Locality Optimizations 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

Irregular scientific codes experience poor cache performance due to their memory access patterns. We examine several data and computation locality transformations including GPART, a new technique based on hierarchical clustering. GPART constructs quality partitions quickly by clustering multiple neighboring nodes in a few passes, with priority on nodes with high degree. Overhead is kept low by considering only edges between partitions. We develop compiler analyses and transformations in SUIF to automatically apply locality transformations, and propose user annotations to locate coordinate information needed by geometric partitioning algorithms. We experimentally evaluate locality optimizations for both static and adaptive codes, where connection patterns dynamically change at intervals during program execution. We derive a simple cost model to guide locality optimizations when access patterns change. Experiments on several irregular scientific codes show locality optimization techniques are effective improving performance by 30–50% on average. In particular, GPART closely matches the performance of more sophisticated partitioning algorithms, with one third of the overhead. With dense data sets with higher edge/node ratios, we find computation reordering becomes more important; using space-filling curves to sort computation (edges) can yield improvements over lexicographic sorting.

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 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. In the adaptive

Submitted to ICS'01. This paper combines earlier workshop papers in LCR'00 and LCPC'00, and adds new results evaluating space-filling curves, impact of dense data inputs, and description of a complete compiler implementation.

```
// Irregular           // Adaptive irregular

do t = 1, time         do t = 1, time
  do i = 1, M            if (change) idx[] = ...
    ... = x[idx[i]]      do i = 1, M
                           ... = x[idx[i]]
```

Figure 1 Regular and Irregular Applications

irregular code, the irregular accesses to x change as the program proceeds. While the program iterates the outer loop, the condition variable $change$ will become true, then the index array idx will have different values, changing the access pattern to x in the inner loop. Changes in access patterns make locality optimizations more difficult. Regular codes benefit because compilers can analyze data access patterns, using estimates of cache performance to guide loop and data transformations to improve locality. 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 transformations on irregular computations [5, 1, 7, 14, 15]. Results have been promising, but a number of issues have not been examined. Our paper makes the following contributions:

- Design a compiler/run-time system for either automatically applying graph-based locality optimizations, or using programmer annotations to guide coordinate-based locality optimizations. Our compilation system, implemented in SUIF, works for both static and adaptive irregular codes.
- Develop GPART, a new locality transformation based on hierarchical clustering which nearly matches the performance of more sophisticated partitioning algorithms, but with a small fraction of the overhead.
- Experimentally evaluate the effectiveness of several locality optimizations for a range of input data and programs on both sequential and parallel architectures.
- Devise a heuristic for applying locality optimizations to adaptive irregular codes. We find a simple cost model may be used to accurately predict a desirable transformation frequency when the rate of change in memory accesses is known. An on-the-fly algorithm can apply the cost model by measuring the per-iteration execution time before and after optimizations.

The remainder of the paper begins with a discussion of our optimization framework and locality optimization algorithms. We experimentally evaluate the impact of locality optimizations, and provide both static and dynamic methods to apply locality optimizations based on a simple cost model. Finally, we conclude with a discussion of related work.

2 Locality Optimizations

2.1 Data & computation transformations

Irregular applications frequently suffer from high cache miss rates due to poor memory access behavior. However, program transformations may exploit dormant locality in the codes. The main idea behind these locality optimizations is to change computation order and/or data layout at runtime, so that irregular codes can access data with more temporal and

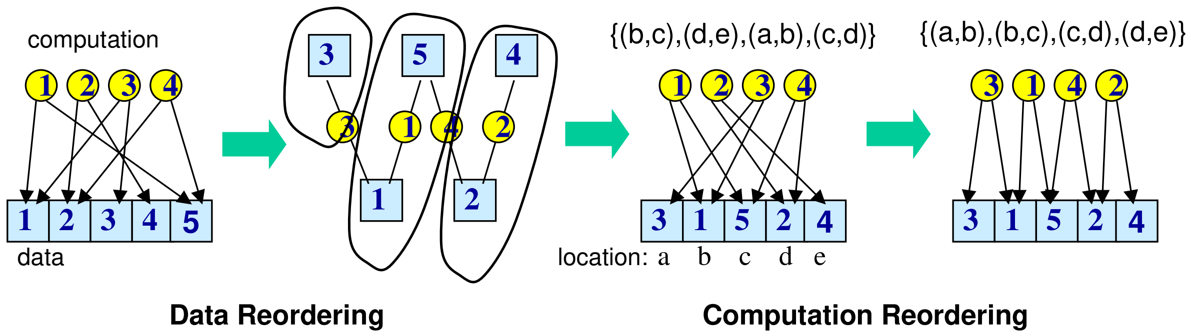

Figure 2 Graph Partitioning & Lexicographic Sort

spatial locality. Fortunately, irregular scientific computations are typically composed of loops performing *reductions* such as SUM and MAX, so loop iterations can be safely reordered [5, 7]. Data layouts can be safely modified as well, as long as all references to the data element are updated. Data layouts can be transformed directly by inserting library routines that rearrange elements in data arrays, such as the host array x in Figure 1. Such updates are straightforward unless pointers are used. Computation is reordered *indirectly* by rearranging the list of edges, such as the index array *idx* in Figure 1. By rearranging elements in the index arrays, the program executes the computation in a different order.

2.2 Framework and application structure

A key decision is when computation and data reordering should be applied. For locality optimizations, we believe the number of distinct irregular accesses made by each loop iteration can be used to choose the proper optimization algorithm. Codes may access either a single or multiple distinct elements of the array x on each loop iteration. In the simplest case, each iteration makes only one irregular access to each array. The NAS integer sort (IS) and sparse matrix vector multiplication found in conjugate gradient (CG) fall into this category. Locality can be optimally improved by sorting computations (iterations) in memory order of array x [5, 7, 15].

More often, scientific codes access two or more distinct irregular references on each loop iteration. Such codes arise in PDE solvers traversing irregular meshes or N-body simulations, when calculations are made for pairs of interacting particles. An example is shown in Figure 2. Circles represent computations (loop iterations), squares represent data (array elements), and arrows represent data accesses. Initially memory accesses are irregular, but either computation or data may be reordered to improve temporal and spatial locality. Notice that since each iteration accesses two array elements, computations can be viewed as edges connecting data nodes, resulting in a graph. Locality optimizations can then be mapped to a graph partitioning problem. Partitioning the graph and putting nodes in a partition close in memory can then improve spatial and temporal locality. Applying lexicographic sorting after partitioning captures even more locality. Finding optimal graph partitions is NP-hard, thus several optimization heuristics have been proposed [5, 1, 7]. When each iteration accesses more than two array elements, a graph can be formed by completely connecting the elements and the same graph partitioning techniques can be used.

2.3 Data reordering algorithms

In this section, we describe a number of locality optimization algorithms which have been proposed for irregular codes.

Consecutive Packing (CPACK) 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 [7]. CPACK can be applied with very low overhead, since it just requires one pass through the loop marking the order in which data is accessed. However, CPACK does not explicitly take into account the underlying graph or geometric structure of the data.

Reverse Cuthill-McKee (RCM) Reverse Cuthill-McKee (RCM) methods have been successfully used in sparse matrix reorganization [4]. RCM simply uses reverse breadth-first search (BFS) to reorder data in irregular computations, by viewing iterations as graph edges. Performance is improved with low overhead [5, 1]. Both CPACK and RCM are *traversal* methods which reorder based on a single pass over the computation.

Space-filling curve (MORTON) Space-filling curves (*e.g.*, Morton) are continuous, non-smooth curves that pass through every point in a finite k-dimensional space [11, 14]. Because interactions tend to be local, arranging data using space-filling curves reduces the distance (in memory) between two geometrically close points in space, yielding better locality [14]. When geometric coordinate information is available, sorting data and computation using space-filling curves is the locality optimization of choice, since it achieves good locality with very low overhead simply using a multidimensional bucket sort. However, geometric coordinate information is not necessarily available, and will probably require user intervention to identify.

Recursive Coordinate Bisection (RCB) When data is unevenly distributed, an alternative to space-filling curves is recursive coordinate bisection (RCB), which recursively splits each dimension in two by finding the median of data coordinates in that dimension [2]. Previously used to reduce interprocessor communication, we show RCB can also help improve locality. After partitioning, data items are stored consecutively within each partition. Subpartitions are also arranged consecutively, creating a recursive hierarchical layout similar to Z-ORDERING, a space-filling curve for dense array layouts. RCB has higher overhead, but is likely to work well for unevenly distributed data. Figure 3 shows an example of using RCB and MORTON.

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. Even if it is available, user annotations may be needed for compilers to understand the association between coordinates and data nodes. In comparison, graph partitioning algorithms can be applied automatically based on the loop data access pattern. Partitioning algorithms were first developed for load balancing in parallel codes. More recently, people have employed multi-level graph partitioning algorithms encapsulated in library packages such as METIS [13] achieve good partitions with low overhead. These algorithms compute a succession of coarsened graphs (with fewer nodes) which approximate the original graph, compute a partition on the coarsened graph, and then expand back to the original graph. Figure 4 provides an example of multi-level graph partitioning applied to a 2D domain.

2.4 Computation reordering algorithms

Independent of data reordering, we can also reorder loop iterations (computations). If each iteration performs only one irregular access, sorting the loop iterations by data accessed results in optimal locality without data reordering.

Lexicographical sort (LEX-COMP) For computations with two or more irregular accesses, lexicographically sorting the loop iterations will improve temporal locality [7]. Sorting is so effective that data meshes provided by applications

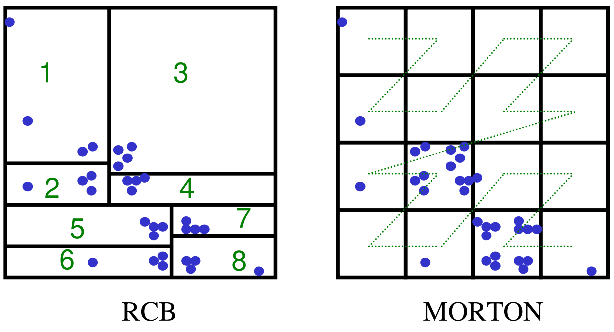

Figure 3 Geometric Coordinate Partitioning: RCB & MORTON

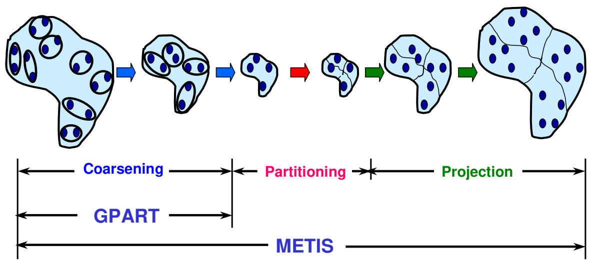

Figure 4 Graph Partitioning: METIS & GPART

writers are frequently presorted, eliminating the need to apply sorting at run time. Thus in our experiments, input data is presorted using lexicographical sort.

Bucket sort Bucket sorting with cache-sized buckets can also be used, producing similar improvements with lower overhead, since order in a bucket may be arbitrary. Bucket sorting has been shown to work well for codes with only one irregular access per loop iteration [15]. It can also be extended for loops with multiple irregular accesses, by viewing each reference as an additional dimension and buckets as tiles, then lexicographically ordering the buckets.

Space-filling curve sort (SFC-COMP) We can also use more sophisticated computation reordering algorithms by applying bucket sort, then arranging buckets in space-filling curve order [14]. This provides better locality, since lexicographic order jumps from the end of a row to the beginning of the next row. For sparse data sets, space-filling curve order may show similar result to lexicographical order, since the most neighboring buckets may be empty. For dense data sets, however, space-filling curve order can exploit more locality. Space-filling curve sort has higher overhead than lexicographical bucket sort, so it should be used in situations where high overhead can be amortized over many iterations.

2.5 Low-overhead graph partitioning (GPART)

Previous graph partitioning algorithms were designed for load-balancing in parallel computations, VLSI circuits, and other high cost applications. As a result, greater processing time is acceptable. Since our goal is to improve cache performance at run-time, we desire partitioning algorithms with lower overhead. We thus developed a hierarchical graph clustering algorithm (GPART) that has low overhead and produces quality ordering for cache locality.

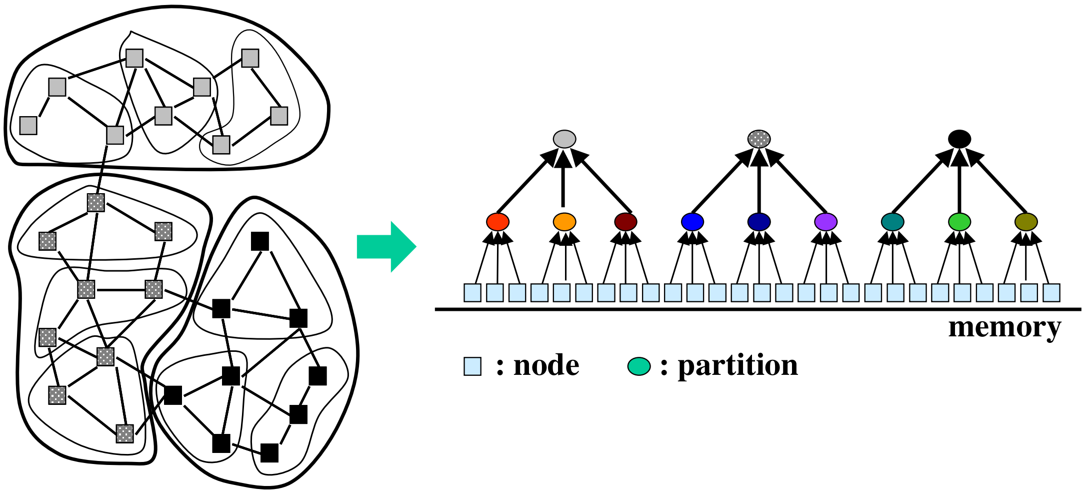

Figure 5 Graph Partitioning Using Hierarchical Clustering (GPART)

```
1:  sort nodes by descending degree
    pLimit = cacheLineSize
    while (pLimit <= maxLimit) {
2:    for each node N (in sorted order) {
        P = partition containing N
        if (size(P) < pLimit) {
          for each neighbor M of N in adj. list {
            // pick partner in random order
            Q = partition containing M
            if (P == Q)
              mark M as merged neighbor
            else if (size(P) + size(Q) <= pLimit) {
              merge Q into P
              mark M as merged neighbor
              if (size(P) == pLimit)
                break (exit for loop)
      } } } }
3:    eliminate merged neighbors in adjacency list
4:    pLimit = pLimit*growFac // new partition size
    }
5:  recursively lay out nodes according to hierarchy
```

Figure 6 GPART Algorithm

GPART is derived from the coarsening phase from METIS, shown in Figure 4, but it has been modified for greater efficiency and better locality. The GPART algorithm performs a number of passes. On each pass, it clusters several adjacent nodes into a single partition. Each partition is then treated as a single node in the next pass, creating a hierarchical structure. The GPART algorithm consists of the following steps:

1) SORT Nodes are sorted once by their vertex degrees (number of incipient edges) from most number of neighbors to least. Sorting improves the likelihood that closely connected nodes are clustered first through multiple passes, improving the quality of the partitions produced.

2) MERGE Visiting nodes in order of vertex degree, randomly choose partition containing neighboring node on adjacency list. If combined size is less than *pLimit* (current partition size limit), merge partitions containing the node and its chosen neighbor. Mark the neighbor as merged. Random selection may result in lower quality partitions than more sophisticated methods, but reduces overhead.

3) FILTER While visiting nodes to cluster, mark neighbors as merged if they have already been merged into the partition of the current node. After visiting all nodes, marked neighbors are eliminated from the adjacency list. Fewer nodes need to be searched on each pass, speeding up future passes.

4) LOOP Repeat STEP 2 and STEP 3, increasing the current limit on partition size (*pLimit*) by factor *growFac*, up to the maximum desired partition size (*maxLimit*). Initial *pLimit* is set to the cache line size, and *maxLimit* is set to L1 cache size divided by data accessed on each loop iteration. If *pLimit* is small, overhead increases as the algorithm iterates many times. If *pLimit* is large, the quality of partition degrades. *growFac* is experimentally set to eight.

5) LAYOUT Finally, arrange nodes according to the resulting hierarchical layout. Nodes in a partition are recursively laid out nearby in memory, maintaining the partition hierarchy shown in Figure 5.

The full GPART algorithm is shown in Figure 6. The partitions produced by GPART are used to improve locality by generating data layouts that put neighboring nodes in nearby memory 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 together in memory, which further improves locality. Since applications visit nodes according to the storage order due to lexicographical sorting, maintaining the hierarchical structure of the clustered graph increases the chances of visiting neighbor nodes that have already been recently accessed. These nodes are more likely to remain in cache, improving performance.

GPART is similar to the coarsening phase in METIS, but fewer clustering passes are required because more nodes are collapsed at once. Because we form clusters in a bottom-up fashion, we capture most of the locality immediately and can stop after a small number of passes. Sorting by degree before merging loops helps to cluster with right neighbors, even though GPART clusters large nodes at once. The quality of the partition produced is slightly less than that produced by METIS, but the overhead is significantly less.

We chose our parameters for GPART experimentally by performing a large number of experiments. We found that stopping after partitions reach the size of L1 cache and using appropriately large coarsening factor reduced the number of passes without significantly degrading the quality of the partitions. Choosing an effective coarsening factor depends on several factors. Merging a few nodes (small collapsing rate) yield quality partitions, but require more passes. Merging many nodes (large collapsing rate) requires fewer passes, but may produce poor quality partitions. Collapsing rates are also dependent on the ratio of edges to nodes (E/N). Since GPART randomly selects neighboring nodes to merge, the quality of ordering is probabilistic. The higher ratio of edges to nodes, the lower probability of choosing the most desirable neighboring nodes. According to our experiments, selecting a collapsing rate that merges most neighboring nodes after the second loop yields the highest quality partitions.

```
obj x[N]              // host array, N elements
int idx1[E]           // index array, E edges
int idx2[E]           // index array, E edges

#pragma coord(loc,x)  // user annotation
real loc[N,2]         // coordinates for elements

idx1[...] = ...       // initialize edge list
idx2[...] = ...

inspect(N,E,sizeof(obj),x,idx1,idx2,loc)
                      // inspector reorders x, idx

do t = 1, time            // time-step loop
   if (change)            // adaptive code
      idx1[] = ...        // modify index array
      idx2[] = ...
      inspect( ... )      // reapply inspector

   do i = 1, M            // computation (executor)
      ... = x[idx1[i]]    // data/edges reordered
      ... = x[idx2[i]]
```

Figure 7 Compiler Insertion of Inspectors

3 Compiler Support for Locality Optimizations

3.1 Inspector-executor approach

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 calls to an *inspector* to process memory access patterns at run-time to identify non-local data needed by each processor [6].

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 inspectors are expensive, but can be also amortized over many time steps. Researchers on data-parallel compilers have discussed compiler techniques for automatically generating inspectors [9, 12]. We use simplified versions of those techniques for locality optimizations.

We also use techniques to identify when memory access patterns change for adaptive computations [8, 12]. When access patterns change such as in adaptive computations, locality inspectors may also 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.

3.2 Compilation algorithm

Our implementation in the SUIF compiler applies locality optimizations in three steps as follows:

1) IDENTIFY The compiler first makes a pass through the program to identify irregular reductions. It looks for array reductions that are accessed using *index arrays* in reduction statements. For each irregular reduction statement, the compiler identifies both the *index array* and the *host array*(the array accessed using index array). Summary information is collected and merged for all statements in a basic block and passed upward along the tree structure of program blocks. The compiler also identifies index arrays used with multiple host arrays.

2) LOCATE The compiler next looks for appropriate locations for inserting inspectors. Performing a top-down walk of the AST, it identifies the highest blocks that contain writes to the index arrays, but do not include corresponding irregular array reductions. These blocks are marked so that inspectors can be inserted following them. Occasionally the host array may be modified between changes to the index array and the array reduction. In this case the compiler must insert inspectors only after the last block that makes change to host array. Otherwise data reordering performed on host array is nullified and may generate incorrect results.

3) INSERT Finally, the compiler actually inserts the call the the inspector. A call to the appropriate library function is inserted, and passed as parameters the host arrays, index array, size of both arrays, and size of host array elements. When the inspector is executed at run-time, it rearranges the contents of the index and host arrays, resulting in the desired data and computation transformations. The inspector may also need to be converted to a conditional call so that the inspector is called only when actual changes happen. As mentioned in earlier section, computation order is indirectly changed by reordering the index arrays.

An example of our compiler output is shown in Figure 7. The compiler identifies host and index arrays from the irregular reduction, finds two places in the program where the index arrays are modified (and thus require inspectors following), and inserts a call to the inspector with all the needed parameters.

Note that our compiler algorithm for inserting inspectors also works well for adaptive codes. Since inspectors are inserted whenever index arrays are changed, it finds both the initialization block that initializes index array at run-time, and the adaptive block that changes index array while program proceeds. The compiler then inserts inspectors at both places. Inside the inspectors, locality transformation will be applied if benefit is expected. The cost model in later section will discuss this in detail.

3.3 Geometric coordinate information

Since all computation/data reordering algorithms are implemented in a library, inspectors can use appropriate algorithms by calling corresponding functions. Some data reordering algorithms such as RCB and MORTON require geometric coordinate information. We find coordinate information is often represented with an array whose elements correspond to coordinates of host array in program context.

To use geometric partitioning algorithms, our compiler needs to determine which arrays contain coordinates information, and pass them as parameters to the inspector. Unfortunately, unlike graph partitioning there many not be any way for the compiler to identify this information. We thus propose a simple user annotation to identify coordinate arrays and the host arrays they refer to. This information can be gather by the compiler in the first IDENTIFY compiler pass. When the inspector is generated, the appropriate coordinate array may then also be passed as a parameter, and used in a geometric partitioning locality optimization. Figure 7 presents an example of the annotation and show how it can be used identify the coordinate information for a host array.

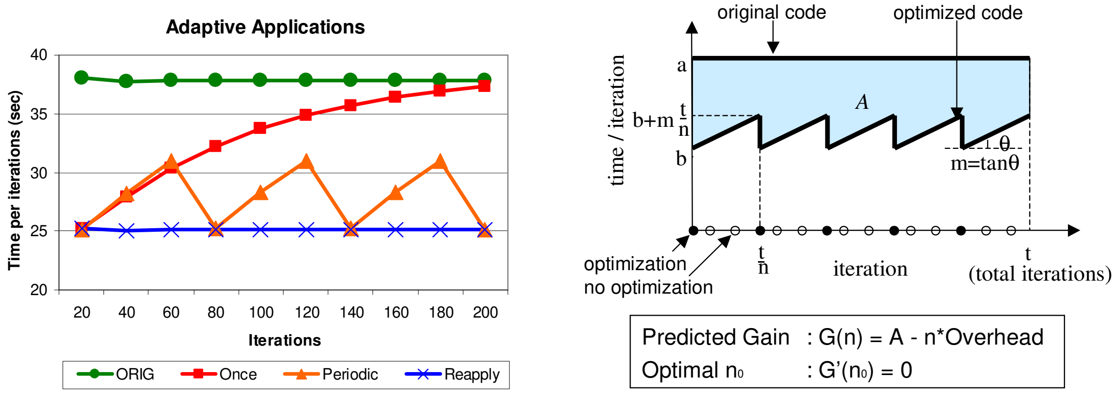

Figure 8 Adaptive Computations : Impact on Performance & Analytical Model

4 Optimizing Adaptive Applications

Another 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. For instance, the example in Figure 1 is adaptive, since condition `change` may be satisfied on some iterations of the time-step loop, modifying elements of the index arrays `idx`. Fortunately, changing data access patterns reduces locality and degrades performance, but does not affect legality. Locality optimizations are not reapplied after every change, but only when it is deemed profitable.

4.1 Impact of adaptation

To evaluate the effect of adaptivity on performance after computation/data reordering, we timed MOLDYN, a molecular dynamic application. Adaptive behavior in MOLDYN is actually dependent on the initial position and velocity of molecules in the input data. Unfortunately molecules move very slowly in the default data set. To mimic more active adaptive codes, we modified MOLDYN by periodically changing the positions of 10% of the molecules randomly every 20 iterations. We realize this may not be realistic, but it allows us to perform experiments in a reasonable amount of time. Future research will need to conduct a more comprehensive study of adaptive behavior in scientific programs.

Results for our modified version of MOLDYN is shown in the first graph of Figure 8. The x-axis marks the passage of time in the computation, while the y-axis measures execution time. The execution time does not include the overhead of optimization to focus on the performance change of the core computation. ORIG is the original program. *Once* represents the program where data and computation are reordered exactly once at the beginning. In comparison, *Periodic* represents the program where the optimization is periodically applied. *Reapply* represents the program where the optimization is applied whenever access pattern changes. It is intended to demonstrate the effectiveness of locality optimizations after adaptive behavior.

Results show that without reapplying locality reordering, performance degrades and eventually matches with the unoptimized program (*Once*). In comparison, reapplying partitioning algorithms after every access pattern change can preserve the performance benefits of locality (*Reapply*), if overhead is excluded. In practice, however, we do need to take into account overhead if adaptive rate is high. By periodically applying reordering, we can maximize the net

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

$$
P_{optimal} = \frac{t}{n_0} \quad = \quad \sqrt{\frac{2O_v}{m}} \qquad\qquad (G'(n_0) = 0) \qquad (2)
$$

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

Figure 9 Calculating Net Gain and Adaptation Frequency

benefit of locality reordering. Results show performance begins to degrade, but recover after locality optimizations are reapplied (*Periodic*). Even after overheads are included, the overall execution time is improved by reapplying locality optimizations. The key question is what frequency of reapplying optimizations results in maximum net benefit. In the next section we attempt to calculate the frequency and benefit of reordering.

4.2 Cost model for optimizations

Based on the observation of performance changes in adaptive application, we devise a simple cost model shown in the second diagram of Figure 8. This cost model is used to find the best optimization rate and predict its benefits. We begin by showing how to calculate optimal period when all parameters such as optimization overhead, optimization benefit, access change frequency, and performance degradation rate are known. Later we show how this method may be adopted to work in practice by collecting and fixing some parameters and gathering the remaining information on the fly through instrumentation.

We consider the performance of two versions of a program, *original code* and *optimized code*. The second diagram of Figure 8 plots execution time per iteration, excluding the overhead. The upper straight line corresponds to the *original code* and the lower saw-tooth line corresponds to the *optimized code*. For the original code, we assume randomly initialized input data, which makes execution times per iteration stay constant after access pattern changes. For the optimized code, locality optimization is performed at the beginning and periodically reapplied. For simplicity, we assume execution times degrades linearly (m) as the nodes change and periodically drop to the lowest (optimized) point after the optimization is reapplied. Execution times per iteration do not include the overhead of the locality reordering, but we will take it into account later in our benefit calculation. The net performance gain ($G(n)$) from periodically applying optimization can be calculated as in Figure 9. Since the area below the line is the total execution time, the net performance gain will the area between two lines (A) minus the total overhead (nO_v). Taking the differential equation of $G(n)$, we can calculate the point where the net gain is maximized. Then we can calculate the optimal period ($P_{optimal}$) and the gain (G_{max}). In practice, data access patterns often change periodically instead of after every iteration, since scientists often accept less precision for better execution times. We can still directly apply our model in such cases, assuming one iteration in our model corresponds to a set of loop iterations where data access patterns change only on the last iteration.

```
do t = 1, n          // time-step loop
  if (change)        // change accesses
    idx1[]=
    access_changed() // note access changes
  cost_model()       // apply cost model

  do i = 1, edges    // main computation
    ...
```

Figure 10 Applying Cost Model Dynamically

4.3 On-the-fly application of cost model

The cost model described in the previous section requires several parameters to calculate the optimal period and net benefit of locality optimizations. In practice, some of the information needed is either expensive to gather (e.g., overhead for each optimization) or nearly impossible to predict ahead of time (e.g., degradation rate of performance). In this section, we show how our cost model can be applied by gathering information on the fly by inserting a few timing routines in the program. Figure 10 shows an example code that uses the on-the-fly cost model. At every iteration, `cost_model()` is called to monitor the execution time of every iteration and calculate an optimal period based on the information monitored so far.

Now we will describe in detail how the on-the-fly cost model works. First, we apply only GPART as locality reordering, since considering all optimization algorithms requires too much overhead. Thus, we only calculate the frequency of reordering, not the expected gain from each algorithm. Since GPART has low overhead but yields high quality ordering, it should be suitable for adaptive computations. Second, performance degradation rate (m) is gathered at run time and adjusted by monitoring performance (execution time) of every iteration. The rate is chosen so that actual elapsed time since the last reordering is the same as calculated elapsed time under the newly chosen linear rate. Third, for the first a few iterations, the cost model accumulates performance information enough to calculate an optimal period. The cost model does not monitor the first iteration to avoid noisy information from cold start. On the second iteration, it applies GPART and monitors the overhead and the optimized performance (b). In the third iteration, it does not apply reordering and monitors the performance degradation. The cost model now has accumulated enough information to calculate the desired period of reapplying locality optimizations. From this point on, the cost model keeps monitoring performance of each iteration, adjusting degradation rate and produces new periods based on accumulated information. When the desired interval is reached, it reapplies GPART.

Since irregular scientific applications may change mesh connections periodically (e.g., every 20 iterations), an optimization for the on-the-fly cost model is detecting periodical access pattern changes. Rounding up the predicted optimization frequency so transformations are performed immediately after the next access pattern change can then fully exploit the benefit of locality reordering. For this purpose, the compiler can insert a call to *access_changed()* as shown in Figure 10. This function call notifies the cost model when access pattern changes. The cost model then decides when to reapply optimizations based on access patterns changes. Our on-the-fly cost model thus works well in practice with limited information. It is flexible since it keeps adjusting model parameters and can easily adapt to new situations.

```
** 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 11 Key Kernels of Applications

Name	# Nodes	# Edges	3d Mesh of
FOIL	144649	1074393	parafoil
AUTO	448695	3314611	GM Saturn
MOL1	131072	1179648	molecules (sml)
MOL2	442368	3981312	molecules (lg)

Table 1 Input Data Sets for Irregular Kernels

5 Experimental Evaluation

5.1 Experimental environment

Our experimental results are obtained on a number of platforms, including a Sun HPC10000 with 400 MHz UltraSparc2 processors, and a DEC multiprocessor with 275 MHz Alpha 21064 processors. Our prototype compiler is based on the Stanford SUIF compiler [17]. It can identify and parallelize irregular reductions using *pthreads*. and has been extended to automatically generate inspectors like Chaos [9] for both sequential and parallel programs. Our experiments predate our compiler implementation, and use mostly hand-optimized codes even though our compiler can now generate the optimizations automatically.

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. Statistics and timings are collected after the initialization section and the first iteration 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. NBF is a kernel abstracted from the GROMOS molecular dynamics code [9]. NBF computes a force between two molecules and applied to velocities of both molecules. MOLDYN is abstracted from the non-bonded force calculation in CHARMM, a key molecular dynamics application used at NIH to model macromolecular systems. The key kernels of these codes are shown in Figure 11.

To test the effects of locality optimizations, we chose a variety of input data meshes. FOIL and AUTO are 3D meshes of a parafoil 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*. Resulting input meshes are rather sparse, compared to the usually used inputs in molecular dynamics. Thus, we also generate denser meshes by changing the *cutoff radius* to 3.5 angstrom. We will

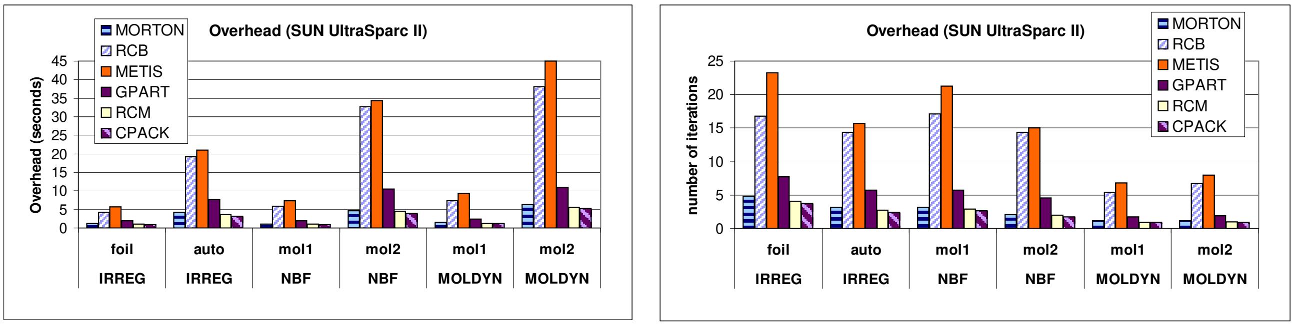

Figure 12 Overhead of Data Reordering (both in seconds and relative to 1 iteration of ORIG)

present the impact on performance of both sparse and denser inputs. All meshes are initially lexicographically sorted, so computation reordering is applied only after data reordering. The sizes of each input data set is shown in Table 1.

5.2 Overhead of optimizations

With these optimization parameters, it is interesting to see the relative processing overhead of each locality optimization. Figure 12 displays the costs of data reordering techniques, both in seconds and measured relative to the execution time of one iteration of the time step loop. 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. RCM has almost same overhead as CPACK. In comparison, partitioning algorithms are quite expensive when used for cache optimizations. RCB and METIS are quite expensive when used for cache optimizations, on the order of 5–45 times higher than CPACK. The overhead of GPART is much less than METIS and RCB, but 3–5 times higher than CPACK. MORTON is slightly higher than CPACK and RCM but less than GPART.

5.3 Impact on sequential performance with sparse inputs

Next we look at sequential execution times on the UltraSparc2. Figure 13 presents normalized execution times calculated relative to the execution time of ORIG. Results are presented for programs executing different numbers of iterations. Since results include the overhead of locality optimizations performed at the beginning of each program, performance improves as the number of iterations increases.

The set of results when the number of iterations is *infinity* represent the quality of each optimization when overhead is ignored. The results show partitioning algorithms perform better than RCM and CPACK in quality of data reordering. MORTON, RCB, METIS, and GPART achieve 30–50% improvements. In comparison, RCM improves less than the partitioning algorithms, but a bit better than CPACK. CPACK shows the least improvement among all algorithms.

When applying locality optimizations in practice, we must consider overheads for each optimization. Even though RCB and METIS generate higher quality orderings, they require hundreds of iterations to be competitive with RCM or CPACK. Meanwhile, MORTON and GPART begins to outperform RCM and CPACK around 40 iterations due to their low overhead. RCB and METIS will be most useful for long running scientific applications where their high overheads can be

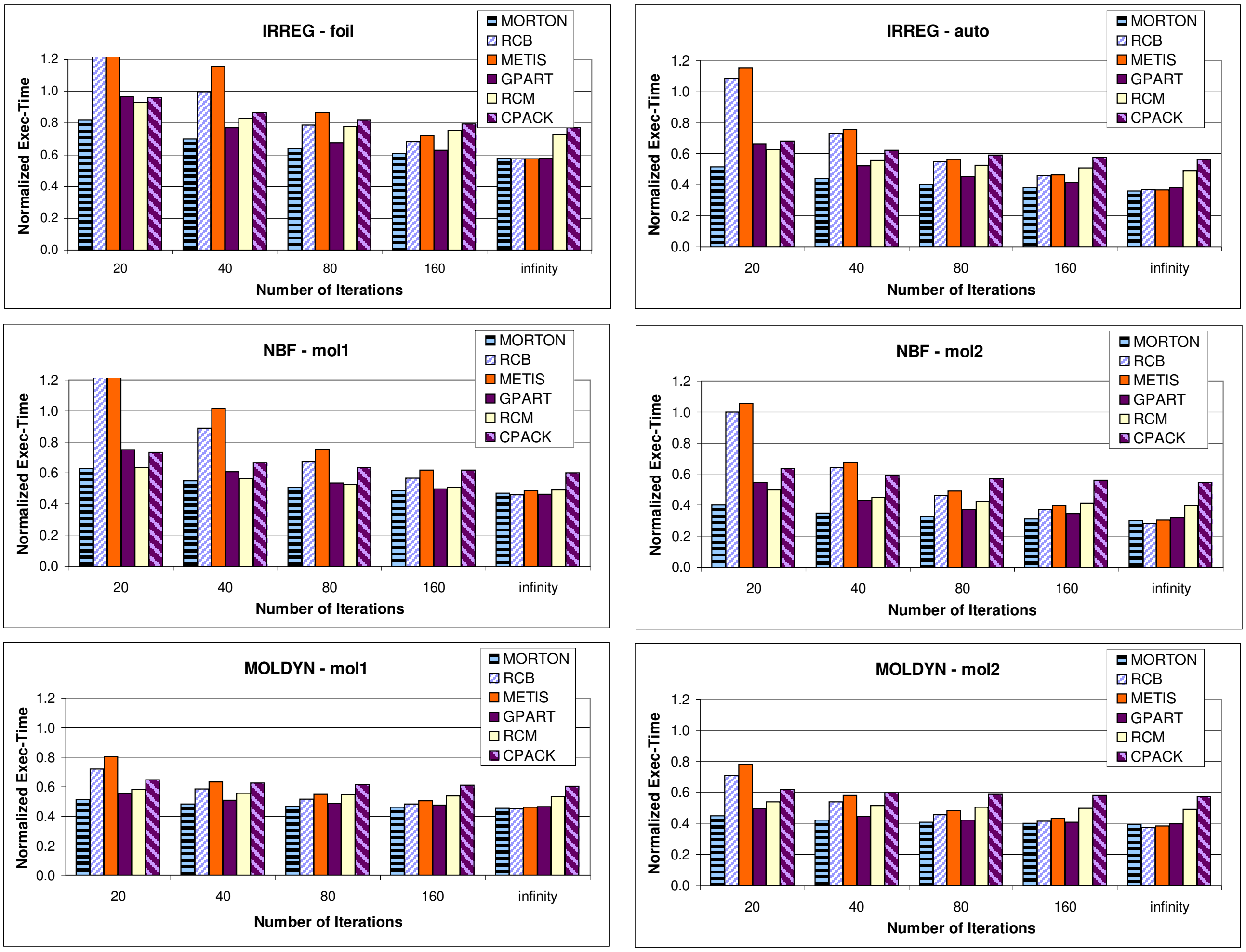

Figure 13 Normalized Sequential Execution Time on HPC 10000 (ORIG = 1, overhead included)

---

can be amortized over many loop iterations. GPART, RCM, and CPACK will be more effective for applications with small numbers of iterations due to their low overhead. MORTON are the most desirable because it produces good performance with low overheads.

Table 2 summarizes our execution results for locality optimizations. Overheads are normalized relative to CPACK. Performance improvements are relative to the original program, averaged over all applications. Improvements at infinity exclude preprocessing overhead, while improvements on average represent the average computed over programs executing from 20 to an infinite number of time steps.

5.4 Impact of dense data sets

The effect of locality optimizations is determined by characteristics of the input data set. In particular, we find data set density (edge/node ratio) affects the relative performance of data reordering algorithms. So far our experiments involved data sets from sparse meshes for PDE solvers, which have low edge/node (E/N) ratios around 10. Many molecular dynamics codes, however, have denser data sets with higher E/N ratios around 100. In this section we compare locality transformations with both sparse and dense data sets ($9 \leq E/N \leq 124$), but fixed number of nodes

Opt. Version	Normalized Overhead (vs CPACK)	Sequential Improvement at ∞	Sequential Improvement avg	Parallel Improvement at ∞	Parallel Improvement avg
MORTON	1.25	57%	53%	58%	36%
RCB	6.44	58%	38%	58%	–45%
METIS	7.64	57%	33%	61%	–60%
GPART	2.23	57%	49%	60%	21%
RCM	1.10	48%	44%	29%	8%
CPACK	1.00	39%	36%	38%	19%

Table 2 Summary of Optimization Results

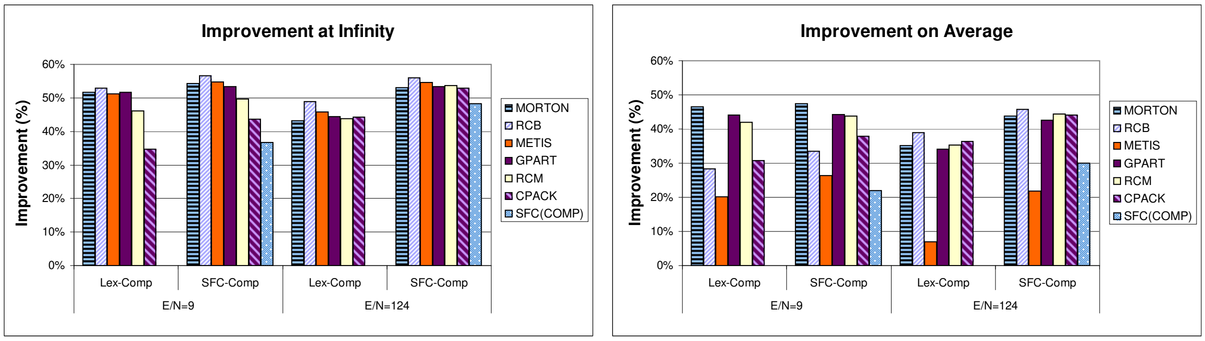

Figure 14 Performance Improvement on Different E/N Ratio Inputs

(N=70K). We have results for individual programs over a range of E/N ratios, but due to lack of space only present averaged results over all applications for two E/N ratios.

Figure 14 presents average performance improvement of all three applications for one sparse (E/N=9) and one dense (E/N=124) data set. The first graph presents improvements ignoring overhead (infinite iterations assumed), the second graph presents improvements when average over versions executing 20, 40, 80, 160, and infinite iterations. For each input, the first set of bars represent the improvement when data reordering is combined by computation reordering using lexicographic sort (LEX-COMP). The second set of bars represent improvement using space-filling curve sort (SFC-COMP) instead. The last bar in each second set represents improvement when SFC-COMP is applied alone without any data reordering.

A number of observations can be made from the results. First, SFC-COMP improves performance over LEX-COMP by about 3% for sparse data sets and 8% for dense data sets, when overhead is not included. SFC-COMP thus provides better quality computation partitions, though the benefit is more noticeable with more edges. Second, dense data sets diminishes differences between data reordering algorithms. Both the quality of the data partitions and the overhead of the partitioning algorithm become less important, because at high E/N ratios the size of the data set is relatively small. Most processing overhead and locality benefits are derived from computation reordering.

Third, dense data sets increases overhead for graph-partitioning locality algorithms. METIS performance is significantly reduced when overheads are considered, as is apparent when results are averaged over a number of iterations. In comparison, GPART maintains its good performance by keeping overhead low. Finally, data reordering is still useful even for dense data sets, as its performance outstrips applying SFC-COMP alone.

5.5 Impact on parallel performance

We also evaluated the effect of locality optimizations on parallel programs by measuring performance for each program on a 8-processor Sun HPC 10000. Results are included in Table 2, and show locality optimizations also improve performance of parallel programs, with higher quality partitions become more important. Intuitively these results make sense, since better locality results in less communication and false sharing between processors.

Optimizations such as CPACK and RCM which yield slightly less improvements than RCB and METIS for uniprocessors perform substantially worse on parallel codes. In comparison, MORTON and GPART achieve roughly the same performance as the more expensive partitioning algorithms, even for parallel codes. When overhead is taken into account (in the averaged results), we see that the high overhead of RCB and METIS are exacerbated in parallel programs, actually leading to negative improvements (slowdowns) for small number of operations.

Overall, our experiments show that if geometric coordinate information is available, MORTON deliver the best performance, with its low overheads and quality partitions. Otherwise, results show GPART is the best choice since it has only one third the overhead of more expensive partitioning algorithms, but produces partitions whose performance is within 1–2% of the best partitions, and 9–18% better than traversal based methods like RCM and CPACK. Once processing overhead is taken into account, other than MORTON GPART is the clear winner over a range of program execution lengths. Our optimization approach is to thus use either MORTON or GPART, depending on whether user annotations are available to specify geometric coordinate information.

5.6 Performance of adaptive computations

To evaluate the precision of our static cost model and on-the-fly cost model, we performed experiments with the three kernels on a DEC Alpha 21064 processor. To verify our static cost model, we ran experiments with three kernels. One of results is shown in the first graph of Figure 15. We show one application and one input results due to the limitation of space. However, other applications also show similar results. The y-axis represents the percentage of net performance gain over original execution time. The x-axis represents the number of transformations applied throughout the whole time steps. Different curves correspond to different locality reordering, varying numbers of transformations applied. The vertical bars represent the numbers of transformations chosen by our cost model and the percentage numbers under the bars represent the predicted gain. The optimization frequency calculated by the cost model is not an integer and needs to be rounded to an integer in practice. Nonetheless, results show our static cost model predicts precise performance gains, and always selects nearly the maximal point of each measured curve.

The second graph on Figure 15 shows the performance improvements over original programs, using both static cost model and on-the-fly cost model. For a reference comparison, the graph also shows the best performance among all possible periodic optimization. The performance gain from both cost models closely matches with the best performance gain using periodic optimization. Thus, our cost model precisely finds an optimal period statically or on the fly. The improvements from on-the-fly cost model are also close to the improvements from static cost model, verifying on-the-fly cost is as precise as static cost model.

The numbers on top of bars in the graph show the number of transformation applied in the best periodic optimization, or static/on-the-fly cost model. The numbers of transformations from on-the-fly cost model also closely match with the numbers from static cost model. In some cases the numbers do not match exactly with the numbers from the best periodic optimization, but the performance gains are not much different. Due to the simplicity in both cost model, they

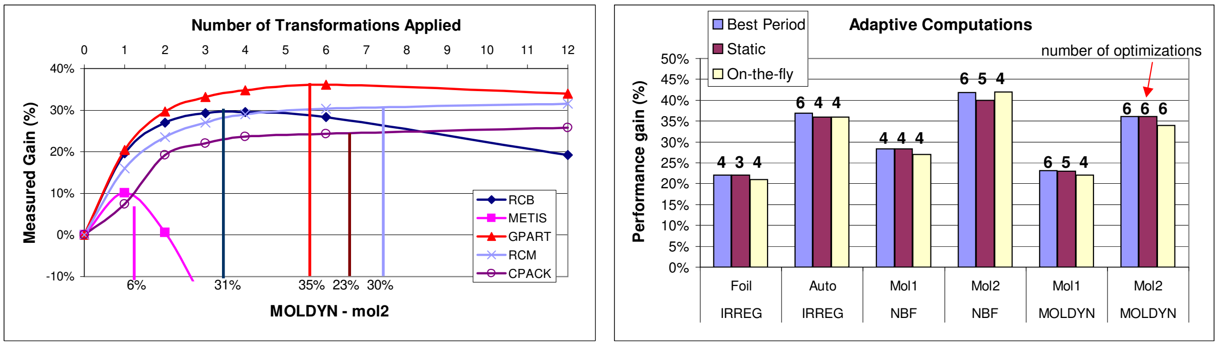

Figure 15 Static Cost Model & On-the-fly Cost Model

might not be able to pin-point the optimal period of optimization. However, both cost models produce roughly good periods, losing only small amount of the performance.

6 Related Work

The importance of irregular scientific computations is well established. Irregular reductions have been recognized as being particularly vital. Researchers have investigated compiler analyses and ways to provide efficient run-time and compiler support as well as efficient implementations of parallel irregular reductions [8, 18]. The inspector-executor paradigm used by locality optimizations was first developed for message-passing multiprocessors and employed in the CHAOS run-time system [6].

Most research on improving data locality has focused on dense arrays with regular access patterns. Locality optimizations have also been developed in the context of sparse linear algebra [4]. Some researchers have investigated improving locality for irregular scientific applications. Das *et al.* investigated data/computation reordering for unstructured euler solvers [5]. They combined data reordering using RCM and lexicographical sort for computation reordering with communication optimizations. Al-Furaih and Ranka studied partitioning data using METIS and BFS to reorder data in irregular codes [1]. They conclude METIS yields better locality.

Ding and Kennedy explored applying dynamic copying (packing) of data elements based on loop traversal order [7]. They were able to automate most of their transformations in a compiler, using user provided information. For adaptive codes they reapply transformations after every change. In comparison, we show partitioning algorithms can yield better locality, albeit with higher processing overhead. We also develop a more sophisticated on-the-fly algorithm for deciding when to reapply transformations for adaptive codes.

Mellor-Crummey *et al.* manually applied a geometric partitioning algorithm based on space-filling curves to map multidimensional data to memory, showing large improvements in performance [14]. Hu *et al.* show similar benefits for parallel irregular codes [10]. In comparison, we automate our optimizations in a compiler and run-time library, and also use graph-based partitioning techniques.

Mitchell *et al.* improved locality using bucket sorting to reorder loop iterations in irregular computations [15]. They improved the performance of two NAS applications (CG, and IS) and a medical heart simulation. Bucket sorting works only for computations containing a single irregular access per loop iteration. In comparison, we investigate more complex cases where two or more irregular access patterns exist. For simple codes lexicographic sort yields improvements similar to bucket sorting.

Researchers have previously proposed on-the-fly algorithms for improving load balance in parallel systems, but we are not aware of any such algorithm for guiding locality optimizations. Bull uses a dynamic system to improve load balance in loop scheduling [3]. Execution times of parallel loop iterations are recorded and used for load balancing.

Nicol and Saltz investigated algorithms to remap data and computation for adaptive parallel codes to reduce load imbalance [16]. They use a dynamic heuristic which monitors load imbalance on each iteration, predicting time lost to load imbalance. Data an computation is remapped using a greedy *stop-on-rise* heuristic, when a local minima is reached in predicted benefit. We adapt a similar approach for our on-the-fly optimization technique, but use it to guide transformations to improve locality instead of load imbalance.

7 Conclusions

Our research demonstrates the benefits of improving data locality in irregular scientific applications by applying compiler and run-time transformations to partition data and computation. We develop GPART,a new partitioning algorithm based on hierarchical graph partitioning with low overhead. Experiments find partitioning algorithms (RCB, METIS, GPART) generate better quality ordering for cache locality than simple traversing algorithms (RCM, CPACK), particularly on sparse inputs. GPART produces partitions which almost match the performance of the best partitioning strategies, but with much lower overhead. We also present an on-the-fly cost model to dynamically select how often to reorder data in adaptive codes.

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.

8 Acknowledgments

We are very grateful to Prof. George Karypis at the University of Minnesota for providing the code for METIS and application meshes used in our experiments. We also wish to thank Prof. Pugh for an improvement to GPART and Prof. Saltz for several helpful suggestions on adaptive codes.

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, April 1998.

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

[3] J. M. Bull. Feedback guided dynamic loop scheduling: Algorithms and experiments. In *Proceedings of the Fourth International Euro-Par Conference (Euro-Par'98)*, Southhampton, UK, September 1998.

[4] 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.

[5] R. Das, D. Mavriplis, J. Saltz, S. Gupta, and R. Ponnusamy. The design and implementation of a parallel unstructured euler solver using software primitives. In *Proceedings of the 30th Aerospace Sciences Meeting & Exhibit*, Reno, NV, January 1992.

[6] 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, September 1994.

[7] 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.

[8] 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, October 1998.

[9] 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.

[10] Y. Hu, A. Cox, and W. Zwaenepoel. Improving fine-grained irregular shared-memory benchmarks by data reordering. In *Proceedings of SC'00*, Dallas, TX, November 2000.

[11] 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.

[12] 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.

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

[14] 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.

[15] 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, October 1999.

[16] D. Nicol and J. Saltz. Dynamic remapping of parallel computations with varying resource demands. *IEEE Transactions on Computers*, 37(9):1073–1087, September 1988.

[17] R. Wilson et al. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29(12):31–37, December 1994.

[18] H. Yu and L. Rauchwerger. Adaptive reduction parallelization techniques. In *Proceedings of the 2000 ACM International Conference on Supercomputing*, Santa Fe, NM, May 2000.