Improving Locality For Adaptive Irregular Scientific Codes

Hwansoo Han, Chau-Wen Tseng

Department of Computer Science
University of Maryland
College Park, MD 20742

{*hshan, tseng*}*@cs.umd.edu*

Abstract

An important class of scientific codes access memory in an irregular manner. Because irregular access patterns reduce temporal and spatial locality, they tend to underutilize caches, resulting in poor performance. Researchers have shown that consecutively packing data relative to traversal order can significantly reduce cache miss rates by increasing spatial locality. In this paper, we investigate techniques for using partitioning algorithms to improve locality in adaptive irregular codes. We develop parameters to guide both geometric (RCB) and graph partitioning (METIS) algorithms, and develop a new graph partitioning algorithm based on hierarchical clustering (GPART) which achieves good locality with low overhead. We also examine the effectiveness of locality optimizations for adaptive codes, where connection patterns dynamically change at intervals during program execution. We use a simple cost model to guide locality optimizations when access patterns change. Experiments on irregular scientific codes for a variety of meshes show our partitioning algorithms are effective for static and adaptive codes on both sequential and parallel machines. Improved locality also enhances the effectiveness of LOCALWRITE, a parallelization technique for irregular reductions based on the owner computes rule.

1 Introduction

Computational science is increasingly becoming an important tool for scientists and engineers performing research and development. Fast yet inexpensive microprocessors and commercial multiprocessors provide the computing power they need for research and development. Compilers play an important role by automatically customizing programs for complex processor architectures, improving portability and providing high performance to non-expert programmers.

As scientists attempt to model more complex problems, computations with irregular memory access patterns become increasingly important. These computations arise in several application domains. In computational fluid dynamics (CFD), meshes for modeling large problems are sparse to reduce memory and computations requirements. In n-body solvers such as those arising in molecular dynamics, data structures are by nature irregular because they model the positions of particles and their interactions.

As microprocessors become increasingly fast, memory system performance begins to dictate overall performance. The ability of applications to exploit locality by keeping references to cache becomes a major (if not the key) factor affecting performance. Unfortunately, irregular computations have characteristics which make it difficult to utilize caches efficiently.

Consider the example in Figure 1. In the regular code, accesses to x are made to consecutive memory locations (since Fortran is column-major). This spatial locality allows the code to take full advantage of long cache lines to reuse each cache line multiple times before it is flushed from cache. In comparison, in the irregular code, accesses to x are irregular, dictated by the contents of the index array *idx*. It is unclear whether spatial locality exists or can be exploited by the cache.

Compounding the problem, regular codes benefit because compilers can analyze data access patterns, using estimates of cache performance to guide loop and data transformations to improve locality [37, 50, 17]. In comparison, there is relatively little information at compile time concerning the locality properties of irregular programs.

Researchers have demonstrated that the performance of irregular programs can be improved by applying a combination of computation and data layout transforma-

University of Maryland, Dept. of Computer Science Technical Report CS-TR-4039, September 1999.

This research was supported by NSF CAREER Development Award #ASC9625531 in New Technologies. The IBM SP-2 and DEC Alpha Cluster were provided by NSF CISE Institutional Infrastructure Award #CDA9401151 and grants from IBM and DEC.

```
// Regular                // Irregular
do t = 1, time            do t = 1, time
  x(N,N)                    x(N), idx(M)
  do i = 1, N               do i = 1,M
    do j = 1, N               ... = x(idx(i))
      ... = x(j,i)
```

Figure 1 Regular and Irregular Applications

tions [14, 38]. The compiler identifies irregular computations which can be reordered, then inserts calls to run-time routines which reorder computations based on lexicographically sorting the edges in the mesh, then consecutively *packing* data in memory according to the traversal order. Space-filling curves can be used to reorganize data when geometric coordinate information is available. The overall premise of preprocessing data to improve performance is based on the inspector/executor paradigm, first used to parallelize irregular computations for message-passing machines [13].

In this paper, we improve on existing methods in several ways. Run-time partitioning algorithms (e.g., RCB, METIS) can obtain better locality by exploiting geometric information or graph structure. We show how such algorithms may be tuned to improve performance while balancing overhead. We present a new graph partitioning algorithm (GPART) based on hierarchical clustering, and show how to tune it to improve locality with low overhead. We also examine the impact of adaptivity on locality optimizations, and derive heuristics for deciding when locality optimizations should be performed. Experiments demonstrate locality and performance are improved for several irregular scientific codes for a variety of application meshes. Our paper makes the following contributions:

- Develop a new graph partitioning technique based on graph clustering that balances overhead with precision.
- Selecting effective parameters for partitioning techniques.
- Devise cost models for guiding locality optimizations for adaptive irregular codes.
- Experimental evaluation of locality optimizations for adaptive irregular and parallel irregular codes.

The remainder of the paper begins with a discussion of algorithms for improving data layouts for irregular codes. We experimentally select parameters for partitioning techniques and evaluate the qualities of locality optimizations using different algorithms. We examine the effect of adaptivity on locality optimizations, present a simple cost model to guide optimizations, then evaluate our model experimentally. We also investigate interaction with previous algorithms for parallelizing irregular reductions. Finally, we conclude with a discussion of related work.

2 Background

Previous research has shown that computation and data transformation can significantly improve the locality of irregular scientific codes [14, 38]. Two techniques are computation reordering and consecutive packing.

2.1 Computation reordering

Computation reordering works as follows. If each loop iteration accessed one data item, sorting the loop iterations by the addresses of the data items would yield optimal temporal and spatial locality. In many irregular scientific applications, each loop iteration tends to compute results for a single edge in meshes or interaction between two bodies, resulting in accesses to a pair of data items. The pair of accesses may be viewed as a tuple (x, y), where x and y are the addresses of the pair of data items accessed. Rearranging the loop iterations by applying radix (lexicographic) sort to the tuples then yields a quality solution which improves both temporal and spatial locality. Sorting is so effective that data meshes provided by applications writers are frequently presorted, eliminating the need to apply sorting at run time. In our research, all input meshes are presorted, so the base performance represents the performance of computation reordering.

Data reordering algorithms we will describe in the following sections alter the original order of nodes, making presorted input meshes no longer lexicographically sorted according to the new order of nodes. Thus, the computation reordering is required whenever data ordering algorithms are applied. Since computation reordering is orthogonal to the data reordering, computation reordering can be applied in combination with any data reordering algorithm. In our research, we always apply computation reordering after data reordering.

2.2 Consecutive packing (CPACK)

In addition to changing the order of accesses, the compiler can also reorganize data layout. Ding and Kennedy proposed *consecutive packing* (CPACK), where data is moved into adjacent locations in the order they are first accessed (first-touch) by the computation [14]. The motivation is that the original data access order is likely to be based on the logical affinity among data items. By rearranging the data according to the order in the temporal access sequence, spatial locality is likely improved over the original storage order. Experiments seem to indicate CPACK can improve spatial locality of data references and can reduce conflict and capacity misses, particularly if input meshes are presorted or computation reordering is applied before packing. CPACK is very efficient in terms of overhead, but it does not explicitly take into account structures in program input data. As a result, our experiments show it produces lower quality orderings than other heuristics.

Besides CPACK, Ding and Kennedy also proposed two other heuristics, *group packing* and *consecutive group packing* to capture the structure of input data [14]. *Group packing* classifies data according to their average reappearance distances and *consecutive group packing* applies *group packing* within a limited range of access sequence. Miss rates of applications using these two other heuristics are slightly lower than using CPACK only when many consistent reuse patterns exist. These two other heuristics worked slightly better for one of their application, MESH, but worked worse for another application, MOLDYN. Furthermore, these two other heuristics showed worse performance even for the MESH, when computation reordering was used together [14]. As a result, we believe *group packing* and *consecutive group packing* are not versatile heuristics for capturing the existing structures of input data. We think CPACK is a better heuristic than their other heuristics in general.

3 Partitioning Algorithms

A different class of data transformation algorithms we use in our research attempt to improve the locality of irregular computations by *partitioning* the data. Partitioning algorithms exploit the fact most interactions (and mesh connections) in scientific computations are local, between nearby elements. For instance, in a molecular dynamics the computation of the interactions may be calculated only between molecules within a given cutoff radius (e.g., 3 angstroms). By creating partitions containing most of the neighboring data elements, interactions tend to remain within the partition. Storing all elements in a partition together in memory thus increases the probability they can remain in cache, yielding better reuse.

Partitioning algorithms were originally developed for applications in such areas as load-balancing parallel computations [27], VLSI design [2], and database storage [44]. We adapt them for improving cache performance for irregular codes, where overhead of partitioning is more important. In this section, we examine three partitioning algorithms: recursive coordinate bisection, multi-level graph partitioning, and hierarchical graph clustering.

3.1 Recursive coordinate bisection (RCB)

In scientific applications, data items are usually logically located in two or three dimensions. The 1D storage order is simply an artifact of storing data items in memory. One method for partitioning data is to take into account the actual geometric proximity between nodes. Because interactions tend to be local, putting neighboring nodes into a partition can improve locality.

Recursive coordinate bisection (RCB) is based on geometric coordinate information. RCB works by recursively selecting the longest dimension, then splitting the dimension into two partitions by finding the median of data coordinates in that dimension. Because the median is used, split partitions are guaranteed to have roughly equal number of data items. The process is recursively repeated as desired [3, 4]. Once all partitions are selected, data items are stored consecutively within each partition, and partitions within an upper level partition are also arranged consecutively, constructing a hierarchical structure. Thus, RCB produces not only a partition but also an order among partitions that is similar to Z-ORDERING, a space-filling curve for dense array layout. Computation can then be reordered by the resulting partition.

Space-filling curves also use geometric coordinate information to attempt to ensure proximity in memory. Data are laid out in memory according to space-filling curves (e.g., Hilbert, Morton), which are continuous, non-smooth curves that pass through every point in a finite k-dimensional space [24, 38, 46]. Figure 2 shows an example of using the Hilbert space-filling curve. Each point in the k-dimensional space is mapped to its location on the 1D space-filling curve using a sequence of bit-level operations on its k-dimensional coordinates. These mappings attempt to minimize the distance (in memory) between two geometrically close points in space.

Space-filling curves requires information on the coordinates of each point, but can yield major performance improvements over randomly placed data [38].

Figure 2 compares RCB and a space-filling curve (HILBERT) for an example where nodes are unevenly distributed. Space-filling-curves work best when nodes are evenly distributed in a fixed finite space, because they assume fixed size grid to partition nodes. However, in scientific computations such as astrophysics, celestial bodies may be distributed unevenly. Thus, space-filling curves may fail to partition the bodies due to the coarse resolution of grid, putting large set of bodies in the same grid partition and giving no order among those large set of bodies in the same grid.

Selecting an appropriate resolution of grid to handle unevenly distributed nodes is not trivial and using excessively fine resolution increases the run-time overhead. Space-filling curves also do not guarantee load balance

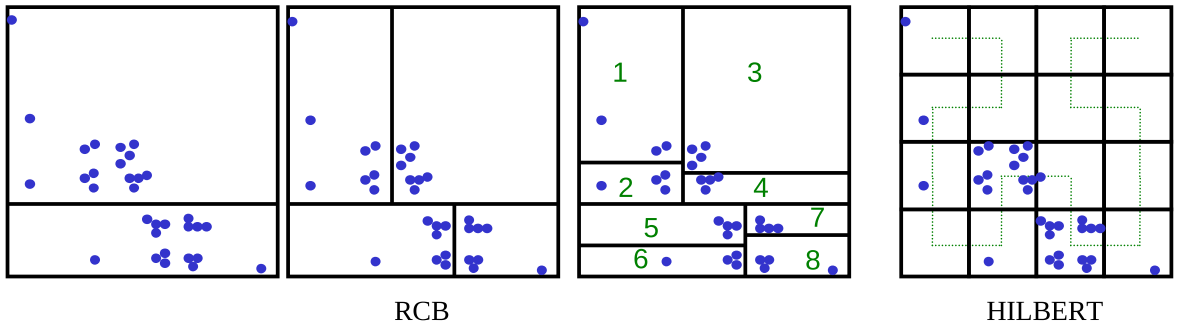


Figure 2 Recursive Coordinate Bisection (RCB) vs. A HILBERT Space-Filling Curve

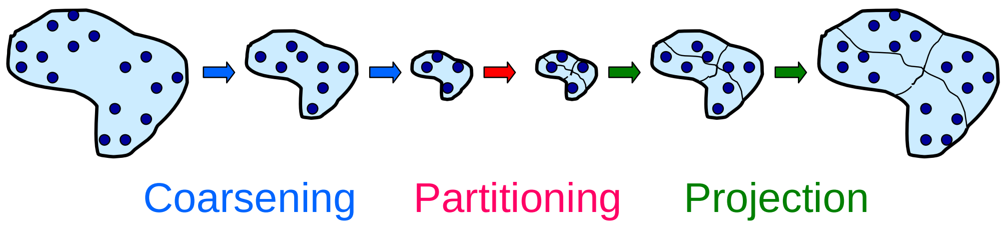


Figure 3 Multi-level Graph Partitioning (METIS)

among partitions. However, most scientific applications are executed in parallel where load balance is critical to their performance.

In recursive bisection, each step incurs run-time overhead, so RCB should be halted as quickly as possible. However, partitions must be small enough so that they can remain in cache, or else locality benefits will be lost. Compared to CPACK and space-filling curves, overhead is high since more passes of sorting are required.

3.2 Multi-level graph partitioning (METIS)

A major limitation of RCB and space-filling curves is that geometric coordinate information is needed for each node. This information may not be available for some applications where two or more access patterns exist in the same loop without any notion of geometric proximity, which is more general assumption for applications we have to deal. Even if it is available, user annotations may be needed since the compiler may not be able to automatically derive coordinate information for each data item. Instead of relying on coordinate information, partitions may be computed using the underlying graph structure that is constructed by connecting data elements accessed in the same loop iteration. Spectral methods can be effective but are computationally intensive [45]. More recently, people have employed multi-level graph partitioning algorithms encapsulated in library packages such as METIS [28, 29].

Multi-level graph partitioning algorithms work by first computing a succession of coarsened graphs (with fewer nodes) which approximate the original graph. Graphs are coarsened by randomly choosing edges in a graph and matching (collapsing) its endpoints to form a node in the new coarsened graph. By only selecting edges connecting unmatched nodes, the number of nodes is reduced roughly by half at each stage. Once the graph reaches a reasonable number of nodes, k-way graph partitioning algorithms are used to split the coarsened graph into k partitions. The partition is then successively mapped back towards the original graph, periodically refining (adjusting) the partition. Figure 3 provides an example of multi-level graph partitioning applied to a 2D domain.

The multi-level algorithm improves efficiency, since the computationally expensive k-way graph partitioning algorithm can be applied to a much smaller reduced graph. Analysis and measurements show multi-level graph partitioning algorithms are reasonably fast and produce good partitions, as measured by the number of *cut edges* which cross partitions [31, 29]. The quality of the partitions are fairly close to that achieved by applying the k-way graph partitioning algorithm to the original graph, but at a fraction of the expense.

3.3 Hierarchical graph clustering (GPART)

Since our goal is to improve cache performance at run-time for irregular applications, we desire partitioning algorithms with lower overhead. We have developed a hier-

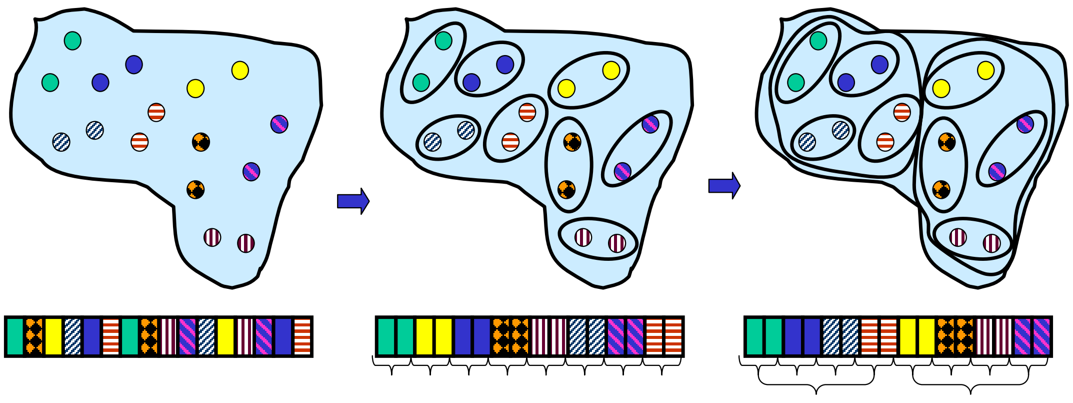


Figure 4 Graph Partitioning Using Clustering (GPART)

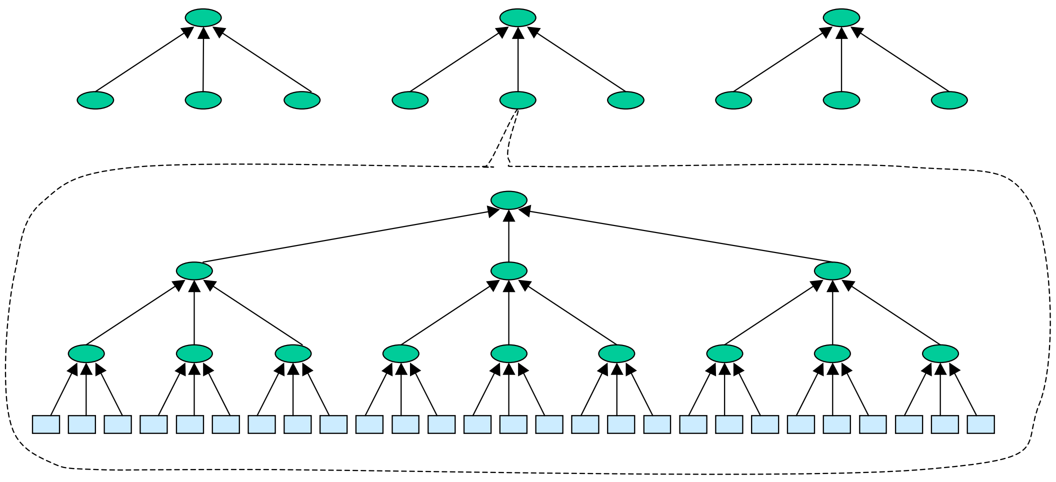


Figure 5 Hierarchical Graph Clustering (GPART)

archical graph clustering algorithm (GPART) that has low overhead and produces quality ordering for cache locality. Previous multi-level graph partitioning algorithms were designed for improving locality in parallel computations, VLSI circuits, and other high cost applications. As a result, greater processing time is acceptable. Experiments indicate the overhead of preprocessing for RCB and METIS are 7–45 times higher than consecutive packing.

Figure 4 and Figure 5 depict how GPART works. GPART collapses several nodes into a single partition in each pass, then a partition is treated as a single node in the next pass. The same clustering algorithm is repeatedly applied through several passes, keeping a hierarchical structure as depicted in Figure 5. The main characteristics of GPART are as follows:

- We randomly pick neighboring nodes to collapse them together. Random selection may result in less quality partitions than other sophisticated methods, but reduces overhead.
- The nodes that are collapsed together in a pass make a partition in that pass. The nodes in the same partition are laid out in nearby memory locations.
- We keep hierarchical structures of partitions as in Figure 5. Smaller partitions (subpartitions) in the previous pass are stored in consecutive locations, if they belong to the same partition in the current pass.
- Large number of nodes are collapsed in each pass. We begin by collapsing enough nodes to fit on a single cache line, then collapse a large number (8 in our research) of nodes at once. We also stop after partitions reach sufficiently large size (usually after the partition size exceeds L1 cache size).
- To achieve improved partitions, we sort nodes once by their vertex degrees (number of incipient edges), and attempt to collapse in the sorted order. Sorting improves the likelihood that partitions keep closely related subpartitions together through multiple passes.

```
sort nodes by degree
limit = cache_line_size
while (limit <= max_limit) {
   for each node N (in sorted order) {
      P = partition for N
      if (size(P) < limit) {
         for each partner M of N {
         /* pick partner in random order */
            Q = partition of M
            if (size(P) + size(Q) <= limit) {
               merge Q into P
               if (size(P) = limit)
                  break (exit for loop)
            }
         }
      }
   }
   sort nodes by partition
   limit = limit * increment
}
```

Figure 6 Hierarchical Graph Clustering Algorithm

The partitions produced generate data layouts which improve locality by putting neighboring nodes in nearby locations. The hierarchical structure in GPART is similar to that of RCB. Each partition in a level keeps all its subpartitions in lower levels close in memory. Keeping hierarchical structures helps store related nodes close in memory, reducing cache misses. Since applications visit nodes according to the storage order, maintaining the hierarchical structure of the clustered graph increases the chance of visiting neighbor nodes which have already been accessed by computations for previous nodes. These nodes are more likely to remain in cache, improving performance.

The overall GPART algorithm is shown in Figure 6. GPART is similar to the coarsening phase in METIS, but fewer clustering passes are required because more nodes are collapsed at once. The quality of the partition produced is slightly less than that produced by METIS, but the overhead is significantly less. In addition, we can adjust the number of clustering passes performed by changing the number of nodes collapsed together in each pass, as well as the largest partitions allowed.

3.4 Complexity comparisons

We pause to take a quick analytical look at the complexity of different locality optimizations, as an function of N, the number of nodes, E, the number of edges, and C, the cache size ($N \approx E$, $N \gg C$ for typical input graphs in our research). The complexity analysis may help in deciding which locality optimization to perform.

Consecutive packing (CPACK) has cost $O(E)$, since it rearranges data based on the order data is first traversed and requires one pass through the data. Using space-filling curves can cost close to $O(N)$, since the problem domain may be uniformly partitioned according to the coordinate space, and simple mapping between coordinates and grid partitions may assign node elements to appropriate grid partitions.

RCB has cost $O(N(\log_2 N)^2)$, since recursively sorting and splitting the data in two at each level of recursion requires up to $O(\log_2(N))$ passes over the data. METIS has cost $O(2N \log_2 N) + O(E)$, where $O(E)$ is the cost of computing a k-way partition on the reduced graph [28]. Up to $O(\log_2 N)$ passes over the data are needed to coarsen the graph, and a similar number of passes are used to refine the partition boundaries after partitioning the coarsened graph.

In comparison, GPART has cost $O(E \log_8(C))$, since the number of clustering passes is dependent on the cache size, not the input data size. As a result, GPART is only a small constant factor more expensive than either CPACK or HILBERT, and should be much more efficient than RCB or METIS for large data sets.

4 Implementation Issues

In scientific applications that traverse meshes or simulate N-body interactions, input data are often represented in the form of graphs. The values related to nodes in meshes or related to bodies are stored in the arrays that are regarded as node structures. Mesh connections or body interactions are stored in separate arrays that are regarded as edge structures. Using node and edge structures, applications implement input data as graphs.

For node structures, arrays are used to keep values associated with each node (*e.g.*, weight, velocity, force, *etc*). One multi-dimensional array may be used instead of several arrays, but only arrays frequently used together should be merged into a multi-dimensional array to fully exploit memory bandwidth and cache line utilization [15].

For edge structures, two type of structures are commonly used; *edge lists* and *partner lists*. Figure 9 demonstrates how a graph is implemented using *edge* and *partner lists*, along with changes due to data and computation reordering. In the example, we store upper triangular part of

```
** edge list **
  do time_step =
    do k = 1, num_edges
      i = left[k]
      j = right[k]
      f = compute(x[i],x[j])
      update(y[i],y[j],f)

** partner list **
  do time_step =
    do i = 1, num_nodes
      do p = start[i],start[i+1]-1
        j = partners[p]
        f = compute(x[i],x[j])
        update(y[i],y[j],f)
```

Figure 7 Examples of *edge list* and *partner list*

```
** original code before reordering **
  do time_step =
    do i = 1, num_nodes
      do p = start[i],start[i+1]-1
        j = partners[p]
        f = compute(x[i],x[j])
        update(y[i],y[j],f)
        z[i] = fn1(y[i],y[j]) // S1
    do i = 1, num_nodes
      z[i] = fn2(y[i])        // S2
```

```
** after reordering x, y **
  REORDER(x, y)
  do time_step =
    do i = 1, num_nodes
      do p = start[i],start[i+1]-1
        j = partners[p]
        i2 = idx[i]       // additional
        j2 = idx[j]       // indirection
        f = compute(x[i2],x[j2])
        update(y[i2],y[j2],f)
        z[i] = fn1(y[i2],y[j2])
    do i = 1, num_nodes
      i2 = idx[i]          // additional
      z[i] = fn2(y[i2])    // indirection
```

Figure 8 Additional Indirection

adjacency matrix, assuming edges have no direction. The *edge list* consists of two arrays, `left` and `right`. A pair of elements from each array represent the end points of an edge. The values stored in each array point to the locations of nodes in node structures. The *partner list* is composed of `partners` and `start`. Instead of explicitly keeping both end points of edges, each node keeps a partner list that stores neighboring nodes. The lists are often combined into one long list, `partners`, for easy management and compact space. The beginning of each list is stored in each element of `start`. The extra element in the last position of `start` plays a *sentinel*, making the codes correctly work for the last node. The values in `partners` are also the positions of nodes in node structures.

Figure 7 shows an example codes that use an *edge list* and a *partner list*, respectively. The codes access edges one by one. It calculates a value, `f` from two node values, `x[i]` and `x[j]`, and updates two node values, `y[i]` and `y[j]`, using the previously calculated value, `f`.

4.1 Data reordering

To automatically apply data layout transformation, compilers need to recognize the code structures shown in Figure 7. Once the compiler recognizes the structures, it can figure out which arrays correspond to node data structures. The compiler can then insert a run-time library call that transforms the layout of node data. For RCB, users may need to provide an extra information about coordinates of nodes. Figure 9 shows an example input graph and node data structures before and after applying a certain data reordering algorithm. The alphabet letters inside the node represent values kept in node structures and the numbers outside the node represent the order of nodes stored in the node structures. The order of node values are changed after the data reordering.

Reordering node data results in another level of indirection for the structures pointing to the node structures to access correct nodes in newly ordered node data. For example, edge structures point to node structures to have an information of connected nodes. Refer to Figure 8. The first segment of codes shows an original code before data reordering and the second segment of codes shows a modified code after data reordering. Additional indirections are introduced to the edge values using an indirection array, `idx`, which points to correct node locations in the reordered node structures, `x` and `y`. To avoid these indirections, we may update edge structures so that they point to the correct positions in the new node structures. However, compilers should guarantee that updating edge structure does not affect the legality of the rest of the computations. Figure 9 also shows an example of updating edge values.

Another source of indirect accesses comes from updating other data structures based on the reordered node data. Refer to the two statements, `S1` and `S2`, in Figure 8. After reordering the node structures, `x` and `y`, indirections are needed when computation involves reordered data structures and other data structures, `z`. In this case, indirections may be eliminated by reordering the other data structures, `z`, in the same way as the reordered data structures. Compilers also need to guarantee that transforming the other data does not affect the legality of the rest of the codes. Once we can eliminate all indirections, the resulting code will be the same as the original code in Figure 8 except for the routines that reorder nodes and update edges outside the `time_step` loop. Discussions about such techniques and compiler support can be found in Ding and Kennedy's research [14].

4.2 Computation reordering

As the codes shown in Figure 7, computations usually proceed according to the order of edges stored in an *edge list* or a *partner list*. Thus, computation reordering can be achieved by reordering the *edge list* or the *partner list*. Compilers should verify that computations are associative to guarantee the legality of transformation. Parallel loops are surely legal to reorder their computations. Since the computation reordering sorts edges according to the order of nodes they are connecting, computation reordering is

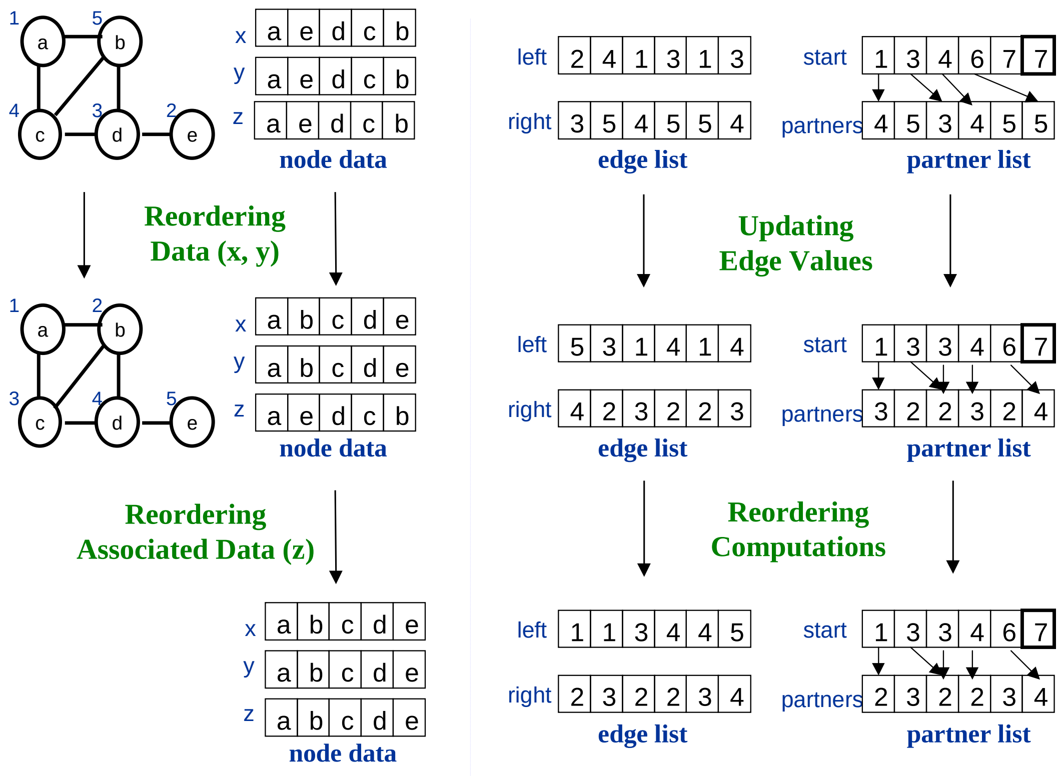


Figure 9 Implementation of Data and Computation Reordering

usually applied after the data reordering.

Once compilers recognize the computation structures shown in Figure 7, compilers can recognize which arrays correspond to edge data structures. Then, the compiler inserts an appropriate run-time library call that lexicographically sorts *edge list* or *partner list*, which is described in Section 2.1. Figure 9 shows an example of computation reordering for *edge list* and *partner list*. The values in `left`, `right`, and `partners` point to the positions of the nodes in node structures. In this example, the input graph is not presorted. After data reordering is applied, the values of edge structures are updated to match with the correct positions in the new node structures. Then, the computation reordering is applied to lexicographically sort the edge structures.

4.3 Inspector/executor paradigm

Locality optimizations for irregular codes can be automated using a combination of compiler and run-time techniques first developed in the context of identifying interprocessor communication for message-passing machines. Saltz *et al.* designed a compiler which can generate an *inspector* to preprocess memory access patterns to identify non-local data needed by each processor [13]. Previous researches on data-parallel compilers such as the Fortran D compiler have discussed compiler techniques for automatically generating inspectors and executors for CHAOS [21, 25]. Simplified versions of those techniques can be used to generate inspectors and executors for locality optimizations. Compilation techniques were also developed to determine when inspectors are needed, as well as when they must be rerun if memory access patterns change for adaptive computations [19, 25].

Inspectors for locality optimizations are given as run-time library calls. These locality inspectors are inserted by compilers and modify the data and computation order at run-time. The overheads of the inspectors are also amortized over many time steps. For example, locality inspectors can be inserted outside the `time_step` loop in Figure 7. Then, the inspectors execute once before the computation begins, benefiting computations throughout the whole `time_step` loop. When access patterns change such as in adaptive computations, locality inspectors may be inserted. However, unlike inspectors for interprocessor communications, locality inspectors need not to be rerun each time the access patterns change. Without rerunning the locality inspectors, the cache performance may degrade, but the codes still generate correct results. Due to this nature, we develop a guide for adaptive computation that decides how often inspectors are to be rerun, considering their benefit and their overhead. We will discuss this issue in more detail later.

```
** IRREG (edge list)
do time_step =
  do i = 1, num_edges
    n1 = left[i]
    n2 = right[i]
    force = (x[n1]-x[n2])/4
    y[n1] += force
    y[n2] += -force
```

```
** NBF (partner list)
do time_step =
  do i = 1, num_atoms
    do p = start[i],start[i+1]-1
      j = partners[p]
      d = x[i]-x[j]
      force = d**(-6)/1000
      y[i] += force
      y[j] += -force
```

```
** Moldyn (edge list)
do time_step =
  do i = 1, num_interactions
    n1 = left[i]
    n2 = right[i]
    d = distance(x[n1],x[n2])
    if (d < cutoff)
      force = d**(-7) - d**(-4)/2
      y[n1] += force
      y[n2] += -force
```

Figure 10 Key Kernels of Applications

Name	# Nodes	# Edges	Description
FOIL	144649	1074393	3D mesh of a parafoil
SUB	214765	1679018	3D mesh of a submarine
AUTO	448695	3314611	3D mesh of GM's Saturn
MOL1	131072	1179648	semi-uniform 3D molecular dynamics mesh (sm)
MOL2	442368	3981312	semi-uniform 3D molecular dynamics mesh (lg)

Table 1 Input Meshes

5 Evaluating Locality Optimizations

5.1 Evaluation environment

We have begun implementation of our prototype compiler using the Stanford SUIF compiler. The prototype can identify and parallelize irregular reductions, generating parallel *pthreads* programs. The compiler can also determine where inspectors need to be inserted, based on when memory access patterns are changed. However, we have not implemented the actual inspector generation phase using techniques developed by Chaos [20, 21]. As a result, we currently insert inspectors by hand for both the sequential and parallel versions of each program.

In our experiments, we measured both cache miss rates and actual sequential and parallel execution times for a DEC multiprocessor with four 275MHz Alpha 21064 processors. Each processor has a 16K direct-mapped L1 cache with 32 byte cache lines, as well as a 4M direct-mapped L2 cache with 64 byte cache lines. Cache miss rates were measured using a cache simulator based on the Shade utility from Sun Microsystems.

5.2 Applications and meshes

We examine three irregular applications, IRREG, NBF, and MOLDYN. All applications contain an initialization section followed by the main computation enclosed in a sequential `time_step` loop. The main computation is thus repeated on each iteration of the `time_step` loop. Statistics and timings are collected after the initialization section and the first few iterations of the `time_step` loop, in order to more closely match steady-state execution.

IRREG is a representative of iterative partial differential equation (PDE) solvers found in computational fluid dynamics (CFD) applications. In such codes, unstructured meshes are used to model physical structures. The mesh is represented by nodes and edges. The main computation kernel iterates over the an *edge list*, computing modifications to its end points. IRREG computes a force which is applied to both endpoints of an edge. Modifications to the value of all nodes are in the form of irregular reductions.

NBF is a kernel abstracted from the GROMOS molecular dynamics code [20]. Instead of an edge list as in IRREG, it maintains a *partner list* for each molecule. Partner lists are more compact than edge lists, but need extra data structures to specify the range of partners for each molecule. NBF computes a force which is applied to both a molecule and its partner.

MOLDYN is abstracted from the non-bonded force calculation in CHARMM, a key molecular dynamics application used at NIH to model macromolecular systems. Like IRREG, an edge list representing interactions between pairs of molecules is maintained. Since the strength of interactions between molecules drops with increasing distance, only molecules within a cutoff distance of each other are assumed to interact. The main computation kernel iterates over all interactions between molecules, computing a single force which is applied to both interacting molecules. The key kernels of these codes are shown in Figure 10.

To test the effects of locality optimizations, we chose a variety of input data meshes. FOIL, SUB, and AUTO are 3D meshes of a parafoil, submarine, and GM Saturn automobile, respectively. The ratios of edges to nodes are between 7–10 for these meshes. MOL1 and MOL2 are small and large 3D meshes derived from semi-uniformly placed molecules of MOLDYN using a 1.5 angstrom cutoff radius.

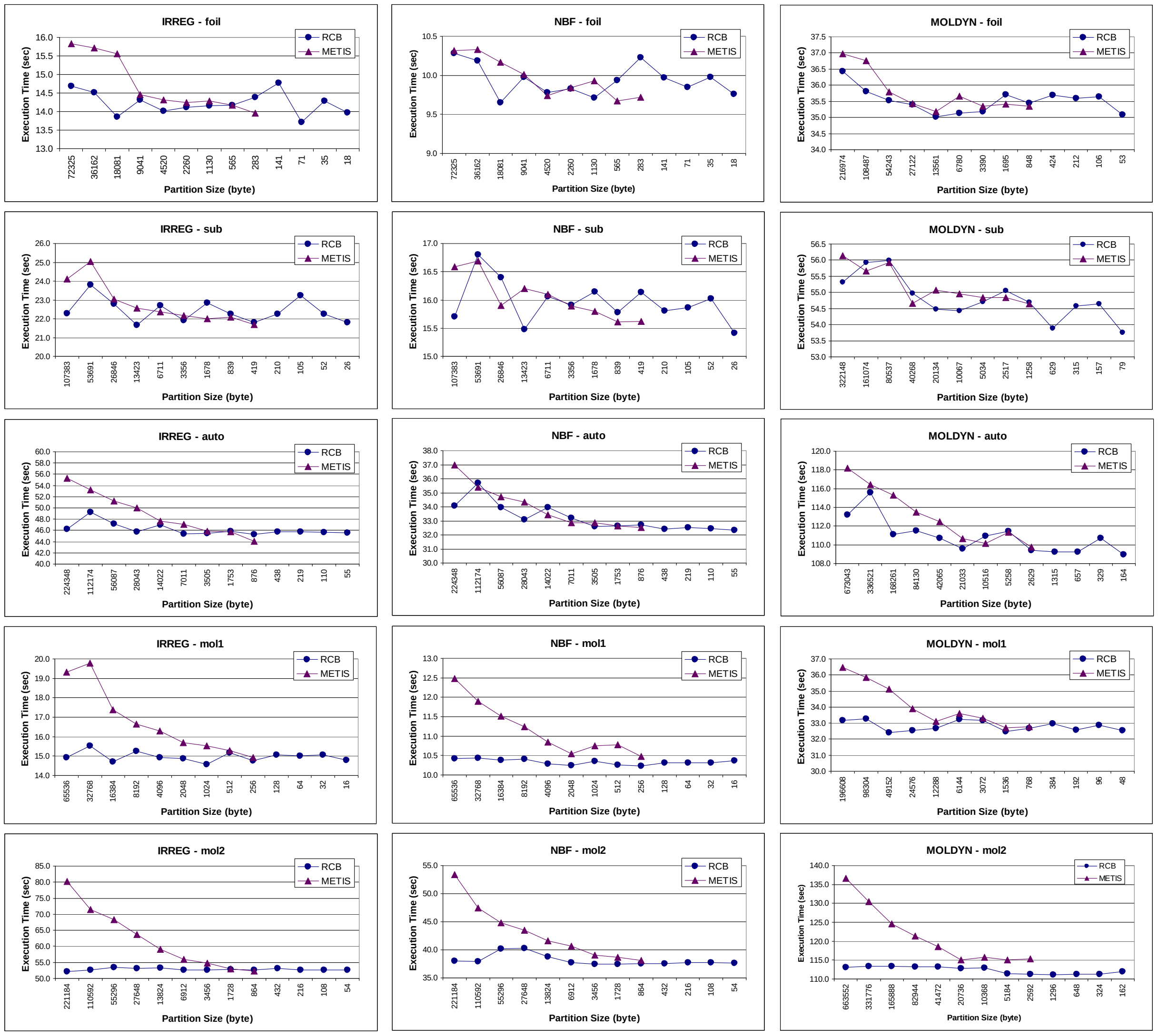


Figure 11 Impact of Partition Sizes of RCB and METIS (overhead excluded)

We applied these meshes to IRREG, NBF, and MOLDYN to test the locality effects of implementing edge lists and partner lists. All meshes are initially sorted, so computation reordering is not required originally, but computation reordering is applied only after data reordering techniques are used.

5.3 Partitioning parameters (RCB, METIS)

An important issue is how to select parameters for partitioning algorithms when used as locality optimizations. For classic partitioning algorithms such as RCB and METIS, the main parameter is how many partitions to create. For RCB, larger domains are recursively divided into halves until the total number of desired partitions is reached. METIS can also produce any number of desired partitions. The input graph is coarsened until the resulting graph is sufficiently small so that a min-cut algorithm can be applied to produce the target number of partitions.

The number of partitions affects the locality of the resulting program. More partitions divide the data into smaller groups, increasing the likelihood that data in a partition can remain in cache. Fewer partitions produce larger number of elements in each partition, reducing the probability elements accessed will still be in cache. However, overhead increases with the number of partitions, so we cannot simply choose an arbitrarily large number of partitions. Smaller partitions may also lead to more accesses outside the partition, so may be counter-productive unless partitions are also grouped hierarchically.

To evaluate desirable partitioning parameters, we performed a number of experiments by choosing different numbers of partitions for each combination of kernel and application mesh. Results are shown in Figure 11. The

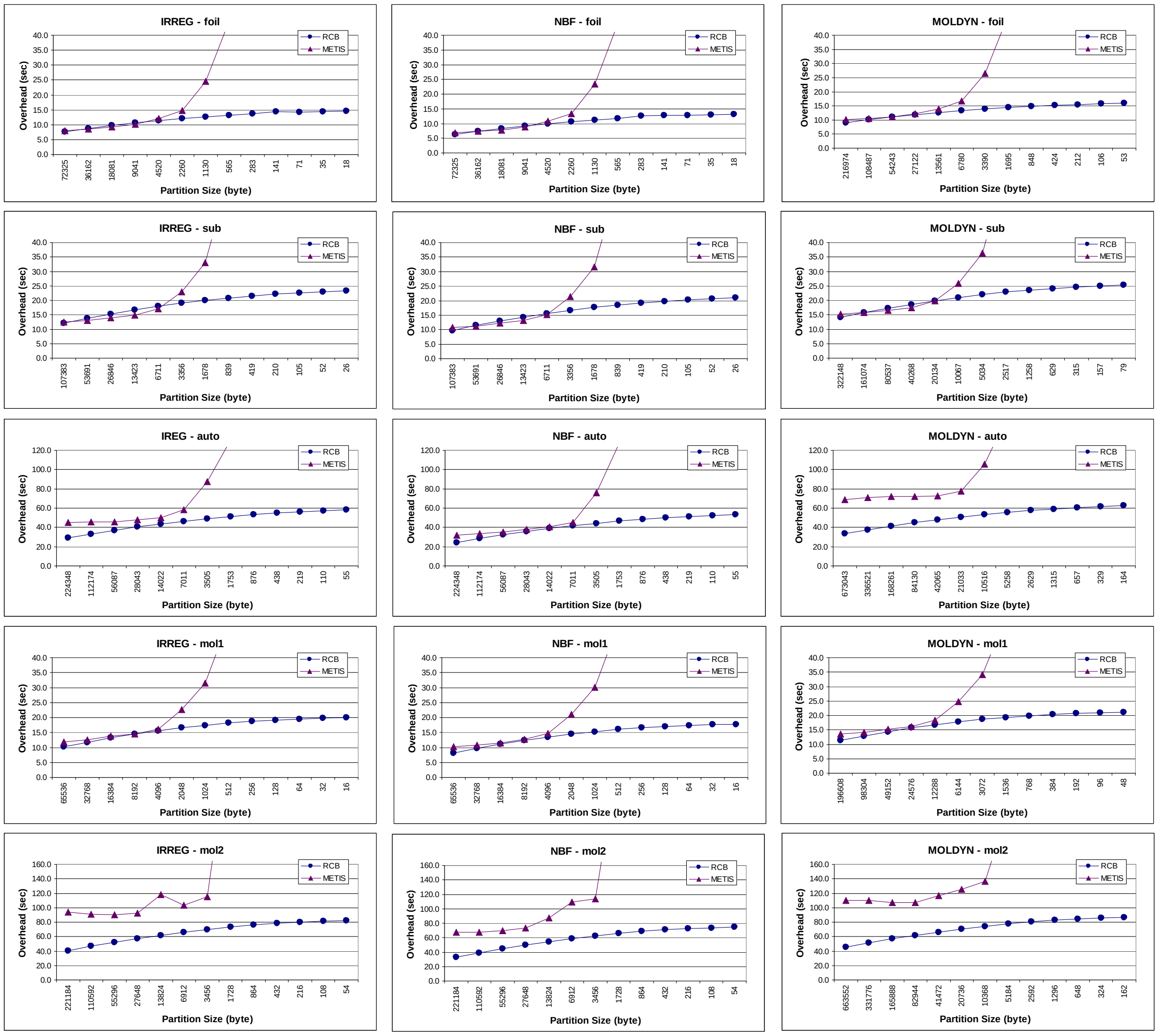


Figure 12 Overhead vs. Partition Size for RCB and METIS

x-axis represents the size in bytes of data in each partition. The y-axis presents execution time in seconds for 40 iterations of the computation, excluding the overhead of layout optimizations. The overheads are excluded to focus on the quality of partitions, versus various partition sizes. As expected, results show execution time generally improves as smaller partition size is selected, but we need to take into account the overhead.

Figure 12 presents the overheads for each partitioning optimization with respect to the partition size. As we can see, overheads go up quickly as the number of partitions increases, particularly for METIS. Thus, we should compromise to get most locality benefit without much overhead.

Based on these experimental results, we find a good criterion for choosing the number of partitions is based on the relationship of partition size to cache size. When the size of all the node data accessed in a partition is roughly on the order of the data cache or smaller, we seem to obtain most of the locality benefits available. The system can thus choose a desirable number of partitions by examining the number of data arrays accessed, as well as the size of each data element. It then choose a number of partitions sufficient to divide data into L1 cache-sized chunks.

5.4 Collapsing parameters (GPART)

Similar to RCB and METIS, a possible parameter for graph clustering is the number of partitions. However, unlike RCB and METIS which begin with a single partition and creates more partitions, GPART begins with each node as a partition and repeatedly clusters them together through multiple passes. Another way of specifying parameters for GPART is thus to specify the desired partition size after

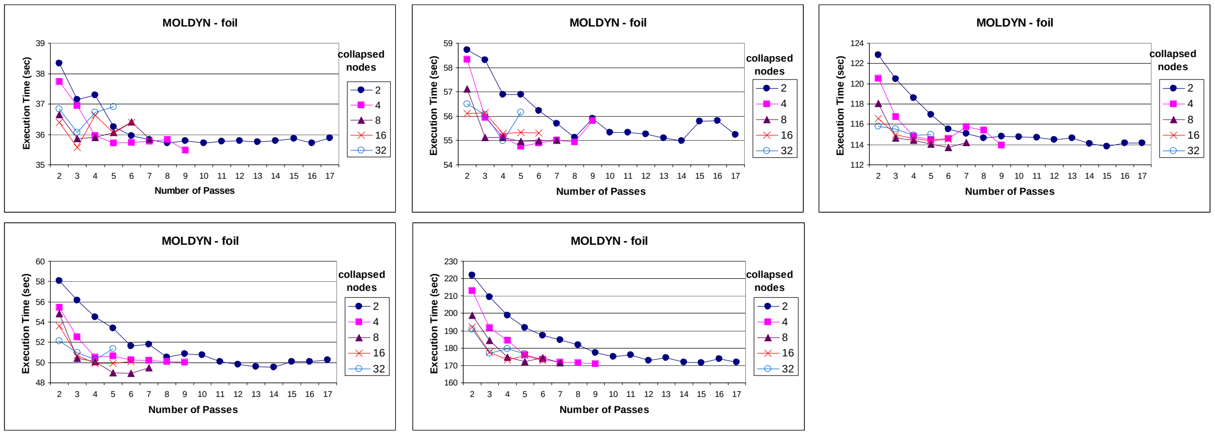


Figure 13 Impact of GPART Parameters (overhead excluded)

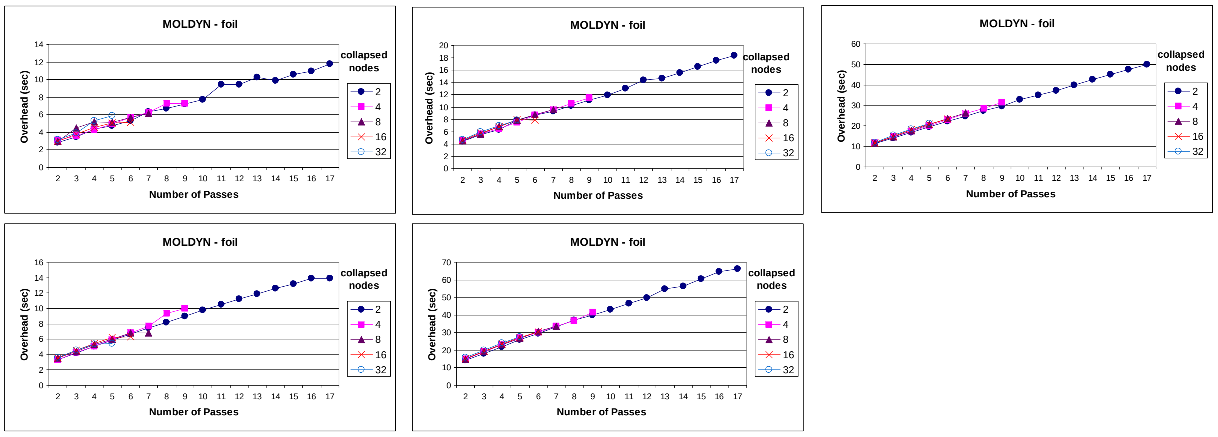


Figure 14 Overhead vs. GPART Parameters

the last clustering pass, and how many nodes to collapse in each clustering pass.

Again, to evaluate desirable partitioning parameters we performed a number of experiments by combining MOLDYN with different application meshes, applying GPART with different collapsing rates. Results are shown in Figure 13. The x-axis represents the number of clustering passes performed. The y-axis presents execution time in seconds for 40 iterations of the computation after optimization, excluding the overhead of layout optimizations. The overheads are excluded again to focus on the quality of partitioning with respect to the various parameters. Different data series represent different collapsing rates, ranging from collapsing 2 to 32 nodes in each clustering pass. The number of nodes in a partition at each pass can then be calculated as the collapsing rate raised to the n_{th} power, where n is the number of passes performed. We see that performance improves with more passes, and collapsing a reasonably large number of nodes on each pass does not hurt performance.

Figure 14 shows the overheads in seconds versus numbers of collapsing passes for various collapsing rates. As expected, the overhead becomes higher when the number of passes increases. One thing to notice is that collapsing rates little affect the overheads. Thus, selecting best performance under the same number of collapsing passes is a right way to decide parameters when you consider the performance including the overhead.

Once again, results indicate most locality benefits may be obtained by having the size of all the data accessed in a partition to be roughly the size of the data cache or larger. The choice of the number of nodes to collapse in each pass is more tricky when you consider the quality of partitions. Small collapsing rates yield quality partitions, but require more passes. Large collapsing rates take fewer passes, but produce poor quality partitions. Our results seem to indicate collapsing eight nodes at a time yields reasonably good locality while keeping the number of passes (and overhead) low.

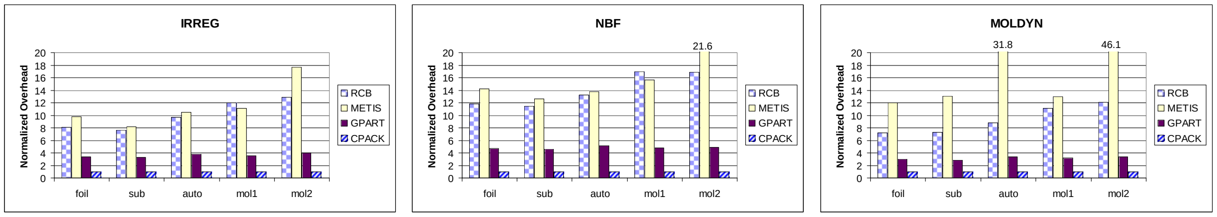


Figure 15 Normalized Overhead of Data Reordering Algorithms (CPACK = 1)

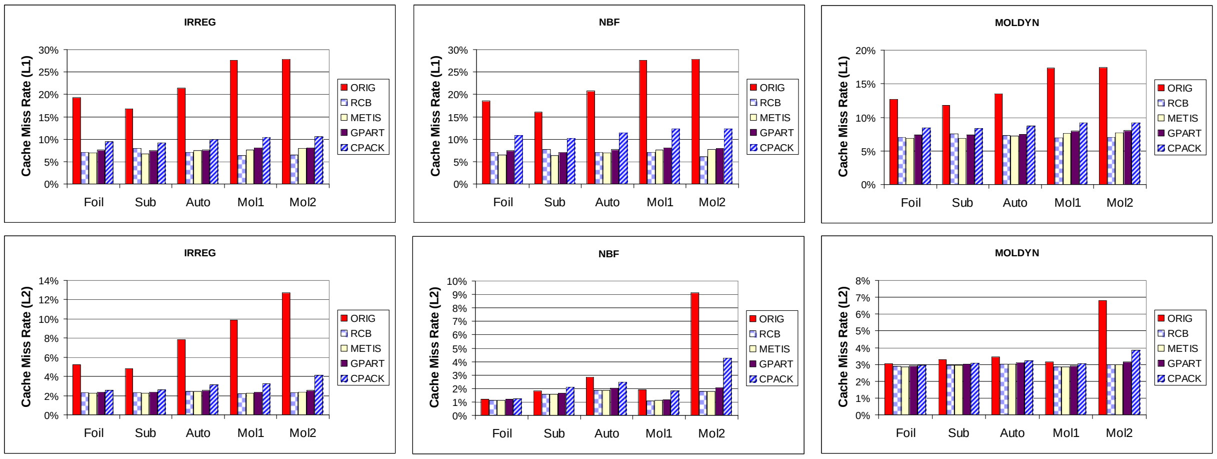


Figure 16 L1 & L2 Cache Miss Rates (percentage of all memory references)

5.5 Optimizations

In our experiments, we evaluated several optimization techniques for each application/mesh combination. The different optimizations are as follows:

- ORIG. The original program. Since input meshes are presorted, it corresponds to the result of computation reordering.
- RCB. Recursive coordinate bisection is applied to rearrange data layout, based on user-provided coordinate information.
- METIS. The multi-level graph partitioning algorithm in the METIS library is used to compute data partitions and rearrange data layout.
- GPART. The graph is partitioned through successive clustering passes. Initial partitions contain 4 nodes, additional passes collapse up to 8 nodes. The sequence of passes thus produces partitions containing 4, 32, 256, 2K, 16K nodes through 5 passes.
- CPACK. The data layout is rearrange by consecutive packing based on the original access order.

Both RCB and METIS use partition parameters described in Section 5.3. The number of partitions selected is based on the partition size for all data arrays accessed in the computation is roughly the L1 cache size.

In GPART we start from a partition containing 4 nodes, since each node in our applications contains an 8-byte double-precision floating point number. Thus, 4 nodes will fit into 32-byte cache line in DEC Alpha processors. We merge 8 nodes on each pass, resulting in 5 passes.

As we mentioned early, computation reordering is applied after any data reordering algorithms are applied.

5.6 Overhead of optimizations

Figure 15 displays the costs of data reordering techniques measured relative to the cost of CPACK. The overhead includes the cost to update edge structures and transform other related data structures to avoid the extra indirect accesses caused by the data reordering. The overhead also includes the cost of computation reordering.

The least expensive data layout optimization is CPACK which we use as the base for comparison. In comparison, METIS is quite expensive when used for cache optimizations, on the order of 10–45 times higher than CPACK. RCB is less expensive than METIS, costing around 8–17 times higher than CPACK. The overhead of GPART is 3–5 times higher than CPACK, but much less than METIS and RCB.

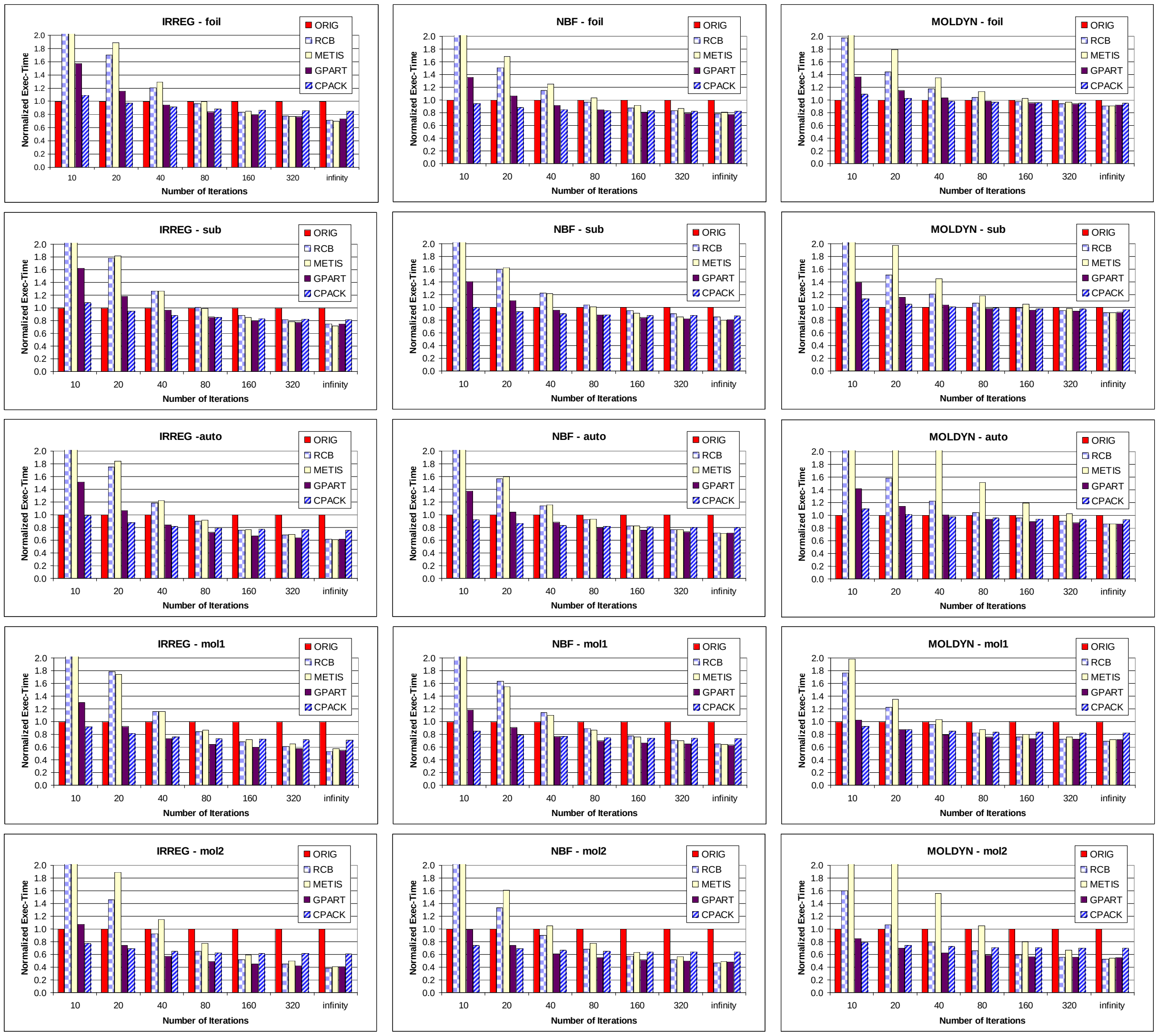


Figure 17 Normalized Execution Time for Optimizations (ORIG = 1, overhead included)

5.7 Impact on miss rates

We first look at simulated L1 cache miss rates for a 16K cache, similar to the L1 (primary) cache for the DEC Alpha 21064. Miss rates are shown in the upper three graphs of Figure 16. Miss rates of original, unoptimized programs are about 12%–28%. All partitioning techniques, RCB, METIS, and GPART reduce miss rates to 6%–8%, outperforming CPACK by 1%–6%. RCB and METIS achieve the best over all performance. GPART obtains nearly the same performance.

We also look at simulated L2 cache miss rates for a 4M cache, similar to the L2 (secondary) cache for the DEC Alpha 21064. The lower three graphs in Figure 16 show miss rates for the L2 cache. Miss rates are calculated as the percentages of all memory references, not the percentages of L2 cache references. L2 miss rates are also reduced by applying partitioning techniques, outperforming CPACK. RCB and METIS show the best performance, but GPART also achieves nearly the same performance. Partitioning techniques achieve lower miss rates than original codes by up to 7.5%.

5.8 Impact on sequential performance

Next we look at sequential execution times on the DEC Alpha. Normalized execution times are presented in Figure 17, calculated relative to the execution time of ORIG. These results include the overhead of data layout optimizations. Since optimizations are performed once at the beginning, the benefits of the optimizations are accumulated through the iterations. Thus, the performance becomes better as the number of iterations increases.

Set of bars at *infinity* in the Figure 17 are shown to

	Normalized overhead	Cache miss rate		Normalized sequential time
		L1	L2	
ORIG	N/A	19.77%	5.15%	1.000
RCB	11.15	7.08%	2.25%	0.692
METIS	16.74	7.21%	2.25%	0.695
GPART	3.87	7.70%	2.37%	0.698
CPACK	1.00	10.04%	2.92%	0.799

Table 2 Averages of Experimental Results (overhead excluded in sequential times)

compare quality of each optimization, ignoring its overhead. Partitioning algorithms perform better than CPACK in quality of data reordering. RCB, METIS, and GPART achieve 10%–60% improvements, while CPACK improves execution times about 5%–40%.

In real situations we should consider the overheads of locality optimizations. Even though RCB and METIS generate better quality ordering, they require hundreds of iterations to be competitive with CPACK. Meanwhile, GPART begins to outperform CPACK around 40 iterations due to its low overhead. Since some scientific applications repeat the computation several hundreds of times without changing access patterns, RCB and METIS will be most beneficial for those applications. The overhead of data layout optimizations can be amortized, just as for inspectors used to reduce communications in parallel codes. GPART and CPACK will be more effective for applications with small numbers of iterations due to their low overhead. Thus, selecting a proper locality optimization algorithm is dependent on the number of iterations expected, but all should prove beneficial. Due to the low overheads of GPART and CPACK, they can be also effectively used in adaptive applications that occasionally change access patterns. We will discuss adaptive computations in more detail later.

5.9 Summary of experimental results

Since we present many experimental results for different applications and different input meshes, we try to summarize the results by averaging all combinations. Table 2 shows the averages of the results for the original code and the different optimization algorithms The second column shows the average overhead of each optimization relative to the average overhead of CPACK. The third and forth columns show average L1 and L2 cache miss rates, respectively. The fifth column presents average sequential execution times relative to the average sequential execution time of the original code. To summarize the results again, RCB and METIS produce the best quality data ordering, achieving 30% improvement in sequential execution times, but require high overheads. GPART achieves nearly the same performance with a third overhead of RCB or with a quarter overhead of METIS. CPACK is the least overhead algorithm, but improves only 20% for the sequential execution times.

6 Parallel Codes

6.1 Parallelizing irregular reductions

In addition to improving locality of irregular codes for sequential execution, locality can also be improved for parallel execution. The core of irregular scientific applications is frequently comprised of *reductions*, associative computations (e.g., SUM, MAX) which may be reordered and parallelized [41, 34, 48]. Compilers for shared-memory multiprocessors generally parallelize irregular reductions by having each processor compute a portion of the reduction, storing results in a local *replicated buffer*. Results from all replicated buffers are then combined with the original global data, using synchronization to ensure mutual exclusion [18, 41].

An example of the REPLICATEBUFS technique is shown in Figure 18. If large replicated buffers are to be combined, the compiler can avoid serialization by directing the run-time system to perform global accumulations in sections using a pipelined, round-robbin algorithm [18]. REPLICATEBUFS works well when the result of the reduction is to a scalar value, but is less efficient when the reduction is to an array, since the entire array is replicated and few of its elements are effectively used.

Previously we introduced LOCALWRITE, a new compiler and run-time technique for parallelizing irregular reductions [19]. LOCALWRITE avoids the overhead of replicated buffers and mutual exclusion during global accumulation by partitioning computation so that each processor only computes new values for locally-owned data. It simply applies to irregular computations the *owner-computes* rule used in distributed-memory compilers [22]. LOCALWRITE is implemented by having the compiler insert inspectors to ensure each processor only executes loop iterations which write to the local portion of each variable. Values of index arrays are examined at run time to build a list of loop iterations which modifies local data.

An example of LOCALWRITE is shown in Figure 19. Computation may be replicated whenever a loop iteration assigns the result of a computation to data belonging to multiple processors (*cut edge*). The overhead for LOCALWRITE should be much less than classic inspector/executors, because the LOCALWRITE inspector does not build communication schedules or perform address translation. Besides, LOCALWRITE does not perform global accumulation for the non-local data. Instead, LOCALWRITE replicates computation, avoiding expensive communications across processors.

```
global x[nodes],y[nodes]
local  ybuf[nodes]
do t =             // time-step loop
  ybuf[] = 0       // init local buffer
  do i = {my_edges} // local computation
    n = idx1[i]
    m = idx2[i]
    force = f(x[m], x[n])
    ybuf[n] += force   // updates stored in
    ybuf[m] += -force  //   replicated ybuf
  reduce_sum(y, ybuf)  // combine buffers
```

Figure 18 REPLICATEBUFS Example

```
global x[nodes],y[nodes]
inspect(idx1,idx2) // calc local_edges/cut_edges
do t =            // time-step loop
  do i = {local_edges} // both LHS's are local
    n = idx1[i]
    m = idx2[i]
    force = f(x[m], x[n])
    y[n] += force
    y[m] += -force
  do i = {cut_edges}   // one LHS is local
    n = idx1[i]
    m = idx2[i]
    force = f(x[m], x[n]) // replicated compute
    y[n] += force or y[m] += -force
```

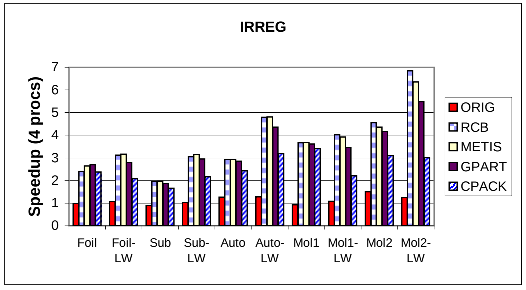
Figure 19 LOCALWRITE Inspector Example

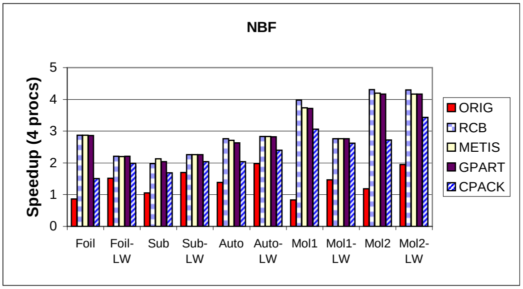
The LOCALWRITE algorithm inspired our techniques for improving cache locality for irregular computations. Conventional compiler analysis cannot analyze, much less improves locality of irregular codes because the memory access patterns are unknown at compile time. The lightweight inspector in LOCALWRITE, however, can reorder the computations at run time to enforce local writes. It is only a small modification to change the inspector to reorder the computations for cache locality as well as local writes. We can use all of the existing compiler analysis for identifying irregular accesses and reductions (to ensure reordering is legal).

Currently data reordering is sequentially performed since we do not have parallel version of those algorithms yet. Once we parallelize the algorithms, we can expect lower overhead in parallel executions.

6.2 Impact on parallel performance

Figure 20 displays 4-processor speedups for each mesh, calculated versus the original, unoptimized program. We exclude overheads to investigate the impact of the qualities of locality optimizations. A cost model that will be described in later sections should be used to decide which optimization should be applied. There are two parallelization options, the original program using REPLICATEBUFS, and a transformed version using LOCALWRITE. We present speedups for each version of the program. In Figure 20, the bars only with input mesh names correspond to the speedups for the REPLICATEBUFS versions and the





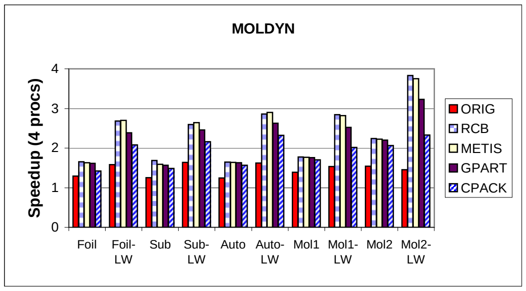


Figure 20 Parallel Speedups (based on unoptimized sequential execution time, overhead excluded)

bars with input mesh names and -LW correspond to the speedups for the LOCALWRITE versions. We see that in all cases, the versions optimized for locality achieved better performance. Locality optimizations are thus carried over to the parallel versions of each program. Table 3 summarizes the parallel speedups by averaging all speedups for different locality optimizations and different parallelization options.

In addition, we found that with locality optimizations, programs parallelized using LOCALWRITE achieved much better speedups than the original programs using REPLICATEBUFS, except for NBF with smaller meshes (FOIL and MOL1). The LOCALWRITE algorithm tends to be ineffective with smaller graphs where duplicated computations are relatively high compared to computations performed locally. In general, however, the LOCALWRITE algorithm benefited more from the enhanced locality. Intuitively these results make sense, since the LOCALWRITE optimization can avoid replicated computation and communication better when the mesh displays greater locality.

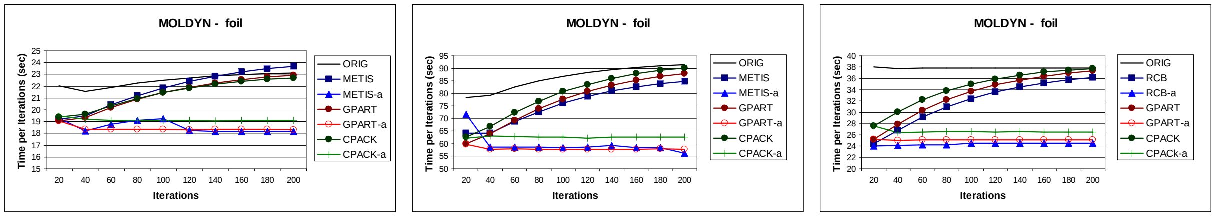


Figure 22 Impact of Adaptivity (-a : transformations are applied whenever access pattern changes, overhead excluded)

	Parallel speedups	
	REPLICATEBUFS	LOCALWRITE
ORIG	1.18	1.48
RCB	2.69	3.40
METIS	2.67	3.36
GPART	2.63	3.10
CPACK	2.15	2.40

Table 3 Average Parallel Speedups (based on unoptimized sequential execution time, overhead excluded)

7 Adaptive Computations

A problem confronting locality optimizations for irregular codes is that many such applications are *adaptive*, where the data access pattern may change over time as the computation adapts to data. The example in Figure 21 is adaptive because condition `change` may be satisfied on some iterations of the time-step loop, modifying elements of the index arrays `idx1` and `idx2`, changing overall data access patterns as a result. For adaptive codes, how quickly the application mesh changes thus affects the locality improvements from data layout optimizations.

7.1 Impact of adaptivity on optimizations

When compiling adaptive irregular computations for distributed memory machines, the compiler must rerun the inspector whenever the connection pattern changes, since new communication may result [25, 19]. Performance suffers for highly adaptive computations with many frequent changes in access patterns.

```
x[nodes],y[nodes]   // data in nodes
do t =              // time-step loop
  if (change)       // change accesses
    idx1[] =
    idx2[] =
  do i = 1, edges   // work on edges
    n = idx1[i]
    m = idx2[i]
    force = f(x[m], x[n])  // computation
    y[n] += force          // update edge
    y[m] += -force         //   endpoints
```

Figure 21 Example Adaptive Irregular Computation

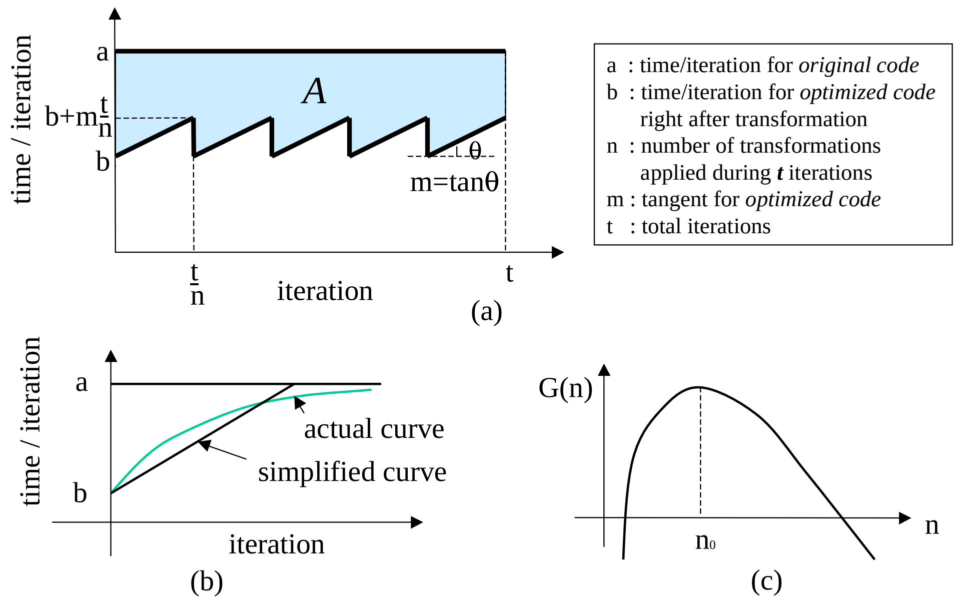
Fortunately, locality optimizations can be more relaxed. First, changes in the access pattern reduce locality and degrade performance, but do not affect the legality of locality transformations. Second, degradations in locality is a function of the amount of change, not the frequency of change. Locality transformations thus do not need to be repeated each time the access pattern changes, only when it is profitable.

To evaluate the effect of adaptivity on locality optimizations, we performed a number of experiments by combining MOLDYN with different application meshes, then periodically swapping the positions of molecules to create an adaptive code. Results are shown in Figure 22. The x-axis marks the passage of time in the computation in groups of 20 iterations. 20% of the nodes are randomly swapped after every 20 iterations. The y-axis measures the execution time per 20 iterations of the kernel, excluding the overhead of the locality transformations. By excluding the overhead, we can better understand the role of locality transformation in adaptive computations.

Each data series represents a different locality optimization. ORIG is the original program. RCB, METIS, GPART, and CPACK represent versions of the program where locality optimizations are applied once at the beginning of program execution. In comparison, RCB–a, METIS–a, GPART–a, and CPACK–a represent adaptive versions of each program where locality optimizations are applied whenever access patterns are changed.

Results show that without reapplying locality transformations, all optimized versions degrade in performance and eventually match the performance of the unoptimized program. In comparison, reapplying partitioning algorithms after access pattern changes can preserve the performance benefits of locality, if overhead is excluded. Performance for the original version of FOIL and AUTO also degrades, because the initial data set has been presorted.

The difficult question is when data should be reordered, if overhead of transformation must be taken into account. Fortunately, it appears that the degradation in performance is mostly proportional to the fraction of data changed. As a result, it should be possible to predict the effectiveness



time / iteration
a
b+m$\frac{t}{n}$
b
A
θ
m=tanθ
$\frac{t}{n}$
iteration
t
(a)
a : time/iteration for *original code*
b : time/iteration for *optimized code* right after transformation
n : number of transformations applied during ***t*** iterations
m : tangent for *optimized code*
t : total iterations
time / iteration
a
b
actual curve
simplified curve
iteration
(b)
G(n)
n
n₀
(c)

Figure 23 Analytical Model for Adaptive Computations

of locality optimizations, based on the rate at which the underlying connection structure becomes disordered due to changes. In the next section we attempt to estimate the benefit using a cost model which takes into account the overhead.

7.2 Adaptive optimization

To guide locality optimizations for adaptive codes, we present a simple cost model for calculating the benefits of locality optimizations for irregular computations. It can also be used to predict how often locality optimizations should be applied and which locality optimization should be used.

To make a simple analytical model we assume input graphs are randomly initialized so that there is almost no locality between nodes. We also assume that a constant amount of edges are randomly selected and changed, after each iteration. We assume two analytical model programs, *original code* and *optimized code*. Figure 23 (a) plots execution time per iteration for both *original code* and *optimized code* excluding the overhead. The upper straight line corresponds to the *original code* and the lower saw-tooth line corresponds to the *optimized code*.

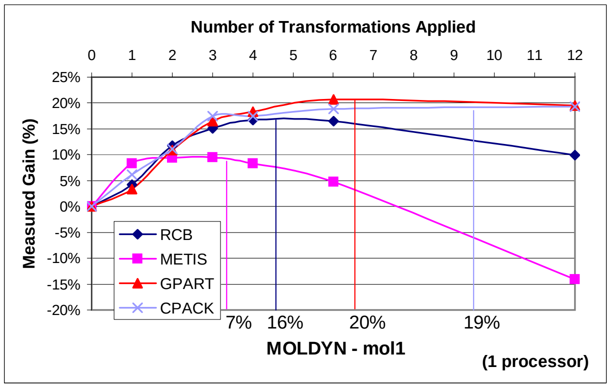
Since input graphs are already randomly initialized, further changes to graphs do not increase the randomness of graphs. Thus, execution times per iteration for the *original code* stay constant. In the *optimized code*, a locality transformation is applied at the beginning of the program and the transformation is periodically applied as the program proceeds. So, the execution times per iteration increase as input graphs change toward random graphs, but periodically drop to the lowest point at the iterations where the locality transformations are applied again. Resulting execution times per iteration will plot saw-tooth shape. Execution times per iteration do not include the overhead of the locality transformation but we will take into account the overhead in our cost model.

Note we are using a simplified model. In the *optimized code*, execution times per iteration approach the constant line of the *original code* with a non-linear rate. However, we approximate this behavior with a linear rate increase as in Figure 23 (b) for the sake of simplicity. Since the performance of *optimized code* is changing at a constant rate (r), we choose the tangent of the line (m) to be $r(a-b)$. For example, if a optimized graph randomly changes 20% of edges at every iteration, then r will be 0.2, becoming a totally random graph after 5 iterations. With the tangent value selected in our model, the execution time per iteration will start from the lowest point (b) and reach the upper constant point (a) after 5 iterations.

The performance gain ($G(n)$) from using periodic locality transformations can be calculated as in the following equation (1). Since the area below the line is the total execution time, the performance gain we can expect is the area between two lines (A) minus the overhead (nO_v).

$$\begin{aligned} G(n) &= A - nO_v \qquad (n \geq 1) \qquad (1) \\ where, & \\ A &= (a-(b+\frac{mt}{2n}))t = (a-b)t - \frac{mt^2}{2n} \\ O_v &= \text{cost of transformation} \\ n &= \text{number of transformations applied} \end{aligned}$$

The performance gain graph is now a function of n, the number of transformations applied, as plotted in Figure



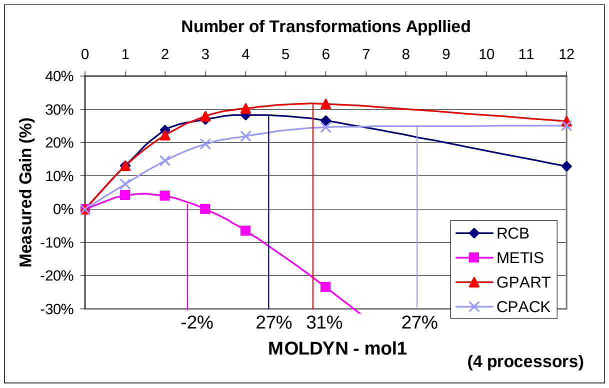


Figure 24 Experimental Verification of Cost Model (vertical bars represent the numbers chosen by the cost model)

23 (c). Finding the point where the performance gain is maximized is straight forward as derived in equation (2) and (3). The maximal point can be determined using its differential equation.

$$n_0 = \left(\sqrt{\frac{m}{2O_v}}\right) t \qquad (G'(n_0) = 0) \tag{2}$$

$$where,$$

$$G'(n) = \frac{d}{dn}G(n) = \frac{mt^2}{2n^2} - O_v$$

Then, maximum gain(G_{max}) can be obtained according to the value of n_0 as in the following equation (3).

$$G_{max} = \begin{cases} G(n_0) = (a-b)t - (\sqrt{2mO_v})t & (n_0 \geq 1) \\ G(1) = (a-b)t - \frac{mt^2}{2} - O_v & (n_0 < 1) \end{cases} \tag{3}$$

Considering the equation (2), n_0 can be less than 1 when the graphs change slowly (m is small) and the overhead (O_v) is high under the given number of iterations (t). In such case, we may apply the transformation only at the beginning in order to get the maximum gain ($G_{max} = G(1)$). On the contrary, if graph changes rapidly (m is large) then we can apply locality transformations multiple times ($n_0 \geq 1$) depending on their overhead. The smaller overhead the more transformation we can apply in order to get the maximum gain.

In practice, graphs often change periodically, not every iteration, trading off precision for execution time. Even in such cases, we can directly apply our model, assuming one iteration in our model corresponds to a set of iterations that do not change graphs except for the last iteration. Another consideration in practice is picking number of transformations based on n_0 that is generated by the cost model. Since our cost model produces a real number for n_0, we may pick a closest integer number for actual uses. However, if n_0 falls in around the midpoint of two integers, the system may pick either. Experimental result showed both choices produce nearly same performance. Our cost model can be also applied to non-adaptive computations. Using the equation 1 and setting the graph change rate (m) to zero, you may directly get an expected performance gain. Since the transformation will be applied once only at the beginning, the gain function will become much simpler. Based on the calculated gain, the system can decide which locality transformation should be used.

To experimentally verify our cost model, we ran a program MOLDYN with a input graph MOL1 on both 1 processor and 4 processors. The input graph randomly swaps 20% of edges every 20 iterations, and the application iterates 240 time steps. Results are shown in the Figure 24. The first graph shows the measured performance gain on 1 processor and the second graph shows the gain on 4 processors, parallelizing with LOCALWRITE. The y-axis represents the percentage of performance gain over original execution time. The x-axis represents the number of transformations applied throughout the 240 time steps. Different curves correspond to the measured performance gains for different locality transformations, varying numbers of transformation applied. The vertical bars represent the numbers of transformation chosen by our cost model. The percentage numbers under the vertical bars represent the calculated performance gain by our cost model. Results show our cost model selects quite precise numbers to decide how often locality transformations should be applied to get the maximum performance gain. Our cost model also predicts relatively precise performance gains.

8 Related Work

8.1 Irregular computations

In scientific applications, significant amount of computations are reductions (*e.g.*, SUM, MAX, *etc.*). Suppose the numerical errors caused by changing computation orders are negligible, reductions can be efficiently parallelized. The techniques of identifying and parallelizing reductions are well established in several researches [23, 34, 41, 48]. In irregular applications, reductions are also core parts of computations, taking up dominant portions of execution times. Researches in irregular computations have focused on general techniques supporting irregular computation including irregular reductions. Researches are categorized in two classes depending on their target systems; message-passing systems and software DSM systems.

8.1.1 Message-passing systems

Efficient run-time primitives are first developed in CHAOS library [13]. The CHAOS library provides primitives that efficiently move data between processors and manage copies of remote data. Using CHAOS primitives, inspector/executor paradigm is applied to parallelize irregular computations. CHAOS primitives are inserted as inspectors before irregular computations begin. The inspectors gather non-local data that will be used later in irregular computations, storing them in local buffer. Then, irregular computations are executed, accessing only locally owned data and locally stored remote data. After the execution, other CHAOS primitives are inserted to scatter the values of non-local data to appropriate owners. For adaptive computations where access patterns change during the whole execution, inspectors need to rerun whenever the access patterns change to get new communication schedules. Even for adaptive applications, the CHAOS system has been shown to scale well on Intel iPSC/860 machines [25]. Furthermore, researchers investigate automatically inserting CHAOS primitives using compiler analysis [25, 20, 21]. Based on the compiler analysis of producer/consumer of data, communication schedules are generated between processors. Appropriate primitives are then inserted in proper places, aggregating necessary communications.

Another implementation of run-time library is found in the PILAR system, where non-local data accesses are represented as *intervals* instead of individual elements [32]. Using more compact representation, PILAR reduces the amount of communications and improves performance. PILAR also unifies communication schedules for regular computations and irregular computations, exploiting a possible communication aggregation across regular and irregular computations [7].

8.1.2 Software DSM systems

Irregular computations are as easily executed as regular computations on software DSM systems. However, efficiently executing irregular computations requires similar techniques used in message-passing systems. With the knowledge of data access patterns, shared-memory compilers recognize and prefetch non-local data before actual irregular computations use them [8, 35]. Furthermore, communication for non-local data is aggregated to reduce overheads. The performance of irregular applications on software DSM systems shows nearly the same as message-passing systems, but with simpler compiler support.

Several researches also investigated combining software DSMs and explicit message passing aided by compilers in order to enhance general performance of software DSM systems [8, 16, 11]. This hybrid approach improves the performance of regular computations as well as irregular computations, making software DSMs more attractive on message-passing machines. Without compiler support, software DSMs also exploit iterative nature of scientific applications by prefetching the same non-local data used in the previous iteration to eliminate access misses [30, 49].

8.2 Locality optimizations for irregular codes

Data locality has been recognized as a significant performance issue for modern processor architectures. Most researchers have focused on loop transformations on dense-matrix codes [37, 47, 43, 50], though recent work has focused on data layout transformations for both arrays [26, 42] and pointer-based data structures [10].

Recent researches show several different approaches to optimize the locality of codes. One approach is improving data locality with hardware support. Using smart memory controllers, irregular accesses to memory are merged and passed to caches. Another approach is rearranging data structures in application level, improving data locality from the source of locality problems. Data locality directly affects the performance, since data accesses with poor locality pollute cache blocks and waste memory bus bandwidth. In the following sections, we will look at different approaches that improve data locality.

8.2.1 Locality optimization with hardware support

Using smart memory systems, irregular memory accesses can be avoided. In such systems, irregular data are remapped to another contiguous space, making data accesses rather regular.

Carter *et al.* proposed a memory controller that can remap scattered data into contiguous space in *shadow memory*, unused physical memory address space [6]. When programs access data in shadow memory, memory controller gathers data from actual locations using a separate address table in memory controller. Gathered data are then passed in compact forms through memory bus. Effectively using their memory controller requires compiler or user assistance. Compilers or users should insert remapping system calls for the data with scattered accesses so that the data are remapped to shadow pages with contiguous placement. Computation parts also need modifications to use different data remapped in shadow pages. Using hardware simulation, they show improvement in sparse matrix applications.

Luk and Mowry provided a hardware mechanism that automatically fetches reordered data, using a *forward bit* that is added to each address [36]. If the forward bit is set for an address, the actual value is fetched from another address pointed by the value in the current address. Using this mechanism, data can be freely reordered without changing the rest of the codes. Memory forwarding mechanism will make data reordering easy to use.

Both hardware supports, however, introduce additional level of indirection in hardware, which means we need to pay extra cost whenever we access irregular data from memory. In comparison, source level modifications do not need extra cost, once modifications are done with more cost and effort. In the following section we will take a look at other researches that directly modify data structures in source level.

8.2.2 Locality optimization for array data

Locality optimizations for irregular applications mainly focus on rearranging data layout for improved locality. Computation reordering is also used along with data reordering to further improve performance.

Al-Furaih and Ranka studied using METIS and breadth-first-search (BFS) to reorder data in irregular computations and improve locality [1]. BFS and its variances have been successfully used in sparse matrix computations to reorder the sparse matrices [12, 33]. They extended the applicability of BFS and graph partitioning algorithms to irregular graph orderings, but did not combine computation reordering or compare against other types of techniques.

Ding and Kennedy explored applying dynamic copying (packing) of data elements based on loop traversal order, and show major improvements in performance [14]. They were able to automate most of their transformations in a compiler using user provided information. Their technique has low overhead, but the quality of rearrangement can be further improved, which motivates us to develop high quality but low overhead algorithms. Ding and Kennedy also discussed reorganization of single arrays into multi-dimensional arrays depending on how closely they are accessed in computation, and found that their technique improves the performance of irregular applications [15]. They also showed their technique does not hurt the performance of regular applications if back-end compilers are smart enough to extract the existing instruction level parallelism in transformed codes.

Mellor-Crummey *et al.* used a geometric ordering algorithm based on space-filling curves to map multidimensional data to memory [38]. They showed significant improvement for large randomly generated data sets. However, space-filling curves cannot guarantee evenly balanced partition when data is unevenly distributed, which may cause significant performance degradation in parallel execution. They also used computation blocking by restructuring computations with multiple loop nests such as tiling in regular codes. A block of computations is finished before the next block of computations begins. They could improve performance by computation blocking, but the performance was somewhat less than that of sorting computation according to space-filling curves. The best performance comes when data is ordered with space-filling curves and computation is ordered according to the data ordering. Since irregular codes usually deal sparse data structures, there are not many benefits for computation blocking. A simple lexicographical sort according to the data ordering will do better for computation ordering.

Mitchell *et al.* used a bucket sorting to reorder irregular computations [39]. They improved the performance of two NAS applications (CG, and IS) and a medical simulation of heart. Their technique, however, works only when a single irregular access pattern exists for a given data array. In comparison, we investigate more complex cases where two or more irregular access patterns exist. In applications we use, two access patterns typically exist, letting our algorithms recognize them as edges between two data elements accessed in the same iteration.

8.2.3 Locality optimization for heap objects

In modern programming languages, data objects are dynamically allocated from heap, a free memory space. Such objects are often linked with pointers if they are logically related. The placements of the objects in memory, however, do not reflect their logical affinity. Thus, accessing logically related objects could show poor memory access behavior. Researchers try to rearrange objects by putting them together if they have temporal and spatial locality.

Calder *et al.* proposed *cache conscious data placement* [5]. They use profiled information to find temporal locality between data, then rearrange data placement for improved locality. For static data, executable files are directly modified to rearrange the static data. For dynamically allocated data, a special heap allocation routine is used to allocate heap data in proper places guided by profiled information. They reduced miss rates for SPEC integer programs and two SPEC floating point programs. However, their data placement technique does not handle rearranging data elements within an array or a heap object.

Chilimbi *et al.* developed a technique to rearrange pointer-based data by clustering data elements into a cache block [10]. They focused on tree data structures including linked lists. Putting a parent node and its child nodes near in memory, they improved tree maintenance/search applications. Using `ccmorph`, a run-time tree optimization routine, a tree structure is converted to an optimized tree structure for locality. They also present an alternative heap allocation routine, `ccmalloc`, that allocates a memory space close to a user specified heap location. The new allocation routine constructs data structures with improved locality. Chilimbi *et al.* also suggest internally reorganizing the fields of data structures with assistant of profiling [9]. They distinguish frequently used fields from other fields, split them out, and pack them into a cache block. When structures are too large to fit in cache block, they reorder the fields inside the structures according to temporal affinity. Using field reorganization, they improved applications programmed in object-oriented languages.

8.3 Contribution of our research

Our research is one of efforts to improve irregular applications by directly transforming applications in source level. We currently focus on the applications that use array data, but will extend to the applications that use pointer-based data. For locality transformation, we adapt partitioning algorithms that are originally developed for other purposes, such as load balancing and interprocessor communication reduction in parallel programs, VLSI design, and database storage [29, 40, 46]. We adapt RCB [4] and METIS [28] for our partitioning algorithms. We also developed a new partitioning algorithm (GPART) based on hierarchical graph clustering. GPART has low overhead but produces nearly the same quality ordering as other precise algorithms. Compared to existing researches on locality optimizations for irregular codes, we propose a suite of algorithms that differ in precision and overhead.

Low overhead algorithms are vital for adaptive computations, where transformations are repeatedly applied whenever profitable. Unlike other inspectors in irregular computations, locality transformations are not necessary after access pattern changes, since the transformations only affect the performance, not the legality of codes. To maximize performance, we use a cost model to guide how often locality transformations to rerun and which transformation to use. We test effects on both randomized graphs and meshes from real applications for a number of computation kernels.

9 Conclusions

In this paper, we present some techniques to improve the locality of irregular computations using a combination of compile and run-time techniques. We establish that partitioning algorithms can yield data layouts better than consecutive packing for both sequential and parallel codes, and develop a new algorithm with lower overheads which approaches the quality of more precise algorithms. We experimentally evaluate how to choose parameters such as the number of partitions and graph clustering factor. We also investigate how locality optimizations may be used for adaptive codes, using a cost model to select how often to reorder data and which optimization to be applied.

As processors speed up relative to memory systems, using graph partitioning to improve data layout should increase in importance, since processing costs go down while benefits increase. For very large graphs, we should also obtain benefits by reducing TLB misses and paging in the virtual memory system. By improving compiler support for irregular codes, we are contributing to our long-term goal: making it easier for scientists and engineers to take advantage of the benefits of high-performance computing.

10 Acknowledgments

We are very grateful to Prof. George Karypis at the University of Minnesota for providing the application meshes in our experiments.

References

[1] I. Al-Furaih and S. Ranka. Memory hierarchy management for iterative graph structures. In *Proceedings of the 12th International Parallel Processing Symposium*, Orlando, FL, Apr. 1998.

[2] C. Alpert and A. Kahng. Recent directions in netlist partitioning. *Integration, the VLSI Journal*, 19(1–2):1–81, 1995.

[3] M. Berger and S. Bokhari. A partitioning strategy for pdes across multiprocessors. In *Proceedings of the 1985 International Conference on Parallel Processing*, Aug. 1985.

[4] M. Berger and S. Bokhari. A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Transactions on Computers*, 37(12):570–580, 1987.

[5] B. Calder, C. Krintz, S. John, and T. Austin. Cache-conscious data placement. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, San Jose, CA, Oct. 1998.

[6] J. Carter, W. Hsieh, L. Stoller, M. Swanson, L. Zhang, E. Brunvand, A. Davis, C. Kuo, R. Kuramkote, M. Parker, L. Schaelicke, and T. Tateyama. Impulse: Building a smarter memory controller. In *Proceedings of the 5th IEEE International Symposium on High Performance Computer Architecture*, Jan. 1999.

[7] D. Chakrabarti, N. Shenoy, A. Choudhary, and P. Banerjee. An efficient uniform run-time scheme for mixed regular-irregular applications. In *Proceedings of the 1998 ACM International Conference on Supercomputing*, Melbourne, Australia, July 1998.

[8] S. Chandra and J. Larus. Optimizing communication in HPF programs for fine-grain distributed shared memory.

In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Las Vegas, NV, June 1997.

[9] T. Chilimbi, B. Davidson, and J. Larus. Cache-conscious structure definition. In *Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation*, Atlanta, GA, May 1999.

[10] T. Chilimbi, M. Hill, and J. Larus. Cache-conscious structure layout. In *Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation*, Atlanta, GA, May 1999.

[11] A. Cox, S. Dwarkadas, H. Lu, and W. Zwaenepoel. Evaluating the performance of software distributed shared memory as a target for parallelizing compilers. In *Proceedings of the 11th International Parallel Processing Symposium*, Geneva, Switzerland, Apr. 1997.

[12] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 24th National Conference of the ACM, ACM Publication P-69*, Association for Computing Machinery, NY, 1969.

[13] R. Das, M. Uysal, J. Saltz, and Y.-S. Hwang. Communication optimizations for irregular scientific computations on distributed memory architectures. *Journal of Parallel and Distributed Computing*, 22(3):462–479, Sept. 1994.

[14] C. Ding and K. Kennedy. Improving cache performance of dynamic applications with computation and data layout transformations. In *Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation*, Atlanta, GA, May 1999.

[15] C. Ding and K. Kennedy. Inter-array data regrouping. In *Proceedings of the Twelveth Workshop on Languages and Compilers for Parallel Computing*, San Diego, Aug. 1999.

[16] S. Dwarkadas, A. Cox, and W. Zwaenepoel. An integrated compile-time/run-time software distributed shared memory system. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, Boston, MA, Oct. 1996.

[17] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: An analytical representation of cache misses. In *Proceedings of the 1997 ACM International Conference on Supercomputing*, Vienna, Austria, July 1997.

[18] M. Hall, S. Amarasinghe, B. Murphy, S. Liao, and M. Lam. Detecting coarse-grain parallelism using an interprocedural parallelizing compiler. In *Proceedings of Supercomputing '95*, San Diego, CA, Dec. 1995.

[19] H. Han and C.-W. Tseng. Improving compiler and run-time support for adaptive irregular codes. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, Paris, France, Oct. 1998.

[20] R. v. Hanxleden. Handling irregular problems with Fortran D — A preliminary report. In *Proceedings of the Fourth Workshop on Compilers for Parallel Computers*, Delft, The Netherlands, Dec. 1993.

[21] R. v. Hanxleden and K. Kennedy. Give-N-Take — A balanced code placement framework. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, Orlando, FL, June 1994.

[22] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, Aug. 1992.

[23] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Preliminary experiences with the Fortran D compiler. In *Proceedings of Supercomputing '93*, Portland, OR, Nov. 1993.

[24] Y. Hu, S. L. Johnsson, and S.-H. Teng. High Performance Fortran for highly irregular problems. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Las Vegas, NV, June 1997.

[25] Y.-S. Hwang, B. Moon, S. Sharma, R. Ponnusamy, R. Das, and J. Saltz. Runtime and language support for compiling adaptive irregular programs on distributed memory machines. *Software—Practice and Experience*, 25(6):597–621, June 1995.

[26] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. Improving locality using loop and data transformations in an integrated framework. In *Proceedings of the 31th IEEE/ACM International Symposium on Microarchitecture*, Dallas, TX, Nov. 1998.

[27] G. Karypis and V. Kumar. Analysis of multilevel graph partitioning. In *Proceedings of Supercomputing '95*, San Diego, CA, Nov. 1995.

[28] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. In *Proceedings of the 24th International Conference on Parallel Processing*, Oconomowoc, WI, Aug. 1995.

[29] G. Karypis and V. Kumar. Multi-level k-way hypergraph partitioning. In *Proceedings of SC'98*, Orlando, FL, Nov. 1998.

[30] P. Keleher and C.-W. Tseng. Enhancing software DSM for compiler-parallelized applications. In *Proceedings of the 11th International Parallel Processing Symposium*, Geneva, Switzerland, Apr. 1997.

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

[32] A. Lain and P. Banerjee. Exploiting spatial regularity in irregular iterative applications. In *Proceedings of the 9th International Parallel Processing Symposium*, Santa Barbara, CA, Apr. 1995.

[33] W. Liu and A. Sherman. Comparative analysis of the cuthill-mckee and the reverse cuthill-mckee ordering algorithms for sparse matrices. *SIAM Journal on Numerical Analysis*, 13(2):198–213, Apr. 1976.

[34] B. Lu and J. Mellor-Crummey. Compiler optimization of implicit reductions for distributed memory multiprocessors. In *Proceedings of the 12th International Parallel Processing Symposium*, Orlando, FL, Apr. 1998.

[35] H. Lu, A. Cox, S. Dwarkadas, R. Rajamony, and W. Zwaenepoel. Compiler and software distributed shared memory support for irregular applications. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Las Vegas, NV, June 1997.

[36] C. Luk and T. Mowry. Memory forwarding: Enabling aggressive layout optimizations by guaranteeing the safety of data relocation. In *Proceedings of the 26th International Symposium on Computer Architecture*, Atlanta, GA, May 1999.

[37] K. S. McKinley, S. Carr, and C.-W. Tseng. Improving

data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.

[38] J. Mellor-Crummey, D. Whalley, and K. Kennedy. Improving memory hierarchy performance for irregular applications. In *Proceedings of the 1999 ACM International Conference on Supercomputing*, Rhodes, Greece, June 1999.

[39] N. Mitchell, L. Carter, and J. Ferrante. Localizing non-affine array references. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, Newport Beach , LA, Oct. 1999.

[40] C. Ou et al. Fast and parallel mapping algorithms for irregular and adaptive problems. In *Journal of Supercomputing*, 1994.

[41] W. Pottenger. The role of associativity and commutativity in the detection and transformation of loop level parallelism. In *Proceedings of the 1998 ACM International Conference on Supercomputing*, Melbourne, Australia, July 1998.

[42] G. Rivera and C.-W. Tseng. Data transformations for eliminating conflict misses. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, Montreal, Canada, June 1998.

[43] V. Sarkar. Automatic selection of higher order transformations in the IBM XL Fortran compilers. *IBM Journal of Research and Development*, 41(3):233–264, May 1997.

[44] S. Shekhar and D.-R. Liu. Partitioning similarity graphs: A framework for declustering problems. *Information Systems Journal*, 21(4), 1996.

[45] H. Simon. Partitioning of unstructured mesh problems for parallel processing. In *Proceedings of the Conference on Parallel Methods on Large Scale Structural Analysis and Physics Applications*. Permagon Press, 1991.

[46] J. P. Singh, C. Holt, T. Totsuka, A. Gupta, and J. Hennessy. Load balancing and data locality in adaptive hierarchical n-body methods: Barnes-hut, fast multipole, and radiosity. *Journal of Parallel and Distributed Computing*, June 1995.

[47] Y. Song and Z. Li. New tiling techniques to improve cache temporal locality. In *Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation*, Atlanta, GA, May 1999.

[48] T. Suganuma, H. Komatsu, and T. Nakatani. Detection and global optimization of reductions operations for distributed parallel machines. In *Proceedings of the 1996 ACM International Conference on Supercomputing*, Philadelphia, PA, May 1996.

[49] G. Viswanathan and J. Larus. Compiler-directed shared-memory communication for iterative parallel computations. In *Proceedings of Supercomputing '96*, Pittsburgh, PA, Nov. 1996.

[50] M. E. Wolf and M. Lam. A data locality optimizing algorithm. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, Canada, June 1991.