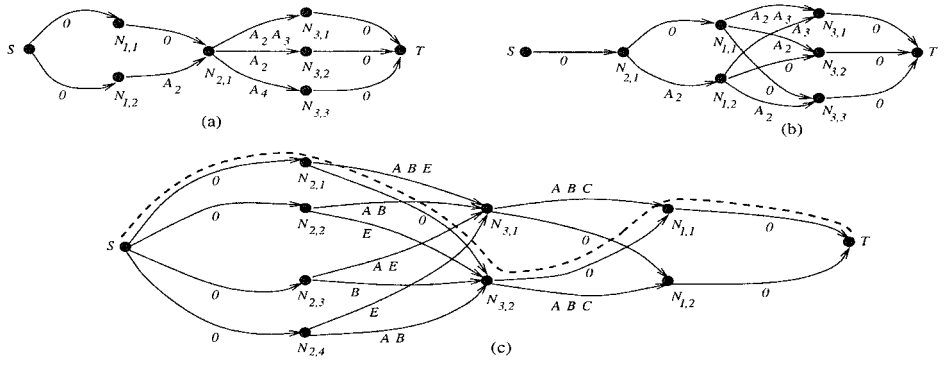

FIG. 8. DAGs for different examples. Each edges is annotated by the number of arrays whose local layouts differ in the local alternatives. The thick dashed curve in (c) shows the shortest path.

node of the shortest path (excluding the source and target) is denoted by δ_i. Suppose that there is a conflict between δ_i and δ_{i+1} on an array $\mathscr{A}$. In order to resolve this conflict the layout for $\mathscr{A}$ should be changed either in δ_i or in δ_{i+1}, as we do *not* consider array redistribution in memory. Our approach decides the alternative to be changed by considering all nodes along the shortest path. The algorithm traverses the shortest path and records, for each array with conflicting layout demands, the number of r-m and c-m layout demands. Then, in an attempt to satisfy the majority of the nests, it chooses the layout that occurs most frequently. Notice that this is exactly the same procedure used for solving the simpler case of the general problem (see Table 2b). After that, the local layouts (in suggested local alternative) of a nest which are different from global layouts are changed accordingly. If desired, *local* can be run once more for the suboptimal nests by taking into account the globally optimized layouts.

To sum up, after the third phase, the suggested local alternatives for each nest, and after the fourth phase, the global layouts for the whole program, are determined, and then the local layouts are adjusted accordingly.

Since a loop nest can have at most $m+1$ local alternatives, if we assume s loop nests in the program, the total number of nodes in the DAG is at most $s(m+1)+2$. The additional term represents the source and target nodes. The total number of edges, on the other hand, is $(s-1)(m+1)^2+2(m+1)$. This is because there are at most $(m+1)(m+1)$ edges between two neighboring loop nests in the DAG, and there are a total of $(s-1)$ such neighboring pair of loop nests. The term $2(m+1)$ comes from the source and target nodes. Overall, the total space complexity of the DAG is $\Theta(sm^2)$.

It is important to note the difference between our approach and one that uses an integer linear programming (ILP) formulation such as that of Kremer and Kennedy [26]. The ILP formulation used by Kennedy and Kremer [26] uses an enumerated set of local candidates (for distributions of arrays) for each phase; each phase (determined a priori) specifies only a partial distribution and there can be changes across phases, resulting in the need for ILP. In our approach, we order the nests for processing using greedy heuristics. Therefore once the nests are ordered, what we have is simply a version of the shortest path problem in which constraints are propagated from one nest to a later nest in order. Note that it is possible to use an ILP formulation in our case as well.

$\mathscr{N}_1$:

```
DO i = 1, N
 DO j = 1, N
  DO k = 1, N
   A(i,j) = B(j,k) + C(i,j)
  END DO
 END DO
END DO
```

$\mathscr{N}_2$:

```
DO i = 1, N
  DO j = 1, N
    DO k = 1, N
      DO l = 1, N
        D(i,j) = E(k,i) + A(k,j) + B(l,i)
      END DO
    END DO
  END DO
END DO
```

$\mathscr{N}_3$:

```
DO i = 1, N
  DO j = 1, N
    DO k = 1, N
      B(i,j) = E(j,i) + A(j,i) + C(k,i)
    END DO
  END DO
END DO
```

FIG. 9. A simple example consisting of three loop nests. For the sake of illustration, we assume that *local* returns 2, 4, and 2 local alternatives for $\mathscr{N}_1$, $\mathscr{N}_2$, and $\mathscr{N}_3$, respectively.

TABLE 3

Local Layout Assignments for the Program Fragment in Fig. 9

	A	*B*	*C*	*D*	*E*
$\mathcal{N}_1$	r-m	c-m	r-m		
	c-m	r-m	c-m		
$\mathcal{N}_2$	r-m	c-m		r-m	c-m
	r-m	c-m		r-m	c-m
	r-m	r-m		c-m	r-m
	c-m	r-m		c-m	r-m
$\mathcal{N}_3$	c-m	r-m	c-m		c-m
	r-m	c-m	r-m		r-m

Note. c-m denotes column-major and r-m denotes row-major.

7.4. Example

In order to demonstrate the technique on a program fragment, we consider a simple example shown in Fig. 9 which consists of three consecutive nests. Assume that when *local* is run for each nest of this program it returns the alternatives shown in Table 3. The first phase of the algorithm returns only a single connected component. In the second phase, `min-edge` gives two preferred orders, $\{\mathcal{N}_2, \mathcal{N}_1, \mathcal{N}_3\}$ and $\{\mathcal{N}_2, \mathcal{N}_3, \mathcal{N}_1\}$, both with 18 edges on the corresponding DAGs. The `max-accuracy` heuristic, on the other hand, returns $\{\mathcal{N}_2, \mathcal{N}_3, \mathcal{N}_1\}$ and $\{\mathcal{N}_3, \mathcal{N}_2, \mathcal{N}_1\}$. Since $\{\mathcal{N}_2, \mathcal{N}_3, \mathcal{N}_1\}$ is common to both heuristics, we select that order for the DAG to be constructed. The resulting DAG is shown in Fig. 8c. The shortest path algorithm returns the path $\{\mathcal{N}_{2,1}, \mathcal{N}_{3,2}, \mathcal{N}_{1,1}\}$ (shown as a thick dashed curve) with a zero pair-wise cost. It should be noted that, for this example, that assignment happens to be optimal as well. That is, the optimal global memory layouts for arrays *A*, *B*, *C*, *D*, and *E* are r-m, c-m, r-m, r-m, and r-m, respectively.

8. PERFORMANCE RESULTS

In this section we shall present our experimental results on seven regular matrix codes: `risch`, a transpose routine from NWChem [23], a large computational chemistry package; `vpenta` and `btrix` from Spec 92 [15]; `lu`, an LU decomposition program which makes use of three matrices; `adi` from Livermore Kernels [33]; `mxm`, the classical *ijk* matrix multiplication routine; and `mxmxm`, a routine from [12] which multiplies three two-dimensional matrices.

Our experimental results are obtained on the SGI Origin 2000 distributed-shared-memory machine and the IBM SP-2 message-passing architecture. Each

processor of the Origin 2000 is 195 MHz R10000 MIPS processor, with 32 Kbyte L1 data cache and 4 Mbyte L2 unified cache. R10000 is an advanced superscalar processor which can fetch and decode four instructions per cycle and can run them on five pipelined functional units. Both caches are two-way associative and nonblocking. For L1 cache hits, the latency is 2 cycles; and for L1 misses that hit in L2, the latency is 8 to 10 cycles. The C versions of the transformed programs are compiled using the *native* compiler using the `-O3` option (the blocking and unrolling are disabled explicitly as they blur temporal reuse). The IBM SP-2 is a distributed-memory message-passing machine and has RS/6000 Model 590 processors, each with a 256 Kbyte data cache. As we mentioned earlier, our approach is implemented using Parafrase-2 [37] as a source-to-source translator. The translated programs are then compiled using the `-O2` option with the native compiler.

We first evaluate the `mxm` routine in detail on the Origin and then present the overall results for all programs. In evaluating our technique we mainly concentrate on four parameters: execution `cycles`, `mflops` rate, primary cache misses (`PCM`), and `TLB` misses. Execution cycles and mflops rates are obtained using timing routines in the program whereas the miss rates are obtained using the hardware performance counters on the Origin. For this study, we ignore the number of secondary cache misses.

Table 4a shows the performance of the original `mxm` code on a single node of the Origin 2000 using different `SIZE`×`SIZE` double-precision matrices. All arrays in the original code have row-major memory layouts. Table 4b shows the same parameters for the optimized code using the approach explained in this paper. Table 4c, on the other hand, shows the execution cycles *per processor* and mflops rates for 1000×1000 double-precision matrices using a different number of processors. From Tables 4a and 4b, we observe that there are 22, 21, 21, and 98% improvements in the execution cycles, mflops rates, L1 cache misses, and TLB misses, respectively. In particular, optimizing for cache locality has a huge impact on the number of TLB misses (due to improved memory locality). However, the R10000 processor used in the Origin 2000 has a complex hardware where lots of misses can get overlapped with the ongoing computation; therefore, the overall improvement is around 21%. Table 4c shows that when all processor sizes are considered, we have a 13% improvement in the mflops rates.

Tiling (blocking) is a technique to improve the locality and parallelism in loop nests and is a combination of strip-mining and loop permutation [10, 49, 29, 13, 50]. Due to interference misses, it is difficult to select a suitable blocking factor (tile size). Unless the blocking factor is tailored according to the matrix size and cache parameters, the performance of tiling may be rather poor [13, 29]. We also investigated what happens when we tile the original and the optimized codes for matrix multiply. The results are presented in Table 5. We see in Tables 5a and 5b that in general tiling has a significant impact in the overall performance of the code. Another observation is that when a single node is considered the locality optimization brings a 2% improvement in the mflops rates over the original tiled code. However, we also observe that the improvement in the TLB misses is at around 79%. Therefore, we can expect that in machines with larger remote latency to local latency ratio the impact of locality optimization on tiling will be higher. (In the Origin that

TABLE 4

Performance Results for `m×m` on the SGI Origin 2000

Size	Cycles	secs	mflops	PCM	TLB
		(a) Original code			
250	53, 791, 486	0.279	111.89	643, 683	351
500	474, 620, 174	2.485	100.59	10, 898, 344	37, 839
750	1, 624, 182, 002	13.111	64.36	31, 143, 773	257, 610
1000	3, 794, 084, 796	25.404	78.73	61, 576, 000	1, 119, 772
1250	7, 362, 564, 410	44.361	88.06	89, 704, 228	3, 276, 983
		(b) Optimized code			
250	46, 351, 293	0.247	126.52	634, 047	316
500	365, 675, 339	1.879	131.79	9, 191, 568	1, 601
750	1, 390, 474, 757	11.856	71.17	26, 255, 207	4, 199
1000	3, 141, 423, 468	21.820	91.66	45, 511, 806	11, 720
1250	5, 443, 610, 016	33.769	115.68	71, 222, 154	29, 197

(c) Execution cycles per processor and mflops rates with `Size` = 1000 on different number of processors

	Original		Optimized	
Number of procs	Cycles	mflops	Cycles	mflops
1	3, 794, 084, 796	78.73	3, 141, 423, 468	91.66
2	2, 449, 015, 482	126.45	2, 104, 961, 384	143.95
3	2, 002, 523, 515	156.63	1, 782, 032, 138	176.38
4	1, 763, 195, 747	180.59	1, 597, 983, 884	200.42
5	1, 646, 348, 059	196.14	1, 501, 126, 373	215.49
6	1, 412, 387, 509	204.69	1, 354, 394, 317	228.73
7	1, 401, 777, 158	211.24	1, 234, 491, 198	240.96
8	1, 266, 574, 001	207.66	1, 125, 485, 929	244.80

Note. The execution cycles (per processor) and mflops rates are obtained using timing routines in the program whereas the miss rates are obtained using the hardware performance counters on the Origin. For this study, we ignore the number of secondary cache misses. `PCM` denotes the primary cache misses and `TLB` refers to the TLB misses.

we use this ratio is around 2.) If multiple node performance is considered, the optimized tiled version has an 11% improvement in mflops rates and a 17% improvement in the execution cycles on 8 processors over the original tiled code. These results reveal that optimizing locality before tiling may further enhance the memory performance.

We now turn to the global picture and consider in Fig. 10 the mflops rates of all the programs. For each program, we compile and run three different versions:

TABLE 5

Performance Results for the Tiled m×m Code on the SGI Origin 2000

Size	Cycles	secs	mflops	PCM	TLB
		(a) Original code			
250	39, 755, 847	0.213	146.03	428, 744	396
500	309, 753, 349	1.655	151.06	4, 681, 308	31, 125
750	1, 027, 432, 650	10.480	80.52	10, 353, 755	56, 947
1000	2, 550, 973, 238	18.894	105.86	29, 610, 985	773, 778
1250	4, 961, 294, 521	31.803	122.83	52, 308, 223	2, 282, 477
		(b) Optimized code			
250	38, 958, 125	0.211	147.41	405, 653	345
500	303, 166, 413	1.602	156.06	3, 316, 253	9, 612
750	1, 011, 261, 450	10.169	82.97	13, 481, 989	54, 109
1000	2, 432, 754, 503	18. 691	107.00	36, 443, 931	173, 314
1250	4, 715, 763, 339	31.102	125.60	55, 173, 671	414, 070

(c) Execution cycles per processor and mflops rates with Size = 1000 on different number of processors

	Original		Optimized	
Number of procs	Cycles	mflops	Cycles	mflops
1	2, 550, 973, 238	105. 86	2, 432, 754, 503	107.00
2	1, 802, 352, 157	160.76	1, 781, 115, 880	166.66
3	1, 474, 743, 898	201.07	1, 240, 750, 519	209.02
4	1, 435, 008, 133	213.22	1, 196, 824, 753	223.63
5	1, 366, 130, 001	226.96	1, 112, 705, 892	239.90
6	1, 269, 450, 004	245.25	1, 047, 278, 804	258.44
7	1, 152, 357, 278	263.71	960, 662, 143	277.79
8	1, 139, 057, 256	259.52	940, 619, 346	287.29

Note. The tile size is set to 50 × 50 double-precision elements in all experiments.

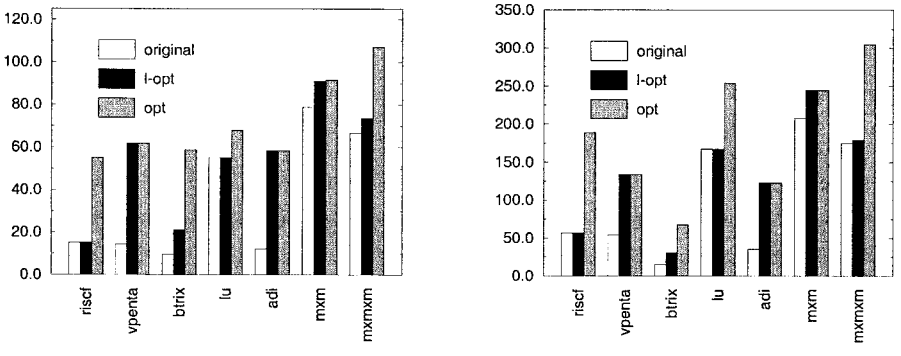

FIG. 10. Mflops rates on (a) a single node and (b) 8 nodes of the SGI Origin. A 35% improvement is obtained over the l-opt version on a single node.

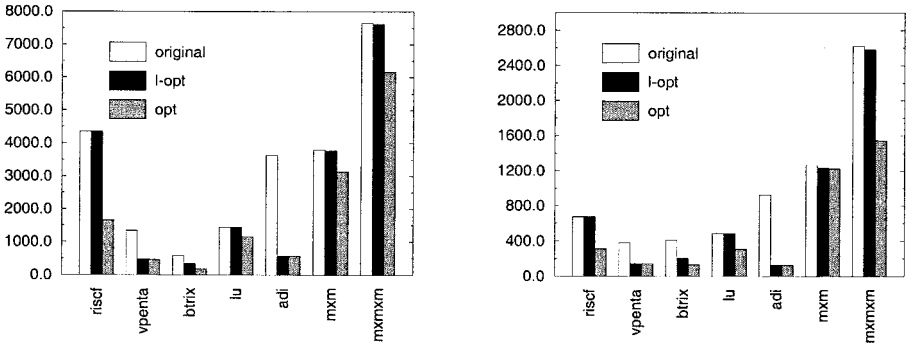

FIG. 11. Per processor execution cycles on (a) a single node and (b) 8 nodes of the SGI Origin. All values are in millions.

original is the original unoptimized code, l-opt is a version that uses only loop transformations to improve locality [32], and opt is the version obtained using the approach discussed in this paper. The l-opt version uses all linear loop transformations (represented by square nonsingular transformation matrices) as well as other loop transformations such as loop fusion and loop distribution. We also implemented this version in the Parafrase-2 [37] framework following the algorithms (height reduction and width reduction) given in Li's thesis [32]. Note that this loop transformation framework uses general nonsingular transformations which subsume unimodular loop transformations [49] and loop permutations [34]. We note that for `riscf` and `lu` the `original` and the l-opt versions result in the same code. In the `riscf` code, within the same loop nest different arrays are accessed with different access patterns so loop transformation is not able to optimize it. In `lu`, the dependences prevent the linear loop transformations which would otherwise improve the performance. We also note that in `adi` and `vpenta` the l-opt and the opt versions result in the same optimized code; that is, for these programs memory layout transformations do not bring any additional improvement. We can see from Figs. 10a and 10b that our approach is quite successful in optimizing locality on both single node and multiple nodes. Considering all programs, the opt almost doubles the mflops rate of the original version and results in a 35% improvement over the l-opt version on a single node showing the importance of a unified approach which takes into account loop as well as data transformations. Another point to emphasize is that unlike most approaches to locality which gives priority to spatial locality, since our approach tries to optimize temporal locality first, in general we have a reduction in the number of loads and stores. For example, on a single node of the Origin with 1000×1000 double-precision matrices, l-opt

TABLE 6

Number of Iterations Executed by Each Program

Program	riscf	vpenta	btrix	lu	adi	m×m	m×m×m
Iterations	83, 886, 080	10, 353, 600	4, 055, 940	106, 000, 000	5, 242, 880	1, 000, 000	2, 000, 000

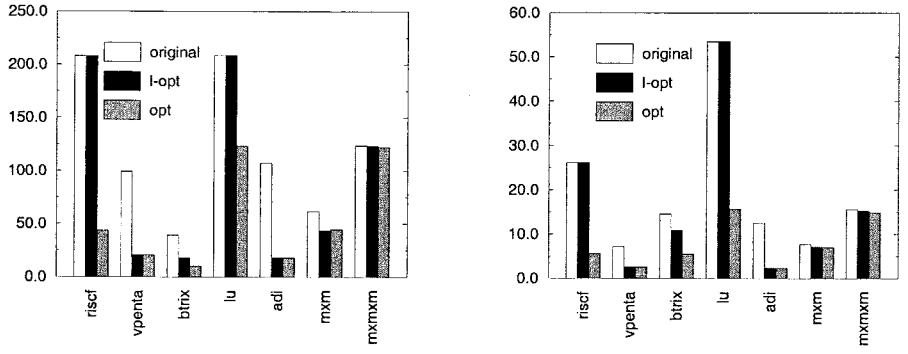

FIG. 12. Primary cache misses (PCM) on (a) a single node and (b) 8 nodes of the SGI Origin. All values are in millions. These results are obtained using the performance counters in the Origin.

generates 2, 052, 597, 824 load operations whereas opt generates 2, 016, 720, 224 load operations. Similarly, for store operations l-opt results in 12, 228, 912 stores while our approach produces only 6, 026, 224 due to the fact that we start optimizing from the left-hand side references and attempt to optimize each reference first for temporal locality then for spatial locality.

Figures 11a and 11b present the performance figures in execution cycles per processor (in millions). As a reference, we also give in Table 6 the number of iterations executed by each program. These results show the impressive reductions in cycles achieved by our unified approach to locality and parallelism.

Figures 12a and 12b give the primary cache misses in millions. We note that except for the mxm code on a single node, the opt version achieves the lowest miss rates for the L1 cache. Figures 13a and 13b on the other hand, present the TLB misses under three versions. The figures on top of some bars give the number of misses (again in thousands) whose corresponding bars cannot fit in the space. It is easy to see that on a single node, the TLB misses of original is not even comparable to that of the opt version except for the adi and the lu codes (in the lu code the improvement mainly comes from the reduction in PCM). Again except for vpenta, lu, and adi the TLB misses of the l-opt version is not comparable to that of opt. The results on 8 nodes are also similar. It is interesting to note that in codes such

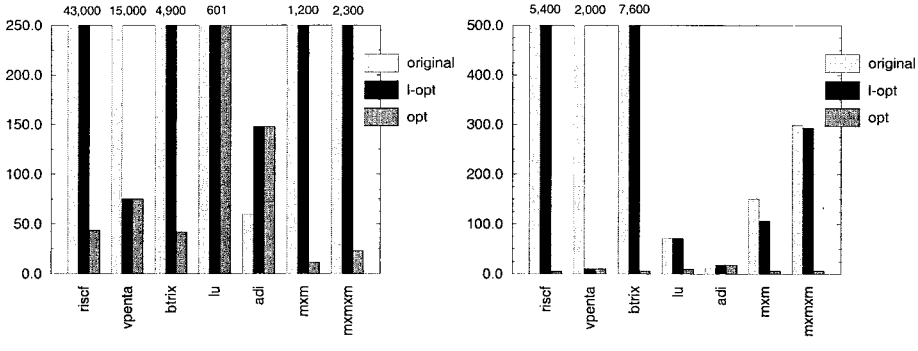

FIG. 13. TLB misses on (a) a single node and (b) 8 nodes of the SGI Origin. All values are in thousands. These results are obtained using the performance counters in the Origin.

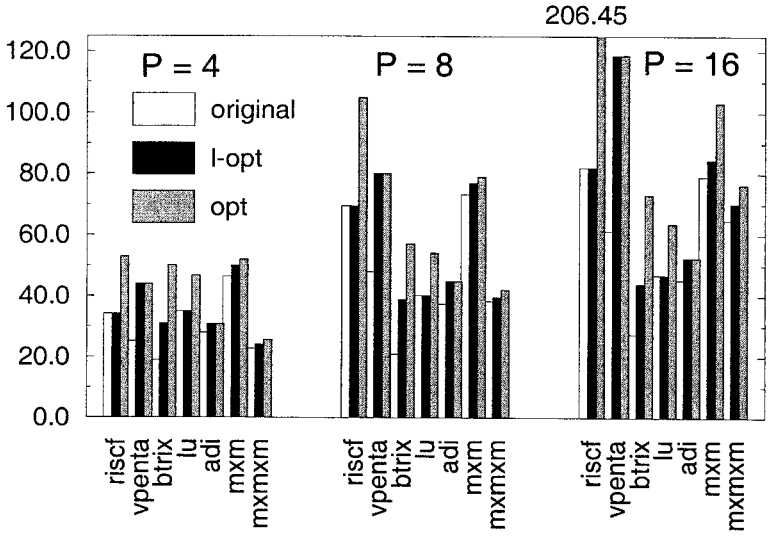

FIG. 14. Performance of the programs on the IBM SP-2. Overall the opt version achieves a 53% improvement over the original version and a 29% improvement over the l-opt version.

as `mxm` and `mxmxm` the main locality improvement comes from the TLB misses rather than the L1 misses. In the `adi code` the optimized version increases the number of TLB misses. But in that case that increase is more than compensated by the reduction in the L1 and L2 misses (not shown).

Finally, Fig. 14 shows the mflops rates performance of the programs on the IBM SP-2. Overall the opt version delivers a 53% improvement over the original version and a 29% improvement over the l-opt version. We should emphasize that the results presented in this figure are conservative in the sense that the unoptimized programs are also parallelized (if possible) such that the maximum granularity of parallelism is obtained; moreover, the data are distributed across the memories of the processors so as to eliminate the interprocessor communication as much as possible. In particular, in four out of the seven codes the data is distributed so that all potential communications are eliminated. Since this may not always be the case for the unoptimized programs, we believe that the performance improvement obtained by our algorithm will be higher in general.

9. SUMMARY

The broad variety of parallel architectures renders designing unified compiler techniques difficult. However, although the underlying hardware facilities are different, we believe that all the parallel architectures will benefit from compiler optimizations that are aimed at improving locality and deriving large-granularity of parallelism. In this paper, we describe a compiler algorithm that handles locality, parallelism, and data distribution in a unified manner and that can be embedded in a compilation framework for parallel machines. Our algorithm derives the entries of a loop transformation matrix and determines suitable memory layouts for each array such that good locality, outermost loop parallelism, and minimum

communication are obtained. Our experiments with several benchmarks on the SGI Origin 2000 distributed-shared memory multiprocessor and the IBM SP-2 message-passing machine show the effectiveness of the unified approach.

Further research will involve investigating compiler algorithms which can handle complex data distributions (e.g., distributions along more than one dimension and diagonal-wise distributions) and locality in a unified manner for both single-nest and multiple-nest cases. We believe that the results presented in this paper provide encouraging evidence that the challenges posed by different parallel systems can be handled with unified compilation techniques; that is, it seems possible to design unified compilation techniques for distributed and shared address space machines.

ACKNOWLEDGMENTS

This work is supported in part by NSF Young Investigator Award CCR-9357840, NSF CCR-9509143. The work of J. Ramanujam is supported in part by an NSF Young Investigator Award CCR-9457768 and by NSF Grant CCR-9210422.

REFERENCES

1. A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. Kranz, J. Kubiatowicz, B. Lim, K. Mackenzie, and D. Yeung, The MIT Alewife machine: architecture and performance, *in* "Proc. 22nd International Symposium on Computer Architecture," 1995.
2. J. M. Anderson, S. P. Amarasinghe, and M. S. Lam, Data and computation transformations for multiprocessors, *in* "Proc. 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming," July 1995.
3. J. M. Anderson and M. S. Lam, Global optimizations for parallelism and locality on scalable parallel machines, *in* "Proc. SIGPLAN Conference on Programming Language Design and Implementation," pp. 112–125, June 1993.
4. V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer, An interactive environment for data partitioning and distribution, *in* "5th Distributed Memory Computing Conference, Charleston, SC, April 1990."
5. P. Banerjee, J. A. Chandy, M. Gupta, E. W. Hodges IV, J. G. Holm, A. Lain, D. J. Palermo, S. Ramaswamy, and E. Su, The PARADIGM compiler for distributed-memory multicomputers, *IEEE Comput*. **28**, 10 (October 1995), 37–47.
6. A. J. C. Bik and H. A. G. Wijshoff, "On a Completion Method for Unimodular Matrices," Technical Report 94-14, Dept. Computer Science, Leiden University, Netherlands, 1994.
7. W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu, Parallel programming with Polaris, *IEEE Comput*. **29**, 12 (December 1996), 78–82.
8. Z. Bozkus, L. Meadows, D. Miles, S. Nakamoto, V. Schuster, and M. Young, Techniques for compiling and executing HPF programs on shared-memory and distributed-memory parallel systems, *in* "Proc. 1st International Workshop on Parallel Processing, Bangalore, India, December 1994."
9. T. Brewer and G. Astfalf, The evolution of the HP/Convex Exemplar, *in* "Proc. COMPCON Spring'97: 42nd IEEE Computer Society International Conference," pp. 81–86, February 1997.
10. S. Carr and K. Kennedy, Blocking linear algebra codes for memory hierarchies, *in* "Proc. 4th SIAM Conference on Parallel Processing for Scientific Computing, Chicago, IL, December 1989."
11. R. Chandra, D. Chen, R. Cox, D. Maydan, N. Nedeljkovic, and J. M. Anderson, Data-distribution support on distributed-shared memory multiprocessors, *in* "Proc. Programming Language Design and Implementation (PLDI), Las Vegas, NV, 1997."

12. M. Cierniak and W. Li, Unifying data and control transformations for distributed shared memory machines, *in* "Proc. SIGPLAN'95 Conference on Programming Language Design and Implementation, La Jolla, CA, June 1995."
13. S. Coleman and K. McKinley, Tile size selection using cache organization and data layout, *in* "Proc. SIGPLAN'95 Conference on Programming Language Design and Implementation, La Jolla, CA, June 1995."
14. T. Cormen, C. Leiserson, and R. Rivest, "Introduction to Algorithms," The MIT Press, Cambridge, MA, 1990.
15. K. M. Dixit, New CPU benchmark suites from SPEC, *in* "Proc. COMPCON'92—37th IEEE Computer Society International Conference, San Francisco, CA, February 1992."
16. J. J. Dongarra, J. D. Croz, S. Hammarling, and I. Duff, A set of level 3 basic linear algebra subprograms, *ACM Trans. Math. Software* **16**, 1 (March 1990), 1–17.
17. D. Gannon, W. Jalby, and K. Gallivan, Strategies for cache and local memory management by global program transformations, *J. Parallel Distrib. Comput.* 5 (1988), 587–616.
18. J. Garcia, E. Ayguade, and J. Labarta, A novel approach towards automatic data distribution, *in* "Proc. Supercomputing'95, San Diego, December 1995."
19. M. R. Garey and D. S. Johnson, "Computers and Intractability: A Guide to the Theory of NP-Completeness," Freeman, New York, 1979.
20. M. Gerndt, Updating distributed variables in local computations, *Concurrency Practice Experience* **2**, 3 (September 1990), 171–193.
21. M. Gupta and P. Banerjee, Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers, *IEEE Trans. Parallel Distrib. Systems* **3**, 2 (March 1992), 179–193.
22. J. L. Hennessy and D. A. Patterson, "Computer Architecture: A Quantitative Approach," second ed., Morgan Kaufmann, San Mateo, CA, 1995.
23. High Performance Computation Chemistry Group, "NWChem: A Computation Chemistry Package for Parallel Computers," version 1.1, Pacific Northwest Laboratory, Richland, WA, 1995.
24. S. Hiranandani, K. Kennedy, and C.-W. Tseng, Compiling Fortran D for MIMD distributed-memory machines, *Commun. Assoc. Comput. Mach.* **35**, 8 (August 1992), 66–88.
25. Y.-J. Ju and H. Dietz, Reduction of cache coherence overhead by compiler data layout and loop transformations, *in* "Proc. 4th Workshop on Languages and Compilers for Parallel Computing, Santa Clara, CA, August 1991."
26. K. Kennedy and U. Kremer, Automatic data layout for High Performance Fortran, *in* "Proceedings of Supercomputing'95, San Diego, CA, December 1995."
27. K. Kennedy and K. S. McKinley, Optimizing for parallelism and data locality, *in* "Proc. 1992 ACM International Conference on Supercomputing (ICS'92), Washington, D.C., July 1992."
28. U. Kremer, "Automatic Data Layout for Distributed Memory Machines," Ph.D. thesis, Rice University, Houston, TX, 1995.
29. M. S. Lam, E. Rothberg, and M. E. Wolf, The cache performance and optimizations of blocked algorithms, *in* "Proc. 4th International Conference on Architectural Support for Programming Languages and Operating Systems, April, 1991."
30. J. Laudon and D. Lenoski, The SGI Origin: A CC-NUMA highly scalable server, *in* "Proc. 24th Annual International Symposium on Computer Architecture, May 1997."
31. D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy, The DASH prototype: implementation and performance, *in* "Proc. 19th International Symposium on Computer Architecture, Gold Coast, Australia, May 1992," pp. 92–103.
32. W. Li, "Compiling for NUMA Parallel Machines," Ph.D. Thesis, Cornell University, Ithaca, NY, 1993.
33. F. McMahon, "The Livermore Fortran Kernels: A Computer Test of the Numerical Performance Range," Technical Report UCRL-53745, Lawrence Livermore National Laboratory, Livermore, CA, 1986.

34. K. McKinley, S. Carr, and C. W. Tseng, Improving data locality with loop transformations, *ACM Trans. Progr. Languages Systems* **18**, 4 (July 1996), 424–453.

35. M. O'Boyle and P. Knijnenburg, Non-singular data transformations: definition, validity, applications, *in* "Proc. 6th Workshop on Compilers for Parallel Computers, Aachen, Germany, 1996," pp. 287–297.

36. D. Palermo and P. Banerjee, Automatic selection of dynamic data partitioning schemes for distributed-memory multicomputers, *in* "Proc. 8th Workshop on Languages and Compilers for Parallel Computing, Columbus, OH, 1995," pp. 392–406.

37. C. Polychronopoulos, M. B. Girkar, M. R. Haghighat, C. L. Lee, B. P. Leung, and D. A. Schouten, Parafrase-2: an environment for parallelizing, partitioning, synchronizing, and scheduling programs on multiprocessors, *in* "Proc. the International Conference on Parallel Processing, St. Charles, IL, August 1989," pp. 39–48.

38. J. Ramanujam, Non-unimodular transformations of nested loops, *in* "Proc. Supercomputing 92, Minneapolis, MN, November 1992," pp. 214–223.

39. J. Ramanujam and A. Narayan, Integrating data distribution and loop transformations for distributed memory machines, *in* "Proc. 7th SIAM Conference on Parallel Processing for Scientific Computing, February 1995" (D. Bailey *et al*., Eds.), pp. 668–673.

40. J. Ramanujam and A. Narayan, Automatic data mapping and program transformations, *in* "Proc. Workshop on Automatic Data Layout and Performance Prediction, Houston, TX, April 1995."

41. V. Sarkar, G. R. Gao, and S. Han, Locality analysis for distributed shared-memory multiprocessors, *in* "Proc. the Ninth International Workshop on Languages and Compilers for Parallel Computing, Santa Clara, CA, August 1996."

42. A. Schrijver, "Theory of Linear and Integer Programming," Wiley-Interscience Series in Discrete Mathematics and Optimization, Wiley, New York, 1986.

43. T. J. Sheffler, R. Schreiber, J. R. Gilbert, and S. Chatterjee, Aligning parallel arrays to reduce communication, *in* "Frontiers '95: The 5th Symposium on the Frontiers of Massively Parallel Computing, McLean, VA, February 1995," pp. 324–331.

44. C.-W. Tseng, J. Anderson, S. Amarasinghe, and M. Lam, Unified compilation techniques for shared and distributed address space machines, *in* "Proc. 1995 International Conference on Supercomputing (ICS'95), Barcelona, Spain, July 1995."

45. R. Thekkath, A. P. Singh, J. P. Singh, S. John, and J. Hennessey, An evaluation of a commercial CC-NUMA architecture: the CONVEX Exemplar SPP-1200, *in* "Proc. 11th International Parallel Processing Symposium, Geneva, Switzerland, April 1997."

46. E. Torrie, C.-W. Tseng, M. Martonosi, and M. W. Hall, Evaluating the impact of advanced memory systems on compiler-parallelized codes, *in* "Proc. International Conference on Parallel Architectures and Compilations Techniques (PACT), Limassol, Cyprus, June 1995."

47. R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy, SUIF: An infrastructure for research on parallelizing and optimizing compilers, *ACM SIGPLAN Notices* **29**, 12 (December 1994), 31–37.

48. M. Wolf and M. Lam, A loop transformation theory and an algorithm to maximize parallelism, *IEEE Trans. Parallel Distrib. Systems* **2**, 4 (October 1991), 452–471.

49. M. Wolf and M. Lam, A data locality optimizing algorithm, *in* "Proc. ACM SIGPLAN 91 Conf. Programming Language Design and Implementation, June 1991," pp. 30–44.

50. M. Wolfe, "High Performance Compilers for Parallel Computing," Addison–Wesley, Reading, MA, 1996.

51. H. Zima and B. Chapman, Compiling for distributed-memory systems, *Proc. IEEE* **81**, 2 (1993), 264–287.