Software Support For Improving Locality in Scientific Codes

Hwansoo Han, Gabriel Rivera, Chau-Wen Tseng

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

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

Abstract

We propose to develop and evaluate software support for improving locality for advanced scientific applications. We will investigate compiler and run-time techniques needed to achieve high performance on both sequential and parallel machines. We will focus on two areas. First, iterative PDE solvers for 3D partial differential equations have poor locality because accesses to nearby elements in higher-level dimensions are spread far apart in memory. Careful tiling and padding can frequently recapture such reuse. Second, computations on adaptive meshes and sparse matrices experience many cache misses because they access data in an irregular manner. Data layout and access order can be rearranged according to mesh connections or geometric location to improve locality, with cost models used to guide frequency of transformations for adaptive computations.

1 Introduction

From astrophysics to biochemistry, scientists are increasingly using computers to conduct research and development. Fueling the trend is the wide availability of inexpensive commodity microprocessors and soon multiprocessor PCs. Though computers are quite fast, scientists continue to seek greater computing power to perform ever more complex simulations. Parallelism has always been accepted as a technique for greatly improving available computation power, at least for amenable applications. In recent years, *locality* has joined parallelism as a key ingredient to achieving high performance, due to the increasing disparity between memory and processor speeds.

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

To remain successful, computers not only need to be fast, but easy to use. Their computing power should be accessible without expert knowledge of the underlying architecture. Software support at both compile and run time plays an important role by automatically customizing programs for complex processor architectures, improving portability, and providing high performance to non-expert programmers. Compiler transformations for improving data locality have proven to be quite successful, either rearranging computation through loop transformations (e.g., loop permutation, fusion, fission) [52, 76], or changing data layout through data transformations (e.g., padding, transpose) [5, 35, 61]. Experiments show compilers can improve the performance of many scientific programs, some times dramatically.

Despite these successes, new problems are beginning to arise. Scientific codes are becoming increasingly large and complex as more difficult multi-disciplinary problems are attempted. Features such as 3D arrays and adaptive irregular memory accesses make utilizing caches effectively difficult, leading to poor performance. Traditional program transformations are frequently inapplicable or insufficient.

We believe exploiting locality for advanced scientific applications will require developing novel compile-time analyses and sophisticated run-time systems to perform data layout and computation transformations. Run-time systems are needed because many programs are not analyzable statically, while compiler support is still crucial both for inserting interfaces to the run-time system and for directly applying program transformations where possible. Cooperation between the compiler and run-time will be critical for advanced scientific codes.

This proposal aims to develop and evaluate software support for improving locality for advanced scientific applications, for both sequential and parallel machines. To illustrate how compiler and run-time software support can be combined to improve locality, we examine problems and preliminary solutions for two types of scientific codes (3D iterative solvers, adaptive irregular computations) on both sequential and parallel computers.

We begin by quickly reviewing some basic concepts regarding caches and locality, as well as existing locality optimizations. We then examine each of the three types of advanced scientific codes in detail. For each we provide a motivating problem, describe possible solutions, and show some preliminary experimental results. We summarize our proposed research and research plan, before concluding with a comparison with related work.

1.1 Caches and locality

The rise in the importance of data locality is driven by the growing disparity between processor and memory speeds. As processers become ever faster relative to memory, computer architects have created in modern processors a *memory hierarchy* where higher levels (e.g., registers, primary cache) are fast and small, while lower levels (e.g., secondary cache, memory) are slow and large. For a more concrete example of how memory latencies affect performance, consider recent microprocessors such as the DEC Alpha or the MIPS R10000. Data in primary (on-chip) cache can be accessed in a few cycles, secondary (off-chip) cache accesses take 10–20 cycles, and local memory accesses take 60–100 cycles. Processors can now issue multiple instructions per cycle, making long memory latencies even worse obstacles to good performance. Applications that do not use caches effectively spend most of their time waiting for memory accesses.

Because of hardware limitations, caches are divided into a small number of cache lines. Caches can thus be characterized by the length of their cache lines and capacity (size as determined by the product of cache line size and total number of cache lines). Another important property of caches is that they usually have a small fixed *set associativity*, where a k-way associative cache means an address can be mapped into one of k possible cache lines. Limited associativity means a memory access may be mapped on top of a useful cache line, flushing it from the cache despite sufficient capacity in the cache. These *conflict misses* can cause problems when trying to exploit reuse.

In order to exploit the greater speed of higher levels of the memory hierarchy, applications must possess *data locality*, where an application reuses data already brought in. Reuse may be in the form of *temporal locality*, where the same data is accessed multiple times, or *spatial locality*, where nearby data is accessed together. With respect to the program, locality may be in the form of *self-reuse* caused by a single variable reference, or *group-reuse* caused by multiple variable references. For parallel programs, locality also means accessing data in local memory instead of remote memory on another processor node.

Many compiler techniques exist for improving locality in both sequential and parallel scientific programs. Loop transformations (e.g., loop permutation, fusion, tiling) for sequential dense-matrix codes with regular memory access patterns has proven useful [16, 25, 42, 43, 52, 51, 63, 67, 76, 77]. Data layout optimizations (e.g., transpose, padding) also help [3, 5, 12, 15, 35, 61, 62], even for irregular programs [1, 20, 53]. Despite the major advances made in providing software support for improving locality for both sequential and parallel programs, more work remains. In the following sections, we discuss three important types of codes where locality support can be improved.

2 3D Iterative PDE Solvers

2.1 Motivation

An important class of scientific programs attempts to compute solutions to systems of partial differential equations (PDEs) using finite differencing techniques. These iterative PDE solvers are also called *stencil* codes because they compute point values based of averaging neighboring points. Historically PDE solvers have targeted 2D domains, but as computers became more powerful scientists began writing programs to solve PDEs in three dimensions. Numerical analysts then discovered that 3D PDE codes have particularly poor memory behavior with respect to microprocessor caches [7, 68].

To see why 3D codes pose greater difficulties for caches, we look at 2D and 3D versions of a simple Jacobi kernel. Jacobi is a simple example of an iterative PDE solver. Multigrid solvers which use a succession of grid sizes are more realistic, since they converge on solutions in $O(1)$ time. However, the most computationally expensive part of multigrid codes is the smoothing step, which is simply a stencil computation such as Jacobi or red-black Gauss-Seidel applied to the component mesh, so our Jacobi example is highly relevant. For instance, over 90% of the execution time in the SPEC95/NAS MGRID benchmark is spent in a 27-point 3D jacobi kernel.

Consider the Jacobi codes shown in Figure 1. In the 2D Jacobi kernel, three adjacent columns of array B are accessed together. Since the columns are adjacent, the cache only needs to be able to hold three columns of B to exploit group reuse (assuming a write-through cache, so A does not interfere). Even a relatively small 16K primary (L1) cache can thus hold all the data needed up to a 682xM array (where M can be any value), assuming double precision (8 byte) elements. For 2D arrays of column size 682 or smaller, values of B only need to be brought into cache once for each iteration of the outer time-step loop.

In comparison, the 3D Jacobi kernel accesses six columns of B in three adjacent planes. Because three entire NxN

```
// 2D Jacobi Iteration
A(N,N), B(N,N)
do J=2,N-1
do I=2,N-1
  A(I,J) = R*
    (B(I-1,J)+B(I+1,J)+
     B(I,J-1)+B(I,J+1))
```

```
// 3D Jacobi Iteration
A(N,N,N), B(N,N,N)
do K=2,N-1
do J=2,N-1
do I=2,N-1
  A(I,J,K) = R*
    (B(I-1,J,K)+B(I+1,J,K)+
     B(I,J-1,K)+B(I,J+1,K)+
     B(I,J,K-1)+B(I,J,K+1))
```

```
// Tiled 3D Jacobi
A(N,N,N), B(N,N,N)
do KK=2,N-1,TK
do JJ=2,N-1,TJ
do II=2,N-1,TI
  do K=KK,min(KK+TK-1,N-1)
  do J=JJ,min(JJ+TJ-1,N-1)
  do I=II,min(II+TI-1,N-1)
    A(I,J,K) = R*
      (B(I-1,J,K)+B(I+1,J,K)+
       B(I,J-1,K)+B(I,J+1,K)+
       B(I,J,K-1)+B(I,J,K+1))
```

Figure 1 Example stencil codes

planes now need to remain in cache, only 3D arrays of size 26x26xM can fully exploit reuse for a 16K L1 cache. Even for the larger 512K secondary (L2) cache, group reuse is lost for 3D arrays larger than 147x147xM. Data for array B will need to be brought into cache two or more times each time the kernel is executed, reducing performance for larger problem sizes.

2.2 Approach

Tiling (blocking) is a transformation which combines strip-mining with loop permutation to form small tiles of loop iterations which are executed together to exploit data locality [10, 33, 78]. An example of tiling is shown in Figure 1. Tiling is very effective with linear algebra codes [16, 42, 43, 46, 63], but tiling for iterative PDE solvers is a relatively new area. One reason is that when loops are permuted to exploit temporal and spatial reuse [52, 76], group-temporal and spatial reuse in 2D stencils can usually be obtained without tiling, though padding may be necessary to preserve group reuse [51, 62]. As we have already seen, this is no longer the case for 3D stencil codes. Tiling can be used to exploit reuse across outer time-step loops [67], but this is not possible in multigrid codes because each time step contains sweeps over a sequence of grids of different sizes.

We believe tiling can be used to preserve group-reuse and spatial locality in 3D iterative PDE solvers, especially when combined with array padding. We apply tiling to all three dimensions as shown in Figure 1 to exploit reuse carried on all three loops [76]. The tiled code then executes a chunk of the original code at a time. Because fewer iterations of the I and J loops are executed, reuse on the K loop is more likely to be exploited by the cache.

Tiling preserves the original loop permutation, which has already been selected to exploit data locality. In addition, the tile-controlling loops (KK, JJ, II) retain the same permutation as the tile loops (K, J, I). This reduces the loss of reuse along tile boundaries. Which loops need to be tiled is a function of the reuse in a loop. Many methods exist for calculating reuse carried by each loop [52, 76]. Our choice of tile dimensions is guided by a cost model which estimates the reuse gained and lost for different tile dimensions. This is accomplished by estimating the number of cache misses that arise from a tile size, favoring tile sizes which result in fewer cache misses. For instance, for the tiled 3D Jacobi code in Figure 1 a TIxTJxTK tile will access approximately $(TI + 2)(TJ + 2)(TK + 2)$ elements of array B, due to the six-point stencil pattern. Assuming no conflict misses, the cost for the whole nest is $N^3(TI+2+L_s-1)(TJ+2)(TK+2)/(L_s TITJTK)$, where L_s is the number of elements per cache line. The cost model enables the compiler to compare the effectiveness of different tile sizes in improving reuse.

2.2.1 Tile size selection

Limited cache associativity may cause conflict misses, severely degrading the performance of tiled codes [46]. For instance, choosing cubic tiles equal in size to the L1 cache (the TILE algorithm evaluated later) often results in self-interfering tiles. A method for avoiding conflicts is to carefully select tile dimensions tailored to individual array dimensions so that no conflicts occur. For 2D arrays, the Euclidean remainder algorithm may be used to quickly compute a sequence of nonconflicting tile dimensions through a simple recurrence [16, 63].

An alternative algorithm (LRW) by Lam, Rothberg, and Wolf finds the largest square nonconflicting 2D tile using an greedy algorithm which expands tile dimensions while checking no conflicts occur [46]. It can also be used to find a nonconflicting rectangular tile subsuming the largest square tile, by expanding additional tile dimensions after growth in one dimension is limited by conflicts. We adapt this algorithm in a straightforward manner for finding nonconflicting 3D tiles. We begin with $TI = TJ = TK = 1$. We then iteratively attempt to increase each tile dimension until none may be increased without introducing conflicts. On each iteration, we first consider increasing TI followed by TJ and TK.

Note that this method only computes nonconflicting *array* tile dimensions, i.e. the dimensions of a 3D subarray without self-interference. We compute the actual (iteration space) tile dimensions TI', TJ', and TK' based on the resulting array tile dimensions and the access patterns of the tiled loop nest. For instance, for the nest in Figure 1, we need to trim each of the array tile dimensions by 2 (i.e. $TI' = TI - 2, TJ' = TJ - 2, TK' = TK - 2$) or else the iteration space tile will exceed the boundaries of the array tile through accesses to `B`. Note that writes to `A` will not affect the cache given a non write-allocate cache.

In more complicated kernels, where multiple references bring data into cache, we need to partition the nonconflicting array tile between arrays. This can be accomplished by reducing one tile dimension and then applying *inter-variable padding* so that each array accesses data mapping to its own partition in the array tile [62].

2.2.2 Array padding

Even when choosing nonconflicting tiles, performance of tiling may suffer for certain array dimensions, particularly powers of two. In such cases, non-conflicting tiles result where one or more dimensions are pathologically small, reducing the benefits of tiling. One possible solution is to use padding to enable better tile sizes [5, 61]. Padding increases the size of leading array dimensions, increasing the range of non-conflicting tile shapes. It has proven to be very useful for improving tiling for 2D linear algebra codes [63].

To combine padding with tile size selection, we need only test a small set of pads. Our algorithm (PAD) increases each lower dimension by 0–16 elements, generating tile sizes for each padded array. The version with the best tile according to our cost model is selected. For 2D tiles, this algorithm is both inexpensive and effective [63]. For 3D tiles we need to evaluate a much larger space of possible pads, but the compilation time is still relatively low. In our experiments the additional memory required for padding is quite small, around 10% for reasonably large 3D arrays.

2.3 Preliminary results

2.3.1 Evaluation environment

To demonstrate our approach, we use three representative kernels (JACOBI, REDBLACK, SUBK) commonly found in iterative PDE solvers. JACOBI is the simple six-point finite-difference stencil in three dimensions, followed by a loop to copy values between the two arrays. As we saw previously, if group reuse can be exploited for accesses to three different planes in the K dimension, array B only needs to be brought into memory once for each iteration of the time-step loop. REDBLACK performs Gauss-Seidel solves on alternating points. Because accesses have a stride of two elements, exploiting spatial locality can significantly reduce memory references. Both JACOBI and REDBLACK are commonly used in multigrid codes. SUBK forms one-third of Erlebacher, a 3D tridiagonal solver which performs forward and backward substitutions in all three dimensions. Unlike point algorithms such as JACOBI and REDBLACK, SUBK contains relatively little reuse in the K dimension, simply scanning back and forth along each plane. It is thus representative of PDE solvers applying line relaxation along the K dimension, and provides a challenge to locality optimizations.

To compare tiling algorithms, we obtained performance and cache miss rate data for a range of problem sizes, focusing on problems sizes for which tiling can preserve reuse at both cache levels. Tiling targeted the L1 cache, since doing so can improve locality at both cache levels. For JACOBI and SUBK, the problem sizes NxNx32 were used, varying N while fixing the third dimension at 32 to reduce measurement times; this had no impact on tile conflicts. For REDBLACK, NxNxN problem sizes were used, where N varied from 75 to 150. Measurements consisted of timings made on a Sun UltraSparc I and a SGI Origin2000 with multiple R10000 processors, as well as uniprocessor cache simulation results.

2.3.2 Impact on performance

We first measured execution time performance in millions of floating point operations (MFLOPS) on the Sun UltraSparc I. Results for varying problem sizes are shown in Figure 2 for the different tiling optimizations described in Table 1. Table 1 also summarizes these results, giving average improvement in performance. In addition, cache simulation results appear for a 16K L1 and a 512K L2 direct-mapped cache, the configuration present in the UltraSparc I.

These results indicate a substantial improvement in each program as a result of tiling, from 10–23% for JACOBI (Jac.) and SUBK and from 40–61% for REDBLACK (RB). From the performance advantage of PAD in Table 1, we find padding to be essential for obtaining the greatest possible performance and cache miss rate improvement. We observe the cause of this in Figure 2, in which TILE and LRW obtain poor performance for numerous pathological problem sizes. By padding array dimensions, PAD is able to avoid tile self-interference, unlike TILE, and is able to find well-shaped tiles which minimize the cost model, unlike LRW. In further experiments, the PAD algorithm has also been effective, improving by 11% the average performance of the `RESID` subroutine dominating the execution time of the SPEC95/NAS benchmark MGRID.

We also measured performance of JACOBI on a SGI Origin2000 with eight R10000 processors, as shown in Figure 3. The first graph gives performance in MFlops

Alg.	Algorithm Description	Average % Perf. Improvement			Average % Miss Rate					
					L1 Cache			L2 Cache		
		Jac.	Subk	RB	Jac.	Subk	RB	Jac.	Subk	RB
ORIG	Original program				29	9.6	13	4.4	4.0	4.9
TILE	Cubic tile	15	16	40	27	6.5	7.7	3.5	2.8	1.2
LRW	Non-conflicting rectangular tile	10	13	39	27	6.6	5.5	3.3	2.8	1.1
PAD	LRW + padding (between 0–16)	23	20	61	26	6.2	3.8	3.2	2.6	1.1

Table 1 Program versions and performance statistics

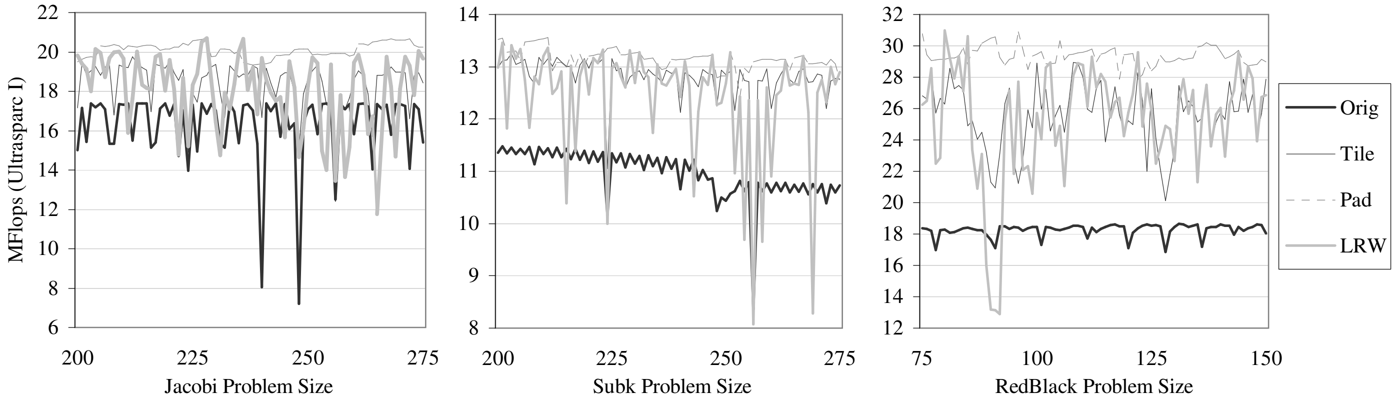

Figure 2 Uniprocessor performance of tiled 3d codes

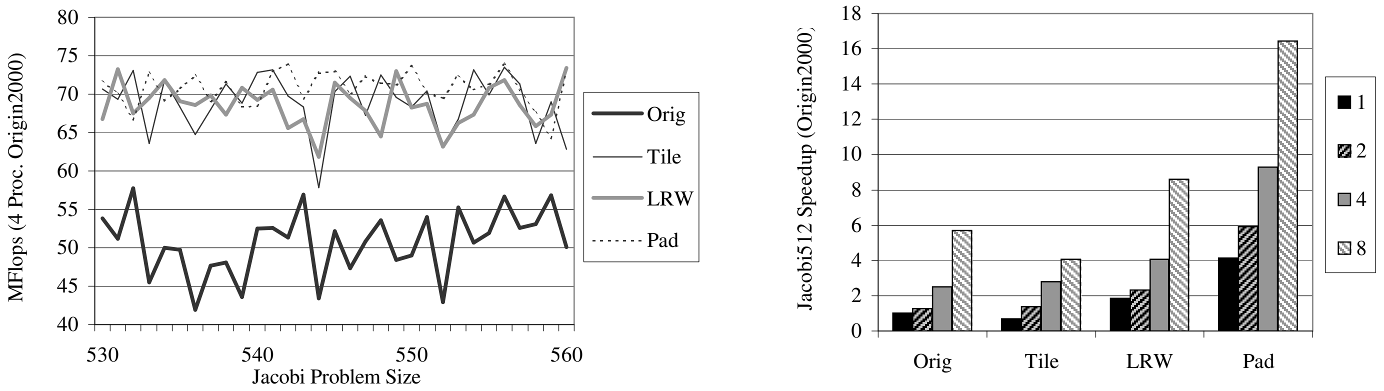

Figure 3 Multiprocessor performance of JACOBI

for 4 processors. Results indicate locality optimizations are also important for parallel codes, with tiling improving performance by about 30%. Since L1 and L2 caches are 2-way set associative, padding is less critical for performance, but still avoids lesser pathological problem sizes such as 544. For powers of 2, padding is still essential, as shown in the second graph, which gives speedups at problem size 512. By combining padding with tiling, we improve performance by over 4 before increasing the number of processors. With 8 processors, the speedup exceeds 16. LRW and especially TILE are much less effective in improving locality since they do not pad array dimensions.

```
// Regular        // Irregular         // Adaptive          // Moldyn
x(N,N)            x(M)                 x(M), idx(M)         x(M),y(M),e(2,N)
do t = 1, time    do t = 1, time       do t = 1, time       do t = 1, time
  do i = 1, N       do i = 1,M           if (change)          if (recalc)
    do j = 1, N       ... = x(idx(i))      idx(...) = ...       edges(...) = ...
      ... = x(j,i)                       do i = 1, M          do i = 1, N
                                           ... = x(idx(i))      d = x(e(1,i))-x(e(2,i))
                                                                force = d**(-7)-d**(-4)
                                                                y(e(1,i)) += force
                                                                y(e(2,i)) += -force
```

Figure 4 Example regular and irregular codes

3 Adaptive Irregular Computations

3.1 Motivation

As computational scientists attempt more complex simulations, codes with irregular memory access patterns are becoming 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 astrophysics and molecular dynamics, data structures are by nature irregular because they model the positions of particles and their interactions. Unfortunately, irregular computations have poor temporal and spatial locality, and do not utilize processor caches efficiently.

Consider the example codes in Figure 4. In the regular code, accesses to x are made to consecutive memory locations (since Fortran is column-major). This spatial locality allows the code to take full advantage of long cache lines to reuse each cache line multiple times before it is flushed from cache. In comparison, in the irregular and adaptive codes, accesses to x are irregular, dictated by the contents of the index array *idx*. For all these codes, irregular memory access patterns mean spatial locality is unlikely to be obtained if the data is larger than the cache.

Researchers recently showed locality can be improved for irregular codes using compiler and run-time data layout and computation reordering transformations [1, 20, 53]. What has not been investigated in detail are locality transformations for *adaptive* irregular codes, where the data access pattern may change over time over the course of the computation. Memory access patterns may change because the computation mesh is selectively refined to speed convergence, or because interaction pairs change as particles move around in space. Figure 4 provides an example of an adaptive code where condition `change` may be satisfied on some iterations of the time-step loop, modifying some elements of the index array `idx1` and changing overall data access patterns.

Because of the overhead of locality optimizations, we expect it will be usually too expensive to reapply transformations every time memory access patterns change. Heuristics must be devised to decide how often transformations should be applied depending on characteristics of the application and underlying machine architecture. Which locality optimization should be applied must also be decided, since transformation overhead becomes more important for adaptive codes. Even more emphasis must be placed on developing efficient locality transformations with low overhead.

3.2 Approach

3.2.1 Locality optimizations

Three computation and data transformations used to improve locality in irregular codes are computation reordering, consecutive packing, and using space-filling curves [1, 20, 53]. Computation reordering works because in many irregular scientific applications, each loop iteration tends to compute results for a single edge or interaction, resulting in accesses to a pair of data items. The pair of accesses may be viewed as a tuple (x, y), where x and y are the addresses of the pair of data items accessed. Rearranging the loop iterations by applying radix (lexicographic) sort to the tuples then yields a quality solution which improves both temporal and spatial locality. Sorting is so effective that data meshes provided by applications writers are frequently presorted, eliminating the need to apply sorting at run time.

In addition to changing the order of memory accesses, the compiler can also reorganize data layout. Using *consecutive packing* (CPACK), data is moved into adjacent locations in the order they are first accessed (i.e., first-touch) by the computation. CPACK improves spatial locality of data references, and can reduce conflict and capacity misses. It is very efficient, but requires the execution of one iteration of the computation. It also does not take into account the structure of the program data. If geometric coordinate information is available, an alternative method is to lay out data in memory according to space-filling curves (e.g., Hilbert, Morton) which attempt to minimize the distance (in memory) between two geometrically close

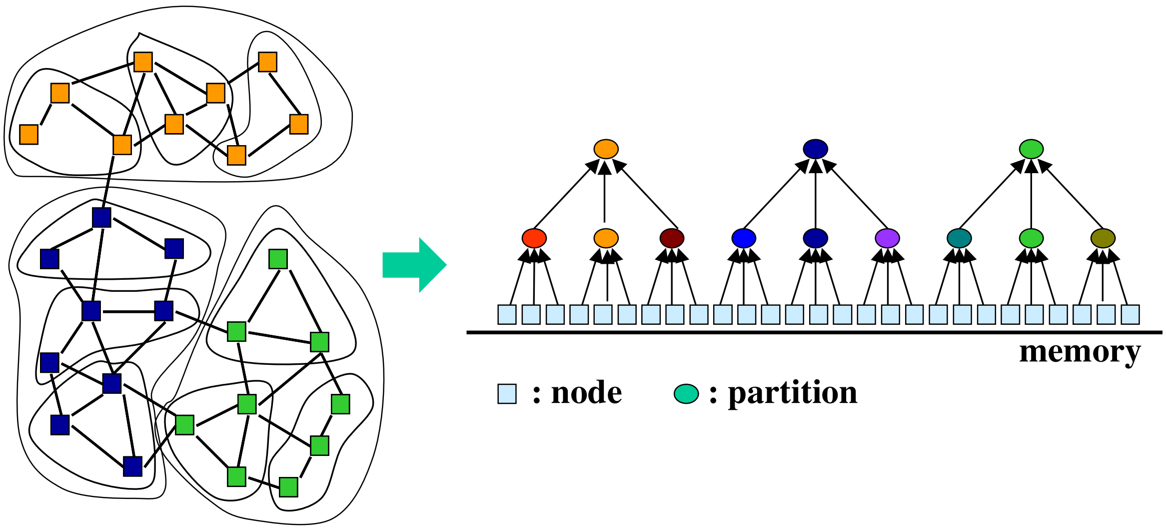

Figure 5 Graph Partitioning Using Hierarchical Clustering (GPART)

points in space [31, 53, 66].

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.* discovered compilers can generate an *inspector* to preprocess memory access patterns, then use this information in an *executor* to perform computation and communication [19]. Inspectors are expensive, but their cost can be amortized over many iterations of the computation if the memory access pattern is stable. Simplified techniques can be used to generate inspectors and executors for locality optimizations [20].

3.2.2 Partitioning algorithms

Instead of consecutive packing and space-filling curves, the compiler and run-time system can *partition* the data and computation. Partitioning algorithms exploit the fact most interactions (and mesh connections) in scientific computations are local, between nearby elements. Nodes and edges accessed in a partition thus have a higher probability of remaining in cache, yielding more reuse. Lexicographical sort is applied after partitioning to improve locality further. Partitioning algorithms were originally used for load-balancing parallel computations [37], VLSI design [2], and database storage [65]. We adapt them for improving cache performance, where overhead of partitioning is much more important.

Partitioning may be guided by either geometric location or the underlying graph structure. Recursive coordinate bisection (RCB) uses geometric coordinate information to recursively split partitions in two along the longest dimension by finding the median data point in that dimension. RCB improves on space-filling curves because partitions are guaranteed to have roughly equal number of data items even for uneven distributions, but is also limited because it relies on geometric coordinate information, which may not be available for some applications. Even if it is available, user annotations may be needed since the compiler may not be able to automatically derive coordinate information for each data item.

An alternative is to compute partitions using the underlying graph structure. An efficient algorithm is to apply multi-level graph partitioning algorithms encapsulated in library packages such as METIS [40, 38, 39]. METIS works by first computing a succession of coarsened graphs (with fewer nodes) which approximate the original graph. Graphs are coarsened by randomly choosing edges in a graph and matching (collapsing) its endpoints to form a node in the new coarsened graph. By only selecting edges connecting unmatched nodes, the number of nodes is reduced roughly by half at each stage. Once the graph reaches a reasonable number of nodes, k-way graph partitioning algorithms are used to split the coarsened graph into k partitions. The partition is then successively mapped back towards the original graph, periodically refining (adjusting) the partition. The multi-level algorithm improves efficiency, since the computationally expensive k-way graph partitioning algorithm can be applied to a much smaller reduced graph.

Both RCB and METIS have relatively high overheads to choose partitions, making them less suitable for adaptive codes. To reduce overhead, we propose another graph-based partitioning algorithm based on the hierarchical clustering (GPART). Figure 5 illustrates how GPART works. GPART collapses several nodes into a single partition in each pass, then a partition is treated as a single node

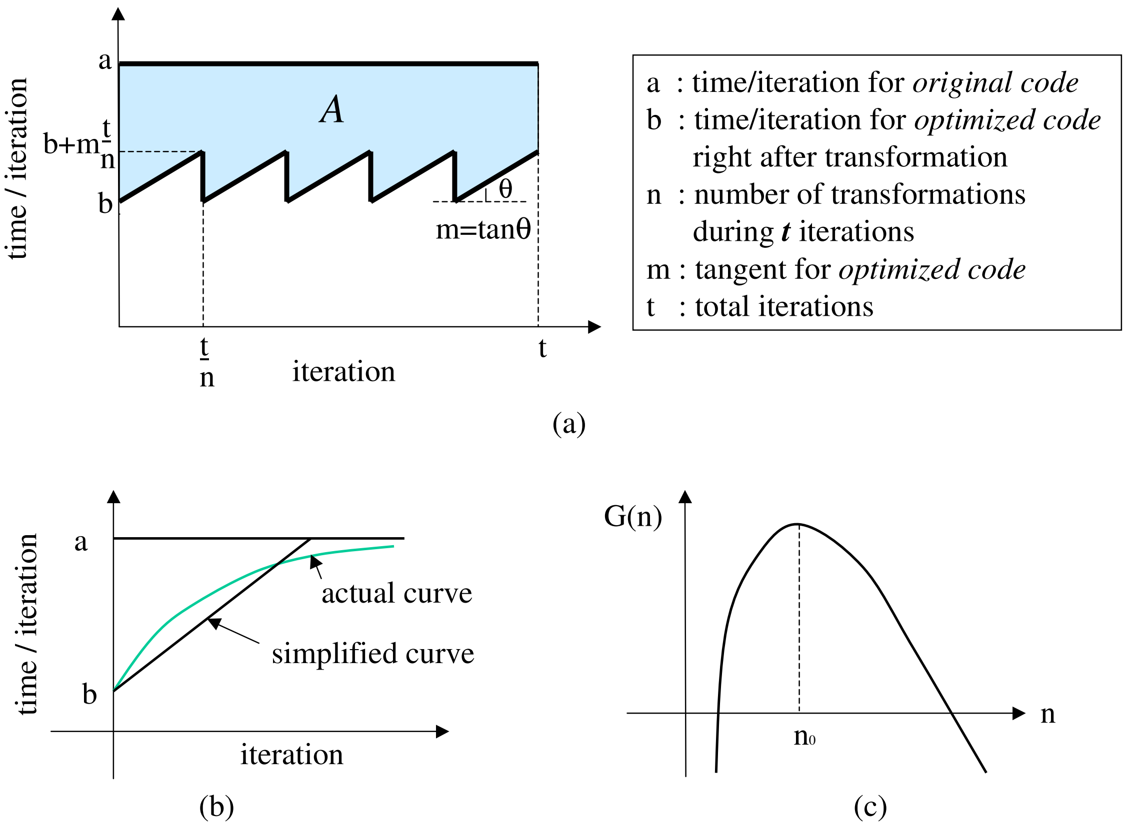

Figure 6 Analytical model for adaptive applications

in the next pass. The same clustering algorithm is repeatedly applied through several passes, keeping a hierarchical structure as depicted in Figure 5. After partitioning with GPART, nodes which are clustered (collapsed) together are placed in the same partition. Overhead is much less, since for the purposes of locality optimizations we can halt after a small fixed number of clustering passes, regardless of the data size. By clustering (collapsing) a larger number of nodes in each step, we can speed up the clustering process significantly.

In addition to using partitions generated by graph clustering to order data, we can also use the hierarchical structure to order nodes within a partition. Subpartitions in the previous pass are stored in consecutive locations, if they belong to the same partition in the current pass. Keeping hierarchical structures in memory helps store related nodes close in memory, reducing cache misses. Since applications visit nodes according to the storage order, maintaining the hierarchical structure of the clustered graph increases the chance of visiting neighbor nodes that have already been accessed by computations for previous nodes. These nodes are more likely to remain in cache, improving performance.

3.2.3 Adaptive algorithms

When compiling adaptive irregular computations for message-passing machines, the compiler must rerun the inspector whenever the connection pattern changes, since new communication may result [32, 27]. Locality optimizations can be more flexible, since changes in the connection pattern degrade performance, but do not affect the legality. In addition, performance is affected by the amount of data changed, not the number of times it is changed. Locality transformations thus do not need to be repeated each time the connection pattern changes, only when it is profitable.

Choosing when data should be repartitioned is difficult, since the impact on performance is hard to judge. The overhead of partitioning must also be estimated and be taken into account. To guide locality optimizations for adaptive codes, we make the simplifying assumption that performance degradation is roughly proportional to the fraction of data changed. Experiments seem to indicate this assumption is roughly correct, as we show later. The benefit from locality optimizations can then be calculated as follows.

Consider Figure 6(a). The y-axis displays the execution time to perform each time step of an adaptive irregular code. We assume a constant fraction of data accesses

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

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

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

Table 2 Analytical model for calculating adaptation frequency

is changed each iteration, but locality optimizations are applied only at intervals. The execution time for each iteration will thus steadily increase, dropping whenever data is repartitioned, yielding the sawtooth pattern. The benefit of locality optimization is then the shaded area above the sawtooth but below the time taken by the original unoptimized code. The desirable interval between applying transformations can be computed by maximizing the shaded area while taking the overhead of optimizations into account.

Our model is only approximate, since performance degradations due to data access changes is not linear, but curved as shown in Figure 6(b). Using our simplifying assumptions, we can calculate the low-cost repartitioning by taking a derivative and taking its maximum, as in Figure 6(c). The frequency calculated is shown in Table 2.

We experimentally validated our cost model by comparing with actual timings for different transformation frequencies. We find actual improvements are larger than predicted, due to our simplifying assumptions. However, the model is sufficiently precise to pick the best number of times to apply locality transformations. Since our cost model produces a floating point number for n_0, we have to pick an integer number of transformations for actual use. Experimental result showed either rounding up or down produce nearly same performance.

3.3 Preliminary results

3.3.1 Evaluation environment

To evaluate locality transformations, we ran a number of experiments on a DEC Alpha multiprocessor with four 21064 processors for MOLDYN, a program abstracted from the non-bonded force calculation in CHARMM, a key molecular dynamics application used at NIH to model macromolecular systems. MOLDYN maintains an edge list representing interactions between pairs of molecules. Since the strength of interactions between molecules drops with increasing distance, only molecules within a cutoff distance of each other are assumed to interact. The main computation kernel, shown in Figure 4, iterates over all interactions between molecules, computing a single force which is applied to both interacting molecules. At intervals the interaction list is updated, as the position of molecules changes over time.

We used a variety of input meshes to test the impact of locality optimizations for different data placements. FOIL, SUB, and AUTO are 3D meshes of a parafoil, submarine, and GM Saturn automobile, respectively, while MOL1 and MOL2 are meshes derived from interactions between semi-uniformly placed molecules from MOLDYN using a 1.5 angstrom cutoff radius. Problem sizes ranged from 130K–450K nodes and 1M–4M edges.

3.3.2 Impact on static codes

We begin by examining the impact of locality transformations on MOLDYN when data access patterns are static. Figure 7 presents the sequential and parallel execution times for each locality optimization. We see that locality transformations can improve the execution time of irregular scientific programs from 10–38%, with benefits increasing with large input data sets. RCB and METIS achieving the best performance (but also the highest overhead).

We found locality transformations are more successful when applied to a parallel version of MOLDYN. Percentage improvements over original parallel programs are from 20–60% whereas improvements in sequential programs are from 10–40%. Locality is more important factor in

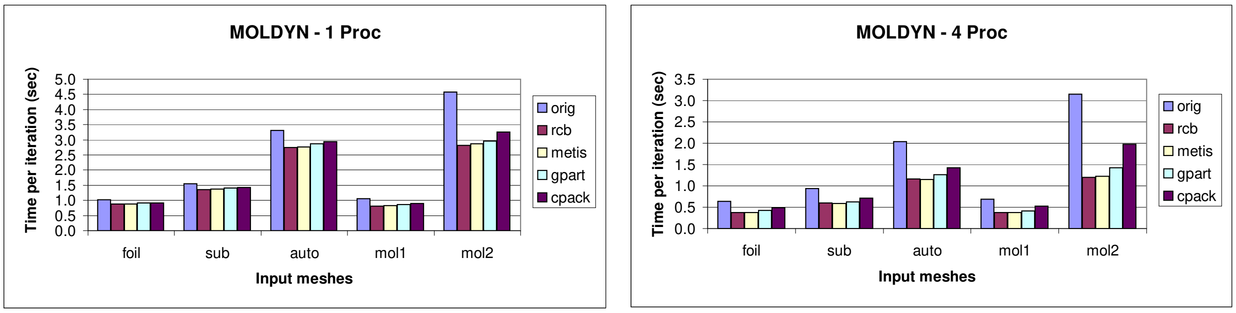

Figure 7 Impact on execution time of static codes

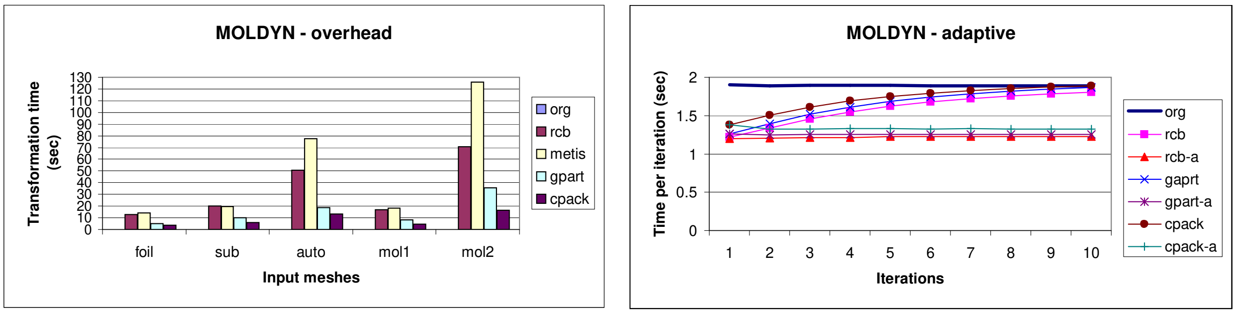

Figure 8 Overhead of optimizations, impact of adaptation

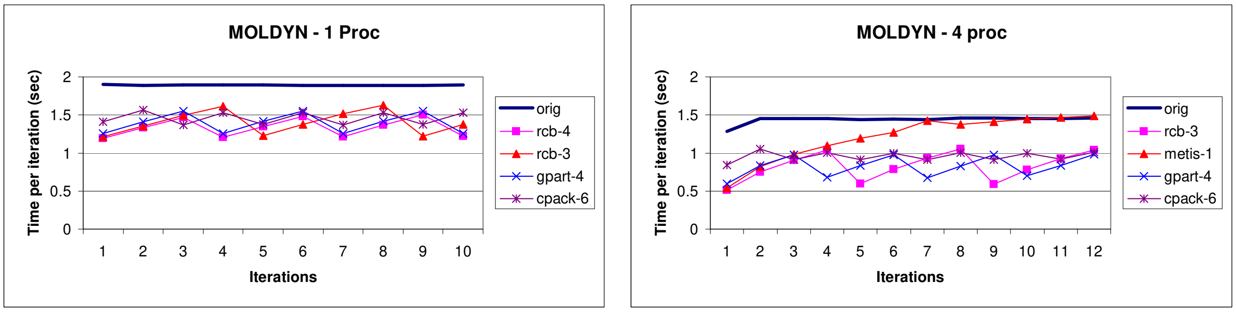

Figure 9 Impact on execution time of adaptive codes

parallel programs than in sequential programs. Slight differences in effectiveness of transformations yield bigger differences in parallel programs.

The first graph in Figure 8 shows overhead of each transformation in each input mesh. Overhead is highest for RCB and METIS, moderate for GPART, and low for CPACK. For short running programs, GPART and CPACK will be proper choices whereas RCB and METIS will achieve more improvements for long running programs with many iterations. Fortunately scientific applications tend to execute for many time steps, allowing overhead to be amortized.

3.3.3 Impact on adaptive codes

To evaluate the effect of adaptivity on locality optimizations, we created an adaptive version of MOLDYN by randomly swapping 10% of the nodes after every 20 iterations. We then evaluated its performance for the MOL1 input mesh.

We first examined the effect of applying locality optimizations once, after every change, or not at all. Results are shown in the second graph in Figure 8. The x-axis marks the passage of time in the computation in groups of 20 iterations. The y-axis measures the execution time per iteration of the time step, excluding optimization overhead. Each data series represents a different locality optimization. ORIG is the original program. RCB, METIS, GPART, and CPACK represent versions of the program where locality optimizations are applied once at the beginning of program execution. In comparison, RCB-a, METIS-a, GPART-a, and CPACK-a represent adaptive versions of each program where locality optimizations are applied whenever connection patterns are changed (overhead of locality transformations are not included).

Results show that without reapplying locality transformations, all optimized versions degrade in performance and eventually match the performance of the unoptimized program. In comparison, reapplying partitioning algorithms after changes can preserve the performance benefits of locality. It appears that the degradation in performance is generally proportional to the fraction of data changed, so our cost model can be effective.

We then used our cost model to predict how often to apply locality optimizations for each algorithm. Results are shown in Figure 9. Each data series represents a version of the program. RCB-3 means data is repartitioned 3 times during the whole 10 sets of iterations, and is selected according to our cost model. Similarly, RCB-4, METIS-1, GPART-4 and CPACK-6 mean data is repartitioned 4, 1, 4 or 6 times, respectively.

For the sequential adaptive code, once overhead of transformations is included we found program performance improved 18.6%, -6.6%, 21.9%, and 21.9% respectively for RCB, METIS, GPART, and CPACK. Improvements of 14.5%, -20% 20.2%, and 19.5% were achieved on the 4-processor SGI Origin2000 for the same optimizations. Results show all transformations but METIS were able to yield improvements. Since the overhead of METIS is greater than others, it is only applicable to long running and rather static codes. The higher overhead transformation needs to be applied the less frequently to achieve maximum gain. CPACK the lowest overhead transformation, is applied 6 times and GPART the medium overhead transformation, is applied 4 times and RCB the high overhead transformation, is applied 3 times to yield maximum achievable gain.

Besides the frequency of transformation application, our cost model picks GPART and CPACK as appropriate transformations to yield maximum gain, which is correct considering the experimental results. GPART with 4 times transformations and CPACK with 6 times transformations yield maximum gain over other transformations.

4 Related Work

There has been much work on improving locality in scientific applications [5, 22, 23, 24, 35, 36, 52, 61, 62, 71, 76] as mentioned in the introduction. Here we will focus on the work which is most relevant to our proposed research.

A number of researchers have investigated tiling as a means of exploiting reuse. Lam, Rothberg, Wolf show conflict misses can severely degrade the performance of tiling [46]. Wolf and Lam analyze temporal and spatial reuse, and apply tiling when necessary to capture outer loop reuse [76], Coleman and McKinley select rectangular non-conflicting tile sizes [16] while others focus on using a portion of cache [77]. Temam *et al.* analyze the program to determine whether a tile should be copied to a contiguous buffer [72].

Kodukula *et al.* present a technique called data shackling which is very effective at generating tiled code for complex linear algebra codes [42]. Sarkar describes data locality optimizations used in the IBM XL Fortran compilers, including loop transformations and tiling [64]. Mitchell *et al.* discussed the interactions of multi-level tiling for several goals, such as cache, TLB, and parallelism [54]. They found explicitly considering multiple levels of the memory hierarchy (cache and TLB) led to the choice of compromise tile sizes which can yield significant improvements in performance.

Song and Li extending tiling techniques to handle multiple loop nests [67]. In many cases their technique can exploit reuse across multiple iterations of the time-step loop, yielding major improvements. Their technique does not currently apply to 3D arrays, and they concentrate on only L2 cache since L1 cache is too small to provided reuse for their necessarily large tiles. We intend to investigate combining our tiling techniques with theirs. Rüde *et al.* explored program transformations for improving the performance numerical algorithms on hierarchical memories [7, 68]. They applied transformations by hand based on domain-specific applications knowledge. We attempt to develop similar techniques which can be automated in a compiler.

In comparison with all these researchers, we attempt to find compiler and run-time solutions to locality problems specific to 3D iterative solvers. We point out several difficult problems and propose solutions.

Researchers examining memory system effects for parallel machines have mostly focused on program transformations to eliminate false sharing and co-locate data and computation. Torrie *et al.* investigated cache behavior for compile-parallelized programs and found memory costs were responsible for over a third of overall execution time [74]. Kennedy and McKinley integrate loop transformations which target parallelism and the memory hierarchy [41]. The impact of false sharing has been studied extensively [6, 21, 26, 47, 73]. Program transformations to co-locate data and computation can reduce true sharing misses, and is a topic of active interest to many researchers [4, 48, 40, 75].

Jeremiassen and Eggers automatically eliminate false sharing in explicitly parallel programs in a compiler [34]. Cierniak and Li examined combining array transpose and loop transformations to improve locality for parallel programs [15]. Amarasinghe *et al.* demonstrated the utility of array reindexing for parallel applications [3]. Bugnion *et al.* presented page coloring as a technique to improve cache utilization on multiprocessors [9].

Many researchers have investigated support for irregular computations. Irregular computations generally perform reductions. The importance of identifying and parallelizing reductions in scientific applications is well established [30, 49, 59, 70]. Researchers have investigated both efficient run-time [19, 32] and compiler [13, 28, 50, 60] support. Most research on compiling irregular codes has been based on the CHAOS run-time system [32]. Compilers can also automatically generate calls to CHAOS run-time routines [14, 28, 29]. Researchers have also examine efficiently supporting irregular computations on software DSMs [8, 50, 55], including hybrid regular-irregular codes [11, 44]. For certain irregular codes, spatial locality can be exploited by considering irregular data as composed of sets of intervals instead of individual elements [45, 69]. Computation and data for irregular reductions may be partitioned so they perform writes separate sections of an array, eliminating buffering and synchronization on multiprocessors [27].

Domain decomposition algorithms such as RCB have been used by researchers for load balancing, reducing interprocessor communication, and many other uses [39, 57, 66]. Ou *et al.* found spectral partitioning using eigenvectors to be effective but expensive, while space-filling curves achieved similar performance with much less overhead [58].

Das *et al.* investigated data/computation reordering for parallel unstructured euler solvers [18]. In addition to communication optimizations using the PARTI/CHAOS run-time system, they applied the Reverse Cuthill-McGee method [17] for data reordering and lexicographical sort for computation reordering, achieving two and three-fold improvements for parallel performance on an Intel iPSC/860.

Al-Furaih and Ranka studied using METIS and breadth-first-search to partition irregular computations and improve locality, but did not combine computation reordering or compare against other techniques [1]. Ding and Kennedy explored applying dynamic copying (packing) of data elements based on loop traversal order, and show major improvements in performance [20]. They were able to automate most of their transformations in a compiler.

Mellor-Crummey *et al.* use a geometric partitioning algorithm based on space-filling curves to map multidimensional data to memory [53]. They show significant improvements for large randomly generated data sets for MOLDYN. However, space-filling curves cannot guarantee evenly balanced partition like RCB when data is distributed unevenly.

Nicol and Saltz investigated dynamic computation remapping as a means of reducing load imbalance in adaptive parallel computations [56]. Analytically modeling average processor idle time, they established a policy that dynamically decides when to apply remapping by measuring execution time per time step. Computation is remapped when the first local minima in the idle time cost is detected. In simulation experiments, they show their algorithm is near optimal and outperforms the best static remapping frequency. Our current model can also be extended to a dynamic prediction model, monitoring performance and overhead changes.

Compared to existing research on locality optimizations for irregular codes, we propose a suite of algorithms which differ in precision and overhead, including new algorithms designed for low overhead and high precision. These algorithms are vital for adaptive codes, where low overhead is crucial. We test effects on both randomized graphs and meshes from real applications for a number of computation kernels.

5 Acknowledgments

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

References

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

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

[3] J. Anderson, S. Amarasinghe, and M. Lam. Data and computation transformation for multiprocessors. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, July 1995.

[4] J. Anderson and M. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, Albuquerque, NM, June 1993.

[5] D. Bacon, J.-H. Chow, D.-C. Ju, K. Muthukumar, and V. Sarkar. A compiler framework for restructuring data declarations to enhance cache and TLB effectiveness. In *Proceedings of CASCON'94*, Toronto, Canada, October 1994.

[6] W. Bolosky and M. Scott. False sharing and its effect on shared memory performance. In *Proceedings of the USENIX Symposium on Experiences with Distributed and*

Multiprocessor Systems (SEDMS IV), San Diego, CA, September 1993.

[7] T. Bonk and U. Rüde. Performance analysis and optimization of numerically intensive programs. SFB Bericht 342/26/92 A, Institut für Informatik, TU München, November 1992.

[8] P. Brezany, M. Gerndt, V. Sipkova, and H. Zima. SUPERB support for irregular scientific computations. In *Proceedings of the 1992 Scalable High Performance Computing Conference*, Williamsburg, VA, April 1992.

[9] E. Bugnion, J. Anderson, T. Mowry, M. Rosenblum, and M. Lam. Compiler-directed page coloring for multiprocessors. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, Boston, MA, October 1996.

[10] S. Carr and K. Kennedy. Compiler blockability of numerical algorithms. In *Proceedings of Supercomputing '92*, Minneapolis, MN, November 1992.

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

[12] R. Chandra, D.-K. Chen, R. Cox, E. Maydan, N. Nedeljkovic, and J. Anderson. Data distribution support on distributed shared memory multiprocessors. In *Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation*, Las Vegas, NV, June 1997.

[13] 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.

[14] C. Chase, K. Crowley, J. Saltz, and A. Reeves. Compiler and runtime support for irregularly coupled regular meshes. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, Washington, DC, July 1992.

[15] M. Cierniak and W. Li. Unifying data and control transformations for distributed shared-memory machines. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, La Jolla, CA, June 1995.

[16] S. Coleman and K. S. McKinley. Tile size selection using cache organization and data layout. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, La Jolla, CA, June 1995.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] S. J. Eggers and T. E. Jeremiassen. Eliminating false sharing. In *Proceedings of the 1991 International Conference on Parallel Processing*, St. Charles, IL, August 1991.

[22] J. Ferrante, V. Sarkar, and W. Thrash. On estimating and enhancing cache effectiveness. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing, Fourth International Workshop*, Santa Clara, CA, August 1991. Springer-Verlag.

[23] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformation. *Journal of Parallel and Distributed Computing*, 5(5):587–616, October 1988.

[24] 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.

[25] S. Ghosh, M. Martonosi, and S. Malik. Precise miss analysis for program transformations with caches of arbitrary associativity. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, San Jose, CA, October 1998.

[26] E. Granston and H. Wishoff. Managing pages in shared virtual memory systems: Getting the compiler into the game. In *Proceedings of the 1993 ACM International Conference on Supercomputing*, Tokyo, Japan, July 1993.

[27] 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.

[28] 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.

[29] 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.

[30] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Preliminary experiences with the Fortran D compiler. In *Proceedings*

of Supercomputing '93, Portland, OR, November 1993.

[31] 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.

[32] 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.

[33] F. Irigoin and R. Triolet. Supernode partitioning. In *Proceedings of the Fifteenth Annual ACM Symposium on the Principles of Programming Languages*, San Diego, CA, January 1988.

[34] T. Jeremiassen and S. Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, July 1995.

[35] 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.

[36] M. Kandemir, J. Ramanujam, and A. Choudhary. A compiler algorithm for optimizing locality in loop nests. In *Proceedings of the 1997 ACM International Conference on Supercomputing*, Vienna, Austria, July 1997.

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

[38] 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.

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

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

[41] K. Kennedy and K. S. McKinley. Optimizing for parallelism and data locality. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, Washington, DC, July 1992.

[42] I. Kodukula, N. Ahmed, and K. Pingali. Data-centric multilevel blocking. In *Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation*, Las Vegas, NV, June 1997.

[43] I. Kodukula and K. Pingali. An experimental evaluation of tiling and shacking for memory hierarchy management. In *Proceedings of the 1999 ACM International Conference on Supercomputing*, Rhodes, Greece, June 1999.

[44] A. Lain and P. Banerjee. Techniques to overlap computation and communication in irregular iterative applications. In *Proceedings of the 1994 ACM International Conference on Supercomputing*, Manchester, England, July 1994.

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

[46] M. Lam, E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, Santa Clara, CA, April 1991.

[47] H. Li and K. Sevcik. NUMACROS: Data parallel programming on NUMA multiprocessors. In *Proceedings of the USENIX Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS IV)*, San Diego, CA, September 1993.

[48] J. Li and M. Chen. The data alignment phase in compiling programs for distributed-memory machines. *Journal of Parallel and Distributed Computing*, 13(2):213–221, October 1991.

[49] 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.

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

[51] N. Manjikian and T. Abdelrahman. Fusion of loops for parallelism and locality. *IEEE Transactions on Parallel and Distributed Systems*, 8(2):193–209, February 1997.

[52] 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.

[53] 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.

[54] N. Mitchell, L. Carter, J. Ferrante, and K. Högstedt. Quantifying the multi-level nature of tiling interactions. In *Proceedings of the Tenth Workshop on Languages and Compilers for Parallel Computing*, Minneapolis, MN, August 1997.

[55] 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.

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

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

[58] C. Ou, M. Gunwani, and S. Ranka. Architecture independent locality-improving transformations of computational graphs embedded in k-dimensions. In *Proceedings of the 1995 ACM International Conference on Supercomputing*, Barcelona, Spain, July 1995.

[59] 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.

[60] 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.

[61] 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.

[62] G. Rivera and C.-W. Tseng. Eliminating conflict misses for high performance architectures. In *Proceedings of the 1998 ACM International Conference on Supercomputing*, Melbourne, Australia, July 1998.

[63] G. Rivera and C.-W. Tseng. A comparison of compiler tiling algorithms. In *Proceedings of the 8th International Conference on Compiler Construction (CC'99)*, Amsterdam, The Netherlands, March 1999.

[64] 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.

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

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

[67] 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.

[68] L. Stals, U. Rüde, C. Weiss, and H. Hellwagner. Data local iterative methods for the efficient solution of partial differential equations. In *Proceedings of the Eighth Biennial Computational Techniques and Applications Conference*, Adelaide, Australia, September 1997.

[69] E. Su, A. Lain, S. Ramaswamy, D. J. Palermo, E. W. Hodges IV, and P. Banerjee. Advanced compilation techniques in the PARADIGM compiler for distributed-memory multicomputers. In *Proceedings of the 1995 ACM International Conference on Supercomputing*, Barcelona, Spain, July 1995.

[70] 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.

[71] O. Temam, C. Fricker, and W. Jalby. Cache interference phenomena. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement & Modeling Computer Systems*, Santa Clara, CA, May 1994.

[72] O. Temam, E. Granston, and W. Jalby. To copy or not to copy: A compile-time technique for assessing when data copying should be used to eliminate cache conflicts. In *Proceedings of Supercomputing '93*, Portland, OR, November 1993.

[73] J. Torrellas, M. Lam, and J. Hennessy. False sharing and spatial locality in multiprocessor caches. *IEEE Transactions on Computers*, 43(6):651–663, June 1994.

[74] E. Torrie, C.-W. Tseng, M. Martonosi, and M. Hall. Evaluating the impact of advanced memory systems on compiler-parallelized codes. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, Limassol, Cyprus, June 1995.

[75] S. Wholey. Automatic data mapping for distributed-memory parallel computers. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, Washington, DC, July 1992.

[76] 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.

[77] M. E. Wolf, D. Maydan, and D.-K. Chen. Combining loop transformations considering caches and scheduling. In *Proceedings of the 29th IEEE/ACM International Symposium on Microarchitecture*, Paris, France, December 1996.

[78] M. J. Wolfe. More iteration space tiling. In *Proceedings of Supercomputing '89*, Reno, NV, November 1989.