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

M. Kandemir† J. Ramanujam‡ A. Choudhary§

Abstract

Distributed memory message passing machines can 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, some scalable architectures 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 the different parallel architectures, issues such as data decomposition are unique to specific types of architectures. One of the most important challenges compiler writers face is to design compilation techniques that can work 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 outermost loops can be run in parallel across processors; (2) decomposes each array across processors; (3) optimizes interprocessor communication by vectorizing it whenever possible; and (4) optimizes locality (cache performance) by assigning appropriate storage layout for each array. Depending on the underlying hardware system, some or all of these steps can be applied in a unified framework. We present simulation results for cache miss rates, and empirical results on SUN SPARCstation 5, IBM SP-2, SGI Challenge and Convex Exemplar to validate the effectiveness of our approach on different architectures.

1 Introduction

Optimizing for parallelism and locality in a unified framework is important for UMA architectures (e.g. SGI Challenge), shared-memory NUMA architectures (e.g. Convex Exemplar, Stanford DASH and KSR-1) and distributed memory multicomputers (e.g. Intel Paragon, IBM SP-2 and Thinking Machines CM-5). Optimizing parallelism leads to tasks of larger granularity with lower synchronization and communication costs and is beneficial for parallel machines. Since individual nodes of contemporary parallel machines have some form of cache hierarchy, optimizing cache locality of scientific codes on these machines results in high speedups. Additionally, as distribution of data is an important issue for distributed memory multicomputers and some shared-memory NUMA machines, optimizing data decomposition has a large impact on the overall performance of these machines.

*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.

†EECS Dept., Syracuse University, Syracuse, NY 13244. mtk@ece.nwu.edu

‡Department of Electrical and Computer Engineering, Louisiana State University, Baton Rouge, LA 70803. jxr@ee.lsu.edu

§Department of Electrical and Computer Engineering, Northwestern University, Evanston, IL, 60208. choudhar@ece.nwu.edu

Recent years have witnessed a tremendous increase in processor speeds. Unfortunately, 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 due to not exploiting locality suitably in these programs [26]. Among the issues that challenge compiler writers is maximizing parallelism, minimizing communication via loop level optimizations and block transfers and optimizing locality. Since these issues are interrelated, they should be handled in a unified framework. For example, given a loop nest where a number of arrays are accessed, the locality optimizations may imply a preferred order for the loops whereas the parallelism optimizations may suggest another. In this paper, we present an automatized strategy by which a compiler can optimize programs for both locality and parallelism. Specifically our optimizations perform the following:

- Maximizing the granularity of parallelism by transforming the loop nest such that the outermost loops can run parallel on a number of processors.
- Vectorizing communication, *i.e.,* performing communication in large chunks of data in order to amortize the high startup cost.
- Reorganizing data layouts in memory to match loop order: we believe that matching loop order to individual array layouts on memory is key to obtaining high performance in 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 [24]. In order to eliminate the memory bottleneck, cache locality should be exploited as much as possible. One way of achieving this is to transform the loop nest 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 layouts for multidimensional arrays.

The remainder of this paper is organized as follows. In Section 2, the mathematical framework used throughout the paper is presented, and in Section 3 the related work is summarized. A locality optimization algorithm is introduced in Section 4. An algorithm that maximizes granularity of parallelism is revised in Section 5. In Section 6, 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. Our preliminary results are given in Section 8, and finally the conclusions are presented in Section 9.

2 Preliminaries

In this section we briefly mention about memory layouts and loop transformation theory. The memory layout for an h-dimensional array can be in one of the $h!$ forms, each of which corresponding to layout of data on memory linearly by a nested traversal of the axes in some predetermined order. The innermost axis is called the *fastest changing dimension*. As an example, for row-major memory layout the second dimension is the fastest changing dimension. We focus on loop nests where both array subscripts and loop bounds are affine functions of enclosing loop indices. A reference to an array X is represented by $X(\mathcal{L}\vec{I}+\vec{b})$ where $\mathcal{L}$ is a linear transformation matrix called *array reference matrix*, $\vec{b}$ is offset vector and $\vec{I}$ is a column vector representing the loop indices i_1, i_2,...,i_n starting from the outermost loop.

Linear mappings between iteration spaces of loop nests can be modeled by non-singular transformation matrices [16]. If $\vec{I}$ is the original iteration vector, after applying linear transformation T, the new iteration vector is $\vec{J} = T\vec{I}$. Similarly if $\vec{d}$ is the distance/direction vector, after the transformation, $T\vec{d}$ is the new distance/direction vector. A transformation is *legal* if and only if $T\vec{d}$ is lexicographically positive for every $\vec{d}$ [29]. On the other hand, since $\mathcal{L}\vec{I} = \mathcal{L}T^{-1}\vec{J}$, after the transformation, $\mathcal{L}T^{-1}$ is the new array reference matrix. We denote T^{-1} by Q. For the rest of the paper, the reference matrix for array X will be denoted by $\mathcal{L}^X$ whereas the i^{th} row of the reference matrix for array X will be denoted by $\vec{\ell}_i^{\,X}$.

3 Related Work

Loop transformations have been used for optimizing cache locality in several papers [16, 28]. Results have shown that on several architectures the speedups achieved by loop transformations alone can be quite large.

McKinley *et al.* [18] offer an optimization technique consisting of loop permutation, loop fusion and loop distribution.

Li [16] describes a data reuse model and a compiler algorithm called *height reduction* to improve cache locality. He introduces the concept of a *data reuse vector* and defines its *height* as the number of dimensions from the first non-zero entry to the last entry. The non-zero 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. The 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. The height reduction algorithm both reduces the sensitivity of tiling to the tile size, and places the loops carrying reuse into innermost positions. In comparison, our algorithm (Section 6) more aggressively exploits 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 [28, 18, 16], in the rest of the paper we compare our algorithm to Li's algorithm (denoted W-Opt).

Data transformations, on the other hand, deal with data layout and array accesses rather than iteration space re-ordering. Only a few works have considered data and loop transformations together to optimize locality. Ju and Dietz [11] present a systematic approach that integrates data layout optimizations and loop transformations to reduce cache coherence overhead. Anderson *et al.* [1] offer a simple algorithm to transform the data layout to make the region accessed by each processor contiguous.

Cierniak and Li [4] 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 and transformation matrix. As will be explained later, our approach is more accurate, as it does not restrict the search space of possible loop transformations. Our approach does not depend on any new reuse abstraction such as stride vector. Our extension to multiple nests (global optimization) is also different from the one offered by Cierniak and Li [4] for global optimization.

In a different approach, the data space is transformed using linear non-singular transformation matrices; but the transformed space for each array is stored on memory in a fixed storage order such as column-major or row-major. O'Boyle and Knijnenburg [19] have applied this technique to improve locality of programs. Let $\mathcal{Y}$ be a linear non-singular data transformation matrix. Omitting the shift-type transformations, the data transformation denoted by $\mathcal{Y}$ is applied in two steps: (1) The original reference matrix A is transformed to $\mathcal{Y}A$, and (2) The data layout on memory is also transformed by using $\mathcal{Y}$, and the array declaration statements are changed accordingly. Since we assign different layouts (such as row-major or column-major) for different arrays, we do not need to transform the reference matrices or change array declarations.

Previous work on parallelism has concentrated, among other topics, on compilation techniques for multicomputers [3, 30, 9], automatic discovery of parallelism [27, 20]. The loop-level optimization techniques to improve locality and communication are offered in several papers [18, 27].

4 Algorithm for Optimizing Locality

In this section, we present an algorithm which automatically transforms a given loop nest to exploit spatial locality and assigns appropriate memory layouts for arrays in a unified framework. This algorithm can be used for optimizing locality in uniprocessors and shared-memory multiprocessors. It can also be employed as part of a unified technique for optimizing locality and parallelism in distributed memory machines.

4.1 Explanation

The algorithm is shown in Figure 1. In the algorithm, C is the array reference on the LHS whereas A represents an array reference from the RHS. The symbol $\times$ denotes the *don't care* condition. Let i_1,i_2,...,i_n be the loop indices of the original nest and j_1,j_2,...,j_n be the loop indices of the transformed nest, starting from outermost position. The following is a brief explanation of our algorithm.

• Our transformation matrix should be such that the LHS array of the transformed loop should have the innermost index as the only element in one of the array dimensions and that index should not appear in any other dimension for this array. In other words, after the transformation, the LHS array C should be of the form $C(*, ..., *, j_n, *, ..., *)$ where j_n (the new innermost loop index) is in the r^{th} dimension and $*$ indicates a term independent of j_n. This means that row r of the transformed reference matrix for C is $(0, ..., 0, 1)$ and all entries of the last column, except the one in r^{th} row, are zero. After that, the LHS array can be stored in memory such that dimension r is the fastest changing dimension. This approach helps to exploit the spatial locality for this reference.

• Then the algorithm works on one reference from the RHS at a time. If a row s in the data reference matrix is identical to r^{th} row of the original reference matrix of the LHS array, then the algorithm attempts to store this RHS array in memory such that the s^{th} dimension will be the fastest changing dimension. We note that having such a row s does not guarantee that the array will be stored

Step 1 Initialize $i = 1$.

Step 2 Set $\vec{\ell_i}^{\,C}.Q = (0,0,...,0,1)$ and $\vec{\ell_k}^{\,C}.Q = (\times,\times,...,\times,0)$ for each $k \neq i$.

Step 3 Set memory layout for C such that i^{th} index position will be the fastest changing dimension.

Step 4 For each array reference A on the RHS that has $\vec{\ell_l}^{\,A} = \vec{\ell_i}^{\,C}$ for some l, try to set memory layout for A such that the l^{th} dimension will be the fastest changing dimension.

Step 5 Choose an array reference A for which the equality in **Step 4** does not hold. Initialize $j = 1$.

Step 6 Set $\vec{\ell_j}^{\,A}.Q = (0,0,...,1,0)$ and $\vec{\ell_k}^{\,A}.Q = (\times,\times,...,\times,0,0)$ for each $k \neq j$. If this step is consistent with the previous steps go to **Step 7**, otherwise increment j and go to the beginning of this step. If there exist inconsistencies for all j values, then initialize $j = 1$, and set $\vec{\ell_j}^{\,A}.Q = (0,0,...,1,0,0)$ and $\vec{\ell_k}^{\,A}.Q = (\times,\times,...,\times,0,0,0)$ for each $k \neq j$, and repeat **Step 6** and so on. If no T^{-1} is found then fill the remaining entries arbitrarily observing the *dependences* and *non-singularity*.

Step 7 Repeat **Step 6** for all reference matrices of a particular A (Of course, all reference matrices for a particular A should have the same memory layout).

Step 8 Repeat **Step 6** for all distinct array references.

Step 9 Record the obtained transformation matrix. Also record, for each array, the loop index position which appears in the fastest changing position for that array.

Step 10 Increment i and go to **Step 2** (try a different memory layout for the LHS array C).

Step 11 Compare all the recorded transformation matrices and their associated layouts, and choose the best alternative.

Figure 1: Compiler algorithm for optimizing locality.

on memory such that the s^{th} dimension will be the fastest changing dimension.

• If the condition above fails for RHS array A, then the algorithm tries to transform the reference to $A(*,...,*,\mathcal{F}(j_{n-1}),*,...,*)$, where $\mathcal{F}(j_{n-1})$ is an affine function of j_{n-1} and other indices except j_n, and $*$ indicates a term independent of both j_{n-1} and j_n. This helps to exploit the spatial locality at the second innermost loop. If no such transformation is possible, the j_{n-2} is tried and so on. If all loop indices are tried unsuccessfully, then the remaining entries of Q are set arbitrarily, observing the data dependences and non-singularity.

• After a transformation and corresponding memory layouts are found, they are recorded and the next alternative memory layout for the LHS is tried and so on. Among all feasible solutions, the one that exploits more spatial locality in the innermost loop, is chosen.

The details of this algorithm can be found in [12]. Note that the algorithm considers all possible memory layouts, of which the row-major and column-major layouts are only two alternatives. Although the algorithm makes a kind of exhaustive search; in practice, number of loops, number of array references inside the nest and number of array dimensions are small values; and the approach is reasonably efficient. It should also be noted that since transformation matrices resulting from the algorithm are not necessarily unimodular, we need more general non-singular transformation theory such as [16] or [21]. And finally, it should be noted that the **Steps 2** and **6** involve solving matrix equations. We can use the method given in [16] with appropriate modifications for completing a partial matrix.

4.2 Example

In this subsection, we illustrate how the locality optimization algorithm works by giving an example. We do not show the steps or parts of steps which lead to unsuccessful trials. Figure 2:A shows the `i-j-k` matrix multiplication routine. The reference matrices for the arrays are as follows: $L^C = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$, $L^A = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$ and $L^B = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$.

The algorithm works as follows: First, it considers column-major layout for C.

$L^C.Q = \begin{pmatrix} 0 & 0 & 1 \\ \times & \times & 0 \end{pmatrix}$. Therefore $q_{11} = q_{12} = q_{23} = 0$ and $q_{13} = 1$.

$L^A.Q = \begin{pmatrix} 0 & 0 & 1 \\ \times & \times & 0 \end{pmatrix}$. Therefore $q_{23} = 0$.

$L^B.Q = \begin{pmatrix} \times & 1 & 0 \\ \times & 0 & 0 \end{pmatrix}$. Therefore $q_{22} = 0$ and $q_{32} = 1$.

At this point $T^{-1} = Q = \begin{pmatrix} 0 & 0 & 1 \\ q_{21} & 0 & 0 \\ q_{31} & 1 & 0 \end{pmatrix}$. By setting $q_{21} = 1$ and $q_{31} = 0$, $T^{-1} = Q = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$. All arrays are column-major and the resulting code is shown in Figure 2:B.

Next the compiler tries the other alternative memory layout, namely row-major, for C.

$L^C.Q = \begin{pmatrix} \times & \times & 0 \\ 0 & 0 & 1 \end{pmatrix}$. Therefore $q_{13} = q_{21} = q_{22} = 0$ and $q_{23} = 1$.

$L^B.Q = \begin{pmatrix} \times & \times & 0 \\ 0 & 0 & 1 \end{pmatrix}$. Therefore $q_{33} = 0$.

$L^A.Q = \begin{pmatrix} \times & 0 & 0 \\ \times & 1 & 0 \end{pmatrix}$. Therefore $q_{12} = 0$ and $q_{32} = 1$.

At this point $T^{-1} = Q = \begin{pmatrix} q_{11} & 0 & 0 \\ 0 & 0 & 1 \\ q_{31} & 1 & 0 \end{pmatrix}$. By setting $q_{11} = 1$ and $q_{31} = 0$, $T^{-1} = Q = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$. All arrays are row-major and the resulting code is shown in Figure 2:C. Notice that our first optimized nest is the same nest obtained by earlier works [18, 16]. Our other optimized nest is the same nest used in Lam *et al.* [15] for row-major layouts.

5 Algorithm for Maximizing Parallelism and Minimizing Communication

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 loops is distributed over the processors; (2) data decomposition across processors is determined for each array; and (3) communication is performed in large chunks, and it is optimized such that all non-local data are transferred to respective local memories before the execution of the innermost loop. Notice that the algorithm can also be used for the shared-memory UMA and NUMA architectures. For the NUMA case, the algorithm performs the functions (1) and (2) listed above; whereas for the UMA case, it performs only the function (1).

5.1 Explanation

As before, let $i_1, i_2,...,i_n$ be the loop indices of the original loop and $j_1, j_2,...,j_n$ be the loop indices of transformed loop. The following is the explanation of the algorithm.

• Our transformation matrix should be such that the LHS array of the transformed loop should have the outermost index as the only element in one of array dimensions. In other words, the LHS array C should be of the form $C(*,*,...,j_1,...,*,*)$ where j_1 (the new outermost loop index) is in the r^{th} dimension. This means that the r^{th} row of the transformed reference matrix for C is $(1,0,...,0,0)$. Then the LHS array can be distributed along the dimension r across processors without any communication occurring.

• Then the algorithm works on one reference from RHS at a time. If a row s of data reference matrix for a RHS array A is identical to a row in the reference matrix for the LHS array, then

```
DO i = 1, n
 DO j = 1, n
  DO k = 1, n
   C(i,j)+=A(i,k)*B(k,j)
  ENDDO k
 ENDDO j
ENDDO i
```
(A)

```
DO u = 1, n
  DO v = 1, n
    DO w = 1, n
      C(w,u)+=A(w,v)*B(v,u)
    ENDDO w
  ENDDO v
 ENDDO u
```
(B)

```
DO u = 1, n
  DO v = 1, n
    DO w = 1, n
      C(u,w)+=A(u,v)*B(v,w)
    ENDDO w
  ENDDO v
ENDDO u
```
(C)

```
DO u = 1, n/p
  DO v = 1, n/p
    receive B(*,v)
    DO w = 1, n
      C(u,v)+=A(u,w)*B(w,v)
    ENDDO w
  ENDDO v
ENDDO u
```
(D)

```
DO u = 1, n/p
  DO v = 1, n
    receive A(*,v)
    DO w = 1, n/p
      C(w,u)+=A(w,v)*B(v,u)
    ENDDO w
  ENDDO v
 ENDDO u
```
(E)

```
DO u = 1, n/p
  DO v = 1, n/p
    receive B(v,*)
    DO w = 1, n
      C(u,w)+=A(u,v)*B(v,w)
    ENDDO w
  ENDDO v
ENDDO u
```
(F)

Figure 2: (A) Matrix multiplication nest. (B)-(C) Locality optimized versions of (A). (D)-(E) Parallelism optimized versions of (A). (E)-(F) Versions obtained by the unified algorithm.

Step 1 Initialize $i = 1$.

Step 2 Set $\vec{\ell}_i^C.Q = (1, 0, ..., 0, 0)$ i.e. distribute LHS array across processors along dimension i.

Step 3 For all array references A on the RHS that have $\vec{\ell}_l^A = \vec{\ell}_i^C$ for some l, distribute array A along the dimension l.

Step 4 Choose an array reference A for which the equality in **Step 3** does not hold. Initialize $j = 1$.

Step 5 Set $\vec{\ell}_j^A.Q = (0, 0, ..., 0, 1)$ and $\vec{\ell}_k^A.Q = (\times, \times, ..., \times, 0)$ for each $k \neq j$. If a valid Q is found, check the determinant of it. If non-zero block transfers are possible for that RHS array, go to **Step 6**. If there are no valid Q or the determinant of Q is zero for all j, block transfers are not possible on that array with the given distribution of the LHS array; increment j and go to **Step 5**.

Step 6 Repeat **Step 5** for all reference matrices of a particular A.

Step 7 Repeat **Step 5** for all distinct array references.

Step 8 Record the obtained transformation matrix. Also record the number of arrays for which there is no communication and the number of arrays for which block transfers are possible.

Step 9 Increment i and go to **Step 2** (try a different distribution for the LHS array).

Step 10 Compare all alternatives and choose the best one.

Figure 3: Compiler algorithm for data decomposition and parallelism.

it is always possible to distribute that array along s^{th} dimension across processors without any communication.

• If the condition above does not hold for a RHS reference for an array A, then the entries for Q should be chosen such that some dimension of that reference consists only of the innermost loop index, and the other dimensions are independent of the innermost loop index. That is, the RHS transformed reference should be of the form $A(*, *, ..., j_n, ..., *, *)$ where $*$ indicates a term independent of j_n. If this condition is satisfied, the communication arising from that RHS reference can be moved out of the innermost loop.

• Then (if desired) the previous step is repeated, this time attempting to move the communication out of the second innermost loop. This process terminates when a loop is encountered outside of which the communication cannot be moved to.

The algorithm is presented in Figure 3. The details of this algorithm can be found elsewhere [20].

5.2 Example

To illustrate the technique we consider again the naive matrix multiplication nest shown in Figure 2:A.
The algorithm works as follows:
$\mathcal{L}^C.Q = \begin{pmatrix} 1 & 0 & 0 \\ \times & \times & \times \end{pmatrix}$. Therefore $q_{11} = 1$, $q_{12} = 0$ and $q_{13} = 0$.
Since $\vec{\ell}_1^A = \vec{\ell}_1^C$, A can be distributed along the first dimension as well.
$\mathcal{L}^B.Q = \begin{pmatrix} 0 & 0 & 1 \\ \times & \times & 0 \end{pmatrix}$. Therefore $q_{31} = q_{32} = q_{23} = 0$ and $q_{33} = 1$.
At this point $Q = \begin{pmatrix} 1 & 0 & 0 \\ q_{21} & q_{22} & 0 \\ 0 & 0 & 1 \end{pmatrix}$. The remaining entries should be selected such that the rank of Q should be 3, and no dependence is violated. In this case the compiler can set $q_{21} = 0$ and $q_{22} = 1$. This results in the identity matrix meaning that no transformation is needed. A and C are distributed by rows, and B by columns. The resulting program with the data transfer call is shown in Figure 2:D. Note that the communication is performed outside the innermost loop.
Next the compiler tries to distribute C in the second dimension.
$\mathcal{L}^C.Q = \begin{pmatrix} \times & \times & \times \\ 1 & 0 & 0 \end{pmatrix}$. Therefore $q_{21} = 1$, $q_{22} = 0$ and $q_{23} = 0$.
Since $\vec{\ell}_2^B = \vec{\ell}_2^C$, B can be distributed along the second dimension as well.
$\mathcal{L}^A.Q = \begin{pmatrix} 0 & 0 & 1 \\ \times & \times & 0 \end{pmatrix}$. Therefore $q_{11} = q_{12} = q_{33} = 0$ and $q_{13} = 1$.
At this point $Q = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ q_{31} & q_{32} & 0 \end{pmatrix}$. The remaining entries should be selected such that the rank of Q should be 3, and no dependence is violated. The compiler sets $q_{31} = 0$ and $q_{32} = 1$. This results in $Q = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$. All arrays are distributed by columns. The resulting program is shown in Figure 2:E. We note that the performance of the loop is similar in both the cases.

6 Unified Algorithm

This section presents a unified greedy algorithm which combines the characteristics of the two algorithms presented in the previous two sections (Sections 4 and 5). Specifically the unified algorithm (1) transforms the nest such that the outermost loop can be run in parallel across the processors; (2) decomposes each array across the processors; (3) optimizes interprocessor communication by vectorizing it whenever possible; (4) optimizes locality (cache performance) by assigning appropriate storage layout for each array, and by transforming the iteration space. For the distributed-memory multicomputers, all four steps can be applied. For the shared-memory NUMA case, the functions (1), (2) and (4) are attempted; and for the UMA case, on the other hand, only the functions (1) and (4) can be performed.

6.1 Explanation

In this subsection we give the details of the unified algorithm. We assume that there is a single reference per array in the loop nest. An array is said to be *optimized for parallelism* if it can be distributed along an array dimension where only j_1 (new outermost loop index) appears without any communication. An array is said to be *degree* α *optimized for communication* if it cannot be optimized for parallelism, but communication for it can be performed before the α^{th} loop, where $1 \le \alpha \le n$. An array optimized for parallelism is said to be degree 0 optimized for communication (essentially meaning that it needs no communication (non-local access)). An array is said to be *degree* β *optimized for locality* if it contains the loop index $j_{n-\beta+1}$ in an array dimension and it can be stored on memory such that this array dimension will be the fastest changing array dimension ($1 \le \beta \le n$).

We associate a pair (α, β) for each array reference where α and β denote the degree of communication and locality respectively. It can be seen that the pair $(0, 1)$ is the *best possible pair* for an array reference. Our algorithm tries to achieve this best possible pair for as many references as possible. If the best pair is not possible, the selection of the next pair to be considered depends on whether parallelism is favored over locality or vice versa. In our case, for example for a 3-deep nest in which 2-dimensional arrays are accessed, we took a modest approach and followed the sequence $(0, 1)$, $(0, 2)$, $(3, 1)$; that is, if an array reference cannot be optimized for parallelism, we checked only for the case where the communication can be taken out of the innermost transformed loop. If $(3, 1)$ is tried unsuccessfully, we chose to apply pure communication or pure locality optimization.

Theoretically, if there are enough loop indices and array dimensions, an array reference C can be transformed to the form $C(*, *, .., *, j_1, *, .., *, j_n, *, .., *, *)$ where $*$ denotes a subscript independent of j_n. If such a transformation is possible, then C can be distributed across processors along the dimension where j_1 occurs alone, and at the same time local portions of it can be stored on memory such that the dimension where j_n occurs will be the fastest changing dimension. The problem is that in most of the cases, number of loops and number of array dimensions are small values, and consequently, the number of entries in T^{-1} is small (e.g. 4, 9 etc.). Once the above form is obtained for a reference, since most of the entries of T^{-1} are already determined, the chances of optimizing the other references would be low. Because of this fact, our algorithm should consider other degrees of communication and locality as well.

Each degree of communication and locality suggest a number of possible transformed reference matrices. For a reference, optimization for pair (α, β) can be formulated as problem of finding a transformed reference matrix which is suitable for both α degree communication and β degree locality. For example, Table 1 presents a few transformed reference matrices for several (α,β) pairs for a three-deep nest in which two-dimensional arrays are accessed. The unified algorithm is given in Figure 4. $\mathcal{L}^i$ denotes the original reference matrix for the i^{th} array in the nest, $i = 1$ corresponding to the LHS array. The j^{th} possible transformed reference matrix for an (α, β) pair is denoted by $\mathcal{R}^j{}_{(\alpha,\beta)}$.

Table 1: Array reference matrices for commonly used (α, β) pairs for a two-dimensional array enclosed in a three-deep loop nest.

(0,1)	(0,2)	(3,1)
$\begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 & 0 \\ \times & 1 & 0 \end{pmatrix}$	$\begin{pmatrix} 0 & 0 & 1 \\ \times & \times & 0 \end{pmatrix}$
$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} \times & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$	$\begin{pmatrix} \times & \times & 0 \\ 0 & 0 & 1 \end{pmatrix}$

Step 1 Initialize $i = 1$. Initialize $(\alpha, \beta) \leftarrow (0, 1)$ (try the best possible optimization).

Step 2 Initialize $j = 1$. (try the first transformed reference matrix for this (α, β) pair).

Step 3 Set $\mathcal{L}^i.Q = \mathcal{R}^j{}_{(\alpha,\beta)}$.

If there is no inconsistency, then go to **Step 4**; else increment j (try the next possible transformed reference matrix for this (α, β) pair) and repeat this step.

If there are inconsistencies for every value of j, then increment (α, β) pair (try the next pair on the trial sequence) and repeat this step.

If there are inconsistencies for all (α, β) pairs, then apply pure communication or pure locality optimization for this reference.

Step 4 Increment i and go to **Step 2** (optimize the next array reference).

Step 5 When a Q is found, record it. Also record the associated (α, β) pairs for each array reference.

Step 6 When all solutions are obtained, choose the best alternative by comparing (α, β) values.

Figure 4: Unified compiler algorithm for optimizing locality, parallelism and communication.

6.2 Example

We consider the original matrix multiplication nest of Figure 2:A once again. As before we only show the successful trials.
$\mathcal{L}^C.Q = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}$ with $(\alpha, \beta) = (0, 1)$. Thus, $q_{11} = 0$, $q_{12} = 0$, $q_{13} = 1$, $q_{21} = 1$, $q_{22} = 0$ and $q_{23} = 0$.
$\mathcal{L}^A.Q = \begin{pmatrix} 0 & 0 & 1 \\ \times & \times & 0 \end{pmatrix}$ with $(\alpha, \beta) = (3, 1)$. Therefore $q_{33} = 0$.
$\mathcal{L}^B.Q = \begin{pmatrix} \times & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$ with $(\alpha, \beta) = (0, 2)$. Therefore $q_{32} = 1$.
At this point $T^{-1} = Q = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ q_{31} & 1 & 0 \end{pmatrix}$. By setting $q_{31} = 0$, $T^{-1} = Q = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$. The resulting program is shown in Figure 2:E. All arrays are column-wise decomposed across processors. The arrays C and B are optimized for parallelism ($\alpha = 0$), whereas the array A is optimized for communication with $\alpha = 3$. The arrays C and A are optimized for locality in the innermost loop, whereas for array B the locality is exploited in the second loop.
Next the algorithm considers the other alternative for the array C.
$\mathcal{L}^C.Q = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$ with $(\alpha, \beta) = (0, 1)$. Therefore $q_{11} = 1$, $q_{12} = 0$, $q_{13} = 0$, $q_{21} = 0$, $q_{22} = 0$ and $q_{23} = 1$.
$\mathcal{L}^A.Q = \begin{pmatrix} 1 & 0 & 0 \\ \times & 1 & 0 \end{pmatrix}$ with $(\alpha, \beta) = (0, 2)$. Therefore $q_{32} = 1$ and $q_{33} = 0$.
$\mathcal{L}^B.Q = \begin{pmatrix} \times & \times & 0 \\ 0 & 0 & 1 \end{pmatrix}$ with $(\alpha, \beta) = (3, 1)$. Therefore $q_{32} = 1$.
At this point $T^{-1} = Q = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ q_{31} & 1 & 0 \end{pmatrix}$. By setting $q_{31} = 0$, $T^{-1} = Q = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$. The resulting program is shown in Figure 2:F. All arrays are row-wise decomposed across processors.

The arrays C and A are optimized for parallelism ($\alpha = 0$), whereas the array B is optimized for communication with $\alpha = 3$. The arrays C and B are optimized for locality in the innermost loop, whereas for array A the locality is exploited in the second loop.

7 Global Optimization Problem

In this section, we concentrate on the global optimization problem; that is, optimizing a number of consecutive loop nests simultaneously. In fact, we will handle a sub-problem, namely optimizing locality across a number of loop nests. The other part of the global problem, data decomposition across processors in multiple nests, was handled extensively [23, 7, 2] and is beyond the scope of this paper. Although the algorithm to be presented in this section can easily be modified to incorporate optimal global data decomposition detection as well, for the sake of clarity we assume in this section that all possible data decompositions across processors are equally acceptable. We plan on integrating the global algorithm to be presented here with the techniques given by [23], [7] and [2]. Throughout this section, we assume that the algorithm presented in 6, henceforth referred as *local()*, is run for each individual loop nest, and all possible optimized memory layouts and loop orders are determined. Due to lack of space, we only give formal definition of the problem; and sketch an approach to attack it.

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 assume that *local()* is run for each nest, and a number of possible optimized layout combinations are obtained for each nest. The problem of finding a global array layout combination that satisfies all the nests is NP-complete even for the restricted case where only row-major (r-m) and column-major (c-m) memory layouts are considered. We search for a near-optimal solution with polynomial time which is good enough in practice. Let $LL^{\ell}_{\mathcal{N}}(\mathcal{A})$ be a local layout for an array $\mathcal{A}$ in a combination ℓ for nest $\mathcal{N}$ and $GL(\mathcal{A})$ be the global layout for array $\mathcal{A}$. We define the following parameter:

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

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

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

Now we can formulate the global layout determination problem as a problem of finding global layout assignments for all arrays (that is, to determine $GL(\mathcal{A})$ for each $\mathcal{A}$) and corresponding local layout assignments for each nest (that is, to determine ℓ for each $\mathcal{N}$) such that $\sum_{\mathcal{N}} \sum_{\mathcal{A}} \mu(\mathcal{A}, \mathcal{N}, \ell)$ is minimized.

Let us now concentrate on an example in which 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 case of tie, choose a layout arbitrarily. The complexity of this heuristic is $\theta(n \times m)$ where n is the number of nests and m is the number of arrays. Unfortunately, *local()* can return multiple alternatives for a single nest. Assuming p alternatives per nest, a simple extension of the above heuristic results in $\theta(p^n \times n \times m)$ complexity which is not acceptable unless n is very small. In what follows we formulate the problem on a DAG (directed acyclic graph) as in [13] and solve it using a *shortest path* algorithm.

Let *alternatives($\mathcal{N}$)* be a function that gives the number of alternative layout combinations for nest $\mathcal{N}$. Similarly let *arrays($\mathcal{N}$)* be a function that gives the number of distinct arrays referenced in nest $\mathcal{N}$. Our approach consists of four steps:

(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 *interference graph* by Anderson *et.al.* [1], and they use it to solve the global data decomposition problem. Then an algorithm to find the 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. The complexity of the connected components algorithm on a bipartite graph is $\theta(n+m)$ where n is the number of nests and m is the number of arrays [6]. The following steps operate on a single connected component at a time.

(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. Different heuristics can be used to determine an order for the loop nests. The two of them are

`min-edge` The idea here is to order the nests such that the total number of the edges in the DAG will be minimized.

`max-accuracy` This heuristic tries to increase the accuracy of the solution at the expense of a more complex DAG.

(**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 alternative. There is a directed edge from $N_{i,j}$ to $N_{i+1,k}$ for all $1 \le j \le alternatives(\mathcal{N}_i)$ and $1 \le k \le alternatives(\mathcal{N}_{i+1})$. 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. 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 $1 \le j \le alternatives(\mathcal{N}_1)$, and an edge from $\mathcal{N}_{n,k}$ to T for all $1 \le k \le alternatives(\mathcal{N}_n)$. Then a shortest path algorithm for this DAG is run from S to T. The path with the minimum cost gives a good local layout combination for each nest.

(**4**) The final phase of the heuristic determines the global memory layouts using the local layout assignments obtained in the previous step. We refer to the shortest path obtained in the previous step by δ; and the i^{th} 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 $\mathcal{A}$. In order to resolve this conflict the layout for $\mathcal{A}$ should be changed either in δ_i or in δ_{i+1}, as we do not consider data redistribution in this paper. 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 for which there are conflicting demands, the number of r-m and c-m 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 problem where there is a single alternative per nest. After that, the local layouts in a nest which are different from global layouts are changed accordingly.

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

```
DO i = 1, n
 DO j = 1, n
  DO k = 1, n
   DO l = 1, n
    A(i,j)+=B(k,i)+C(l,k)
   ENDDO l
  ENDDO k
 ENDDO j
ENDDO i
```

(A)

```
DO u = 1, n
 DO v = 1, n
  receive C(v,*)
  DO w = 1, n, n
   DO y = 1, n
    A(u,y)+=B(w,u)+C(v,w)
   ENDDO y
  ENDDO w
 ENDDO v
ENDDO u
```

(B)

```
DO u = 1, n
 DO v = 1, n
  receive B(v,*)
  receive C(*,v)
  DO w= 1, n
    DO y=1, n
     A(y,u)+=B(v,y)+C(w,v)
    ENDDO y
  ENDDO w
 ENDDO v
ENDDO u
```

(C)

Figure 5: (A) A four-deep loop nest. (B) Optimized version of (A). (C) Optimized version of (A).

8 Simulation Results and Experiments

In this section, we present our simulation and experimental results on three programs: matrix multiplication, a four deep loop nest, and a simple benchmark. The last program is optimized using the global optimization algorithm presented in Section 7 as it contains more than one nest. The algorithms presented in this paper are applied manually to the programs, then the loops pointed by the algorithms are parallelized using `INDEPENDENT` directive of the HPF [3] language. We demonstrate the simulation results obtained by using an enhanced version of DineroIII [8], a trace-driven cache simulator. We simulate the miss rates over a range of cache sizes (4K, 8K, 16K, 32K, 64K, 128K), block (cache line) sizes (8, 16, 32, 64, 128, 256) and set-associativities (direct-mapped, 2-way, 4-way, full-associative) for single processor. Also presented are empirical results obtained on SPARCstation 5, IBM SP-2, SGI Challenge and Convex Exemplar. SPARCstation 5 has a 16K direct-mapped data cache and a 32 MB memory. IBM SP-2 is a distributed-memory message-passing machine and has RS/6000 Model 590 processors, each with a 256 KB data cache. SGI Challenge has a logically and physically shared memory system (a UMA architecture). It uses snoopy write-invalidate cache coherence. Each node has a 16KB direct-mapped data cache and a 4MB L2 data cache, which is 4-way set-associative. The cache line size is 32 bytes on the internal cache and 128 bytes on the L2 cache. In SGI experiments, static scheduling has been employed. The exemplar SP-1200, on the other hand, has 1MB data cache. The line size is 32 bytes, and the cache is direct-mapped. Due to space concerns, we present only a subset of our simulation and experimental results.

Tiling is a technique to improve the locality and parallelism, and is a combination of strip-mining and loop permutation [28, 29, 10, 22]. Due to interference misses it is difficult to select a suitable tile size. In other words, unless the tile size is tailored according to the matrix size and cache parameters, the performance of tiling may be rather poor [5, 15]. Our algorithm improves the performance of tiling as it enhances inter-tile locality.

Matrix Multiplication: Figure 6 demonstrates the miss ratios for the matrix multiplication nest with 500×500 double arrays on a direct-mapped cache. We present four different versions of the program: unoptimized (Unopt), optimized (all arrays column-major,Opt), tiled version of the unoptimized nest (Unopt+Tiled) and tiled version of the optimized nest (Opt+Tiled). The first thing to notice is that the tiled-optimized version outperforms the rest for all cache and block sizes. It is also important to note that, for some cases, even optimized nest without tiling performs better than the tiled-unoptimized version (e.g. with cache size=8K, and block size=128.) The tile size is fixed at 32 elements for the tiled versions. The results clearly show the effectiveness of our approach at improving the cache locality in uniprocessors.

Figure 7 illustrates the *insensitivity* of the optimized tiled version to the tile size. The numbers above the bars denote the tile sizes. Notice that while the miss ratio of the unoptimized-tiled version increases with the tile size, that of the optimized-tiled version is quite stable. We note that for this example, similar results have been obtained by Li [16] as well.

Figures 8:A and B show the execution times for the matrix multiplication nest with different input sizes on SPARCstation 5 and a single node of SGI Challenge respectively; and Figure 8:C gives the execution times on different number of processors on SGI Challenge with 1000×1000 double matrices. We note that for 1000×1000 double matrices there is 20% performance improvement over the unoptimized nest on a single node of SGI Challenge; and on SPARCstation 5 with 500×500 double matrices nearly 200 seconds are saved (40% performance improvement). The improvement on multiprocessors comes from both eliminating false sharing and exploiting spatial locality.

Figure 9:A shows the performance of our approach on SP-2 with 512×512 real arrays using *pghpf* [3] compiler with no optimizations turned on. We report the execution times for four different versions:

Unopt_Nocomm: Unoptimized version without communication. The loop order and memory layouts are not changed; arrays C and A are distributed by rows across the processors; and array B is replicated resulting in no communication.

Unopt_Comm: Unoptimized version with communication. The loop order and memory layouts are not changed; arrays C and A are distributed by rows whereas array B is distributed by columns.

Opt_Nocomm: Optimized version without communication. The loop order is shown in Figure 2:E, and all arrays have column-major layout. Arrays C and B are distributed by columns, while array A is replicated.

Opt_Comm: Optimized version with communication. The loop order is shown in Figure 2:E. All arrays are column-major and distributed by columns across the processors.

Opt_Nocomm has the best execution times as can be seen from Figure 9:A; another point is that, for all processor sizes, it is better than Unopt_Nocomm and Opt_Comm is better than the Unopt_Comm. Clearly, this improvement comes from the locality optimizations. We note that the super-linear speedups in some cases are due to cache effects. It is interesting to observe that the execution times for Opt_Comm are very close to those of Unopt_Nocomm, the average difference being 1.5 seconds. This is because of the fact that the compiler optimizes inter-processor communication aggressively, and consequently locality optimizations become more and more important. This result also implies that if locality is not taken into account, the parallelized versions of scientific nests may not produce the desired speedups on multicomput-

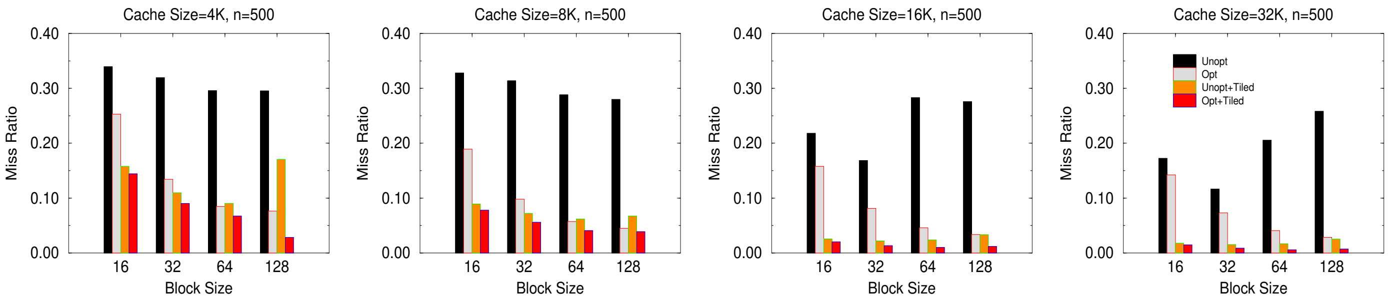


Figure 6: Simulation results for matrix multiplication.

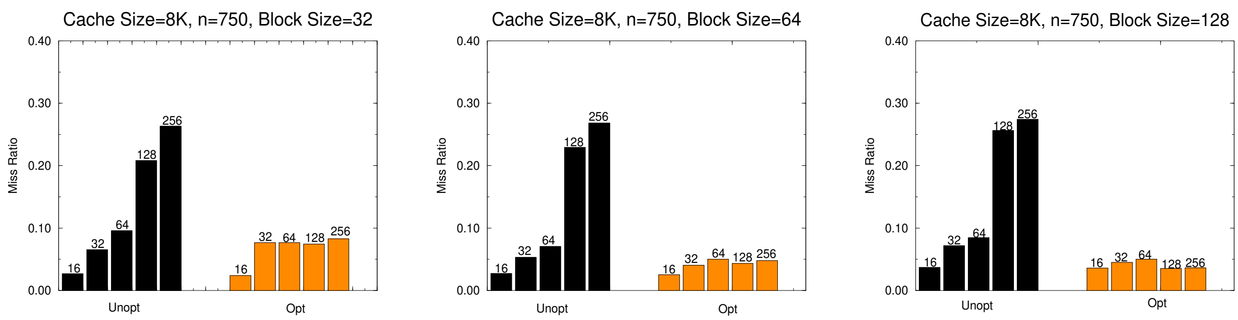


Figure 7: Tile size sensitivity for matrix multiplication.

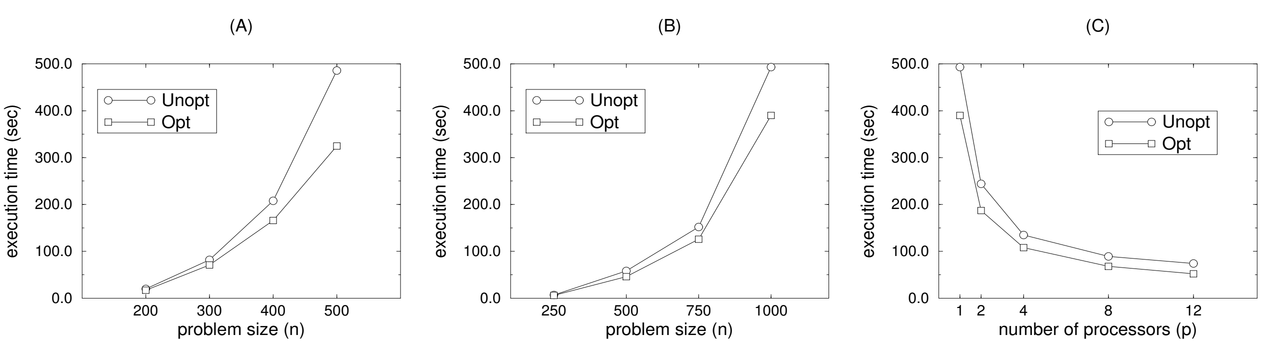


Figure 8: Execution times for matrix multiplication (A) on SPARCstation 5 (B) on a single node of SGI. (C) on multiple nodes of SGI with 1000×1000 double matrices.

ers, even if the maximum degree of parallelism is obtained and the communication is minimized.

Finally, Figure 9:B shows the performance of the matrix multiplication on the Convex Exemplar. The same four versions[1] described earlier were run on up to 32 processors. We note that Opt_Nocomm outperforms all other versions for almost all processor sizes. The reduction in the performance observed beyond 8 processors is not specific to this example and can be attributed to poor inter-hypernode performance in Convex [25].

A Four-Deep Loop Nest: While matrix multiplication does not take benefit of the layout flexibility, the four deep loop nest shown in Figure 5:A does. Array reference matrices for this nest are as follows. $L^A = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$, $L^B = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}$ and $L^C = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$. Application of the unified algorithm results in two optimized node programs as shown in Figures 5:B and C respectively. In Figure 5:B, array A is optimized with $(0,1)$, reference B is optimized with $(0,2)$, and reference C is optimized with $(3,2)$. Arrays A and C have row-major layouts and are distributed by rows, whereas array B is column-major and is distributed by columns. Before the w-loop, communication is performed for C. In Figure 5:C, on the other hand, reference A is optimized with $(0,1)$ and reference C is optimized with $(3,2)$, incurring communication before the w-loop. Array B could only be optimized for locality; and communication is needed for it before the w-loop. Arrays A and C are row-major whereas array B is column-major.

Figure 10 illustrates the lack of sensitivity to the tile size exhibited by the optimized tiled versions. We ran experiments with three different versions: Unopt, unoptimized version (Figure 5:A); W-Opt, optimized version by fixing row-major layouts for all arrays; and Opt one of the versions obtained by our approach (Figure 5:B). As before, the numbers above the bars denote the tile sizes. Notice that while the miss ratio of the unoptimized tiled version is very unstable, that of the optimized version (Figure 5:B) is stable. Also note that our version outperforms W-Opt for all tile, block (cache line) and cache sizes; that is, our approach exploits inter-tile locality better.

Figures 11:A and B present the execution times for this example with different input sizes on SPARCstation 5 and a single node of SGI respectively. Opt-1 and Opt-2 denote the optimized versions obtained by our algorithm (Figures 5:B and C). Figures 11:C and D, on the other hand, show the execution times on multiple nodes of SGI Challenge with 150×150 and 200×200 double arrays respectively. It can be seen that although the approach based on loop transformations alone can improve the performance, our approach results in the best execution times on both uniprocessor and multiprocessors. In SPARC, for example, with 250×250 double matrices our approach (Opt-2) runs in almost 800 seconds less than the W-Opt. On four nodes of the SGI Challenge, with 200×200 double arrays our version (Opt-2) saves 36 more seconds than W-Opt. This example clearly shows that relaxing the memory layouts can save substantial amounts of time for some nests.

Figure 12:A demonstrates the performance of the four deep loop nest on SP-2 with 128×128 real arrays. We compare the versions given in Figures 5:A, B and C respectively. As shown in the figure, the optimized versions perform similarly, and they outperform the unoptimized version for all processor sizes. Finally, Figure 12:B shows the performance of the four deep loop nest on the Convex Exemplar. The versions given in Figures 5:A and C compared. The results illustrate the effectiveness of our approach; although the performance degrades beyond 8 processors for all versions.

A Simple Benchmark: We now show the impact of our global optimization algorithm (Section 7) on a simple benchmark that can benefit from layout optimization. The program shown in Figure 13:A is from [4]. The left part of Figure 14 shows the improvement obtained by our approach in terms of *normalized* miss rates. For each cache size, the three bars from bottom to top correspond to unoptimized version with column-major layouts (Unopt-C), unoptimized version with row-major layouts (Unopt-R) and version optimized by our approach (Opt) respectively. In the optimized version, the loops in the first nest are interchanged; and the following layouts are assigned: A, C, and D are column-major; B and E are row-major. Also, in the optimized version, the outermost loop in each nest is parallelized, and the arrays A, C and D are distributed column-wise while the arrays B and E are distributed row-wise. With these optimizations, the spatial locality for every reference is exploited in the innermost loop and the optimized program is given in Figure 13:B. In simulations, 400×400 double matrices are used.

The right part of Figure 14 shows the execution times on SP-2 with 2048×2048 real arrays. For a single processor, the problem size was too big to fit in the memory; so we ran the experiments on 2, 4, 8 and 16 processors. As can be seen from the figure, the optimized version improves the performance substantially (30% to 50%).

From the simulation results and our experiments we can conclude that our optimizations in general reduce the miss rate and sensitivity of the program to the cache size, and improve the scalability of the program by improving spatial locality. The effect of our algorithm on false sharing, though, is yet to be evaluated quantitively.

9 Conclusions

The broad variety of parallel architectures render designing unified compiler techniques difficult. However, although the underlying hardware facilities are different, we believe all types of parallel architectures will benefit from compiler optimizations which aim good locality and large-granular parallelism. In this paper, we described a constructive algorithm that handles locality, parallelism and data decomposition in a unified manner. Our algorithm derives the terms of a transformation matrix such that the best locality and minimum communication are obtained. Our experiments with message-passing, UMA and NUMA architectures demonstrate the effectiveness of our approach on different parallel platforms.

Further research will involve investigating compiler algorithms which can handle complex data distributions (e.g. distributions along more than one dimension) and spatial locality in a unified manner for both single-nest and multiple-nest cases.

Acknowledgments

We would like to thank Eric Schwabe of Northwestern University, and Ilteris Demirkiran of Syracuse University for enlightening discussions on the global layout optimization problem.

[1] We should emphasize that the data distribution across processors are dictated in the HPF compiler level. How these directives are interpreted in terms of Convex data distribution directives (e.g. *block-shared*) is not investigated here.

References

[1] J. M. Anderson, S. P. Amarasinghe, and M. S. Lam. Data and computation transformations for multiprocessors. In *Proc. Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, July 1995.

[2] J. M. Anderson, and M. S. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *P*roc. SIGPLAN Conference on Programming Language Design and Implementation, pages 112–125, June 1993.

[3] 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*, Banglore, India, December 1994.

[4] M. Cierniak, and W. Li. Unifying data and control transformations for distributed shared memory machines. *Proc. SIGPLAN '95 Conference on Programming Language Design and Implementation*, La Jolla, California, June 1995.

[5] S. Coleman, and K.S. 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.

[6] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*, The MIT Press, Cambridge, MA, 1990.

[7] M. Gupta, and P. Banerjee. Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers. In *IEEE Transactions on Parallel and Distributed Systems*, 3(2):179-193, March 1992.

[8] M. D. Hill, and A. J. Smith. Evaluating associativity in CPU caches. *IEEE Trans. on Computers*, C-38, 12, December 1989, pages 1612–1630.

[9] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiling Fortran D for MIMD distributed memory machines. *Communications of the ACM*, 35(8):66-88, August 1992.

[10] F. Irigoin, and R. Triolet. Supernode partitioning. *Proc. 15th Annual ACM Symp. Principles of Programming Languages*, pages 319-329, San Diego, CA, January 1988.

[11] Y.-J. Ju and H. Dietz. Reduction of cache coherence overhead by compiler data layout and loop transformations. *Proc. 4th Workshop on Languages and Compilers for Parallel Computing*, Santa Clara, CA, August 1991.

[12] M. T. Kandemir, J. Ramanujam, and A. N. Choudhary. A Compiler Algorithm for Optimizing Locality in Loop Nests. In *Proc. 11th ACM International Conference on Supercomputing*, pages 269–276, Vienna, Austria, July 1997.

[13] K. Kennedy, and U. Kremer. Automatic data layout for High Performance Fortran. In *Proceedings of Supercomputing'95*, San Diego, CA, December 1995.

[14] C. Koelbel, D. Lovemen, R. Schreiber, G. Steele, and M.Zosel. *High Performance Fortran Handbook*. The MIT Press, 1994.

[15] 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.

[16] W. Li. Compiling for NUMA parallel machines. Ph.D. Thesis, Cornell University, Ithaca, NY, 1993.

[17] W. Li, and K. Pingali. Access Normalization: Loop restructuring for NUMA compilers. *ACM Transactions on Computer Systems*, November 1993.

[18] K. McKinley, S. Carr, and C.W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems,* 1996.

[19] M. F. P. O'Boyle, and P. M. W. Knijnenburg. Non-singular data transformations: Definition, validity, applications. In *Proc. 6th Workshop on Compilers for Parallel Computers*, Aachen, Germany, 1996.

[20] 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,* D. Bailey *et al.,* Eds., pp. 668–673, February 1995.

[21] J. Ramanujam. Beyond unimodular transformations. *The Journal of Supercomputing*, 9(4):365–389, 1995.

[22] J. Ramanujam, and P. Sadayappan. Tiling multidimensional iteration spaces for multicomputers. *Journal of Parallel and Distributed Computing*, 16(2):108-120, October 1992.

[23] 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 Computation*, pages 324-331, McLean, VA, February 1995.

[24] 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.

[25] R. Thekkath, A. P. Singh, J. P. Singh, S. John, and J. Hennessey. An Evaluation of a commercial CC-NUMA architecture: The CONVEX Exemplar SPP1200. In *Proc. 11th International Parallel Processing Symposium*, Geneva, Switzerland, April 1997.

[26] 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 Compilation Techniques (PACT)*, June 1995.

[27] M. Wolf, and M. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452-471, October 1991.

[28] M. Wolf, and M. Lam. A data locality optimizing algorithm. In *Proc. ACM SIGPLAN 91 Conf. Programming Language Design and Implementation*, pages 30–44, June 1991.

[29] M. Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley Publishing Company, CA, 1996.

[30] H. Zima and B. Chapman. Compiling for distributed-memory systems. *Proceedings of the IEEE*, 81(2):264-287, 1993.

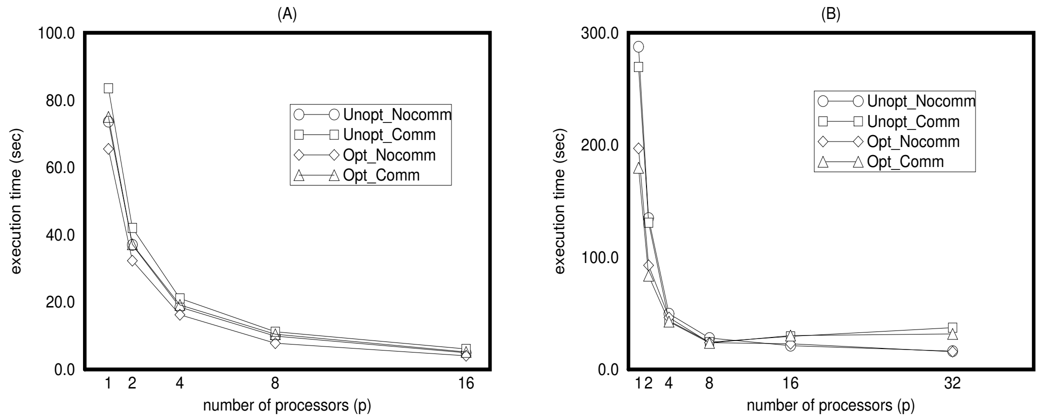


Figure 9: Execution times for matrix multiplication on (A) SP-2 and (B) Convex Exemplar

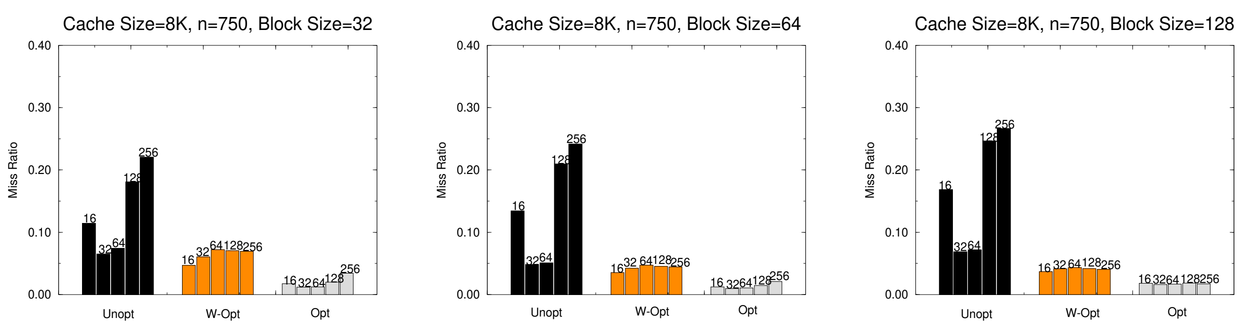


Figure 10: Tile size sensitivity for a four-deep nest.

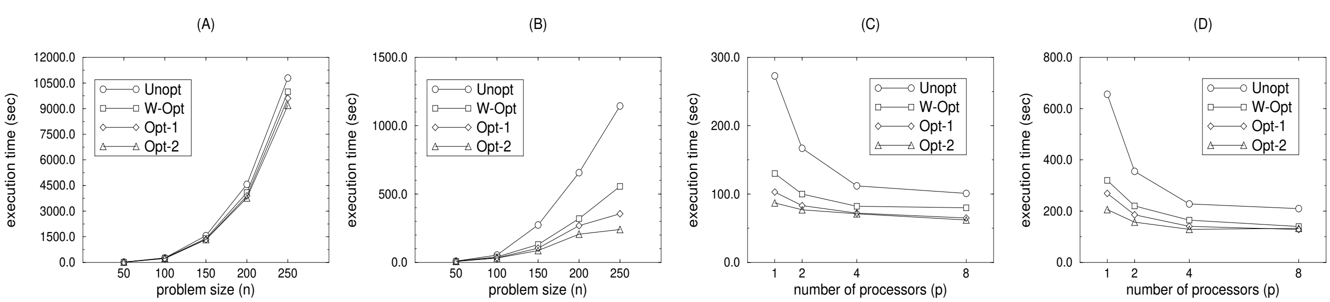
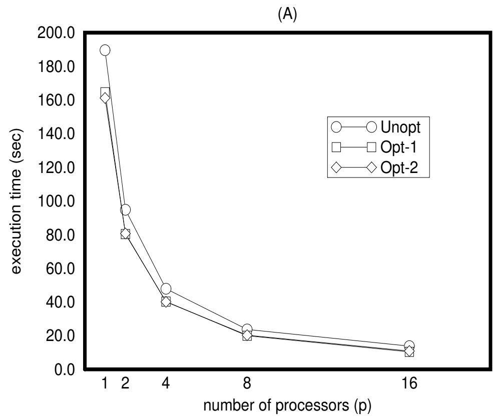


Figure 11: Execution times (A) on SPARCstation 5. (B) on a single node of SGI. (C) on multiple nodes of SGI with 150×150 double arrays. (D) on multiple nodes of SGI with 200×200 double arrays.



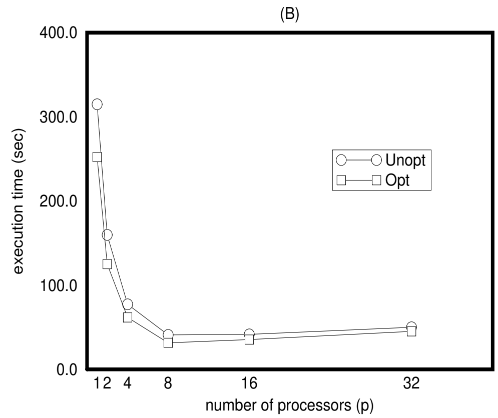


Figure 12: Execution times for a four deep loop nest on (A) SP-2 (B) Convex Exemplar.

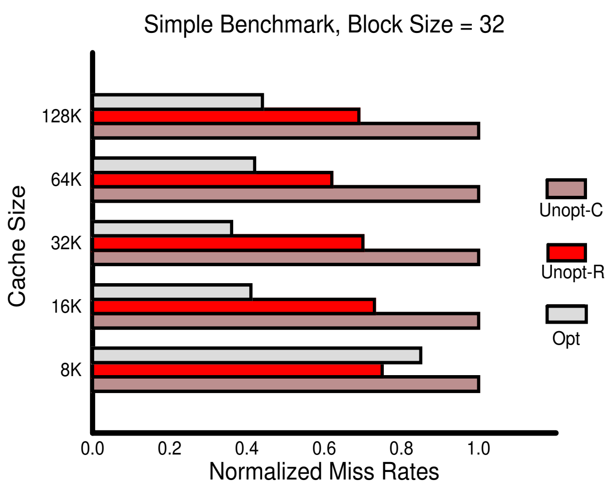
```
DO i = 1, n
  DO j = 1, n
    A[i,j]=B[j,i]*C[i,j]+D[i,j]*LOG(E[j,i])
  ENDDO j
ENDDO i

DO i = 1, n
  DO j = 1, n
    B[i,j]=A[j,i]+E[i,j]
  ENDDO j
ENDDO i
     (A)
```

```
DO j = 1, n
  DO i = 1, n
    A[i,j]=B[j,i]*C[i,j]+D[i,j]*LOG(E[j,i])
  ENDDO i
ENDDO j

DO i = 1, n
  DO j = 1, n
    B[i,j]=A[j,i]+E[i,j]
  ENDDO j
ENDDO i
     (B)
```

Figure 13: (A) A simple benchmark. (B) Optimized version of (A).



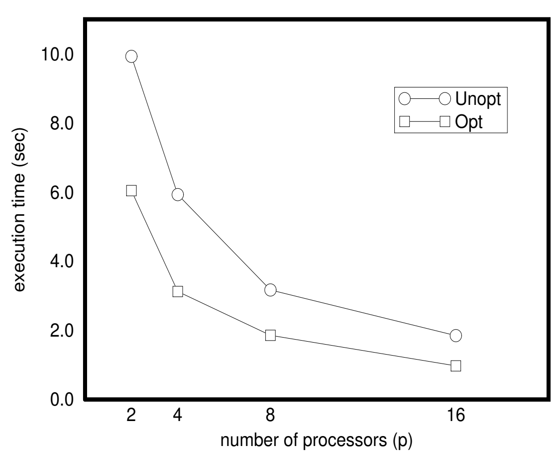


Figure 14: Normalized miss rates for simple benchmark (left) and execution times on SP-2 (right).