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

Irregular scientific codes experience poor cache performance due to their memory access patterns. In this paper, we examine three issues for locality optimizations for irregular computations. First, we experimentally determine efficient parameters for optimizations based on partitioning algorithms. Second, we find locality optimizations can improve performance for parallel codes, but is dependent on the parallelization techniques used. Finally, we show locality optimizations may be used to improve performance even for adaptive codes. We develop a cost model which can be employed to calculate an efficient optimization frequency; it may be applied dynamically instrumenting the program to measure execution time per time-step iteration. Our results are validated through experiments on three representative irregular scientific codes.

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.

This research was supported in part by NSF CAREER Development Award #ASC9625531 in New Technologies, NSF CISE Institutional Infrastructure Award #CDA9401151, and NSF cooperative agreement ACI-9619020 with NPACI and NCSA.

```
// 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

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 irregular code, the irregular accesses to x change as the program proceeds. Whie 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.

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 [32, 46, 13]. 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 [9, 1, 11, 33, 35] and even programs with pointer-based data structures [5, 7]. Results have been promising, but a number of issues have not been examined. Our paper makes the following contributions:

- Experimentally determine effective parameters for locality optimization algorithms that balance overhead with improved locality. We find algorithms which generate roughly L1 cache-sized partitions achieve the best performance.
- Investigate the impact of locality optimizations on parallel programs. We find optimizations which yield better locality have greater impact on parallel performance than sequential codes, especially when parallelizing irregular reductions using local writes.
- 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 then present our experiments on the impact of parameters on locality optimizations, and evaluate the effect of locality optimizations on parallel performance. Next, we evaluate the effect of adaptivity on 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 spatial locality.

Figure 2 gives an example of how program transformations can improve locality. 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.

Fortunately, irregular scientific computations are typically composed of loops performing *reductions* such as SUM and MAX, so loop iterations can be safely reordered [9, 11]. Data layouts can be safely modified as well, as long as all references to the data element are updated. Such updates are straightforward unless pointers are used.

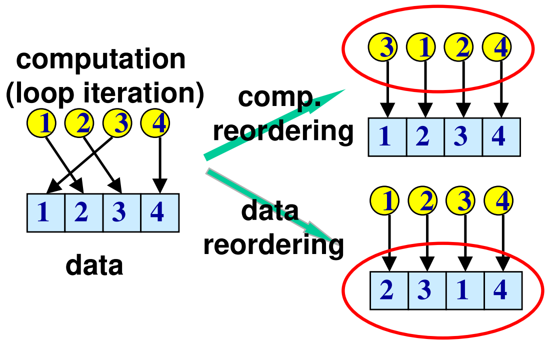

Figure 2 Data & Computation Reordering

```
// 1 access
global x(N)
global idx1(M)

do i = 1, M
   ... = x(idx1(i))
```

```
// 2 accesses
global x(N)
global idx1(M),idx2(M)

do i = 1,M
   ... = x(idx1(i))
   ... = x(idx2(i))
```

Figure 3 Access Pattern of Irregular Computations

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. Figure 3 shows examples of different access patterns. 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, as in Figure 2. 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` [9, 11, 35]. Sorting computations may also be virtually achieved by sorting index array `idx1`.

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 pair of points. An example is shown in Figure 4. 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 [9, 1, 11, 17].

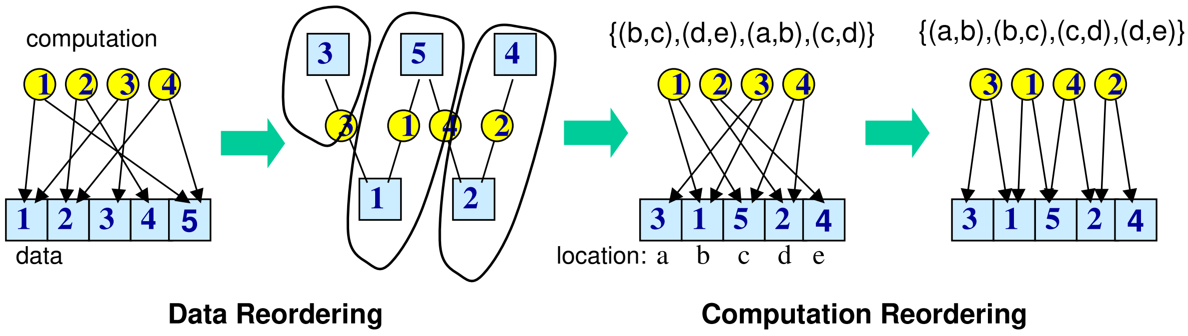

Figure 4 Graph Partitioning & Lexicographic Sort

2.3 Locality optimization algorithms

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 [33]. However, geometric coordinate information is not necessarily available, and will probably require user intervention to identify. We briefly review a number of other locality optimization algorithms which are easier to automate as compiler-directed transformations.

Lexicographical sorting One optimization is to reorder loop iterations(computations). If each iteration performs only one irregular access, sorting the loop iterations by data accessed results in optimal locality. Bucket sorting with cache-sized buckets can also be used, producing similar improvements with low overhead [35]. For computations with two or more irregular accesses, lexicographically sorting the loop iterations will improve temporal locality. Sorting is so effective that data meshes provided by applications writers are frequently presorted, eliminating the need to apply sorting at run time. When we perform experiments, we use presorted input meshes, and always apply lexicographical sorting after data reordering.

Consecutive Packing (CPACK) In addition to computation reordering, 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 [11]. 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 [8, 30, 23]. 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 [9, 1].

Recursive Coordinate Bisection (RCB) RCB recursively splits each dimension into two partitions by finding the median of data coordinates in that dimension. The process is recursively repeated, alternating dimension [2, 3]. After partitioning, data items are stored consecutively within each partition, and partitions within an upper level partition are also arranged consecutively, constructing a hierarchical structure, which is similar to Z-ORDERING, a space-filling curve for dense array layout. RCB has higher overhead, but is likely to work better for unevenly distributed data than space-fillign curves.

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. Multi-level graph partitioning algorithms encapsulated in library packages such as METIS [25, 26] 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.

Low Overhead Graph Partitioning (GPART) We proposed a low overhead graph partitioning algorithm(GPART) which is tuned for runtime locality optimization [17]. GPART 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. When GPART maps the hierarchical structure on memory, it puts the nodes in the same partition on nearby memory locations. This layout principle is observed throughout the all levels of hierarchy. GPART achieves quality cache locality with low overhead, compared to RCB and METIS.

```
** 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 5 Key Kernels of Applications

Name	# Nodes	# Edges	Description
FOIL	144649	1074393	3D mesh of a parafoil
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 Data Meshes

3 Locality Optimizations Parameters

Optimizations such as RCB, METIS, and GPART work by partitioning input data, then rearranging data layout to colocate data in the same partition. Smaller partitions are likely to yield better locality, but processing overhead also increases. In this section, we experimentally measure the impact of partition size on the performance of RCBand METIS, as well as the impact of collapsing factor and passes on GPART. The resulting parameters are used to guide these locality optimizations [17].

3.1 Experimental environment

Our experimental results are obtained on a Sun HPC10000 which has 400 MHz UltraSparc II processors with 16K direct-mapped L1 and 4M direct-mapped L2 caches. Our prototype compiler is based on the Stanford SUIF compiler [45]. It can identify and parallelize irregular reductions using *pthreads*, but does not yet generate inspectors like Chaos [18, 19]. As a result, we currently insert inspectors by hand for both the sequential and parallel versions of each program.

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 [18]. 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 5.

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. We applied these meshes to IRREG, NBF, and MOLDYN to test the locality effects. All meshes are initially sorted, so computation reordering is not required originally, but computation reordering is applied only after data reordering techniques are used, since data reordering makes computations out of order. The sizes of each input data set is shown in Table 1.

3.2 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

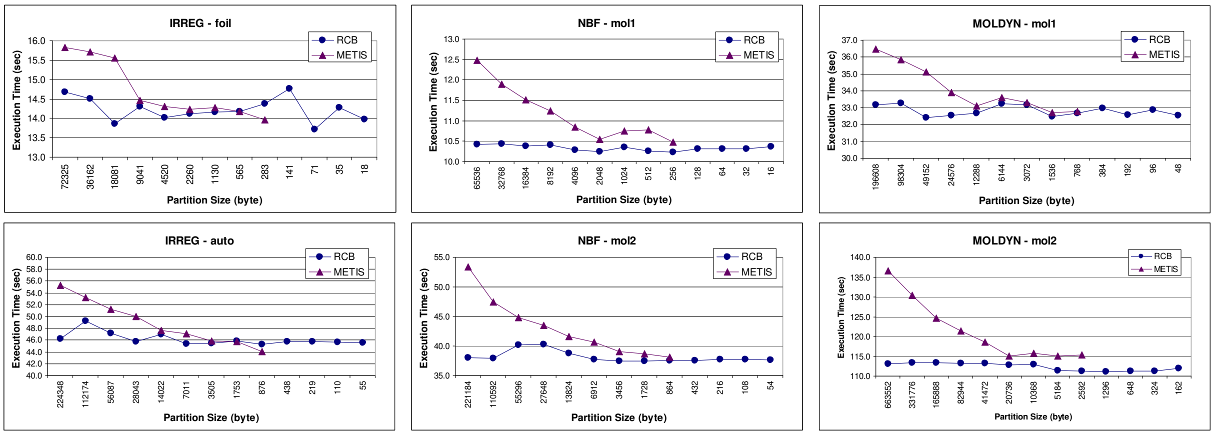

Figure 6 Impact of Partition Sizes of RCB and METIS (overhead excluded)

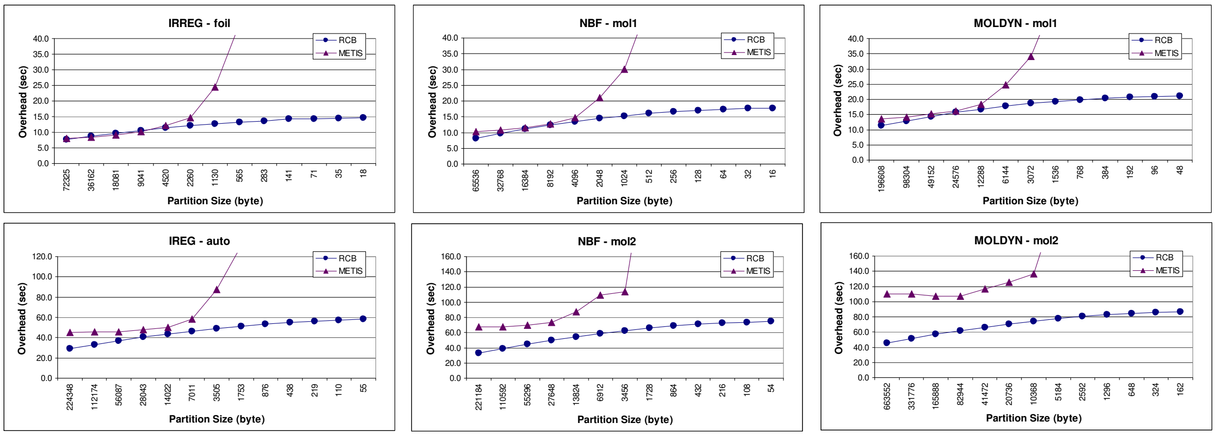

Figure 7 Overhead vs. Partition Size for RCB and METIS

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 6. The 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 7 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. We follow this procedure in our experiments, computing the desired number of partitions by dividing the entire input data size by the L1 cache size, so that each partition created is then roughly the L1 cache size.

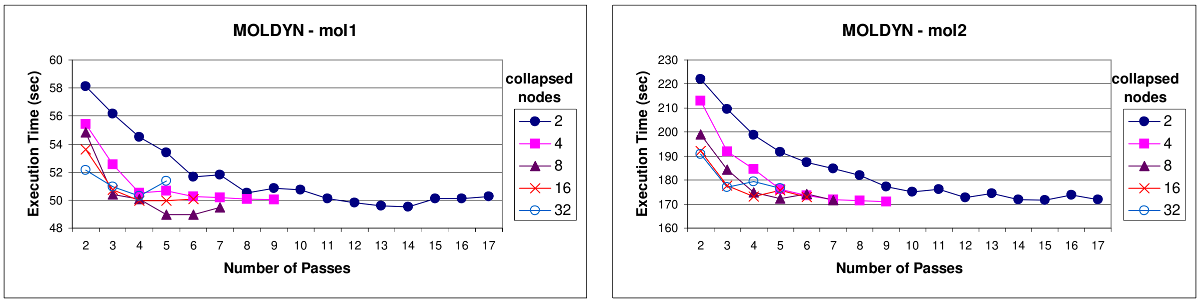

Figure 8 Impact of GPART Parameters (overhead excluded)

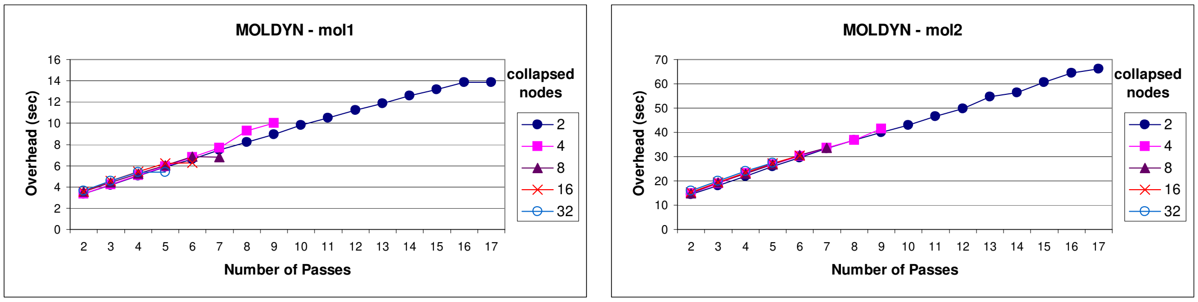

Figure 9 Overhead vs. GPART Parameters

3.3 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 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 8. 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 9 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 different collapsing rates do not much affect overhead. Selecting best collapsing rate for a given number of passes is thus the best way to choose parameters when considering the performance including the overhead.

Once again, results indicate most locality benefits may be obtained when the size of all the data accessed in a partition is roughly the size of the L1 cache. The choice of the number of nodes to collapse in each pass is more tricky when considering 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.

For our experiments we choose five passes with initial partitions comprising four nodes, since each node in our applications contains an 8-byte double-precision floating point number. Thus, four nodes will fit into 32-byte cache line on Sun UltraSparcII processors. For additional passes we collapse up to eight nodes. The sequence of passes thus produces partitions containing 4, 32, 256, 2K, and 16K nodes through five passes.

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 Sun Ultra Sparc processors. We merge 8 nodes on each pass, resulting in 5 passes. As we mentioned early, computation reordering is applied

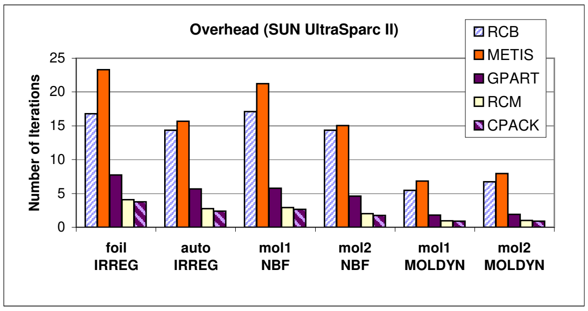

Figure 10 Overhead of Data Reordering (relative to 1 iteration of ORIG)

after any data reordering algorithms are applied.

3.4 Overhead of optimizations

With these optimization parameters, it is interesting to see the relative processing overhead of each locality optimization. Figure 10 displays the costs of data reordering techniques 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.

4 Impact on Parallel Codes

The second issue we consider is the effect of locality optimizations on parallel execution. We find parallel performance is also improved, but the impact is dependent on the parallelization strategy employed by the application. We thus first briefly review two parallelization strategies.

4.1 Parallelizing irregular reductions

The core of irregular scientific applications is frequently comprised of *reductions*, associative computations (e.g., SUM, MAX) which may be reordered and parallelized [38, 31, 44]. 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 [15, 38].

```
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 11 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
    if(own(y[n]) y[n] += force
    if(own(y[m]) y[m] += -force
```

Figure 12 LOCALWRITE Inspector Example

An example of the REPLICATEBUFS technique is shown in Figure 11. 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 [15]. 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.

An alternative method is LOCALWRITE, a compiler and run-time technique for parallelizing irregular reductions we previously developed [16]. 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 [20]. 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 12. 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

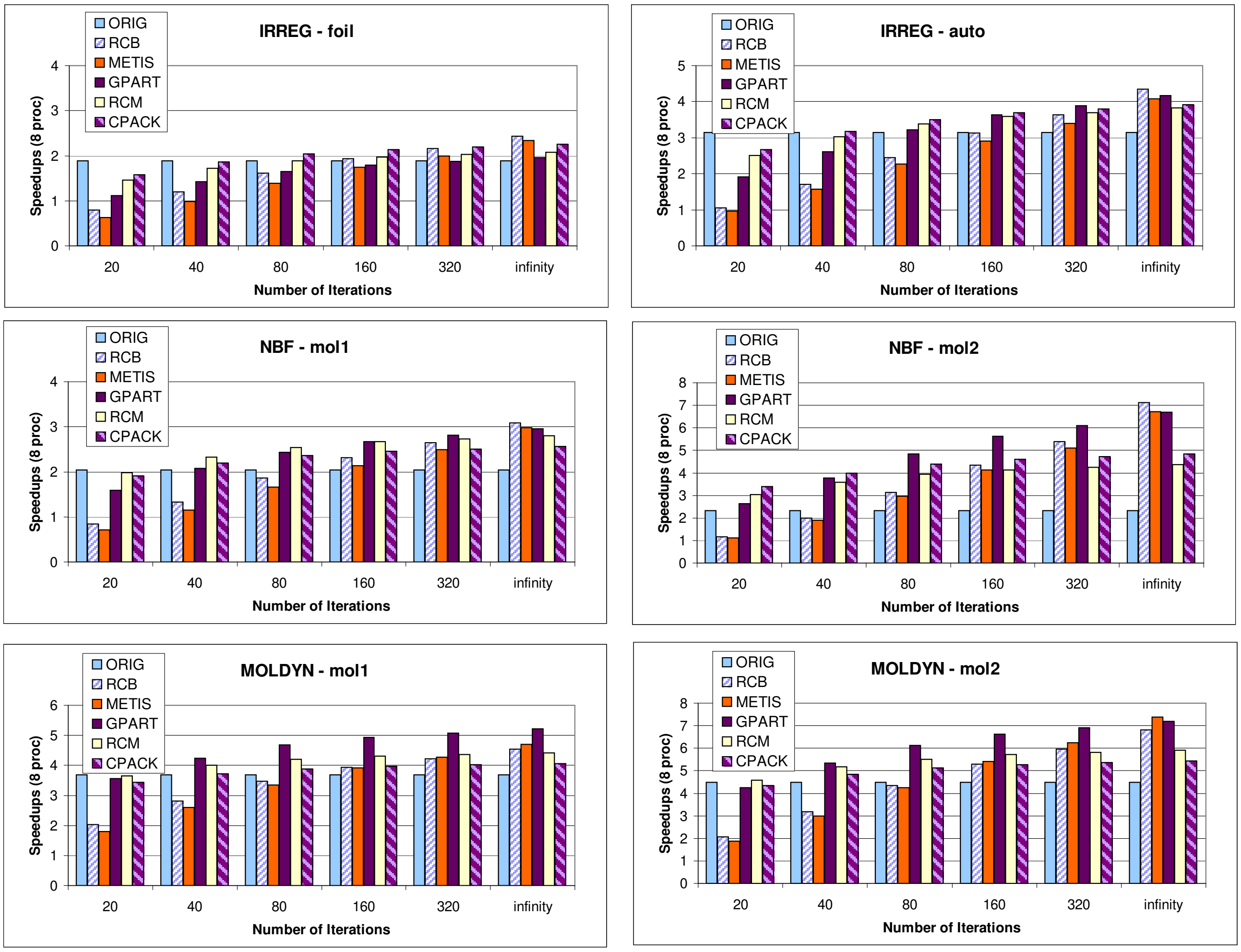

Figure 13 Speedups on HPC 10000 (parallelized with REPLICATEBUFS)

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.

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).

Though targeting parallel programs, our locality optimizations preprocess data sequentially. If overhead is too high, parallel graph partitioning algorithms may be used to reduce overhead for parallel programs.

4.2 Impact on parallel performance

Figure 13 and Figure 14 display 8-processor Sun HPC10000 speedups for each mesh, calculated versus the original, unoptimized program. Figure 13 shows speedups when applications are parallelized using REPLICATEBUFS and Figure 14 shows speedups when LOCALWRITE is used. We include overheads to show how performance changes as the total number of iterations executed by the program increases. We see that when a sufficient number of iterations is executed, the versions optimized for locality achieved better performance. Locality optimizations thus also improve performance of parallel versions of each program.

In addition, we found that with locality optimizations, programs parallelized using LOCALWRITE achieved much

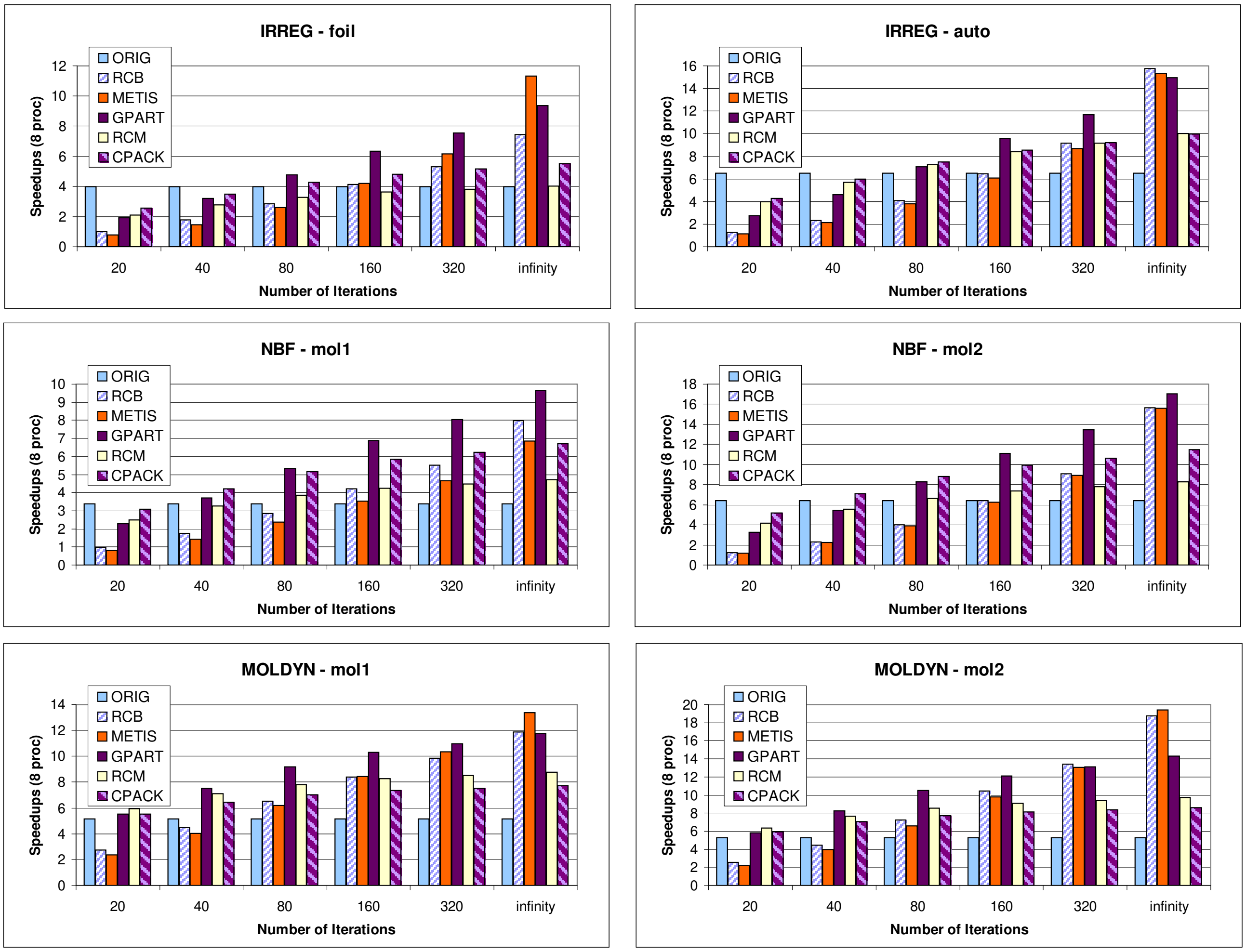

Figure 14 Speedups on HPC 10000 (parallelized with LOCALWRITE)

better speedups than the original programs using REPLICATEBUFS. The LOCALWRITE algorithm tends to be more effective with larger graphs where duplicated computations are relatively few compared to computations performed locally. In general, 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.

Results show higher quality partitions become more important for parallel codes. Optimizations such as CPACK and RCM which yield almost the same improvements as RCB and METIS for uniprocessors [17] perform substantially worse on parallel codes. In comparison, GPART achieves roughly the same performance as the more expensive partitioning algorithms, even for parallel codes.

5 Optimizing 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. 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.

5.1 Impact of adaptation

To evaluate the effect of adaptivity on performance after computation/data reordering, we timed MOLDYN with input MOL1 on a DEC Alpha, periodically swapping 10%

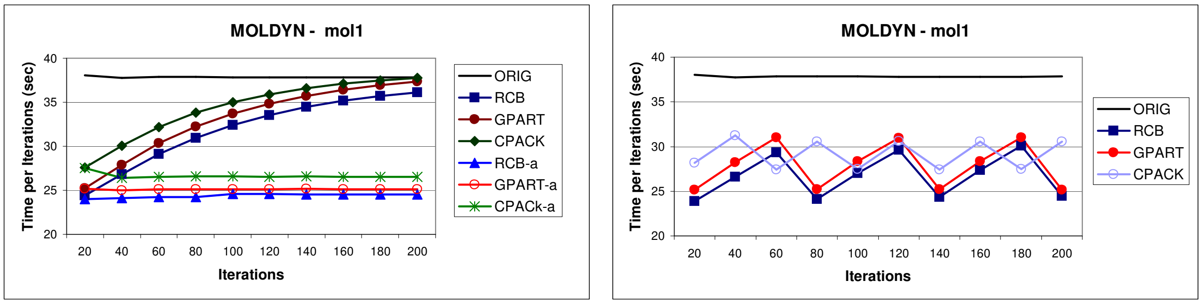

Figure 15 Performance Changes in Adaptive Computations (overhead excluded)

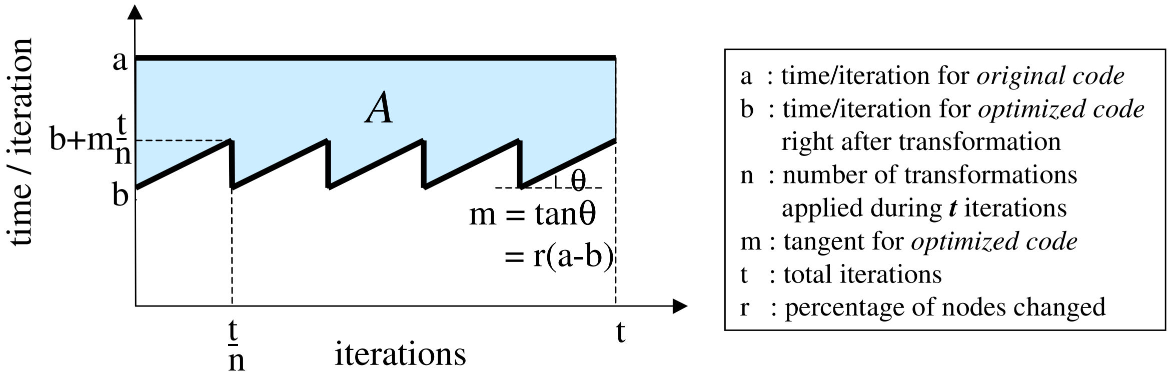

Figure 16 Analytical Model for Adaptive Computations

of the molecules randomly every 20 iterations to create an adaptive code. We realize this is not realistic, but it provides a testbed for our experiments. Adaptive behavior in MOLDYN is actually dependent on the initial position and velocity of molecules in the input data. Future research will need to conduct a more comprehensive study of adaptive behavior in scientific programs.

Results for our synthetic adaptive code is shown in Figure 15. In the first graph, the x-axis marks the passage of time in the computation, while the y-axis measures execution time. ORIG is the original program. RCB, METIS, GPART, and CPACK represent versions of the program where data and computation are reordered for improved locality exactly once at the beginning of program. In comparison, RCB–a, METIS–a, GPART–a, and CPACK–a represent the versions where data and computation are reordered whenever access patterns change. Data points represent the execution times for every 20 iterations of the program. Results show that without reapplying locality reordering, performance degrades and eventually matches with the unoptimized program. In comparison, reapplying partitioning algorithms after every access pattern change can preserve the performance benefits of locality, if overhead is excluded.

In practice, however, we do need to take into account overhead. By periodically applying reordering, we can maximize the net benefit of locality reordering. Performance changes with periodic reordering are shown in the second graph in Figure 15. We reapply CPACK every 40 iterations, and RCB and GPART every 60 iterations. Results show performance begins to degrade, but recover after locality optimizations are reapplied. Even after overheads are included, the overall execution time is improved over not 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.

5.2 Cost model for optimizations

To guide locality transformations for adaptive codes, we need a cost model to predict the benefits of applying optimizations. We begin by showing how to calculate costs when all parameters such as optimization overhead, optimization benefit, access change frequency, and access change magnitude 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 dynamic instrumentation.

First we present a simple cost model for computing the benefits of locality reordering when all information is available. We can use it to both predict improvements for different optimizations and decide how frequently locality transformations should be reapplied.

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

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

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

Figure 17 Calculating Net Gain and Adaptation Frequency

We consider models for the performance of two versions of a program, *original code* and *optimized code*. Figure 16 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 node changes. For the optimized code, locality optimization is performed at the beginning and periodically reapplied. For simplicity, we assume execution times per iteration increase linearly as the nodes change and periodically drop to the lowest (optimized) point after reordering 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.

Performance degradation rate (m) of the optimized code is set to $r(a-b)$, where r is the percentage of changed nodes. For example, if an adaptive code randomly changes 10% of nodes each iteration ($r = 0.1$), then the execution time per iteration becomes that of the original code after 10 iterations. The net performance gain ($G(n)$) from periodic locality transformations can be calculated as in Figure 17. 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. 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.

To verify our cost model, we ran experiments with three kernels on a DEC Alpha 21064 processor. The kernels iterate 240 time steps, randomly swapping 10% of nodes every 20 iterations. We will use these adaptive programs for all the remaining experiments for adaptive codes. Results are shown in Figure 18. 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 240 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 cost model predicts precise performance gains, and always selects nearly the maximal point of each measured curve.

5.3 On-the-fly application of cost model

Though experiments show the cost model introduced in the previous section is very precise, it requires several parameters to calculate the frequency 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., rate of change in data access patterns).

In this section, we show how a simplified cost model can be applied by gathering information on-the-fly by inserting a few timing routines in the program. First, we apply only GPART as locality reordering. Thus, we only calculate the frequency of reordering, not the expected gain. 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 (instead of calculating from the percentage of node changes) and adjusted by monitoring performance (execution time) of every iteration. The new rate is chosen so that actual elapsed time since the last reordering is the same as calculated elapsed time under the newly

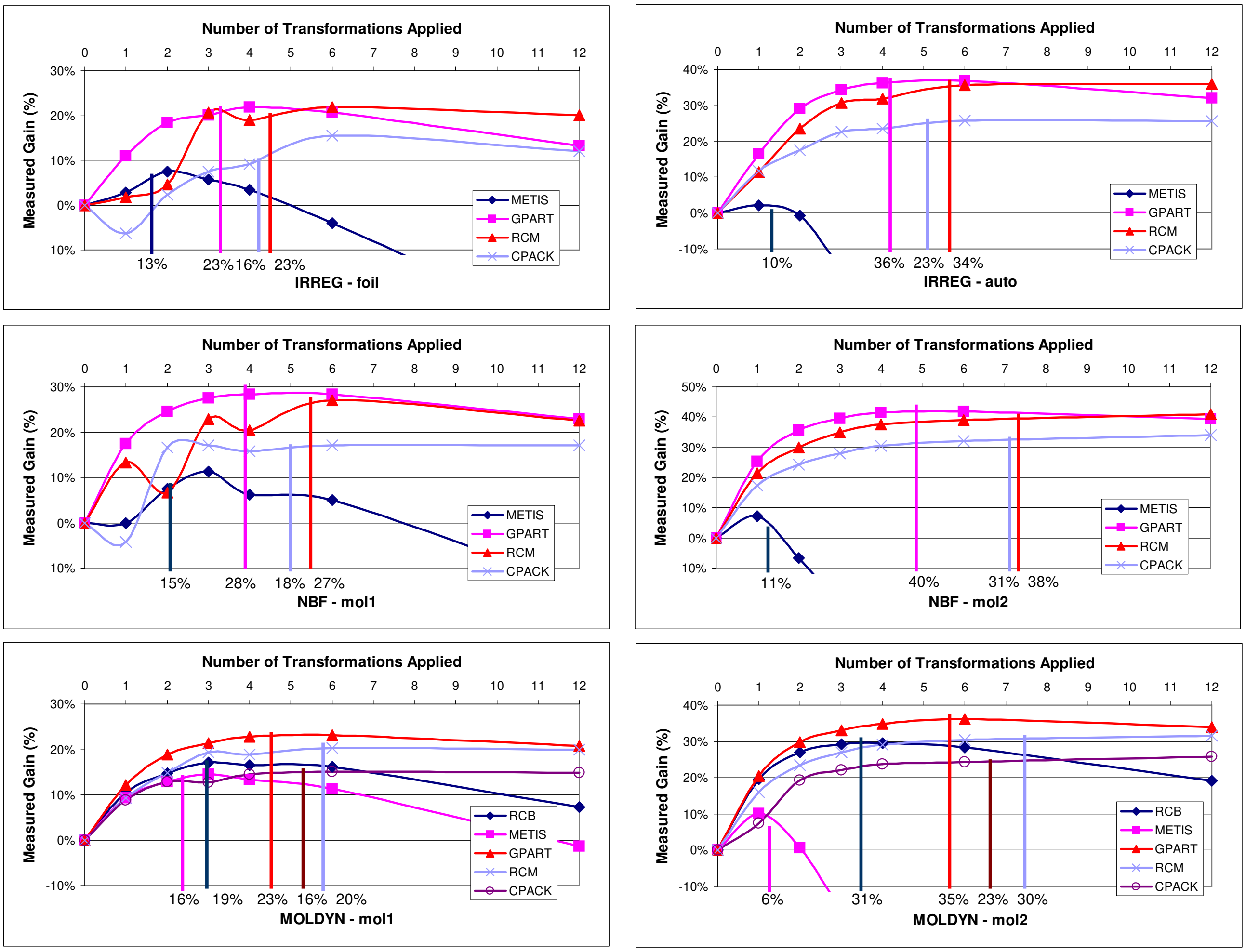

Figure 18 Experimental Verification of Cost Model (vertical bars represent the numbers chosen by the cost model)

```
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 19 Applying Cost Model Dynamically

chosen linear rate.

Figure 19 shows an example code that uses on-the-fly cost model. At every iteration, `cost_model()` is called to calculate a frequency based on the information monitored so far. In our current setup, 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 frequency of reapplying locality optimizations. From this point on, the cost model keeps monitoring performance of each iteration, adjusting degradation rate and produces new frequencies 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 19. This function call notifies the cost model when access pattern changes. The cost model then

	IRREG		NBF		MOLDYN	
	FOIL	AUTO	MOL1	MOL2	MOL1	MOL2
Number of Transformations	4	4	4	4	4	6
Measured Gain (%)	20.7	36.0	27.1	41.5	22.1	33.9

Table 2 Performance Improvement for Adaptive Codes with On-the-fly Cost Model

decides whether access patterns periodically changes.

To evaluate the precision of our on-the-fly cost model, we performed experiments with the three kernels under the same situation. Table 2 shows the performance improvements over original programs that do not have locality reordering. The improvements closely match with the maximum improvements in Figure 18 and the numbers of transformations also match with the numbers in periodic reordering. Our on-the-fly cost model thus works well in practice with limited information.

6 Related Work

The importance of irregular scientific computations is well established [34, 41, 27, 36]. Irregular reductions have been recognized as being particularly vital [21, 31, 38, 44]. Researchers have investigated compiler analyses [28, 29] and ways to provide efficient run-time and compiler support [6, 39], as well as efficient implementations of parallel irregular reductions [14, 16, 47].

The inspector-executor paradigm used by locality optimizations was first developed for message-passing multiprocessors and employed in the CHAOS run-time system [10]. Compiler techniques were developed to automatically generate calls to the run-time routines [18, 19]. An algorithm for building incremental communication schedules was implemented to reduce processing overhead for adaptive irregular codes [22]. In comparison, locality optimizations do not need to reanalyze after each access pattern change because the change affects only locality, not correctness.

Most research on improving data locality has focused on dense arrays with regular access patterns, applying either loop [32, 43, 42, 46] or data layout [24, 40] transformations. Locality optimizations have also been developed in the context of sparse linear algebra. The Reverse Cuthill-McGee (RCM) method [8] improves locality in sparse matrices by reordering columns using reverse breadth-first search (BFS). More sophisticated versions of the algorithm have since been developed [8, 30, 23].

Some researchers have investigated improving locality for irregular scientific applications. Das *et al.* investigated data/computation reordering for unstructured euler solvers [9]. They combined data reordering using RCM and lexicographical sort for computation reordering with communication optimizations to improve performance of parallel codes on an Intel iPSC/860. Al-Furaih and Ranka studied partitioning data using METIS and BFS to reorder data in irregular codes [1]. They conclude METIS yields better locality. They also evaluated different partition sizes for METIS, and found partitions equal to cache size yielded the best performance. However, they did not include computation reordering or processing overhead.

Ding and Kennedy explored applying dynamic copying (packing) of data elements based on loop traversal order, and show major improvements in performance [11]. 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. Ding and Kennedy also developed algorithms for reorganizing single arrays into multi-dimensional arrays depending on their access patterns, improving the performance of many irregular applications [12]. Their technique might be useful for Fortran codes where data are often single dimension arrays, not structures as in C codes.

Mellor-Crummey *et al.* used a geometric partitioning algorithm based on space-filling curves to map multi-dimensional data to memory [33]. They also blocked computation using methods similar to tiling. In comparison, our graph-based partitioning techniques are more suited for compilers, since geometric coordinate information and block parameters for space-filling curves do not need to provided manually. When coordinate information is available, using RCB is better because space-filling curves cannot guarantee evenly balanced partition when data is unevenly distributed, which may cause significant performance degradation in parallel execution.

Mitchell *et al.* improved locality using bucket sorting to reorder loop iterations in irregular computations [35]. 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 [4]. The execution time of each parallel loop iteration is recorded and used to ensure processors are assigned roughly equal amounts of computation on succeeding iterations of the enclosing time-step loop. In comparison, we measure time executed per loop iteration for the entire time-step loop, and apply it to decide when to reapply locality optimizations.

Nicol and Saltz investigated algorithms to remap data and computation for adaptive parallel codes to reduce load imbalance [37]. 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

In this paper, we propose a framework that guides how to apply locality optimizations according to the application access pattern. We find locality optimizations also improve performance for parallel codes, especially when combined with parallelization techniques which benefit from locality. We show locality optimizations may be used to improve performance even for adaptive codes, using an on-the-fly cost model to select for each optimization how often to reorder data.

As processors speed up relative to memory systems, locality optimizations for irregular scientific computations should increase in importance, since processing costs go down while memory costs 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. Saltz for several helpful suggestions.

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 pdes across multiprocessors. In *Proceedings of the 1985 International Conference on Parallel Processing*, August 1985.

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

[4] 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.

[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, October 1998.

[6] S. Chandra and J.R. 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

[13] 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.

[14] E. Gutierrez, O. Plata, and E. Zapata. A compiler method for the parallel execution of of irregular reductions on scalable shared memory multiprocessors. In *Proceedings of the 2000 ACM International Conference on Supercomputing*, Santa Fe, NM, May 2000.

[15] 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, December 1995.

[16] 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.

[17] H. Han and C.-W. Tseng. A comparison of locality transformations for irregular codes. In *Proceedings of the 5th Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, Rochester, NY, May 2000.

[18] 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, December 1993.

[19] 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.

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

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

[22] 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.

[23] E. Im and K. Yelick. Model-based memory hierarchy optimizations for sparse matrices. In *Proceedings of the 1998 Workshop on Profile and Feedback-Directed Compilation*, Paris, France, October 1998.

[24] 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, November 1998.

[25] 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, August 1995.

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

[27] C. Koelbel, P. Mehrotra, J. Saltz, and S. Berryman. Parallel loops on distributed machines. In *Proceedings of the 5th Distributed Memory Computing Conference*, Charleston, SC, April 1990.

[28] Y. Lin and D. Padua. On the automatic parallelization of sparse and irregular Fortran programs. In *Proceedings of the 4th Workshop on Languages, Compilers, and Runtime Systems for Scalable Computers*, Pittsburgh, PA, May 1998.

[29] Y. Lin and D. Padua. Compiler analysis of irregular memory accesses. In *Proceedings of the SIGPLAN '00 Conference on Programming Language Design and Implementation*, Vancouver, Canada, June 2000.

[30] 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, April 1976.

[31] 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, April 1998.

[32] 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.

[33] 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.

[34] R. Mirchandaney, J. Saltz, R. Smith, D. Nicol, and K. Crowley. Principles of runtime support for parallel processors. In *Proceedings of the Second International Conference on Supercomputing*, St. Malo, France, July 1988.

[35] 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.

[36] S. Mukherjee, S. Sharma, M. Hill, J. Larus, A. Rogers, and J. Saltz. Efficient support for irregular applications on distributed-memory machines. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, July 1995.

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

[38] 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.

[39] M. Rinard and P. Diniz. Commutativity analysis: A new analysis technique for parallelizing compilers. *ACM Transactions on Programming Languages and Systems*, 19(6):942–992, November 1997.

[40] 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.

[41] J. Saltz, R. Mirchandaney, and K. Crowley. Runtime parallelization and scheduling of loops. In *Proceedings of the 1st Symposium on Parallel Algorithms and Architectures*, Santa Fe, NM, 1989.

[42] 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.

[43] 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.

[44] 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.

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

[46] 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.

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