Communication-Minimal Partitioning of Parallel Loops and Data Arrays for Cache-Coherent Distributed-Memory Multiprocessors

Rajeev Barua, David Kranz and Anant Agarwal

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139 *

December 19, 1994

Abstract

Harnessing the full performance potential of cache-coherent distributed shared memory multiprocessors without inordinate user effort requires a compilation technology that can automatically manage multiple levels of memory hierarchy. This paper describes a working compiler for such machines that automatically partitions loops and data arrays to optimize locality of access.

The compiler implements a solution to the open problem of finding communication-minimal partitions of loops and data. Loop and data partitions specify the distribution of loop iterations and array data across processors. A good loop partition maximizes the cache hit rate while a good data partition minimizes remote cache misses. The problems of finding loop and data partitions interact when multiple loops access arrays with differing reference patterns. Our algorithm handles programs with multiple nested parallel loops accessing many arrays with array access indices being general affine functions of loop variables. It discovers communication-minimal partitions when communication-free partitions do not exist. The compiler also uses sub-blocking to handle finite cache sizes.

A cost model that estimates the cost of a loop and data partition given machine parameters such as cache, local and remote access timings, is presented. Minimizing the cost as estimated by our model is an NP-complete problem, as is the fully general problem of partitioning. A heuristic method which provides solutions in polynomial time is presented.

The loop and data partitioning algorithm has been implemented in the compiler for the MIT Alewife machine. The paper presents results obtained from a working compiler on a 16-processor machine for three real applications: Tomcatv, Erlebacher, and Conduct. Our results demonstrate that combined optimization of loops and data can result in improvements in runtime by nearly a factor of two over optimization of loops alone.

1 Introduction

Cache-coherent distributed shared memory multiprocessors have multiple levels in their memory hierarchy which must be managed to obtain good performance. These levels include local cache, local memory, and remote memory that is accessed by traversing an interconnection network. Although local memory has a longer access time than the cache, and the remote memory has a longer access time than local memory, the programming abstraction of shared memory hides these distinctions. Consequently, shared-memory programs written without regard to the increasing cost of access to the various levels often suffer poor performance. Fortunately, a compiler can relieve the user from the burden of managing the memory hierarchy for many classes of programs through loop partitioning and data partitioning so that the probability of access from the closest level in the memory hierarchy is maximized.

The goal of *loop partitioning* for applications with nested loops that access data arrays is to divide the iteration space among the processors to get maximum reuse of data in the cache, subject to the constraint of having a good load balance. For architectures where remote memory references are more expensive than local memory references (NUMA machines), the goal of *data partitioning* is to distribute data to nodes such that most cache misses are satisfied out of the local memory.

This paper presents an algorithm to find loop and data partitions automatically for programs with multiple loop nests and data arrays. The algorithm does not resort to square blocking when communication-free partitions do not exist. Rather, it

*Authors' e-mail: {barua,kranz,agarwal}@lcs.mit.edu. Authors' phone: (617)253-8438.

discovers a partitioning of loops and data that minimizes communication cost over the multiple levels of memory hierarchy for the entire program.

Simultaneous partitioning of loops and data is difficult when multiple loops access the same data array with different access patterns. If a loop is partitioned in order to get good data reuse in the cache, that partition determines which processor will access each datum. In order to get good data locality, the data should be distributed to processors based on that loop partition. Likewise, given a partitioning of data, a loop should be partitioned based on the placement of data used in the loop. This introduces a conflict when there are multiple loops because a loop partition may have two competing constraints: good cache reuse may rely on one loop partition being chosen, while good data locality may rely on another.

Making cost tradeoffs in discovering *communication-minimal* partitions in the presence of multiple memory hierarchy levels requires a communication cost model. For a given loop and data partitioning, the communication cost model presented estimates the cost of executing the loop partition given the data partitions of arrays accessed in the loop and the architectural parameters of the machine, such as the cost of local and remote cache misses. Although finding communication-free partitions does not require a communication cost model, real programs often do not admit communication-free partitions.

The cost model is used to drive an iterative search procedure for finding a communication-minimal partitioning for the entire program. Although other search techniques can be used as well, the following solution is implemented in our compiler and described in this paper. We have found that this search procedure does not add much to the compilation time and it yields good results. The iterative solution has two steps: (1) an initial seed partitioning, and (2) an iterative search through the space of loop and data partitions commencing from the initial partitioning.

The iterative solution is seeded with an initial partitioning of each individual loop nest that disregards data locality. This initial loop partitioning is found using the method described in [2]. The iterative solution is also seeded with an initial data partition. This initial partitioning of each array is chosen to match the partitioning of the largest loop that accesses that array. Thus, by first partitioning each loop for cache locality, the initial seeding favors cache locality over data locality.

After the initial seeding, the iterative solution proceeds to use the cost model to repartition the loops according to the data partitions to increase the locality of access to local memory at the expense of cache locality if it results in better performance. It then repartitions the data arrays according to the resulting loop partitions to increase the cache locality. In this manner the loops and data are alternately re-partitioned, thus iteratively improving the solution. The cost model controls the heuristic search.

The algorithm has been implemented in a compiler for cache-coherent multiprocessors with physically distributed memory. The algorithm, however, is general and does not require that the target architecture have coherent caches. Parallelism in the source program is assumed to be specified using parallel **do** loops, either by a programmer or by a previous parallelization phase. The algorithm reported in this paper has been implemented as part of the compiler for the Alewife machine. Results from a working 16-processor machine for several real applications indicate that combined iterative optimization of loops and data can result in a decrease in runtime by nearly a factor of two over optimization of loops alone, and that a partitioning of loops for cache locality followed by partitioning data arrays according to the largest loop that accesses them, can result in improvements in runtime by a factor of about 1.4 over optimization of loops alone.

The rest of the paper describes the algorithms and the implementation, and presents several performance results. Section 2 describes related work. Section 3 overviews the notation and framework for partitioning loops and data. In particular, it shows how to derive loop partitions that minimize cache misses, and data partitions that match a given loop partition, which has the effect of minimizing remote memory accesses for a given loop partition. Note that while the above process minimizes the number of cache misses, it does not minimize overall runtime because the number of remote memory accesses is not necessarily minimized. Section 4 describes a cost model that estimates the communication cost for a given loop partition and a given data partition. This cost model is used to drive a search for a communication-minimal partitioning of loops and data arrays. Section 5 describes the iterative search method, and Section 6 presents performance results on Alewife.

2 Related Work

The problem of loop and data partitioning for distributed memory multiprocessors with global address spaces has been studied by many researchers. One approach to the problem is to have programmers specify data partitions explicitly in the program, as in Fortran-D [7, 12]. Loop partitions are usually determined by the owner computes rule. Though simple to implement, this requires the user to thoroughly understand the access patterns of the program, a task which is not trivial even for small programs. For real medium-sized or large programs, the task is a very difficult one. Presence of fully general affine function accesses further complicates the process. Worse, the program would not be portable across machines with different architectural parameters.

Ramanujam and Sadayappan [10] consider data partitioning in multicomputers and use a matrix formulation; their results do not apply to multiprocessors with caches. Their theory produces communication-free hyperplane partitions for loops with affine index expressions when such partitions exist. However, when communication-free partitions do not exist, they deal only with index expressions of the form variable plus a constant.

Ju and Dietz [8] consider the problem of reducing cache-coherence traffic in bus-based multiprocessors. Their work involves finding a data *layout* (row or column major) for arrays in a uniform memory access (UMA) environment. We consider finding data partitions for a distributed shared memory NUMA machine.

Abraham and Hudak [1] look at the problem of automatic loop partitioning for cache locality only for the case when array accesses have simple index expressions. Their method uses only a local per-loop analysis.

A more general framework for loop partitioning was presented by Agarwal et. al. [2] for optimizing for cache locality. That framework handled fully general affine access functions, i.e. accesses of the form A[2i+j,j] and A[100-i,j] were handled. However, that work found local minima for each loop independently, giving possibly conflicting data partitioning requests across loops in NUMA machines.

Another view of loop partitioning involving program transformations is presented by Carr et. al. [5]. This paper was focused on uniprocessors but their method could be integrated with data partitioning for multiprocessors as well.

The work of Anderson and Lam [4] does a global analysis across loops, but has the following differences with our method: (1) It does not take into account the combined effect on performance of globally coherent caches and local memories. (2) It attempts to find a communication free partition by satisfying a system of constraints, failing which it resorts to rectangular blocking and does not attempt to evaluate competing partitioning alternatives. Because our method uses a cost model, it can evaluate different competing alternatives, each having some amount of communication, and choose between them. (3) We guarantee a load balanced solution.

Gupta and Banerjee [6] have developed an algorithm for partitioning doing a global analysis across loops. They allow simple index expression accesses of the form $c_1 * i + c_2$, but not general affine functions. They do not allow for the possibility of hyperparallelepiped data tiles, and do not account for caches.

Knobe, Lucas and Steele [9] give a method of allocating arrays on SIMD machines. They align arrays to minimize communication for vector instructions, which access array regions specified by subranges on each dimension.

Wolf and Lam [13] deal with the problem of taking sequential nested loops and applying transformations to attempt to convert them to a nest of parallel loops with at most one outer sequential loop. This technique can be used before partitioning when the programming model is sequential to convert to parallel loops, and hence complements our work.

3 Overview of the Partitioning Framework

This section overviews the notation and framework used for partitioning. It describes how the number of cache misses can be estimated for a given loop partition. It also gives a brief summary of the method for loop partitioning to increase cache reuse previously published [2]. This method will be used to find an initial loop partition and an initial data partition to seed the search driven by a cost model for communication-minimal loop and data partitioning.

The framework handles programs with loop nests where the array index expressions are affine functions of the loop variables. In other words, the index function $\vec{g}$ can be expressed as,

$$\vec{g}(\vec{i}) \quad = \quad \vec{i}\mathbf{G} + \vec{\mathbf{a}} \tag{1}$$

where $\mathbf{G}$ is a $l \times d$ matrix with integer entries, $\vec{i}$ is the vector of loop variables and $\vec{\mathbf{a}}$ is an integer constant vector of length d, termed the *offset vector*. Thus accesses of the form A[2i+j,100-i] and A[j] are handled, but not A[i^2], where i,j are nested loop induction variables. Consider the following example of a loop:

```
Doall (i=0:99, j=0:99)
  A[i,j] = B[i+j,j]+B[i+j+1,j+2]
EndDoall
```

A loop partition $\mathbf{L}$ is defined by a hyperparallelepiped at the origin as pictured in Figure 1. Each hyperparallelepiped represents the region executed by a different processor, and the whole loop space is tiled in this manner. The number of iterations contained in matrix $\mathbf{L}$ is $|\det \mathbf{L}|$.

The *footprint* of an iteration tile $\mathbf{L}$ with respect to an array reference is the set of points in the data space accessed by the tile through that reference. This footprint is given by $\mathbf{LG}$, translated by $\vec{\mathbf{a}}$. A set of references to one array in one loop with the same $\mathbf{G}$ but different offsets $\vec{\mathbf{a}}$ are called *uniformly intersecting* references. The footprints associated with such

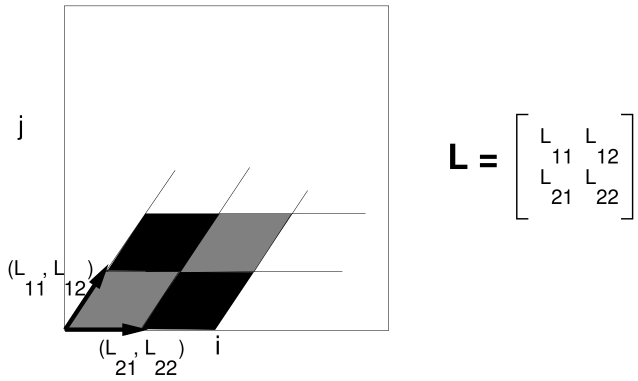


Figure 1: Iteration space partitioning is completely specified by the tile at the origin.

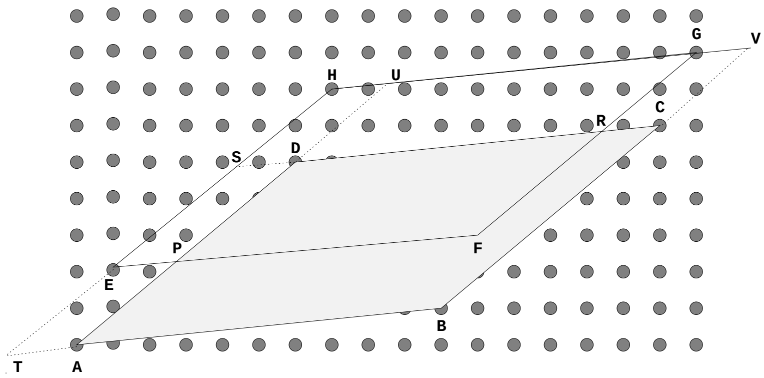


Figure 2: Data footprint wrt $B[i+j,j]$ and $B[i+j+1,j+2]$

sets of references are the same shape, but are translated in the data space. They are said to be in the same *UI-set* (Uniformly Intersecting set).

This is illustrated by the above code fragment. The code has only one UI-set for array B, as the two accesses to B differ only by a constant vector. The **G** matrix for the UI-set is given by

$$\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}.$$

For some loop tile **L** at the origin, the footprint in the data space is the union of **LG** translated by each offset in the UI-set. Since this loop has two references to **B** in the same UI-set with offsets $\vec{\mathbf{a}}$ of (0,0) and (1,2), the footprint looks like that shown in Figure 2.

The total number of cache misses for a given loop nest is the number of its first time data accesses. This number is simply the size of the combined footprint with respect to all the accesses in the loop. [2] shows how the combined footprint in a UI-set can be computed. It also shows how the loop partitioning **L** can be chosen to minimize the number of cache misses. Cache misses are minimized when the combined footprint with respect to all the accesses in the loop has minimum area, as the area is the number of first time data accesses.

For cache-coherent machines with uniform-access memory (UMA), because all cache misses suffer the same cost, loop partitioning alone is sufficient. Loop partitioning performed with complete disregard to data partitioning is termed "random" in our performance results in Section 6. Random refers to random data placement.

The cost model also refers to the data partition **D**, relevant for NUMA machines. A matrix **D** represents a tile at the origin of the data space in the same way as **L** represents a tile at the origin of the iteration space. An array reference in a loop will have good data locality when partitions are chosen such that $\mathbf{LG} = \mathbf{D}$, and the translation offsets of the two differ by at most small constants.

In the above code, both references to B will have simultaneously good probability of being satisfied in local memory when **D** is picked to be **LG**. The only communication for B then, (that is, memory accesses to remote memory), is at the periphery of **LG**, due to small offsets. This periphery is what was minimized in [2]. Indeed, **D** is chosen as shown above to seed the search process.

Data partitioning performed according to the loop partitioning is termed "local" in our performance results. With data partitioning, the probability that a cache miss will be satisfied in the local memory is increased. However, although the number of cache misses satisfied in local memory is increased it is not necessarily maximized.

The following sections discuss how a cost model can be used to make a tradeoff between the number of cache misses and the number of remote memory references, and to discover a communication-minimal partitioning of loops and data (termed "global" in our results) that yields the lowest runtime.

4 The Cost Model

The key to a finding a communication-minimal partitioning is a cost model that allows a tradeoff to be made between cache miss cost and remote memory access cost. This cost model drives an iterative solution and is a function that takes, as arguments, a loop partition, data partitions for each array accessed in the loop, and architectural parameters that determine the relative cost of cache misses and remote memory accesses. It returns an estimation of the cost of array references for the loop.

The cost due to memory references in terms of the architectural parameters is computed by following equation:

$$T_{total\_access} = T_R(n_{remote}) + T_L(n_{local}) + T_C(n_{cache})$$

where T_R, T_L, T_C are the remote, local and cache memory access times respectively, and $n_{remote}, n_{local}, n_{cache}$ are the number of references that result in hits to remote memory, local memory and cache memory. T_C and T_L are fixed by the architecture, while T_R is determined both by the base remote latency of the architecture and possible contention if there are many remote references. T_R may also vary with the number of processors based on the interconnect topology.

n_{cache}, n_{local} and n_{remote} depend on the loop and data partitions. Given a loop partition, for each UI-set consider the intersection between the footprint (**LG**) of that set and a given data partition **D**. First time accesses to data in that intersection will be in local memory while first time references to data outside will be remote. Repeat accesses will likely hit in the cache. A UI-set may contain several references, each with slightly different footprints due to different offsets in the array index expressions. One is selected and called the base offset, or $\vec{\mathbf{b}}$. In the following definitions the symbol $\approx$ will be used to compare footprints and data partitions. $\mathbf{LG} \approx \mathbf{D}$ means that the matrix equality holds. This equality does not mean that all references in the UI-set represented by **G** will be local in the data partition **D** because there may be small offset vectors for each reference in the UI-set.

We define the functions R_b, F_f and F_b, which are all functions of the loop partition **L**, data partition **D** and reference matrix **G** with the meanings given in Section 3. For simplicity, we also use R_b, F_f and F_b, to denote the value returned by the respective functions of the same name.

Definition 1 *R_b is a function which maps* **L**, **D** *and* **G** *to the number of remote references that result from a single access defined by* **G** *and the base offset* $\vec{\mathbf{b}}$.

In other words, R_b returns the number of remote accesses that result from a single program reference in a parallel loop, not including the small peripheral footprint due to multiple accesses in its UI-set. The periphery is added using F_f to be described below.

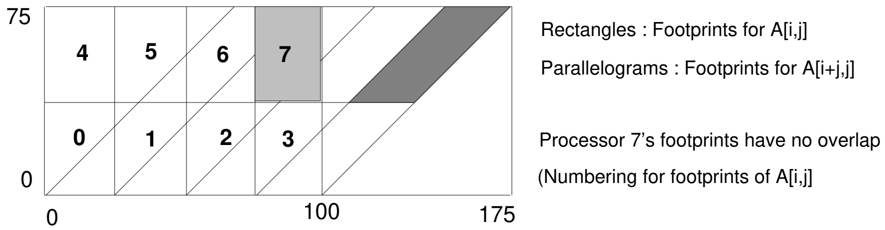
Note that in most cases **G**'s define UI-sets: accesses to an array with the same **G** but different offsets are usually in the same UI-set, and different **G**'s always have different UI-sets. The only exception are accesses with the same **G** but large differences in their offsets relative to tile size, in which case they are considered to be in different UI-sets.

The computation of R_b is simplified by an approximation. One of the two following cases apply to loop and data partitions.

1. Loop partition **L** matches the data partition **D**, *i.e.* $\mathbf{LG} \approx \mathbf{D}$. The references in the periphery due to small offsets between references in the UI-set are considered in F_f. In this case $R_b = 0$.
2. **L** does not match **D**. This is case where the **G** matrix used to compute **D** (perhaps from another UI-set), is different from the **G** for the current access, and thus **LG** and **D** have different *shapes*, not just different offsets. In this case all references for **L** are considered remote and R_b = $|\mathrm{Det\ L}|$.

```
Doall (i=0:100, j=0:75)
   B[i,j] = A[i,j] + A[i+j,j]
EndDoall
```

(a) Code fragment



(b) Data space for Array A(8 processors)

Figure 3: Different UI-sets have no overlap

This is a good approximation because **LG** and **D** each represent a regular tiling of the data space. If they differ, it means the footprint and data tile differ in shape, *and do not stride the same way*. Thus, even if **L**'s footprints and **D** partially overlap at the origin, there will be less overlap on other processors. For a reasonably large number of processors, some will end up with no overlap as shown in the example in Figure 3. Since the execution time for a parallel loop nest is limited by the processor with the most remote cache misses, the non-overlap approximation is a good one.

Definition 2 *F_b is the number of first time accesses in the footprint of* **L** *with base offset* $\vec{\mathbf{b}}$*. Hence:*
$F_b = |Det\ L|$

Definition 3 *F_f is the difference between (1) the cumulative footprints of all the references in a given UI-set for a loop tile, and (2) the base footprint due to a single reference represented by* **G** *and the base offset* $\vec{\mathbf{b}}$*. F_f is referred to as the peripheral footprint.*

See [2] for details on how the peripheral footprint is computed.

Theorem 1 *The cumulative access time for all accesses in a loop with partition* **L***, accessing an array having data partition* **D** *with reference matrix* **G** *in a UI-set is*

$$T_{UI-set} = T_R(R_b + F_f) + T_L(F_b - R_b) + T_C(nref - (F_f + F_b))$$

where $nref$ is the total number of references made by **L** *for the UI-set.*

This result can be derived as follows. The number of remote accesses n_{remote} is the number of remote accesses with the base offset, which is R_b, plus the size of the peripheral footprint F_f, giving $n_{remote} = R_b + F_f$. The number of local references n_{local} is the base footprint, less the remote portion, i.e. $F_b - R_b$. Finally, number of cache hits n_{cache} is clearly $nref - n_{remote} - n_{local}$ which is equal to $nref - (F_f + F_b)$.

Sub-blocking The above cost model assumes infinite caches. In practice, even programs with moderate-sized data sets have footprints much larger than the cache size. To overcome this problem the loop tiles are *sub-blocked*, such that each sub-block fits in the cache and has a shape that optimizes for cache locality. This optimization lets the cost model remain valid even for finite caches. It turned out that sub-blocking was critically important even for small to moderate problem sizes.

Finite caches and sub-blocking also allows us to ignore the effect of data that is shared between loop nests when that data is left behind in the cache by one loop nest and reused by another. Data sharing can happen in infinite caches due to accesses to the same array when the two loops use the same **G**. However, when caches are much smaller than data footprints, and the compiler resorts to sub-blocking, the possibility of reuse across loops is virtually eliminated.

This model also assumes a linear flow of control through the loop nests of the program. While this is the common case, conditional control flow can be handled by our algorithm. Although we do not handle this case now, an approach would be to assign probabilities to each loop nest, perhaps based on profile data, and to multiply the probabilities by the loop size to obtain an effective loop size for use by the algorithm.

5 The Multiple Loops Heuristic Method

This section describes the iterative method, whose goal is to discover a partitioning of loops and data arrays to minimize communication cost. We assume loop partitions are non-cyclic. Cyclic partitions could be handled using this method but for simplicity we leave them out.

5.1 Graph formulation

Our search procedure uses bipartite graphs to represent loops and data arrays. Bipartite graphs are a popular data structure used to represent partitioning problems for loops and data[8, 4]. For a graph G = (V_l, V_d, E), the loops are pictured as a set of nodes V_l on the left hand side, and the data arrays as a set of nodes V_d on the right. An edge e∈E between a loop and array node is present if and only if the loop accesses the array. The edges are labeled by the uniformly intersecting set(s) they represent. When we say that a data partition is *induced* by a loop partition, we mean the data partition **D** is the same as the loop partition **L**'s footprint. Similarly, for loop partitions induced by data partitions.

5.2 Iterative Method Outline

We use an iterative local search technique that exploits certain special properties of loops and data array partitions to move to a good solution. Extensive work evaluating search techniques has been done by researchers in many disciplines. Simulated annealing, gradient descent and genetic algorithms are some of these. See [11] for a comparison of some methods. All techniques rely on a cost function estimating some objective value to be optimized, and a search strategy. For specific problems more may be known than in the general case, and specific strategies may do better. In our case, we know the search direction that leads to improvement, and hence a specific strategy is defined. The algorithm greedily moves to a local minimum, does a mutation to escape from it, and repeats the process.

The following is the method in more detail. To derive the initial loop partition, the single loop optimization method described in Section 3 is used. Then an iterative improvement method is followed, which has two phases in each iteration: the first (forward) phase finds the best data partitions given loop partitions, and the second (back) phase redetermines the values of the loop partitions given the data partitions just determined.

We define a boolean value called the progress flag for each array. Specifically, in the forward phase the data partition of each array having a true progress flag is set to the induced data partition of the largest loop accessing it, among those which change the data partition. The method of controlling the progress flag is explained in section 5.2.2. In the back phase, each loop partition is set to be the data partition of one of the arrays accessed by the loop. The cost model is used to evaluate the alternative partitions and pick the one with minimal cost.

These forward and backward phases are repeated using the cost model to determine the estimated array reference cost for the current partitions. After some number of iterations, the best partition found so far is picked as the final partition. Termination is discussed in Section 5.2.2.

5.2.1 An example

The workings of the heuristic can be seen by a simple example. Consider the following code fragment:

```
Doall (i=0:99, j=0:99)
  A[i,j] = i * j
EndDoall
Doall (i=0:99, j=0:99)
```

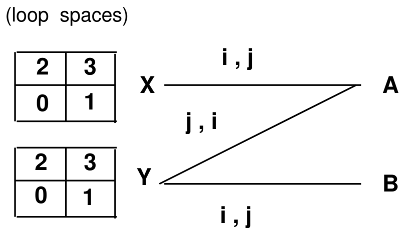


Figure 4: Initial solution to loop partitioning (4 processors)

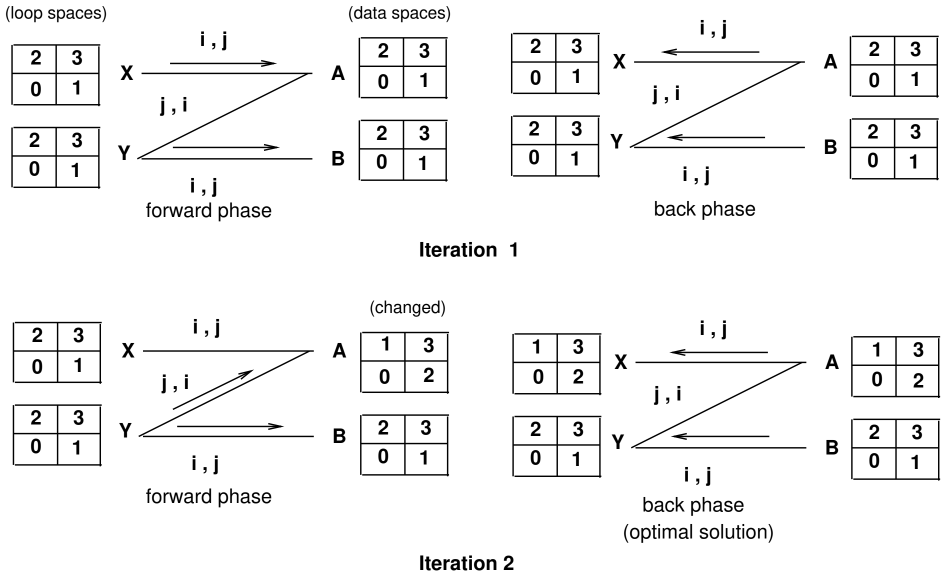


Figure 5: Heuristic: iterations 1 and 2 (4 processors)

```
    B[i,j] = A[j,i]
```
EndDoall

The code does a transpose of A into B. The first loop is represented by X and the second by Y. The initial cache optimized solution for 4 processors is shown in Figure 4. In this example, as there is no peripheral footprint for either array, a default load balanced solution is picked. Iterations 1 and 2 with their forward and back phases are shown in Figure 5.

In iteration 1's forward phase A and B get data partitions from their largest accessing loops. Since both loops here are equal in size, the compiler picks either, and one possible choice is shown by the arrows. In 1's back phase, loop Y cannot match both A and B's data partitions, and the cost estimator indicates that matching either has the same cost. So an arbitrary choice as shown by the back arrows results in unchanged data partitions; nothing has changed from the beginning.

As explained in the next section, the choice of data partitions in the forward phase is *favored in the direction of change*. So now array A picks a different data partition from before, that of Y instead of X. In the back phase loop X now changes its loop partition to reduce cost as dictated by the cost function. This is the best solution found, and no further change occurs in subsequent iterations. In this case, this best solution is also the optimal solution as it has 100% locality. In this example a communication-free solution exists and was found. More generally, if one does not exist, the heuristic will evaluate many solutions and will pick the best one it finds.

5.2.2 Some Implementation Details

The algorithm of the heuristic method is presented in Figure 6. Some details of the algorithm are explained here.

Choosing a data partition different from the current one is preferred because it ensures movement out of local minima. A mutation out of a local minimum is a change to a possibly higher cost point, that sets the algorithm on another path. This rule ensures that the next configuration differs from the previous, and hence makes it unnecessary to do global checks for local minima.

However, without a progress flag, always changing data partitions in the forward phase may change data partitions too fast. This is because one part of the graph may change before a change it induced in a previous iteration has propagated to the whole graph. This is prevented using a *bias rule*. A data partition is not changed in the forward phase if it induced a loop partition change in the immediately preceding back phase. This is done by setting its progress flag to false.

As with all deterministic local search techniques, this algorithm could suffer from oscillations. The progress flag helps solve this problem. Oscillations happen when a configuration is revisited. The solution is to, conservatively, determine if a cost has been seen before, and if it has, simply enforce change at all data partition selections in the next forward phase. This sets the heuristic on another path. There is no need to store and compare entire configurations to detect oscillations.

One issue is the number of iterations to perform. In this problem, the length of the longest path in the bipartite graph is a reasonable bound, since changed partitions in one part of the graph need to propagate to other parts of the graph. This bound seems to work well in practice. Further increases in this bound did not provide a better solution in any of the examples or programs tried.

In all of the small programs we tried, the heuristic found the known optimal solution. For the large applications in section 6 ,we do not know the optimal, but the heuristic found improved solutions with very high locality. Careful manual examination did not result in finding better partitions.

5.3 Algorithm Complexity

Here we show that the above algorithm runs in polynomial time in n and m, the number of loops and distributed arrays in the program. An exhaustive search guaranteed to find the optimal for this NP-complete problem is not practical.

Theorem 2 *The time complexity of the above heuristic is* $O(n^2m + m^2n)$.

To prove this, note that the number of iterations is the length of the longest acyclic path in the bipartite graph, which is upper bounded by $n + m$. The time for one iteration is the sum of the times of the forward and back phases. The forward phase does a selection among m possible loop partitions for each of n loops, giving a bound of $O(nm)$. The back phase does a selection among n possible data partitions for each of m arrays, giving a bound of $O(nm)$. Thus overall the time is $O((n + m)mn) = O(n^2m + m^2n)$.

6 Results

The algorithm described in this paper has been implemented as part of the compiler for the Alewife [3] machine. The Alewife machine is a cache-coherent multiprocessor with physically distributed memory. The nodes are configured in a 2-dimensional mesh network. The approximate average Alewife latencies for a 16 node machine are: 2 cycle cache hit, 11 cycle cache miss to local memory hit and 40 cycle remote cache miss. The last number will be larger for larger machine configurations or when network contention is present.

We compared performance on the following applications:

Tomcatv A code from the SPEC suite. It has 12 loops and 7 arrays, all two dimensional.

Erlebacher A code written by Thomas Eidson, from ICASE. It performs 3-D tridiagonal solves using Alternating Direction Implicit (ADI) integration. It has 40 loops and 22 distributed arrays, in one, two and three dimensions.

Conduct A routine in SIMPLE, a two dimensional hydrodynamics code from Lawrence Livermore National Labs. It has 20 loops and 20 arrays, in one and two dimensions.

These programs were run using each of three compilation strategies:

global Uses the algorithm described in this paper.

```
Procedure Do_forward_phase()
    for all d ∈ Data_set do
        if Progress_flag[d] then
            l ← largest loop accessing d which induces changed Data_partition[d]
            Data_partition[d] ← Partition induced by Loop_partition[l]
            Origin[d] ← Access function mapping of Origin[l]
        endif
        Inducing_loop[d] ← l
    endfor
end Procedure

Procedure Do_back_phase()
    for all l ∈ Loop_set do
        d ← Array inducing Loop_partition[l] with minimum cost of accessing all its data
        Loop_partition[l] ← Partition induced by Data_partition[d]
        Origin[l] ← Inverse access function mapping of Origin[d]
        if Inducing_loop[d] ≠ l then
            Progress_flag[d] ← false
        endif
    endfor
end Procedure

Procedure Partition
    Loop_set : set of all loops in the program
    Data_set : set of all data arrays in the program
    Graph_G : Bipartite graph of accesses in Loop_set to Data_set

    Min_partitions ← φ
    Min_cost ← ∞
    for all d ∈ Data_set do
        Progress_flag[d] ← true
    endfor
    for i= 1 to (length of longest path in Graph_G) do
        Do_forward_phase()
        Do_back_phase()
        Cost ← Find total cost of current partition configuration
        if Cost < Min_cost then
            Cost ← Min_cost
            Min_partitions ← Current partition configuration
        endif
        if cost repeated then               /* convergence or oscillation */
            for all d ∈ Data_set do         /* force progress */
                Progress_flag[d] ← true
            endfor
        endif
    endfor
end Procedure
```

Figure 6: The heuristic algorithm

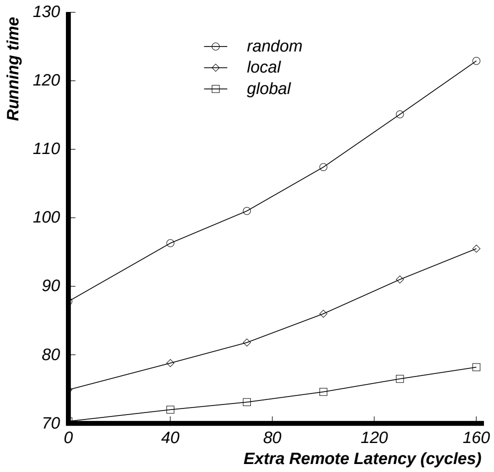


Figure 7: Tomcatv (N = 800)

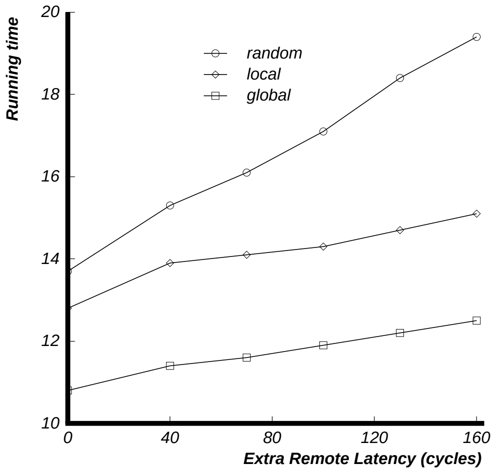


Figure 8: Erlebacher (N = 48)

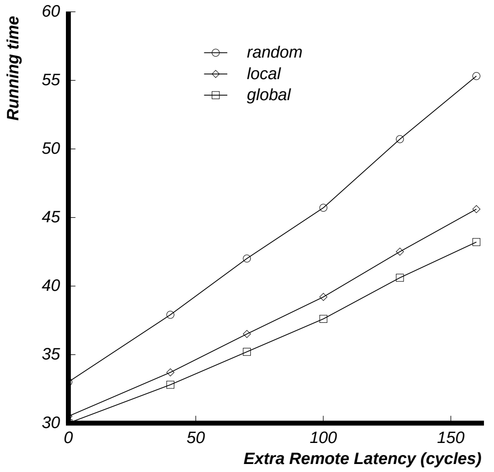


Figure 9: Conduct (480 x 384)

Program	Problem Size	Speedup
Tomcatv	N = 192	15
Erlebacher	N = 48	10
Conduct	153 x 133	11

Figure 10: Speedup (**global**) on 16 processors

local Uses the analysis in [2] to determine the loop partition, and then partitions each array by using the partition induced by the largest loop that accesses that array, to achieve some data locality. This analysis proceeds as in **global** but does only the first iterations' forward phase, and halts.

random Allocates the data across processors randomly.

Figures 7, 8 and 9 show the execution times for each application and partitioning strategy for a variety of remote latencies. The smallest latency uses the default Alewife configuration. The larger latencies were obtained by imposing a hardware-supported delay on remote cache misses. This was done by causing the processor to trap on a remote cache miss. This trap occurs at the same time that the request for data is sent to the remote node. A delay loop was then executed for the number of cycles shown in the graphs.

Due to the relatively short remote access latency in Alewife, the numbers for **local** are close to **global** for the default latency. The difference becomes much more significant for longer latencies that can be found in some other architectures. In Alewife, programs with higher cache-coherency overheads will have longer latencies.

Figure 10 gives the baseline speedup numbers. They represent the default remote latency with global optimization on 16 processors. Because the original problem sizes for some of them were too small to run on a small number of processors, these numbers are for smaller problem sizes as indicated in the table.

7 Conclusions and Summary

We have presented an algorithm to find loop and data partitions automatically for programs with multiple loop nests and data arrays. The algorithm discovers a partitioning of loops and data that minimizes communication cost over the multiple levels of memory hierarchy for the entire program. It does this by balancing the cost of cache misses and remote memory accesses using a cost function. If no communication-free partition is found, the cost function is used to guide a heuristic search through the global space of loop and data partitions. This method has been implemented as part of the compiler for the Alewife machine. We showed results from executing three applications on a real machine with 16 processors. These results indicate that significant performance improvements can be obtained by looking at data locality and cache locality in a global framework.

In the future we would like to add the possibility of copying data at runtime to avoid remote references as in [4]. This factor could be added to our cost model.

References

[1] S. G. Abraham and D. E. Hudak. Compile-time partitioning of iterative parallel loops to reduce cache coherency traffic. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):318–328, July 1991.

[2] Anant Agarwal, David Kranz, and Venkat Natarajan. Automatic Partitioning of Parallel Loops for Cache-Coherent Multiprocessors. In *22nd International Conference on Parallel Processing*, St. Charles, IL, August 1993. IEEE. To appear in IEEE TPDS.

[3] A. Agarwal *et al.* The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor. In *Proceedings of Workshop on Scalable Shared Memory Multiprocessors*. Kluwer Academic Publishers, 1991. An extended version of this paper has been submitted for publication, and appears as MIT/LCS Memo TM-454, 1991.

[4] Jennifer M. Anderson and Monica S. Lam. Global Optimizations for Parallelism and Locality on Scalable Parallel Machines. In *Proceedings of SIGPLAN '93 Conference on Programming Languages Design and Implementation*. ACM, June 1993.

[5] Steve Carr, Kathryn S. McKinley, and Chau-Wen Tzeng. Compiler Optimization for Improving Data Locality. In *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 252–262, October 1994.

[6] M. Gupta and P. Banerjee. Demonstration of Automatic Data Partitioning Techniques for Parallelizing Compilers on Multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):179–193, March 1992.

[7] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiling Fortran D for MIMD Distributed Memory Machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[8] Y. Ju and H. Dietz. Reduction of Cache Coherence Overhead by Compiler Data Layout and Loop Transformation. In *Languages and Compilers for Parallel Computing*, pages 344–358. Springer Verlag, 1992.

[9] Kathleen Knobe, Joan Lukas, and Guy Steele Jr. Data Optimization: Allocation of Arrays to Reduce Communication on SIMD Machines. *Journal of Parallel and Distributed Computing*, 8(2):102–118, August 1990.

[10] J. Ramanujam and P. Sadayappan. Compile-Time Techniques for Data Distribution in Distributed Memory Machines. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):472–482, October 1991.

[11] Bart Selman, Henry Kautz, and Bram Cohen. Noise Strategies for Improving Local Search. In *Proceedings, AAAI*, volume 1, 1994.

[12] C.-W. Tseng. *An Optimizing Fortran D compiler for MIMD Distributed-Memory Machines*. PhD thesis, Rice University, Jan 1993. Published as Rice COMP TR93-199.

[13] Michael E. Wolf and Monica S. Lam. A Loop Transformation Theory and an Algorithm to Maximize Parallelism. In *The Third Workshop on Programming Languages and Compilers for Parallel Computing*, August 1990. Irvine, CA.