Improving Memory Hierarchy Performance for Irregular Applications Using Data and Computation Reorderings*

John Mellor-Crummey†, David Whalley‡, Ken Kennedy†

† Department of Computer Science, MS 132
Rice University
6100 Main
Houston, TX 77005
{johnmc,ken}@cs.rice.edu

‡ Computer Science Department
Florida State University
Tallahassee, FL 32306-4530
whalley@cs.fsu.edu
phone: (850) 644-3506

Abstract

The gap between CPU speed and memory speed in modern computer systems is widening as new generations of hardware are introduced. Loop blocking and prefetching transformations help bridge this gap for regular applications; however, these techniques aren't as effective for irregular applications. This paper investigates using data and computation reorderings to improve utilization of multi-level memory hierarchies for irregular applications. We evaluate the impact of reordering on data reuse at different levels in the memory hierarchy. We focus on coordinated data and computation reordering based on space-filling curves and we introduce a new architecture-independent multi-level blocking strategy for irregular applications. For two particle codes we studied in detail, the most effective reordering reduced overall execution time by a factor of two for the first and a factor of four for the second. Preliminary experience with a third code derived from a large unstructured mesh application showed a factor of two improvement in primary cache misses when using a space-filling curve based ordering of data and computation compared to a random ordering.

Keywords: Memory Hierarchy Optimization, Data Reordering, Computation Reordering, Space-Filling Curves, Multi-Level Blocking

---

* This research was supported in part the National Science Foundation under cooperative agreement CCR-9120008 and grants EIA-9806525 and CCR-9904943, the Department of Energy's Accelerated Strategic Computing Initiative under research subcontract B347884, and by DARPA and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-1-0159. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies or endorsements, either expressed or implied, of DARPA and Rome Laboratory or the U.S. Government.

A preliminary version of this research was described in the *Proceedings of the ACM International Conference on Supercomputing*, June 1999, under the title "Improving Memory Hierarchy Performance for Irregular Applications."

1. Introduction

The gap between CPU speed and memory speed is increasing rapidly as new generations of computer systems are introduced. Multi-level memory hierarchies are the standard architectural design used to address this memory access bottleneck. As the gap between CPU speed and memory speed widens, systems are being constructed with deeper hierarchies. Achieving high performance on such systems requires tailoring the reference behavior of applications to better match the characteristics of a machine's memory hierarchy. Techniques such as loop blocking [1, 2, 3, 4, 5, 6] and data prefetching [4, 7, 8] have significantly improved memory hierarchy utilization for regular applications. A limitation of these techniques is that they aren't as effective for irregular applications. Improving performance for irregular applications is extremely important since large-scale scientific and engineering simulations are increasingly using adaptive irregular methods.

Irregular applications are characterized by patterns of data and computation that are unknown until run time. In such applications, accesses to data often have poor spatial and temporal locality, which leads to ineffective use of a memory hierarchy. Improving memory system performance for irregular applications requires addressing problems of both latency and bandwidth. Latency is a problem because poor temporal and spatial reuse result in elevated cache and translation lookaside buffer (TLB) miss rates. Bandwidth is a problem because indirect references found in irregular applications tend to have poor spatial locality. Thus, when accesses cause blocks of data to be fetched into various levels of the memory hierarchy, items within a block are either referenced only a few times or not at all before the block is evicted due to conflict and/or capacity misses, even though these items will be referenced later in the execution.

One strategy for improving memory hierarchy utilization for such applications is to reorder data dynamically at the beginning of a major computation phase. This approach assumes that the benefits of increased locality through reordering will outweigh the cost of the data movement. Data reordering can be particularly effective when used in conjunction with a compatible computation reordering. The aim of data and computation reorderings is to decrease latency and more effectively utilize bandwidth at different levels of the memory hierarchy by (1) increasing the probability that items in the same block will be referenced close together in time and (2) increasing the probability that items in a block will be reused more extensively before the block is replaced. This paper explores strategies for data reordering and computation reordering along with integrated approaches to evaluate how effectively they improve memory hierarchy utilization on machines with multi-level memory hierarchies. We also introduce multi-level blocking as a new computation reordering strategy for irregular applications.

A common class of irregular applications considers particles or mesh elements in spatial neighborhoods. Figure 1 shows a simple n-body simulation that we use as an example throughout the paper. Although we explain our techniques in terms of this example, they apply more broadly to other types of irregular applications, especially those that simulate physical systems in two or more dimensions. Our sample n-body simulation considers particles within a defined volume, represented here as a two dimensional area for simplicity. Each particle interacts with other particles within a specified cutoff radius. Particles $\mathbf{P_j}$ and $\mathbf{P_k}$ are shown in the physical space along with a cutoff radius surrounding each particle. Interactions are between a particle and other particles within its cutoff radius. The particles can change positions over time in the physical space of the problem. To adapt to these changes, the application periodically recalculates which particles can interact.

Figure 1 also shows the problem data space for this sample application. The information for each particle includes its coordinates in the physical space and other attributes, such as velocity and the force exerted upon it. The interaction list indicates the pairs of particles that can interact. The data for the particles is irregularly accessed since the order of access is determined by the interaction list. The number of interactions is typically much greater than the number of particles. Note that there are many possible variations on how the data space can be organized.

The remainder of this paper has the following organization. First, we introduce related work that uses blocking, data reordering, and space-filling curves to improve the memory hierarchy performance of applications. Second, we outline the general data and computation reordering techniques that we consider in this paper. Third, we describe three irregular programs, explain how we manually apply specific combinations of data and computation reordering techniques by inserting calls to library routines, and present the results of applying these techniques on these programs. Finally, we present a summary and conclusions of the paper.

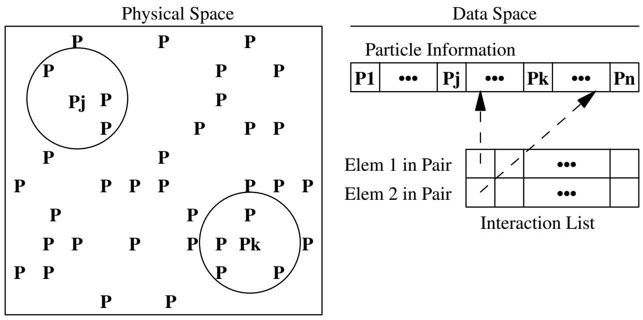

Figure 1: A Classical Irregularly Structured Application

2. Related Work

Blocking for improving the performance of memory hierarchies has been a subject of research for the last few decades. Early papers focused on blocking to improve paging performance [9, 10], but recent work has focused more narrowly on improving cache performance [2, 5, 4, 6]. Techniques similar to blocking have also been effectively applied to improvement of reuse in registers [1]. Most of these methods deal with one level of the memory hierarchy only, although the cache and register techniques can be effectively composed. A recent paper by Navarro *et al.* examines the effectiveness of multi-level blocking techniques on dense linear algebra [11] and a paper by Kodukula *et al.* presents a data-centric blocking algorithm that can be effectively applied to multi-level hierarchies [12].

The principal strategy for improving bandwidth utilization for regular problems, aside from blocking for reuse, has been to transform the program to increase spatial locality. Loop interchange is a standard approach to achieving stride-1 access in regular computations. This transformation has been specifically studied in the context of memory hierarchy improvement by a number of researchers [13, 14].

As described earlier, data reordering can be used to reduce bandwidth requirements of irregular applications. Ding and Kennedy [15] explored compiler and run-time support for a class of run-time data reordering techniques. They examine an access sequence and use it to greedily reorder data aiming to increase spatial locality as the access sequence is traversed. They consider only a very limited form of computation reordering in their work. Namely, for computations expressed in terms of an access sequence composed of tuples of particles or objects, they apply a grouping transformation to order tuples in the sequence to consider all interactions involving one object before moving to the next. Das et al. [16] applied this same computation reordering in an unstructured mesh application. Ding and Kennedy [15] did not specifically consider reordering for multi-level memory hierarchies although they proposed a strategy for grouping information about data elements to increase spatial locality, which has the side effect of improving TLB performance. In our work, we applied this grouping strategy before taking baseline performance measurements. Also, we evaluate Ding and Kennedy's dynamic strategy, first-touch reordering, along with other strategies.

In recent years, space-filling curves have been used for managing locality for both regular and irregular applications. A space-filling curve for some finite space of d dimensions ($d \geq 2$) is a continuous, non-smooth curve that passes arbitrarily close to every point. Each point in a d-dimensional space can be mapped to the nearest position along a 1-dimensional space-filling curve by applying a sequence of bit-level logical operations to its d-dimensional coordinates. A Hilbert space-filling curve is one such mapping. Figure 2 shows a fifth-order Hilbert curve in two dimensions. An important property of this curve, is that its recursive structure preserves locality: points close in the original multi-dimensional space are typically close along the curve. In particular, the successor of any point along the curve is one of its adjacent neighbors along one of the coordinate dimensions.[1] Figure 3 shows a Morton curve. Like a Hilbert curve, a Morton curve also has a recursive structure; however, lattice points along a Morton curve are not always adjacent neighbors, which results in a slightly lower degree of locality. Morton curves are popular because they are simple to compute: a point's position along the curve is determined by a bitwise interleaving of its coordinates.

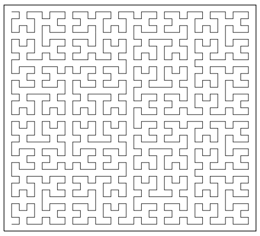

Figure 2: Fifth-order Hilbert curve through 2 dimensions.

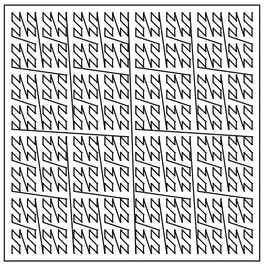

Figure 3: Fifth-order Morton curve through 2 dimensions.

Space-filling curves or related ordering techniques [19] have been used to partition data and computation among processors in parallel computer systems. They have been applied in problem domains that include n-body problems [20, 19], graph partitioning [21], and adaptive mesh refinement [22]. Ordering data elements by their position along a space-filling curve and assigning each processor a contiguous range of elements of equal (possibly weighted) size is a fast partitioning technique that tends to preserve physical locality in the problem domain. Namely, data elements close together in physical space tend to be in the same partition. Ou *et al.* [21] present results that show that other methods, such as recursive spectral bisection and reordering based on eigenvectors, can produce partitionings with better locality according to some metrics;

[1] For more details about the history of space-filling curves, the types of curves, their construction, and their properties, see Sagan [17] and Samet [18].

however, the differences among the methods (in terms of the locality of partitionings produced) diminished when these methods were applied to larger problem sizes. Also, they found that using space-filling curves to compute reorderings is orders of magnitude faster than the other methods they studied.

Several researchers have proposed using recursive data layouts for computation on dense matrices. To improve locality for matrix multiplication, Thottethodi *et al.* [23] explored ordering matrix elements by their position along a space-filling curve rather than typical row-major or column-major orderings, and Frens Wise [24] proposed recursive matrix layouts based on quad trees. The hierarchical locality resulting from these recursively defined orderings is a good match for divide-and-conquer matrix algorithms.

Several researchers have investigated strategies for improving memory hierarchy performance for algorithms on graphs and unstructured meshes. Al-Furaih and Ranka [25] used a simple breadth-first node numbering. Das *et al.* [16] applied breadth-first traversal strategy known as Reverse Cuthill-McKee to order elements in an unstructured mesh to improve locality. This reordering technique was developed by George [26] for a different purpose: bandwidth and profile minimization of sparse matrices. George's strategy was a refinement of a breadth-first ordering technique developed by Cuthill and McKee [27]. The Cuthill-McKee and Reverse Cuthill-McKee orderings use an adjacency list representation of an undirected graph and renumber graph nodes using a breadth-first traversal in which all unnumbered neighbors of a node x are added to a FIFO queue of nodes to be numbered by order of increasing degree. Sloan [28] developed a related but more sophisticated reordering strategy. First, he more carefully selects the first node in the ordering to yield orderings with narrower level structure. Then, at each step instead of simply adding nodes nodes to the queue in order of increasing degree, he uses priorities that are a function of distance to the end node as well as node degree. A principal application of Sloan's method is for ordering elements in a finite element mesh for efficient computation using frontal solution techniques.

Al-Furaih and Ranka [25] also studied the impact of data reorderings based on Hilbert curves for reducing the execution time of particle-in-cell codes. Our work differs from theirs principally in that we consider coordinated data and computation reordering, whereas they consider data reorderings exclusively.

3. Data Reordering Approaches

A data reordering involves changing the location of the elements of the data, but not the order in which these elements are referenced. Consider again the data space shown in Figure 1. A data reordering would changes the order of elements within the particle information vector and updates the interaction list to point to the new particle locations. By placing data elements near one another if they are referenced together, data reordering approaches can improve spatial locality. Temporal locality would not be affected since the order in which data elements are accessed remains unchanged. The following subsections describe the data reordering approaches investigated.

3.1. First Touch Data Reordering

First-touch data reordering is a greedy approach for improving spatial locality of irregular references [15]. Consider Figure 4, which represents the data space in Figure 1 before and after data reordering using the first-touch approach. A linear scan of the interaction list is performed to determine the order in which the particles are first touched. The particle information is reordered and the indices in the interaction list now point to the new positions of the particles. However, the order in which the particles are referenced is unchanged. The idea is that if two particles are referenced near each other in time in the interaction list, then they should be placed near each other in the particle list. An advantage of first-touch data reordering is that the approach is simple and can be accomplished in linear time. A disadvantage is that the computation order (interaction list in Figure 4) must be known before reordering can be performed.

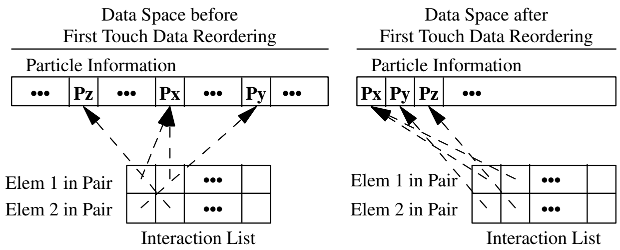

Figure 4: Data Reordering Using a First Touch Approach

3.2. Space Filling Curve Data Reordering

Figure 5 shows an example data space before and after data reordering using a space-filling curve. Assume that the first three particles on the curve are $\mathbf{P_x}$, $\mathbf{P_y}$, and $\mathbf{P_z}$. To use a *k*-level space-filling curve to reorder data for particles whose coordinates are represented with real numbers, several steps are necessary. First, each particle coordinate must be normalized into a *k*-bit integer. The integer coordinates of each particle's position are converted into a position on the space-filling curve by a sequence of bit-level logical operations. The particles are then sorted into ascending order by their position on the curve. Sorting particles into space-filling curve order tends to increase spatial locality. Namely, if two particles are close together in physical space, then they tend to be nearby on the curve. One advantage of using a space-filling curve for data reordering is that data can be reordered prior to knowing the order of the computation. This allows some computation reorderings to be accomplished with no overhead. For instance, if the data is reordered prior to establishing the access order (e.g. an interaction list), then the access order will be affected if it is established as a function of the order of the data. A potential disadvantage of using space-filling curves is that it is possible that the reordering may require more overhead than a first-touch reordering due the sort of the particle information. Of course, the relative overheads of the two approaches would depend on the number of data elements versus the number of references to the data.

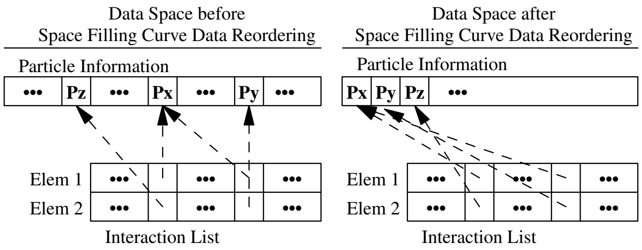

Figure 5: Data Reordering Using a Space Filling Curve

4. Computation Reordering Approaches

A computation reordering involves changing the order in which data elements are referenced, but not the locations in which these data elements are stored. Consider again the data space shown in Figure 1. A computation reordering would reorder the pairs of elements within the interaction list. The vector of particle information accessed by the computation would remain unchanged. Computation reordering approaches can improve both temporal and spatial locality by reordering the accesses so that the same or neighboring data elements are referenced close together in time. The following subsections describe the computation reordering approaches considered in this work.

4.1. Space-Filling Curve Computation Reordering

Reordering a computation in space-filling curve order requires determining the position along the curve for each data element and using these positions as the basis for reordering accesses to these data elements. Figure 6 shows an example data space before and after computation reordering. Assume that the first three particles in space-filling curve order are $\mathbf{P_x}$, $\mathbf{P_y}$, and $\mathbf{P_z}$. To reorder the computation, entries in the interaction list, as shown in Figure 5, are sorted according to the space-filling curve position of the particles they reference. The order of the particle information itself remains unchanged. A space-filling curve based computation reordering can improve temporal locality. For instance, if particle X interacts with a nearby particle Y, then it is likely that particle Y will be referenced again soon since Y in turn will interact with other particles.

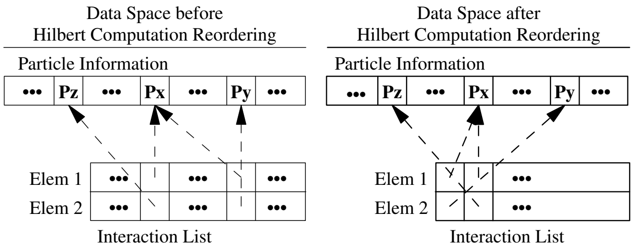

Figure 6: Computation Reordering Using a Space-Filling Curve

4.2. Computation Reordering by Blocking

As described earlier in the paper, blocking computation via loop nest restructuring has been used successfully to improve memory hierarchy utilization in regular applications for multi-level memory hierarchies. Here we describe how blocking can be used as a computation reordering technique for some irregular applications as well.

In terms of our n-body example, the following loop nest is an abstract representation of the natural computation ordering for the given data order:

```
FOR i = 1 to number of particles DO
    FOR j in the set particles_that_interact_with[i] DO
        process interaction between particles i and j
```

To block this computation, we first assign each particle to some block. (One way of computing a block number for a particle is to take its address and ignore some number of low-order bits.) Then, rather than considering all interactions for each particle at once, one can consider all interactions between particles in each pair of blocks while traversing pairs of blocks in some orderly fashion. The following code fragment illustrates this idea for one possible traversal order of the blocks.

```
FOR i = 1 to number of blocks of particles DO
    FOR j = i to number of blocks of particles DO
        process interactions between all interacting
        particle pairs with the first particle in block i
        and the second in block j
```

Mitchell, Carter and Ferrante [29] concurrently developed a related blocking technique for irregular references that they call *bucket tiling*. They improve the locality of a stream of accesses for a single non-affine reference by reordering computation into blocks so that the stream of accesses from the same block of computation falls into the same region of memory. They don't consider orderings for multiple references (such as particle pairs), or hierarchical orderings for multi-level memory hierarchies.

To extend blocking strategies to multi-level memory hierarchies, it is necessary to block for each level in the hierarchy. In an earlier version of this work, we described a *k*-level blocking strategy for *k*-level memory hierarchies [30]. Unfortunately, choosing the best blocking factor for each level is difficult and experimentation is necessary. The best blocking factors for an application depend not only upon the architectural characteristics of the target machine's memory hierarchy but also upon characteristics of the application itself. Architectural characteristics that affect the choice of blocking factor for a cache include the size of the blocks managed by that cache (i.e. line size for data caches or page size for TLB), the number of sets in the cache, the associativity, and even the replacement policy. Application characteristics that affect the choice of blocking factors for a computational kernel include the number of data references in the kernel, whether the access streams for each reference are disjoint or overlapping, and the spatial and temporal reuse among all of the references. For irregular problems, the amount of spatial and temporal reuse achievable is a function of an application's input data and depends on factors such as the average density of particles per unit of space or the average degree of nodes in an unstructured mesh.

Over the last several years, recursive divide and conquer strategies have been advocated for blocking regular computations for machines with multi-level memory hierarchies in an architecture-independent fashion [31, 24]. The rationale for this approach is that if the computation at a particular level of recursion doesn't fit into some level of the memory hierarchy, the computation at some deeper level of recursion will. The divide-and-conquer approach essentially blocks the computation at all possible levels and some of those levels will be an effective blocking for any particular machine.

We can achieve a similar machine-independent multi-level blocking of irregular computations as well by careful computation ordering. In terms of our n-body example, computation order is represented by an interaction list and we can block computation by sorting interactions by the block numbers of the particles they reference. Applying a lexicographical sort [32] to the interaction pairs using [block_of(p1), block_of(p2)] as the sorting key for pair [p1,p2] achieves a single level of blocking. To block for a multi-level memory hierarchy in a machine-independent fashion, we modify the approach slightly. First, we compute a sort key for an interaction using a bit-wise interleaving of the block numbers for the particles in the pair. Next, we sort interactions using these keys. This effectively blocks the interaction list for all possible levels in any memory hierarchy. Forming an interaction's sorting key as the bit-wise interleaving of it's particle block numbers amounts to computing the position of the interaction along a 2D Morton space-filling curve through the space of block pairs. Sorting interactions by their position along a Morton curve recursively blocks the computation. Section 5.1 explains how we accomplish this quickly in practice. An alternative to simply performing a bit-wise interleaving of the block numbers to achieve a recursive blocking based on Morton ordering, a pair of block numbers can be simply treated as coordinates in a two-dimensional space that can be converted to a position along any space-filling curve, such as a Hilbert curve, as we describe in Section 5.1. Sorting by the position along a Hilbert curve will produce a similar recursive blocking. A Morton ordering is faster to compute, but a Hilbert ordering offers more potential locality because it avoids long edges.

5. Applying the Techniques

This section describe our experiences in applying data and computation reordering techniques to improve the performance of two particle codes, *moldyn* and *magi*. Also, we describe our preliminary experiences with a scatter benchmark derived from *CHAD*, a large unstructured mesh application. *Moldyn* is a synthetic benchmark, whereas *magi* and *CHAD* are production programs. These codes are described in more detail in the following subsections. *Moldyn* and *magi* are irregular programs that exhibit poor spatial and temporal locality, which are typical problems exhibited by this class of applications. *CHAD* is in large part a vector computation, but spends a significant fraction of time performing irregular gather/scatter operations to move data between the nodes and edges of an unstructured mesh.

We chose to perform our experiments with *moldyn* and *magi* on an SGI O2 workstation based on the R10000 MIPS processor since it provides hardware counters that enable collection of detailed performance measurements and we were able to use the workstation in isolation. Both programs were compiled with the highest level of optimization available for the native C and Fortran compilers.[2] Table I displays the configurations of the different levels of the memory hierarchy on this machine. Each entry in the TLB contains two virtual to physical page number translations, where each page contains 4KB of data. Thus, the 8KB block size for the TLB is the amount of addressable memory in two pages associated with a TLB entry.

Cache Type	Cache Configuration		
	Cache Size	Associativity	Block Size
L1 Data	32KB	2-way	32B
L2 Data	1MB	2-way	128B
TLB	512KB	64-way	8KB

Table I: SGI O2 Workstation Cache Configurations

[2] Although these compilers can insert data prefetch instructions to help reduce latency, prefetching is less effective for irregular accesses because prefetches are issued on every reference rather than every cache line [8]. Our experience was that data prefetching support in the SGI Origin C and Fortran compilers did not improve performance for the applications we studied and we did not use it in our experiments.

5.1. The *Moldyn* Benchmark

Moldyn is a synthetic benchmark for molecular dynamics simulation. The computational structure in *moldyn* is similar to the nonbonded force calculation in CHARMM [33], and closely resembles the structure represented in Figure 1 of the paper. An interaction list is constructed for all pairs of interactions that are within a specified cutoff radius. These interactions are processed every timestep and are periodically updated due to particles changing their spatial location.

A high-level description of the computation for *moldyn* is shown in Figure 7. The time-consuming portion of the algorithm is the inner **FOR** loop which corresponds to the *computeforces* function in the benchmark. This function traverses the interaction list performing a force calculation for each pair of particles. We applied different data and computation reordering techniques in an attempt to make the *computeforces* function more efficient.

```
Randomly initialize the coordinates of each of the particles.
FOR N time steps DO
    Update the coordinates of each particle based on their
        force and velocity.
    Build an interaction list of particles that are within
        a specified radius every 20th time step.
    FOR each pair of particles in the interaction list DO
        Update the force on each of the particles in the pair.
    Update the velocities of each of the particles.
Print the final results.
```

Figure 7: Structure of the Computation in *Moldyn*

For our experiments, we set the number of particles to 256,000, which resulted in over 27 million interactions. We chose this problem size to cause the data structures to be larger than the secondary cache and the amount of memory that can be contained in the pages associated with the TLB. Figure 8 depicts the data structures used in the *computeforces* function. The coordinates and forces have three elements for each particle since the physical space of the problem is in three dimensions. The length of the interaction list was long enough to contain all interacting pairs of particles. Each of the elements of the coordinates and forces are double precision values and the interaction list elements are integers used as indices into the coordinate and force arrays.

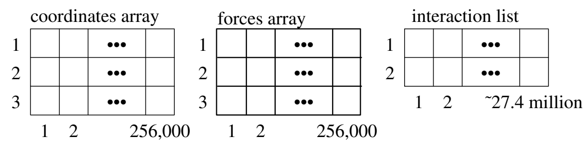

Figure 8: Main Data Structures in the *Moldyn* Benchmark

To make the *moldyn* benchmark more amenable to performing experiments with a large number of particles, we changed the approach for building the interaction list. Previously, a straightforward algorithm with $O(n^2)$ complexity was used to find all the interacting pairs of particles that were within the specified cutoff radius. We used an approach of dividing the physical space into cubes, where the length of each cube side was the size of the cutoff radius. We then assigned each particle to its respective cube. For a given particle, only the particles in current and immediate surrounding cubes had to be checked as possible interaction partners. (This is a well-known technique that is used by the *magi* application as well.) This allowed the interaction list to be built in a couple of minutes instead of several hours.

Before performing experiments with data and computation reorderings, we manually applied three transformations to remove orthogonal memory hierarchy performance problems.

(1) We interchanged the dimensions of the coordinates and the forces arrays so information for each particle would be contiguous in memory.

(2) We fused the coordinates and forces together (approximating an array of structures) to provide better spatial locality.

(3) We adjusted the loop that computes forces so that when a sequence of interactions references the same first particle, the data for the first particle is only loaded from memory once.

The purpose of this static program restructuring was to establish a good performance baseline for our experiments so that improvements in reuse of dynamic data are not "lost in the noise." In our results below, all of our performance comparisons are with respect to this statically tuned version of the program that we refer to as *Baseline*.

Table II shows information about misses in the caches and the TLB for our Baseline version of *moldyn* benchmark. To investigate the nature of the poor memory hierarchy performance, we used the MHSIM memory hierarchy simulator we developed to collect an L1 miss trace for the application. Figure 9 shows a plot of L1 misses over the first 100,000 interactions within the *computeforces* in the Baseline version of *moldyn*. While all memory references were simulated, only the misses associated with the particle information are displayed in the plot. The block numbers in the plot are the portion of the addresses (tag and index) used to access the L1 cache and the interaction numbers indicate on which interaction each miss occurred. The band of misses is initially as wide as the array of particles. Figure 10 shows a plot of L1 misses over 100,000 interactions when a Hilbert curve was used to reorder both the particle data and computation. This plot was drawn at the same scale as the plot in Figure 9 and the total number of misses for the first 100,000 interactions was reduced by a factor of 14. The difference between these plots illustrates the dramatic performance benefits that can be achieved by applying data and computation reorderings.

Cache Type	Baseline Misses	Baseline Miss Ratio
L1	1,613,065,560	0.23439
L2	995,152,174	0.61693
TLB	664,457,217	0.09655

Table II: Miss Information

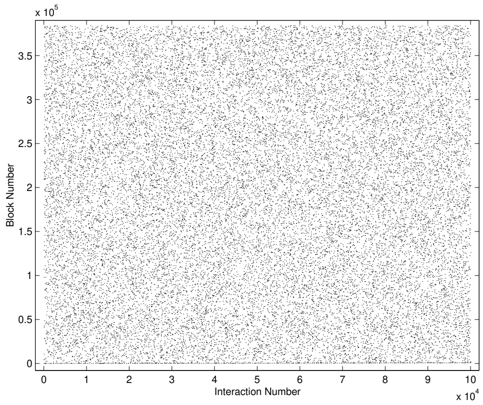

Figure 9: L1 Baseline Misses over the First 100,000 Interactions

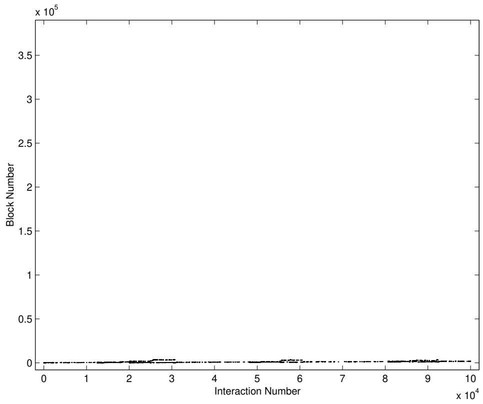

Figure 10: L1 Misses over the First 100,000 Interactions after using Hilbert Curves to Reorder the Data and Computation

To accomplish multi-level blocking of the *moldyn* non-bonded forces computation, the interaction list must be reordered to match the characteristics of the memory hierarchy of the target machine. As described in Section 4.2, we compute a key for each interaction by interleaving the particle block numbers. To sort interaction pairs quickly, we break each key into 4 bit "digits" and then apply a most significant digit radix sort [32]. We chose 4-bit digits to avoid thrashing the TLB for large data sizes. Too many bins for the radix sort implies too many pages.[3]

To show how our multi-level blocking algorithm regularizes the memory accesses of the molecular dynamics force computation, we include Figures 11-13 which show the pattern of L1 misses due to the *computeforces* function for the first 10,000, 100,000, and 1,000,000 primary cache misses collected using a cache simulator configured for the SGI O2. The traces for these plots were collected for the *moldyn* force computation. No reordering has been applied to the data. The computation is blocked using Morton ordering as described in Section 5.2. Note that the scales of these three plots differ on each axis. These plots show only the misses in the particle information; misses for the interaction list were simulated, but not recorded. As the simulation progresses from left to right the increasing interaction numbers mark the progression of the computation. In any vertical slice of the graph, only two blocks are active: one to which the first particle in the pairs belongs, and one to which the second particle belongs. The recursive structure of the blocked computation order can be seen by comparing the three figures at different scales. The recursive structure of the computation causes the figures to have the same form at all scales. Figures 12 and 13 plot enough L1 misses to distinguish interactions between TLB blocks and L2 blocks, respectively. Figure 13 illustrates that four TLB blocks are accessed repeatedly since there are four TLB blocks for each L2 block.

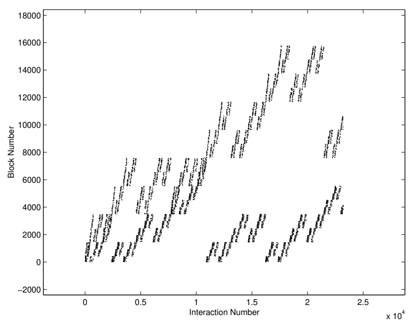

Figure 11: Plot of 10K L1 Misses

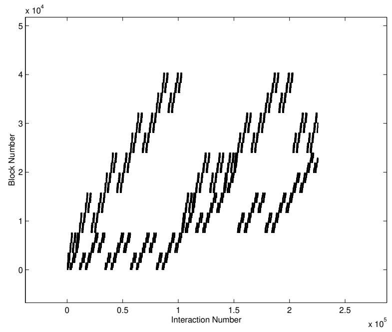

Figure 12: Plot of 100K L1 Misses

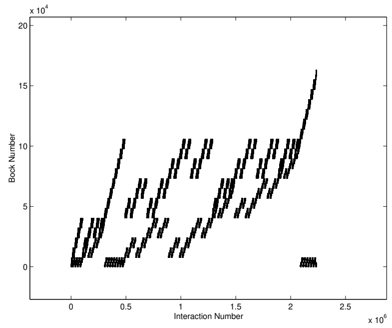

Figure 13: Plot of 1M L1 Misses

Table III shows the results for applying the different combinations of data and computation reorderings to *moldyn* on an SGI O2 workstation. All reorderings were applied by manually inserting calls to general-purpose library routines we developed for computing ordering keys and then permutations based on these keys. RCM stands for the Reverse Cuthill-McKee approach described in Section 2. These results show ratios of end-to-end performance as compared to execution of the Baseline version of *moldyn* without any run-time data or computation reordering.

[3] Prokop [31] describes two alternative sorting algorithms that are cache oblivious with asymptotically optimal reuse that would also be appropriate.

Data Reordering	Computation Reordering	L1 Cache Misses	L2 Cache Misses	TLB Misses	Cycles
RCM	None	0.96441	0.81847	0.49658	0.86650
First Touch	None	0.87487	0.76548	0.31928	0.79069
Hilbert	None	0.87978	0.78074	0.26397	0.80731
None	Hilbert	0.45053	0.12157	0.74006	0.37778
None	Blocking	0.30376	0.23557	0.19278	0.61910
First Touch	Hilbert	0.33735	0.14314	0.00806	0.38773
Hilbert	Hilbert	0.25816	0.10139	0.00624	0.26550

Table III: Results of the Different Data and Computation Reorderings for *Moldyn* (Ratios as Compared to the Baseline Measurements)

There are several aspects of the results that are worth noting. First, data and computation reorderings are most effective at reducing misses for caches with a large block or line size. For this reason reductions in TLB misses were the greatest, and those for L2 were greater than those for primary cache. Second, a combination of data and computation reorderings performed dramatically better than using any specific type of data or computation reordering in isolation. Hilbert data reordering combined with Hilbert computation reordering reduced TLB misses by a factor of 160, L2 misses by a factor of 10, and primary cache misses by a factor of 4. This strategy reduced the miss ratios for L1 cache from 23.4% to 6.1%, for L2 cache from 61.7% to 6.3%, and for TLB from 9.7% to 0.06%.[4] In terms of reducing execution cycles, Hilbert-based data and computation reordering performed the best, yielding a factor of four overall reduction in cycles. While the Morton blocking strategy was competitive even without data reordering, once the data and computation are in Hilbert order for this density of interactions, there is essentially no benefit to blocking. Particles do not have so many neighbors that evaluating all interactions for a single particle causes significant evictions. (The average interaction density in these experiments was 105 interactions per particle.) In addition to the blocking results reported in the table, we also experimented with multi-level blocking based on Hilbert rather than Morton orderings. For the interaction density we studied, the higher overhead of computing Hilbert keys for the interactions masked any potential performance benefits.

5.2. The *Magi* Application

The *magi* application is a particle code used by the U.S. Air Force for performing hydrodynamic computations that focus on interactions of particles in spatial neighborhoods. The computational domain consists of objects comprised of

[4] It is worth noting that since we are measuring end-to-end performance, the miss rates quoted for executions with reordering include all misses incurred performing the reordering as well as misses during the rest of the program execution. When we consider the performance of the *computeforces* routine alone, improvements are far greater.

particles and void space. A 3D rectangular space containing particles is divided into boxes, where the neighboring particles within a sphere of influence of a given particle are guaranteed to be in the same box or an adjacent box. For our experiments, we used DoD-provided test data involving 28,000 particles. For this test case, the size of the data structures is larger than the secondary cache and the amount of memory that can be contained in the pages associated with the TLB.

The utility that created the input data set for *magi* ordered the particles using Sloan's method [28]. This ordering was accomplished by first constructing an undirected graph in which all particles within a cutoff radius are linked as neighbors and then applying Sloan's method to the resulting graph to compute a refined ordering for the particles.

A high-level description of the computation for *magi* is given in Figure 14.

```
Read in the data for each of the particles.
FOR N time steps DO
    FOR each particle i DO
        Create an interaction list for particle i containing
            neighbors within the sphere of influence.
        FOR each particle j within this interaction list DO
            Update information for particle j.
Print the final results.
```

Figure 14: Structure of the Computation in *Magi*

Just as in the *moldyn* benchmark, we manually tuned the *magi* application to improve memory hierarchy performance to provide a better baseline for our experiments.

(1) We transposed several arrays containing particle information so this information would be contiguous in memory.

(2) We fused some arrays together (approximating an array of structures) to provide better spatial locality when different kinds of particle information are referenced together.

Unlike the *moldyn* benchmark, a separate interaction list is created for each particle on each time step and is discarded after being used once. There is never an explicit representation of all the interactions. Therefore, computation reordering techniques that require reordering of the interaction list as presented in the *moldyn* benchmark would not be applicable for *magi*. Likewise, some types of data reordering cannot be accomplished in the same manner since there is no persistent representation of an interaction list that can be updated to point to the new location of the particles. Therefore, we used the following approaches to accomplish data and computation reordering for *magi*.

(1) We used an indirection vector containing the new positions of the particles when applying data reordering without computation reordering so the order in which the particles were referenced would be unaffected. This requires an

additional level of indirection each time information about a particle is referenced, which can potentially have an adverse effect on both the performance of the memory hierarchy and the execution cycles.

(2) Data reordering using a space-filling curve does not depend on the order of the interactions and was performed before the first time step. First-touch data reordering was accomplished by (a) collecting the order of the references during the first time step across the different particle interaction lists and (b) reordering the particles before they are referenced on the second time step.

(3) When applying computation reordering, we simply did not use the indirection vector. Thus, the order of a subsequently generated interaction list is affected by the data reordering of the particle information.

(4) We composed a data reordering using a Hilbert space-filling curve followed by a data reordering using a first-touch approach without using an indirection vector to cause computation reordering. Placing the particles in Hilbert order results in a space-filling curve based computation order, which increases the likelihood that consecutive particles being processed will have many common neighbors in their interaction lists and improves temporal locality. Applying a first-touch reordering to the space-filling curve based computation order after the first time step greedily increases spatial locality. Note this approach is similar to applying computation reordering using a Hilbert space-filling curve approach and data reordering using a first-touch approach as was accomplished in *moldyn*. The only difference is that interaction lists in *magi* are established at the beginning of each time step, which causes the first-touch data reordering to affect the computation order.

Table IV shows the results of applying combinations of data and computation reorderings that were beneficial for the *magi* application. Several of the combinations of data and computation reorderings applied to the *moldyn* benchmark are not shown in this table for two reasons. First, we found that applying data reordering only for *magi* did not improve performance. The cost of accessing data through an indirection vector offset the benefits that were achieved by reordering data. One should note that data reordering without computation reordering can achieve benefits as shown for *moldyn* in Table III. However, achieving such benefits may require that there is an inexpensive method to access the reordered data, such as updating an interaction list once to refer to the new data locations rather than incurring the cost of dereferencing an element of the indirection vector on each data reference. Second, the combinations of data and computation reordering were also restricted by the fact that the interaction list for a particle was regenerated on each time step. Regeneration of the interaction lists prevented direct computation reordering. Likewise, separate and small interaction lists for each particle made the use of blocking inappropriate.

Data Reordering	Computation Reordering	L1 Cache Misses	L2 Cache Misses	TLB Misses	Cycles
First Touch	First Touch	0.42959	0.27032	0.49173	0.56321
Hilbert	Hilbert	0.28621	0.11916	0.15704	0.43751
Hilbert/First Touch	Hilbert/First Touch	0.32670	0.11695	0.13513	0.43607

Table IV: Results of the Different Data and Computation Reorderings for *Magi*
(Ratios as Compared to the Baseline Measurements)

The results in Table IV show that the combination of reordering particle data and and interaction computations according to particle positions along a Hilbert curve (which probabilistically increases spatial and temporal locality) followed by a first-touch data reordering (which greedily improves spatial locality) achieves the lowest L2 and TLB misses and the best overall cycle time by a very slim margin. The table shows that applying a first-touch data reordering after the Hilbert-based reordering amortizes the cost of the first-touch reordering by reducing L2 and TLB misses, but the barely perceptible improvement in overall performance does not justify the additional programming effort.

5.3. Scatter Benchmark from *CHAD*

CHAD is a parallel unstructured mesh application developed at Los Alamos National Laboratory for simulating three-dimensional fluid flows with chemical reactions and fuel sprays. The code operates on a static unstructured mesh composed of arbitrarily mixed hexahedral and lower-order degenerate elements (e.g., pyramids, prisms, or tetrahedra). Although an arbitrary number of elements can be associated with a node, most nodes have degree close to six.

Computation in *CHAD* is principally carried out as dense vector operations on data values associated with the edges and mesh elements. The principal irregular data access patterns occur in gather/scatter operations. On a 16039 node mesh with 47718 edges (the largest test case available to us, but a small one compared to those used in production runs), a sequential version of the code spent 25% of its time performing gather and scatter operations between mesh nodes and endpoints or midpoints of edges. Initially, our aim was to study the effects of data and computation reordering in the context of the entire *CHAD* code; however, working remotely on an unfamiliar code of this size (roughly 88,000 lines) proved to be a bottleneck for completing this investigation. To accelerate our research, we abstracted out a scatter benchmark that represents the data access patterns moving data between nodes and edges.

In our experiments with this benchmark, we used a memory hierarchy simulator to evaluate locality because it enabled us to collect detailed information about misses, including traces. Miss traces enable us to see where and why misses occur and provide insight into the structure of the computation. To collect these traces we used MHSIM, our locally-developed memory hierarchy simulator, which we configured to simulate a 2-way set-associative 32KB primary cache--the configuration found in the MIPS R10K processor used in the SGI O2.

Our scatter benchmark performs a scatter-add that independently accumulates X, Y, and Z quantities at each node from the appropriate endpoint of each incident edge. Figure 15 shows the organization of the nodes of the mesh we used in our experiments. The nodes are closely spaced along radial spines centered at the origin. The ordering of the nodes is largely in terms of concentric shells. Figure 16 shows a plot of node vector position versus distance of the node from the origin. When the original node ordering is drawn by connecting all of the nodes, the resulting figure looks like a ball of string. The structure becomes apparent when viewing the plot at high magnifications, or plots of subsets of the data. Figure 17 shows a plot connecting the positions of 600 consecutive nodes from the node vector in the order given by the original dataset. The line moves through a node on each spine, then repeats the traversal of the spines in the same order at a different radial distance.

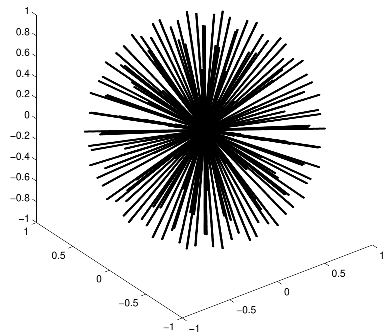

Figure 15: Organization of 16038 mesh nodes for *CHAD* scatter benchmark.

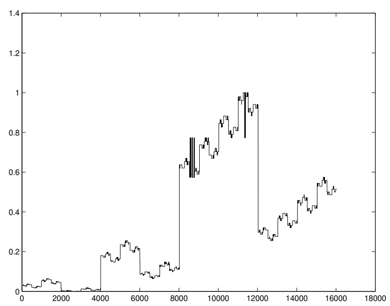

Figure 16: Node vector position versus distance from the origin.

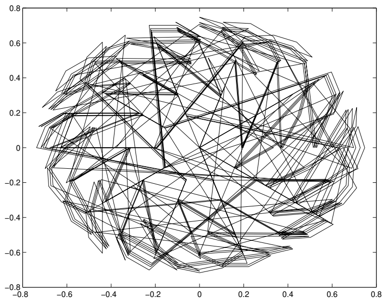

Figure 17: Plot connecting 600 nodes in *CHAD* scatter benchmark mesh (Original Order)

In our experiments, we measured the number of primary cache misses that occurred during one trip through the *CHAD* scatter kernel to accumulate edge-based data at the node endpoints. Figure 18 shows the pattern of misses for one execution of the *CHAD* scatter kernel using the original node order. The irregular accesses for updating the X, Y, and Z node values in the scatter operation appear as three parallel roughly horizontal bands at the top of the figure. At the left edge of these bands are three diagonal lines where the values are initialized to zero. The three central diagonal bands correspond to misses for stride-1 accesses to the separate X, Y, and Z vectors that hold values for each of the edge endpoints. The bottommost diagonal shows the misses to the vector contains a pair of node coordinates for each edge. For this ordering, 12% of the misses come from the irregular access pattern to the node values. However, the irregular misses are important for two reasons. First, they represent an opportunity for temporal reuse. The only other temporal reuse is that each node index for an edge endpoint is used three times to scatter X, Y, and Z data. Second, the other stride-1 misses are predictable and can be mitigated by prefetching.

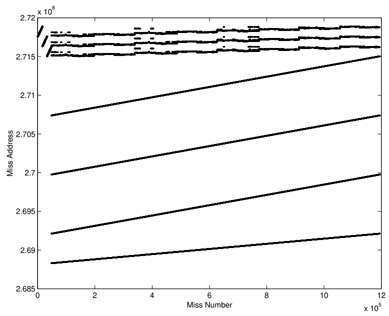

Figure 18: L1 Misses for *CHAD* Scatter Benchmark using Original Node Order

To investigate the impact of data and computation reordering on this scatter benchmark, we investigated several different node and edge ordering strategies. Node and edge orderings determine the spatial and temporal locality of the irregular accesses in the scatter. Without a good node ordering, no spatial locality will be realized for the irregular accesses in the scatter computation. An edge ordering amounts to a computation ordering, since it determines the pattern of irregular access to the nodes during a scatter computation. A good edge order will capitalize on spatial locality in the node ordering and orchestrate temporal reuse of data by bringing multiple irregular accesses to the same node close together in time.

We considered four different node orderings, and three different edge orderings. Not all combinations are considered. The node orderings we compared include the original order from the test dataset, Hilbert order, random order, and the order determined by applying Reverse Cuthill-McKee to the sparse adjacency matrix representing the edges. After a node reordering, edge endpoints must be renumbered. The edge orderings we considered include the original order, lexicographic order (of the (src,dest) edge pairs), and Hilbert order. Comparing different data orderings with random is interesting because the parallel version of the *CHAD* code uses the ParMETIS graph partitioner [34] to partition the nodes and edges of the computational mesh among available processors. ParMETIS computes its partitionings in a hierarchical fashion and swaps nodes and edges between partitions. After partitioning, the locality properties of the mesh pieces are believed to resemble those for meshes with random node orderings [35].

We compute Hilbert order for nodes by normalizing each node's X, Y, and Z coordinates, which are a triple of integer coordinates, each in the range [0..2^21]. We then convert this triple to a position along a 63-bit Hilbert curve running through this space, and sort the nodes by their position along the curve. Computing Hilbert order for edges is analogous: we treat the pair of node numbers identifying the edge endpoints as coordinates in a two-dimensional space, normalize them, and then convert them to Hilbert position and sort them. To improve the quality of lexicographic and Hilbert edge orderings, we flip edges, if necessary, to ensure that the smallest numbered endpoint is always the first in the edge pair.

Table V shows the relative number of primary cache misses measured for a scatter operation performed using different combinations of node and edge orderings for the *CHAD* test mesh. We didn't measure secondary cache or TLB misses at all for this experiment because of the modest data size. The values shown for L1 cache misses are all ratios between the number of misses measured with the simulator for that particular data and computation ordering, divided by the number of misses measured using the original node and computation orderings. Unlike *moldyn* and *magi*, which both have degrees of data reuse on the order of 100 and 20 for each particle in each timestep, the *CHAD* test mesh used by the scatter benchmark has an average degree of 6. Thus, there is less reuse to exploit with good node and edge orderings. For the test mesh, our results show that the original node and edge order is quite good: it is nearly a factor of 2 better than random. Both the space-filling

Node Reordering	Edge Reordering	L1 Cache Misses
original	lexicographic	0.962
Hilbert	original	1.28
Hilbert	lexicographic	0.978
Hilbert	Hilbert	1.03
RCM	lexicographic	0.972
RCM	Hilbert	1.01
random	original	1.61
random	lexicographic	1.92

Table V: Results of the Different Data and Computation Reorderings
for Scatter Benchmark

curve and the Reverse Cuthill-McKee node order produce similar results. In these experiments, the Hilbert edge order is 4-6% slower than the lexicographic order. Comparing the two orders, lexicographic order greedily exploits temporal locality of the first node of an edge pair at the expense of the second. Combined with a good edge order, lexicographic order can be quite effective when the number of incident edges per node is modest (the data for all of a node's partners fits in cache). Hilbert edge order attempts to balance locality between the edge endpoints. This can be beneficial when the number of neighbors per node is high, but here the sacrifice of greedy temporal locality for the first endpoint of an edge is a net loss.

6. Conclusions

Typically, irregular applications make poor use of memory hierarchies and performance suffers as a result. Improving memory hierarchy utilization involves improving reuse at multiple levels, typically including TLB and one or more levels of cache. Our measurements show how coordinated orderings of data and computation can dramatically improve utilization in memory hierarchies at multiple levels. We also have shown that neither data reordering nor computation reordering alone is nearly as effective as a coordinated approach involving both. We introduced multi-level blocking as a new computation reordering strategy for irregular applications and demonstrated significant benefits by combining reordering techniques based on space-filling curves with other data or computation reordering techniques.

Using space-filling curves as the basis for data and computation reorderings offers several benefits. First, reordering data elements according to their position along a space-filling curve probabilistically increases spatial locality. In space-filling curve order, neighboring elements in physical space, which tend to be referenced together during computation, are clustered together in memory. This clustering helps improve utilization of long cache lines and TLB entries. Second, reordering computation to traverse data elements in their order along a space-filling curve also improves temporal locality. By following the space-filling curve, neighboring elements in physical space are processed close together in time, and thus computations that operate on a data element and its spatial neighbors repeatedly encounter the same elements as the computation traverses all elements in a neighborhood. Finally, data reordering based on position along a space-filling curve is fast. The cost of such a reordering is typically small relative to the rest of a program's computation.

With the *moldyn* application, we demonstrated dramatic improvements in memory hierarchy utilization by using Hilbert-based data reordering and either multi-level blocking or a Hilbert-based strategy for reordering computation. The Hilbert-based computation reordering has an advantage over blocking for *moldyn* in that it is accomplished at no cost by simply performing Hilbert-based data reordering before building the interaction list in the canonical fashion. In our experiments, blocking offered no additional benefit over Hilbert computation order because the interaction density was not high enough to cause capacity misses while computing interactions for a single particle. Blocking would offer benefits with higher interaction densities.

With the *magi* application, Hilbert curve based strategies for data and computation reordering improved end-to-end performance by over a factor of two. The best memory hierarchy utilization came from considering particles in space-filling curve order to improve temporal locality, and using that as the basis for a first-touch data and computation reordering that greedily improves spatial locality. It is interesting to note that the improvements we achieved for *magi* with our data reorderings are relative to a baseline computation for which input particle data has already been carefully ordered using Sloan's method for profile minimization [28]. Similarly, space-filling curve based reordering methods provided substantially superior overall performance than the Reverse Cuthill-McKee profile minimization method for *moldyn*.

With the scatter benchmark from the *CHAD* application, the Hilbert and Reverse Cuthill-McKee data orderings combined with lexicographic data ordering produced results closely comparable to the original careful ordering. The key point, is that these ordering strategies achieved this level of performance without any *a priori* knowledge and that the level of locality they achieved is nearly a factor of two better than that achieved for a random ordering. These results suggest that these reordering techniques may provide substantial benefits when applied to pieces of partitioned meshes that are not well ordered.

As the gap between processor and memory speeds continues to grow and large-scale scientific computations continue their shift towards using adaptive and irregular structures, techniques for improving the memory hierarchy performance of irregular adaptive applications will become increasingly important. In this paper, we have demonstrated that data and computation reordering based on space-filling curves can be used to improve the locality of sequential computations. However, these techniques are more broadly applicable. Our colleagues have recently also applied space-filling curve based reorderings to improve the parallel efficiency of shared-memory and software distributed shared memory computations by improving data locality, which reduces communication and false sharing [36, 37]. Our experiences show that good data and computation orders can be achieved for irregular problems using dynamic reorderings, and that the gain in locality from using good data and computation orders can be dramatic.

7. Acknowledgements

Discussions with Vikram Adve and Rob Fowler helped shape this work. Doug Moore provided us with a library for mapping between multidimensional coordinate spaces and positions along a Hilbert space-filling curve. Gina Goff and Ehtesham Hayder provided us access to an SGI O2 workstation for our experiments. Initial experimentation for this work was performed on an SGI Onyx2 at the Biomedical Computation and Visualization Laboratory of the Baylor College of Medicine supported by grant BIR-9512521 from the National Science Foundation. Later experimentation was performed on the Blue Mountain SGI Origin system at Los Alamos National Laboratory.

8. References

[1] D. Callahan, S. Carr, and K. Kennedy, "Improving Register Allocation for Subscripted Variables," *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design & Implementation*, pp. 53-65 (Jun 1990).

[2] 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** pp. 587-616 (1988).

[3] M. S. Lam, E. E. Rothberg, and M. E. Wolf, "The Cache Performance and Optimizations of Blocked Algorithms," *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 63-74 (Apr 1991).

[4] A. K. Porterfield, *Software Methods for Improvement of Cache Performance on Supercomputer Applications,* PhD Dissertation, Rice University, Houston, TX (May 1989).

[5] M. E. Wolf and M. S. Lam, "A Data Locality Optimizing Algorithm," *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pp. 30-44 (Jun 1991).

[6] J. Ferrante, V. Sarkar, and W. Thrash, "On Estimating and Enhancing Cache Effectiveness," *Proceedings of Fourth Workshop on Languages and Compilers for Parallel Computing*, (Aug 1991).

[7] D. M. Tullsen and S. J. Eggers, “Effective cache prefetching on bus-based multiprocessors,” *ACM Transactions on Computer Systems* **13**(1) pp. 57-88 (Feb 1995).

[8] T. C. Mowry, M. S. Lam, and A. Gupta, “Design and Evaluation of a Compiler Algorithm for Prefetching,” *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 62-73 (Oct 1992).

[9] A.C. McKeller and E.G. Coffman, “The Organization of Matrices and Matrix Operations in a Paged Multiprogramming Environment,” *Communications of the ACM* **12**(3) pp. 153-165 (1969).

[10] W. Abu-Sufah, D. J. Kuck, and D. H. Lawrie, “Automatic program transformations for virtual memory computers.,” *Proceedings of the 1979 National Computer Conference*, pp. 969-974 (Jun 1979).

[11] J. J. Navarro, E. Garcia, and J. R. Herrero, *Proceedings of the 10th ACM International Conference on Supercomputing (ICS)*, (1996).

[12] I. Kodukula, N. Ahmed, and K. Pingali, “Data-centric Multi-level Blocking,” *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design & Implementation*, pp. 346-357 (Jun 1997).

[13] J. R. Allen and K. Kennedy, “Automatic loop interchange,” *Proceedings of the SIGPLAN ‘84 Symposium on Compiler Construction SIGPLAN Notices* **19**(6) pp. 233-246 (Jun 1984).

[14] K. S. McKinley, S. Carr, and C.-W. Tseng, “Improving Data Locality with Loop Transformations,” *ACM Transactions on Programming Languages and Systems* **18**(4) pp. 424-453 (Jul 1996).

[15] C. Ding and K. Kennedy, “Improving Cache Performance of Dynamic Applications with Computation and Data Layout Transformations,” *Proceedings of the ACM SIGPLAN ’99 Conference on Programming Language Design & Implementation*, pp. 229-241 (May 1999).

[16] R. Das, D. Mavriplis, J. Saltz, S. Gupta, and R. Ponnusamy, “The Design and Implementation of a Parallel Unstructured Euler Solver Using Software Primitives,” *AIAA Journal* **32** pp. 489-496 (1994).

[17] H. Sagan, *Space-Filling Curves,* Springer-Verlag, New York, NY (1994).

[18] H. Samet, *Applications of Spatial Data Structures: Computer Graphics, Image Processing and GIS,* Addison-Wesley, New York, NY (1989).

[19] J. P. Singh, C. Holt, T. Totsuka, A. Gupta, and J. Hennessy, “Load Balancing and Data Locality in Adaptive Hierarhcical N-body Methods: Barnes-Hut, Fast Multipole, and Radiosity,” *Journal of Parallel and Distributed Computing*, (Jun 1995).

[20] M. S. Warren and J. K. Salmon, “A Parallel Hashed Oct-Tree N-Body Algorithm,” *Proceedings of Supercomputing ’93*, (Nov 1993).

[21] C. Ou, M. Gunwani, and S. Ranka, "Architecture-Independent Locality-Improving Transformations of Computational Graphs Embedded in k-Dimensions," *Proceedings of the International Conference on Supercomputing*, (1995).

[22] M. Parashar and J. C. Browne, "On Partitioning Dynamic Adaptive Grid Hierarchies," *Proceedings of the Hawaii Conference on Systems Sciences*, (Jan 1996).

[23] M. Thottethodi, S. Chatterjee, and A. R. Lebeck, "Tuning Strassen's Matrix Multiplication Algorithm for Memory Efficiency," *Proceedings of SC98: High Performance Computing and Networking*, (Nov 1998).

[24] J. Frens and D. Wise, "Auto-blocking Matrix Multiplication or Tracking BLAS3 Performance from Source Code," *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design & Implementation*, pp. 206-216 (Jun 1997).

[25] I. Al-Furaih and S. Ranka, "Memory Hierarchy Management for Iterative Graph Structures," *Proceedings of the International Parallel Processing Symposium*, (Mar 1998).

[26] A. George and G. Liu, *Computer Solution of Large Sparse Positive Definite Systems,* Prentice Hall, Englewood Cliffs, NJ (1981).

[27] E. Cuthill and J. McKee, "Reducing the Bandwidth of Sparse Symmetric Matrices," *Proc. ACM National Conference*, Association of Computing Machinery, (1969).

[28] S. Sloan, "An Algorithm for Profile and Wavefront Reduction of Sparse Matrices," *International Journal for Numerical Methods in Engineering* **23** pp. 239-251 (1986).

[29] N. Mitchell, L. Carter, and J. Ferrante, "Localizing Non-Affine Array References," *Proceedings of Parallel Architectures and Compilation Techniques '99*, (Oct 1999).

[30] J. Mellor-Crummey, D. Whalley, and K. Kennedy, "Improving Memory Hierarchy Performance for Irregular Applications," *Proceedings of the 1999 ACM International Conference on Supercomputing*, pp. 425-433 (Jun 1999).

[31] H. Prokop, "Cache-Oblivious Algorithms," *Master's thesis,*, MIT Department of Electrical Engineering and Computer Science, (Jun 1999).

[32] D. Knuth, *The Art of Computer Programming Volume 3: Sorting and Searching,* Addison-Wesley, New York, NY (1973).

[33] B.R. Brooks, R.E. Bruccoleri, B.D. Olafson, D.J. States, S. Swaminathan, and M. Karplus, "CHARMM: A Program for Macromolecular Energy, Minimization and Dynamics Calculations," *Journal of Computational Chemistry* **187**(4)(1983).

[34] G. Karypis and V. Kumar, "Parallel Multilevel k-way Partition Scheme for Irregular Graphs," *SIAM Review* **41** pp. 278-300 (1999).

[35] R. Robey, *Personal Communication*Sep 2000.

[36] Y. C. Hu, A. Cox, and W. Zwaenepoel, "Improving Fine-Grained Irregular Shared-Memory Benchmarks by Data Reordering," *Proceedings Supercomputing 2000*, (Nov 2000).

[37] V. Pai and S. Adve, “Code Transformations to Improve Memory Parallelism,” *Proceedings MICRO-32*, (Nov 1999).