Metrics and Models for Reordering Transformations

Michelle Mills Strout
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439 USA
mstrout@mcs.anl.gov

Paul D. Hovland
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439 USA
hovland@mcs.anl.gov

ABSTRACT

Irregular applications frequently exhibit poor performance on contemporary computer architectures, in large part because of their inefficient use of the memory hierarchy. Run-time data- and iteration-reordering transformations have been shown to improve the locality and therefore the performance of irregular benchmarks. This paper describes models for determining which combination of run-time data- and iteration-reordering heuristics will result in the best performance for a given dataset. We propose that the data- and iteration-reordering transformations be viewed as approximating minimal linear arrangements on two separate hypergraphs: a spatial locality hypergraph and a temporal locality hypergraph. Our results measure the efficacy of locality metrics based on these hypergraphs in guiding the selection of data- and iteration-reordering heuristics. We also introduce new iteration- and data-reordering heuristics based on the hypergraph models that result in better performance than do previous heuristics.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*performance measures*

General Terms

Algorithms, Measurement, Performance, Experimentation

Keywords

data locality, locality metrics, run-time reordering transformations, spatial locality graph, temporal locality hypergraph, optimization, inspector/executor

1. INTRODUCTION

Application performance depends on efficient memory hierarchy usage. In almost all modern computers, whenever a memory location is referenced by a program, the data in the referenced location and nearby locations are brought into a fast, but small, data cache. Any additional references to data already in the cache line (before the cache line is evicted from the cache) will be one or two orders of magnitude faster than references to main memory. When such usage occurs during the execution of a program, it is referred to as *spatial locality* for reuse within a cache line and *temporal locality* for reuse of the same data prior to eviction.

As the performance gap between processor and memory speeds grows, inefficient use of the memory hierarchy is becoming the dominant performance bottleneck in many applications. This situation is especially true in applications that do not access memory in a sequential or strided fashion. Such applications are referred to as irregular [31]. Figure 1 shows one possible implementation of iterating over edges in a graph. Iterating over an edgelist exhibits the types of memory references that occur in irregular applications, such as partial differential equation solvers and molecular dynamics simulations.

```
for i=1 to N
  ... X[l[i]] ...
  ... X[r[i]] ...
endfor
```

	i=1	2	3	4	5	6
l	2	4	1	3	4	2
r	6	5	3	2	6	4

	1	2	3	4	5	6
X	A	B	C	D	E	F

Figure 1: Example of an irregular loop and associated data array `X` and index arrays `l` and `r`.

The data locality of such an application can be improved by changing the order of computation (iteration reordering) and/or the assignment of data to memory locations (data reordering) so that references to the same or nearby locations occur relatively close in time during the execution of the program. Run-time reordering transformations use *inspector/executor* strategies [26] to reorder irregular applications effectively. An inspector traverses the memory reference pattern (e.g., edges in the edgelist example) at runtime,

©2004 Association for Computing Machinery. ACM acknowledges that this contribution was authored or co-authored by a contractor or affiliate of the U.S. Government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.
MSP'04, June 8, 2004, Washington, DC, USA.
Copyright 2004 ACM 1-58113-941-1/04/06 ...$5.00.

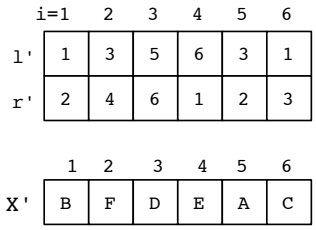

Figure 2: The data array X has been remapped into X'. The pointer update optimization has been used so that the index arrays refer to the new data locations.

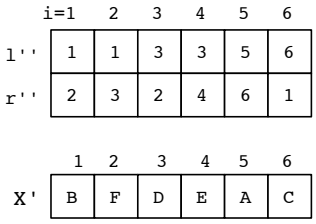

Figure 3: The iterations of the loop can be reordered by lexicographically sorting index arrays l' and r' into l'' and r''.

```
for i=1 to N
  ... X'[l''[i]] ...
  ... X'[r''[i]] ...
endfor
```

Figure 4: The executor that uses the remapped data and index arrays.

generates data-reordering and iteration-reordering functions based on the observed pattern, creates new schedules, and remaps affected data structures accordingly. Figure 2 shows an inspector-generated data remapping of the X data array based on the access pattern in the original l and r index arrays. Pointer update [12] is used to update the values in the index arrays to reference the new data locations. Figure 3 shows how the entries in the l' and r' arrays can then be lexicographically sorted to implement iteration reordering. The executor is a transformed version of the original program that uses the schedules and remapped data structures generated by the inspector. Figure 4 shows how the original code in Figure 1 is transformed to use the remapped data array X' and the updated and remapped index arrays l'' and r''. Run-time reordering transformations are beneficial if the overhead due to the inspector can be amortized over many executions of the improved executor.

Run-time data- and iteration-reordering transformations have been shown to improve the locality and therefore the performance of loops containing irregular access patterns [9, 2, 12, 27, 24, 22, 17, 15]. Selecting the combination of data and iteration reordering transformations that will best improve the performance of the executor while maintaining the ability to amortize the overhead of the inspector for a given irregular application and input is an open problem. In some irregular applications the initial data access pattern is driven by a static entity such as a mesh or molecular interaction list. Since the data access pattern is known statically, a preprocessing step allows the inspector cost to be amortized over many executions of the executor. This preprocessing step can include the evaluation of metrics that compare various data and iteration reordering schemes. Metrics based on the irregular data access pattern avoid the unwieldy and potentially unprofitable alternative of running the entire application with various reorderings.

In this paper, we study experimentally the ability of an existing data-reordering model and corresponding metric to predict performance and cache behavior. We refer to the model as the *spatial locality graph*. We also introduce a new model, the *temporal locality hypergraph*, and corresponding metrics for iteration reordering. We identify one existing iteration-reordering heuristic that implicitly uses the concept of the temporal locality hypergraph, and we develop three new iteration-reordering heuristics that explicitly use the new model and result in improved performance over existing heuristics. We then extend the spatial locality graph to a spatial locality hypergraph for loops that use more than two index arrays to access one or more data arrays. Experimental results show that data reordering heuristics that leverage the spatial locality hypergraph extension result in better performance than those that use only the spatial locality graph model.

2. MODELING DATA REORDERING

In general, run-time *data-reordering* transformations improve the spatial locality in a computation. The typical model used to guide data-reordering transformations is a graph with one vertex for each data item and edges connecting data accessed within the same iteration [30, 18]. We refer to this graph as the *spatial locality graph* since ordering data items that share an edge in this graph consecutively in memory improves the spatial locality of the computation. Many data-reordering algorithms heuristically solve the graph layout problem of minimal linear arrangement [19], or optimal linear ordering [16], for the spatial locality graph. Figure 5 shows the spatial locality graph for the edgelist example in Figure 1. If each data item v is mapped to storage location $\sigma(v)$, then the spatial locality metric based on minimal linear arrangement is

$$\sum_{(v,w)\in G_{SL}(E)} |\sigma(v) - \sigma(w)|,$$

where $G_{SL}(E)$ is the set of edges in the spatial locality graph. Note that this model and corresponding metric take only the spatial locality within a single iteration into account and therefore the metric does not measure spatial locality between iterations.

In our experiments, we calculate the spatial locality metric for various data-reordering heuristics applied to a number of datasets. We use the benchmark irreg [7, 17, 34] to gather execution times for several irregular graphs. The irreg benchmark is a kernel abstracted from a partial differential equation solver. It iterates (10 times) over edges in a graph that are stored in an edgelist data structure such as the one shown in Figure 1. We interleave the data arrays in the base-line code to improve spatial locality [13].

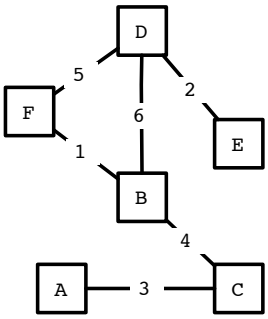

Figure 5: The spatial locality graph for the example in Figure 1. Each vertex (square) represents a data item. Two vertices share an edge if they are accessed the same iteration.

Source	Dataset	ratio	MB
Adams	CylLargeCut.graph	22.70	2.56
Adams	Wing22K.graph	20.20	3.78
Adams	Plate.graph	10.90	3.89
Davis	ex11.graph	32.50	4.37
Davis	li.graph	29.20	5.41
Adams	Cone.graph	32.02	5.72
Davis	rma10.graph	24.43	9.44
COSMIC	foil.graph.txt	7.43	10.40
Adams	Cant.graph	31.58	16.00
Adams	CCSphere.graph	35.56	23.88
Davis	nd6kg.graph	191.09	26.52
Davis	pre2.graph	3.86	29.44
Davis	torso1.graph	34.47	32.32
Davis	cage13.graph	7.90	33.63
Davis	StanfordBerkeley.graph	5.46	38.90
Adams	Sphere150.graph	36.64	45.67
Davis	kim2r.graph	11.90	48.45

Table 1: Datasets used with `irreg` benchmark. The column labeled "ratio" reports the average number of interactions for each molecule. The "MB" column reports the dataset size based on the data structures used in the `irreg` benchmark.

Table 1 lists the datasets used with the `irreg` benchmark and their sources. The Adams datasets are available at the Finite Element Market [1]. The Davis datasets are available at the University of Florida Sparse Matrix Collection [11]. The COSMIC datasets are available at the COSMIC project website [7].

We perform our experiments on a Sun-Fire-280R 1.2 GHz with 1GB memory, 64K of L1 cache, and 1MB of L2 cache, and a Xeon 2.2 GHz with 4GB of memory, 8K of L1 cache, and 512K of L2 cache. The `irreg` benchmark is compiled using the default compiler flags for the Sun 'cc' compiler[1] on the Sun-Fire and 'gcc -O3' [2] on the Xeon. For each combination of benchmark, dataset, data reordering, and iteration reordering, we record the minimum execution time over three runs. All runs are performed on a single processor of these dual processor machines. We use PAPI [23] to collect L1 cache, L2 cache, and TLB miss information on the Xeon.

For each experiment, we use a data-reordering heuristic followed by an iteration-reordering heuristic, which is the same strategy used in [12] and [17]. Other strategies, such as iterating between data and iteration reordering multiple times [34], are not covered in this paper. The spatial locality metric is affected only by the data-reordering heuristic because the metric depends only on the storage mapping $\sigma()$, which maps each data item to a storage location.

We use three data-reordering heuristics:

Consecutive Packing (CPACK) [12]: CPACK is the simplest and fastest data-reordering heuristic. It traverses the edgelist in the current iteration order and packs data into a new ordering on a first-come-first-serve basis. In the spatial locality graph each edge represents an iteration in the loop. CPACK visits the edges in the graph in order and consecutively packs the data items at the edge endpoints.

Breadth-First Search (BFS) [2]: The BFS heuristic converts the edgelist representation of a graph into a graph data structure that stores the neighbors for each data item or node in the spatial locality graph. It then performs a breadth-first search of the nodes in this graph, ordering each data item based on when its corresponding node in the spatial locality graph is visited.

Gpart [17]: Gpart is a graph-partitioning heuristic. It builds the same graph data structure as the one generated by the BFS heuristic to represent the spatial locality graph. Gpart then performs a graph partitioning on the spatial locality graph and orders the data consecutively within each partition. We select the parameters for the Gpart algorithm as described in [17].

The spatial locality metric does not predict the actual execution time or number of cache misses; instead, a lower metric value predicts which strategy results in better executor performance. To determine the effectiveness of the spatial locality metric in selecting the data-reordering heuristic that results in the best performance for a given dataset, we report the geometric mean of the normalized execution times for the data reordering that achieves the lowest metric value on each dataset and the data reordering that achieves the lowest execution time on each dataset. Figure 6 summarizes the results of the `irreg` benchmark on the Sun-Fire-280R. We normalize the execution time of `irreg` on each of 22 different datasets with various data reorderings against the execution time of `irreg` on the original dataset. The execution times do not include the overhead of performing the reorderings (the inspector execution time), because our focus is on modeling the data and iteration reorderings that will result in the fastest executor. Although the iteration reordering does not affect the spatial locality metric, it does affect the cache behavior and therefore the performance of the executor. For the first set of results on the Sun-Fire 280R, we summarize the effectiveness of the spatial locality metric in selecting the data reordering with the lowest execution time when no iteration reordering follows the data reordering versus when lexicographical sorting follows the various data reorderings.

When data reorderings are applied to the original ordering of the datasets there is not much improvement in the execution time, and it is difficult to judge how well the metric predicts the best or close to best data reordering. There-

[1]Sun WorkShop 6 update 2 C 5.3 2001/05/15

[2]gcc version 3.3.3

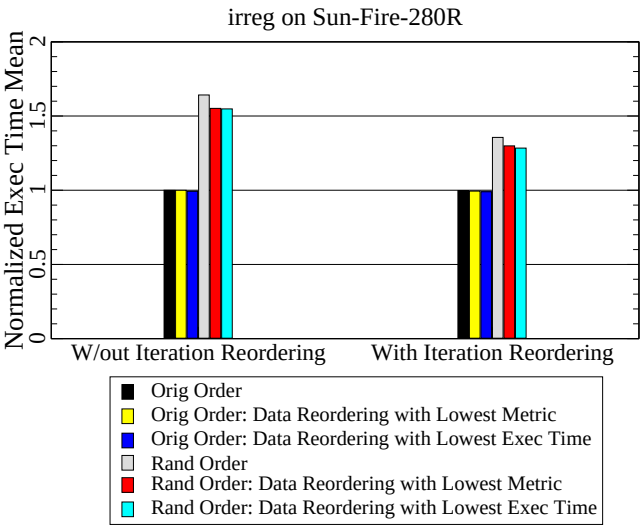

Figure 6: Results for the `irreg` benchmark on the Sun-Fire-280R. Each bar represents the geometric mean of the normalized execution times for the datasets in Table 1. All execution times are normalized to the execution time for the original ordering of the dataset. Random ordering indicates that the data and iterations for the datasets are randomly permuted before performing a data- and iteration-reordering strategy.

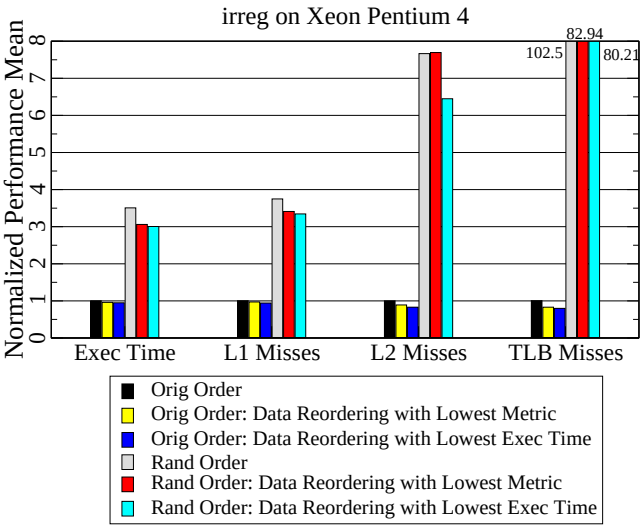

Figure 7: Results for the `irreg` benchmark on the Xeon Pentium 4. Each bar represents the geometric mean of the normalized execution times or miss counts for the datasets in Table 1. All execution times and miss counts are normalized to the execution time or miss counts for the original ordering of the dataset. Random ordering indicates that the data and iterations for the datasets are randomly permuted before performing a data- and iteration-reordering strategy.

fore, we also ran the experiments, in which the original data and iteration orderings are randomly permuted. Techniques such as adaptive mesh refinement [4] and partitioning for parallelism [25] have the effect of perturbing the original ordering; therefore, a complete random ordering represents the extreme of how badly a dataset could perform after the application of such techniques. Starting from the random ordering, it is easier to see that the spatial locality metric does not always select the best data reordering, but the execution time of the executor using the data reordering selected is usually within 2% of the best execution time achieved through data reordering.

Figure 7 shows similar results for the Xeon Pentium 4 except that all the data reorderings are followed by the lexicographical sorting iteration reordering, and we use PAPI to record the L1, L2, and TLB misses. The summarized execution results are similar to those on the Sun-Fire-280R. When the performance is broken down in terms of cache and TLB misses, it is interesting to note that randomly permuting the order has the largest effect on the TLB performance. Also, L2 misses do not correlate well with performance. This suggests that data-reordering heuristics that take into account architectural features such as cache size should also consider using the amount of data referenced in the TLB as a parameter.

Since no data-reordering heuristic results in the lowest execution time in all cases, the spatial locality metric can be used to make a data reordering decision for each dataset. The results that start from a random ordering show that when there is a significant difference between the performance for various data orderings, selecting the data reordering with the lowest spatial locality metric gets close to the best possible performance amongst the data-reordering heuristics used. The fact that all of the datasets in Table 1 do not benefit from data reordering was surprising and suggests the need to determine when the execution time due to the original ordering cannot be improved through heuristic reorderings.

3. MODELING ITERATION REORDERING

Temporal locality occurs when the *same* memory location is reused before its cache line is evicted. In general, runtime *iteration reordering* improves the temporal and spatial locality in an irregular application, by ordering iterations that touch the same data item sequentially in the schedule. Since each edge in the spatial locality graph corresponds to an iteration, reordering the edges in this graph reorders the computation or iterations of the loop. Typically edges are ordered based on some variant of the lexicographical ordering enforced by the new data ordering. We introduce a new model called the *temporal locality hypergraph* and show that heuristics based on this model result in better executor performance than those based on lexicographical sorting. We also show that a metric corresponding to the temporal locality hypergraph can effectively select an iteration-reordering heuristic for a particular dataset that results in either the fastest execution time or close to the fastest execution time.

The *temporal locality hypergraph* models the relationships between iterations of the loop. A *hypergraph* is a generalization of a graph where each hyperedge can involve more than two vertices. It can be described with a set of vertices V and a set of hyperedges E, where each hyperedge is a vertex set. The temporal locality hypergraph has a vertex

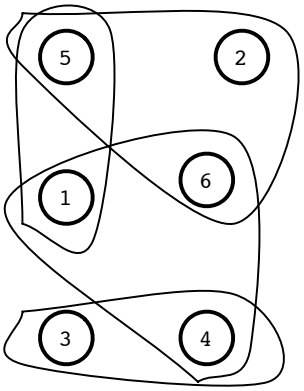

Figure 8: The temporal locality hypergraph for the example in Figure 1. Each vertex (circle) represents an iteration. Two or more vertices belong to the same hyperedge if they access the same data item.

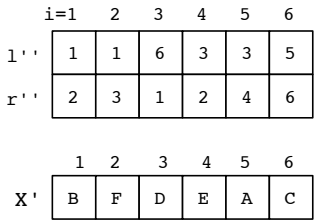

Figure 9: Reordering the iterations (represented by edges in the `l'` and `r'` arrays) using consecutive packing for iterations instead of lexicographical sorting.

i for each iteration and a hyperedge e for each data item. For each data item an iteration accesses, the iteration vertex is included in the hyperedge representing the data item. Figure 8 shows the temporal locality hypergraph for the example in Figure 1. Notice that iterations two, five, and six are in the same hyperedge because `l[2]`, `l[5]`, and `r[6]` index the same location in `X`.

Using a hypergraph representation instead of a graph with only two nodes per edge offers several advantages. The temporal locality hypergraph is significantly more compact than the corresponding temporal locality graph (with edges for all pairs of iterations that access the same memory location). Also, the temporal locality hypergraph is the dual of the spatial locality (hyper)graph. Therefore it can be constructed via a matrix transpose operation, essentially equivalent to converting from a compressed sparse row representation to a compressed sparse column representation. Consequently, the spatial locality graph and temporal locality hypergraph have nearly identical storage requirements.

3.1 Iteration-reordering Heuristics Based on the Temporal Locality Hypergraph

Iteration-reordering heuristics based on the new temporal locality hypergraph model attempt to order iterations within the same hyperedge sequentially. This differs from many existing iteration-reordering heuristics that view iterations as edges in the spatial locality graph and perform some variation of lexicographical sorting.

Lexicographical Sorting (lexSort) [17]: lexSort lexicographically sorts the edges in an edgelist such as the one shown in Figure 1 based on the end points of the edges. Locality grouping [12] and bucket tiling/irregular blocking [27, 25] are variations of lexicographical sorting that require less inspector overhead.

Das *et al.* [10] introduced an iteration-reordering heuristic that groups all the iterations that access the first data item, then the second, etc. We refer to this iteration-reordering heuristic as consecutive packing for iterations (CPACKIter), because one interpretation of this heuristic is that it consecutively packs iterations in the temporal locality hypergraph while visiting the hyperedges according to the data ordering. This differs from lexicographical sorting in that lexicographical sorting treats edges in the spatial locality graph as ordered pairs. In Figure 2, the edges represented by the `l'` and `r'` index arrays are reordered into `l''` and `r''` (see Figure 3) using lexicographical sorting. Using CPACKIter results in the edge $(6, 1)$ being ordered third instead of last (see Figure 9). It is also possible to interpret the Ding and Kennedy [12] locality grouping heuristic as equivalent to CPACKIter.

Consecutive packing for iterations (CPACKIter): CPACKIter visits the hyperedges/data items in order and packs the iterations in each of these hyperedges on a first-come-first-serve basis. This heuristic is analogous to the CPACK data-reordering algorithm that operates on the spatial locality graph. Although CPACKIter is logically based on the temporal locality hypergraph, it is possible to create this iteration reordering with the list of edges in the spatial locality hypergraph without generating the temporal locality hypergraph. Specifically, this heuristic corresponds to sorting the edges based on their minimal endpoint.

In addition to recognizing the temporal locality graph interpretation of CPACKIter equivalent heuristics, we also introduce three new iteration-reordering heuristics based on the temporal locality hypergraph model.

Breadth-first ordering for iterations (BFSIter): BFSIter performs a breadth-first-search ordering on the vertices in the temporal locality hypergraph. Figure 10 lists pseudocode for this algorithm. The algorithm requires the creation of the temporal locality hypergraph $G_{TL}(V, E)$ and uses the spatial locality graph $G_{SL}(V, E)$. For the edgelist example in Figure 1, the edgelist itself represents the spatial locality graph.

Temporal hypergraph partitioning (HPart): PaToH [6] is a hypergraph partitioning package. We use the partitions generated by PaToH to reorder the iterations, analogously to the GPart data-reordering algorithm.

Temporal hypergraph partitioning and CPACK for iterations (CPACKIter-HPart): This heuristic combines CPACKIter with the hypergraph partitioner PaToH. First the iterations are ordered using CPACKIter. The iterations are placed into partitions by the partitioner. Iterations are ordered consecutively within each partition and they maintain their relative CPACKIter ordering.

3.2 Metrics based on the temporal locality hypergraph

We introduce three metrics based on the temporal locality

```
Algorithm BFS_HYPERGRAPH(G_TL(V,E),G_SL(V,E),n)

    ! Initialization
1:  count = 0
2:  select vertex i from G_TL(V) and add to iter-queue
3:  do
      ! While there are still iterations in iter-queue
4:    while (iter-queue is not empty)
5:      i = dequeue(iter-queue)
6:      put i next in iteration ordering
7:      count = count + 1

        ! Determine all hyperedges v and w for which
        ! iteration i belongs
8:      for each (v,w) = E_i where E_i ∈ G_SL(E)
9:        if not visited(v) then
10:         add v to data-queue and mark as visited
11:       if not visited(w) then
12:         add w to data-queue and mark as visited
13:     end for each

        ! Put all iterations in hyperedges corresponding
        ! to v and w into iter-queue
14:     while v = dequeue(data-queue)
15:       for each i in v where v ∈ G_TL(E)
16:         if not visited(i) then
17:           add i to iter-queue and mark as visited
18:       end for each
19:     end while

20:   end while

      ! If the temporal locality hypergraph G_TL is
      ! unconnected then reinitialize the iter-queue
21:   if (count < n) then
22:     add a non-visited iteration to iter-queue
23: while (count < n)
```

Figure 10: BFS_HYPERGRAPH **algorithm that implements both breadth-first search on the temporal locality hypergraph (BFSIter) and breadth-first search on the spatial locality hypergraph (BFS_hyper).**

hypergraph. The absolute distance temporal locality metric is similar to the spatial locality metric. We extend the metric to hypergraphs by summing across all hyperedges the distances between all pairs of iterations belonging to the same hyperedge. Specifically, if each iteration i is mapped to a relative time/ordering $\delta(i)$, then the distance temporal locality metric is

$$\sum_{e \in G_{TL}(E)} \left(\sum_{i_j, i_k \in G_{TL}(E)} |\delta(i_j) - \delta(i_k)| \right),$$

where $G_{TL}(E)$ is the set of edges in the temporal locality graph. The iteration-reordering heuristics should aim to minimize the value of the metric.

We also introduce the *span* and *density* metrics. Preliminary results indicate that these metrics select the iteration-reordering heuristic with the same accuracy as the distance temporal locality metric and they are less expensive to compute. The *span metric* sums across all hyperedges the distance between the minimally and maximally ordered iterations in each hyperedge,

$$\sum_{e \in G_{TL}(E)} \max_{i \in e}(\delta(i)) - \min_{i \in e}(\delta(i)).$$

The *density metric* sums across all hyperedges the span divided by the number of iterations in each hyperedge,

$$\sum_{e \in G_{TL}(E)} \frac{\max_{i \in e}(\delta(i)) - \min_{i \in e}(\delta(i))}{|e|}.$$

3.3 Experimental Results

To compare spatial locality graph and temporal locality hypergraph based iteration-reordering heuristics and to determine the effectiveness of the temporal locality metric in predicting the best iteration reordering, we vary the iteration-reordering heuristic used after data reordering on the `irreg` benchmark. Figures 11 and 12 compare different iteration-reordering heuristics applied to the randomly permuted datasets for `irreg`. Within each iteration reordering (the x axis), we separate how that iteration reordering performs when it is preceded by different data-reordering heuristics. The final two groups of bars summarize the iteration-reordering heuristic that results in the smallest metric value (Low Metric) and lowest execution time (Low Exec) for each dataset. From this graph we can conclude that the iteration-reordering heuristic BFSIter is a significant improvement over existing iteration-reordering heuristics such as lexicographical sorting. BFSIter is one of the new iteration-reordering heuristics that explicitly operates on the temporal locality hypergraph.

The close match between the iteration-reordering heuristic with the smallest metric and the iteration-reordering heuristic with the lowest execution time indicates that the temporal locality metric is useful for selecting a good iteration-reordering heuristic for a particular dataset, even if it does not always select the best.

Figure 13 presents similar results for the molecular dynamics benchmark `moldyn`. `moldyn` performs the nonbonded force calculation as implemented in CHARMM [3]. The `moldyn` benchmark iterates (ten times) over interactions between molecules to determine changes in the force, velocity, and position for each molecule. In the base-line code for `moldyn`, we interleave the data arrays to improve spatial locality [13]. Table 2 lists the molecular dynamics datasets we use with `moldyn`. The HIV, ER-GRE, ApoA1, and POPC datasets are all generated from real biomolecular configurations. The configuration of HIV-1 Nef bound to Thioesterase II (HIV dataset) is distributed by the Scuola Internazionale Superiore di Studi Avanzat (SISSA) [5]; the configurations of estrogen receptor bound to a glucocorticoid response element and Apolipoprotein A-I (ER-GRE and ApoA1 datasets) are distributed by the Theoretical and Computational Biophysics Group, University of Illinois (UIUC) [35]; and the configuration of Bacteriorhodopsin embedded in a POPC membrane (POPC dataset) is distributed by the Leibniz-Rechenzentrum High Performance Computing Group (LRZ) [20, 21]. The mol1 dataset is a quasi-uniform distribution of molecules distributed by the COSMIC group at the University of Maryland [7]. We generate variants of each dataset

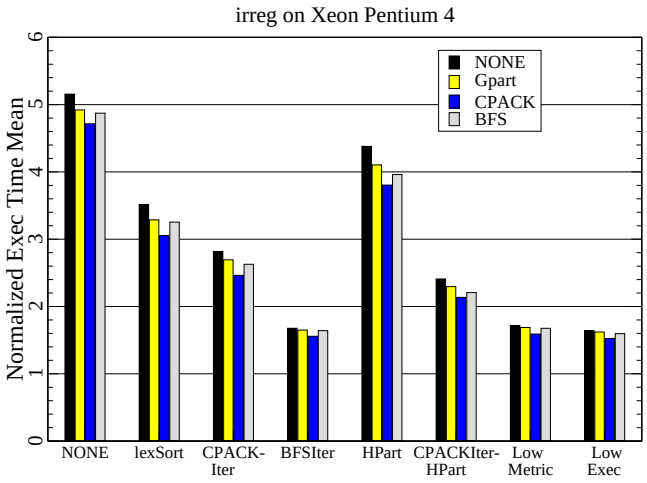

Figure 11: Results that compare various iteration-reordering heuristics applied to the `irreg` benchmark on the Xeon Pentium 4. Each bar represents the geometric mean of normalized execution times for the smallest twelve datasets in Table 1 when a particular data-reordering heuristic is applied before the iteration reordering indicated on the X-axis. We first randomly permute the data and iteration ordering of each dataset.

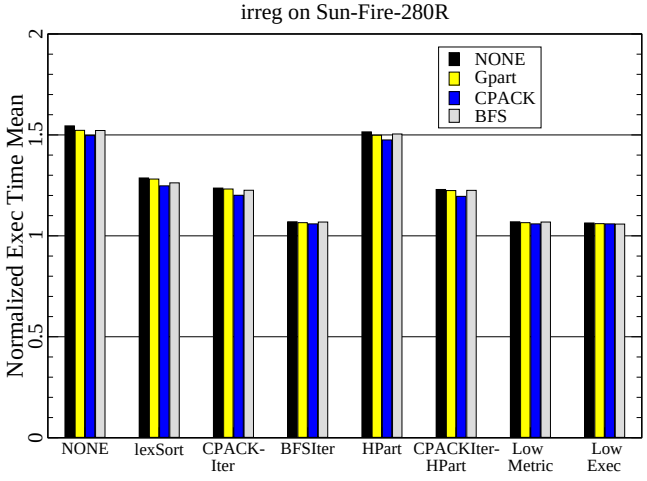

Figure 12: Results that compare various iteration-reordering heuristics applied to the `irreg` benchmark on the Sun-Fire-280R. Each bar represents the geometric mean of normalized execution times for the datasets in Table 1 when a particular data-reordering heuristic is applied before the iteration reordering indicated on the X-axis. We first randomly permute the data and iteration ordering of each dataset.

Source	Dataset	ratio	MB
SISSA	HIV-3.graph	5.62	1.27
SISSA	HIV-4.graph	11.42	1.78
LRZ	popc-br-3.graph	4.10	2.49
LRZ	popc-br-4.graph	10.24	3.66
UIUC	er-gre-3.graph	4.64	3.81
SISSA	HIV-6.graph	36.15	3.93
UIUC	er-gre-4.graph	12.34	5.95
LRZ	popc-br-6.graph	34.12	8.20
UIUC	apoa1-3.graph	4.95	9.81
SISSA	HIV-10.graph	145.03	13.41
UIUC	er-gre-6.graph	40.55	13.82
UIUC	apoa1-4.graph	12.27	14.96
UMD	mol1-1.8.graph	9.00	18.00
SISSA	HIV-12.graph	235.15	21.26
LRZ	popc-br-10.graph	143.99	29.08
UIUC	apoa1-6.graph	41.90	35.82
LRZ	popc-br-12.graph	237.76	46.91
UMD	mol1-3.graph	43.00	52.00
UIUC	er-gre-10.graph	178.03	52.19
UIUC	er-gre-12.graph	298.81	85.89
UIUC	apoa1-10.graph	185.43	136.80
UIUC	apoa1-12.graph	313.22	226.72
UMD	mol1-6.graph	383.00	392.00

Table 2: Datasets used with `moldyn` benchmark. The column labeled "ratio" reports the average number of interactions for each molecule. The "MB" column reports the dataset size based on the data structures used in the `moldyn` benchmark.

by using the same three-dimensional molecule locations and determining the interaction list using different cutoff distances. For example, HIV-3 indicates that the interaction list is built using a 3 angstrom cutoff with the HIV dataset.

Even with a large range of dataset sizes and ratios of interactions to molecules, the CPACK data reordering followed by the BFSIter iteration reordering always results in the lowest execution time for these datasets on this benchmark. The temporal locality metric correctly predicts BFSIter will result in the lowest execution time in all cases.

4. SPATIAL LOCALITY HYPERGRAPH

Many applications, including several phases of scientific simulations, involve iterating over edges, faces, or elements of unstructured meshes. The data access patterns of such applications are best modeled by 2-, 3-, or 4-regular hypergraphs, that is, spatial locality hypergraphs with a constant number of vertices per hyperedge. Data reorderings based on the spatial locality hypergraph model may be more effective than reorderings based on the pairwise spatial locality graph, which can be represented with an edgelist. Similarly, metrics based on the spatial locality hypergraph model may offer a more accurate indication of which reorderings will be most effective. We implement two hypergraph-based data-reordering heuristics.

Consecutive packing for hypergraphs (CPACK): CPACK for spatial locality hypergraphs visits the hyperedges (iterations) in order and packs the data items in each of these hyperedges on a first-come-first-serve basis. This heuristic is equivalent to the CPACK data-reordering algo-

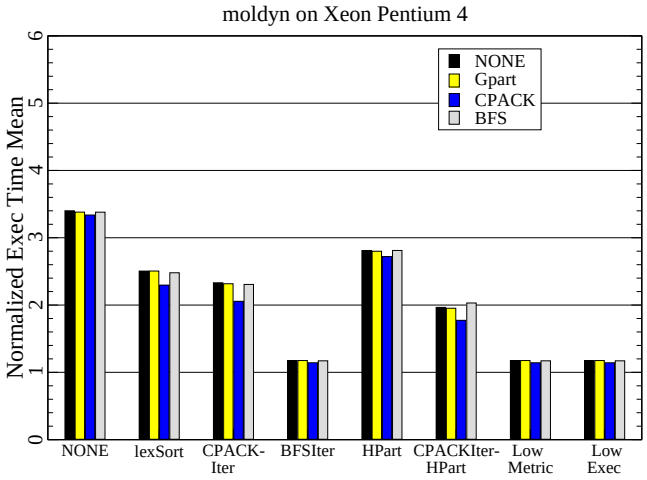

Figure 13: Results that compare various iteration-reordering heuristics applied to the `moldyn` benchmark on the Xeon Pentium 4. Each bar represents the geometric mean of normalized execution times for the datasets in Table 2 when a particular data-reordering heuristic is applied before the iteration reordering indicated on the X-axis. We first randomly permute the data and iteration ordering of each dataset.

rithm when a hyperedge is converted to all edge pairs in the natural way.

Breadth-first search for hypergraphs (BFS_hyper): This data-reordering algorithm applies the algorithm presented in Figure 10 to the spatial locality hypergraph rather than the temporal locality hypergraph.

We apply these hypergraph-based reordering algorithms and two edgelist-based reordering algorithms (GPart and BFS) to a mesh quality improvement application [29]. This application iterates over all of the elements in a mesh and computes various functions for each element. Because all vertices in an element are accessed in the same iteration, we would expect data reorderings based on the spatial locality hypergraph to perform better than orderings based on the spatial locality graph.

Figures 14–16 compare the performance of the mesh quality improvement application on several meshes using the hypergraph-aware data-reordering algorithms (BFS_hyper and CPACK) to the edgelist-based algorithms (BFS and GPart) and no data reordering (NONE). Each data-reordering heuristic is followed by the BFSIter iteration-reordering algorithm. Performance is normalized against the original access pattern with no data or iteration reordering. We perform our experiments on the Xeon 2.2 GHz, a PowerPC G5 1.8 GHz with 8GB memory, 32K of L1 data cache, and 512K of L2 cache, and an Opteron 242 1.6 GHz with 2GB memory, 128K of L1 cache, and 1MB of L2 cache. The execution times reported are the minimum over three runs (twelve for the Opteron, due to anomalous behavior on early runs) on a single processor of these dual processor machines, using 'gcc -O3' [3] to compile the application. Table 3 lists the properties of the meshes used in the experiments. The column labeled "dims" indicates the number of dimensions–2 for a triangular mesh and 3 for a tetrahedral mesh. The column labeled "source" indicates the source of the mesh. The airfoil mesh was supplied by Bijan Mohammadi (BM) of the University of Montpellier as part of a 2-d Navier-Stokes test problem. The other meshes were supplied by Todd Munson (TM) of Argonne National Laboratory and were generated using triangle [32] (2-d meshes) or CUBIT [8] (3-d meshes). The column labeled "MB" indicates the storage required for the mesh quality improvement application, in megabytes. Finally, the arrows in Figures 14–16 indicate the data-reordering heuristic that results in the lowest spatial locality metric value.

[3]gcc version 3.3.3 on the Xeon, version 3.3 on the PowerPC G5, and version 3.3.1 on the Opteron

Source	Mesh	dims	verts	elems	MB
TM	rand10k	2	10400	20394	2.46
TM	turtle	2	18322	36225	4.28
TM	honey8	2	16796	33480	4.06
BM	airfoil	2	22215	43806	5.28
TM	duct	3	177887	965759	97.64
TM	foam	3	190988	964210	83.56

Table 3: Characteristics of the meshes used in the spatial locality hypergraph experiments.

The experimental results show that some applications and datasets can greatly benefit from data and iteration reordering transformations being applied to the original ordering. It is clear that BFS_hyper is the most effective data-reordering heuristic and that the two edgelist-based heuristics, BFS and GPart, are in general not as effective as the hypergraph-aware heuristics. For the Xeon Pentium 4 and the PowerPC G5, the metric either selects the data-reordering heuristic that results in the lowest execution time or one that results in performance within 1.5% of the best. In Figure 16, the metric-selected data reordering is within 1.5% of the lowest execution time except for the honey8 dataset, where the metric-selected data reordering is approximately 8% slower than the data reordering resulting in the fastest execution time.

5. RELATED WORK

Related work falls into three categories: modeling the relationship between data items with a graph equivalent to the spatial locality graph, modeling the effect of data locality on performance, and data-reordering and iteration-reordering heuristics.

Previous work has used a graph equivalent to the *spatial locality graph* to drive data-reordering heuristics. Ou *et al.* refer to this graph as the computational graph. Han and Tseng [18] describe an equivalent graph as the basis for their graph partitioning approach to data reordering. We use the implicitly assumed spatial locality metric to select explicitly among data-reordering heuristics. Our experiments verify that using the data-reordering heuristic with the minimum spatial locality metric results in executor performance that is close to what is provided by the best performing data-reordering heuristic.

The idea of looking at the distance in number of iterations between reuses of data is what underlies our temporal locality metric. Ding and Zhong [14] model temporal local-

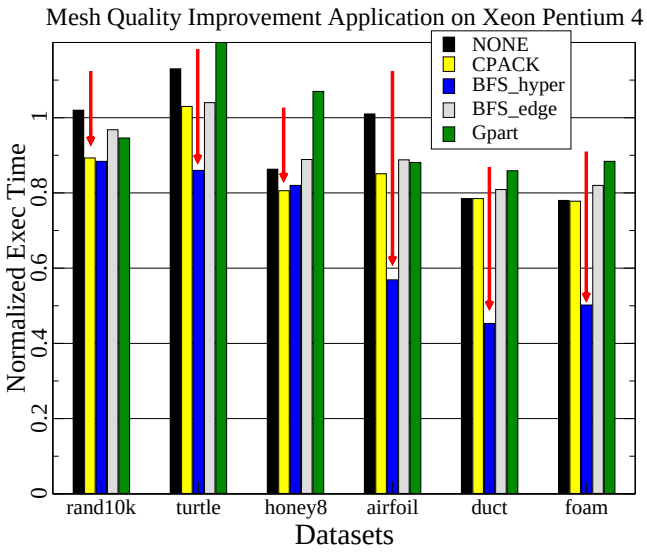

Figure 14: Results that compare various data-reordering heuristics applied to the mesh improvement application on the Xeon Pentium 4. Each bar represents the execution time for that dataset normalized to the execution time for the original ordering of that dataset. Each data reordering is followed by BFSIter for iteration reordering. The arrow indicates which data reordering results in the lowest spatial locality metric value for each dataset.

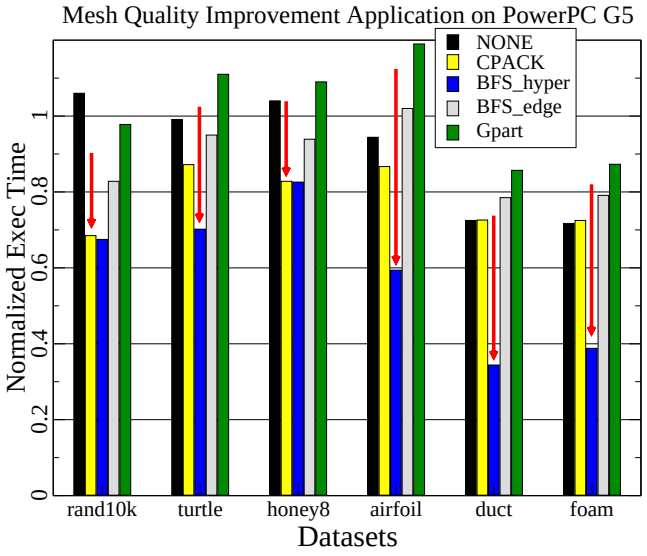

Figure 15: Results that compare various data-reordering heuristics applied to the mesh improvement application on the PowerPC G5. Each bar represents the execution time for that dataset normalized to the execution time for the original ordering of that dataset. Each data reordering is followed by BFSIter for iteration reordering. The arrow indicates which data reordering results in the lowest spatial locality metric value for each dataset.

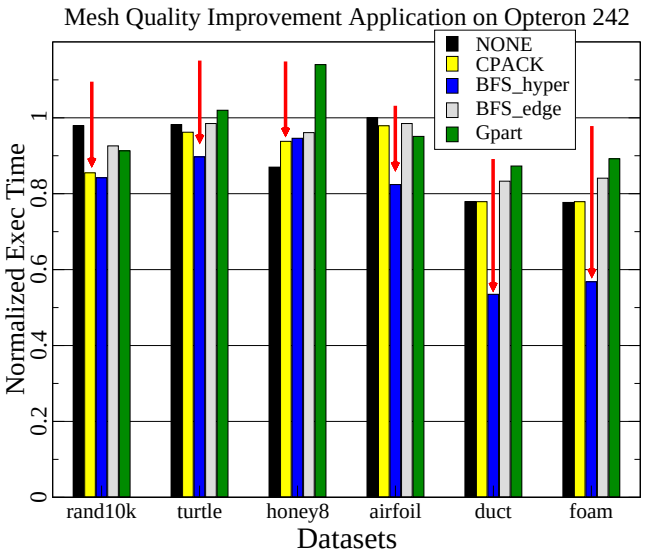

Figure 16: Results that compare various data-reordering heuristics applied to the mesh improvement application on the Opteron. Each bar represents the execution time for that dataset normalized to the execution time for the original ordering of that dataset. Each data reordering is followed by BFSIter for iteration reordering. The arrow indicates which data reordering results in the lowest spatial locality metric value for each dataset.

ity by calculating the reuse distance metric that counts the number of unique data accesses between data reuses. The temporal locality hypergraph metric is not as accurate as the reuse distance metric, because it counts the number of iterations between two accesses to the same data, but does not count the number of distinct data items that are accessed in those intervening iterations. The reuse distance metric is more expensive to compute. In [14], Ding and Zhong show that calculating reuse distance patterns across full programs allows for execution time prediction on the same program for different datasets. This is especially true when the program has regular access patterns. Our work focuses on the situation when the access pattern is irregular. An open problem is whether the increased accuracy provided by reuse analysis would improve our predictions as to which iteration-reordering heuristic is best.

In the domain of data and iteration reordering, [28, 36] propose methods for guidance when some information, such as the data access pattern, is not available until run-time. Hu and Rauchwerger [36] determine attributes of the benchmark and dataset that aid in determining which dynamic parallel reduction strategy will result in the best performance. For example, the data locality experienced by the local reduction computation is modeled with an attribute called "degree of clustering." Although data locality on one processor is taken into account, the characteristics in their model focus on selecting between parallel reduction strategies and not data- and iteration-reordering strategies. Mitchell *et al.* [28] assign a mode of access (sequential, strided, random) to each memory access and then compose a performance prediction based on benchmark experiments with

those modes on the relevant architecture. The spatial locality metric and temporal locality metrics we introduce do not attempt to predict execution time of the full application, but instead select amongst a number of data locality improving heuristics.

Many data-reordering and iteration-reordering heuristics for loops with no dependencies or only reduction dependencies have been developed [9, 10, 33, 2, 12, 27, 24, 17, 15, 22]. Other than those approaches that use space-filling curves [33, 24], the predominant model underlying data-reordering heuristics is a graph equivalent to the spatial locality graph. Data reorderings based on space-filling curves require coordinate information for each node of data and then put nodes that are proximal in some physical space, consecutive as much as is possible in memory as well. Much of the related work uses iteration-reordering heuristics based on lexicographical sorting after a data reordering, either spatial locality graph based or space-filling curve-based. However, Das *et al.* [10] perform an iteration-reordering heuristic equivalent to the CPACKIter heuristic we describe, which has an interpretation on the temporal locality hypergraph.

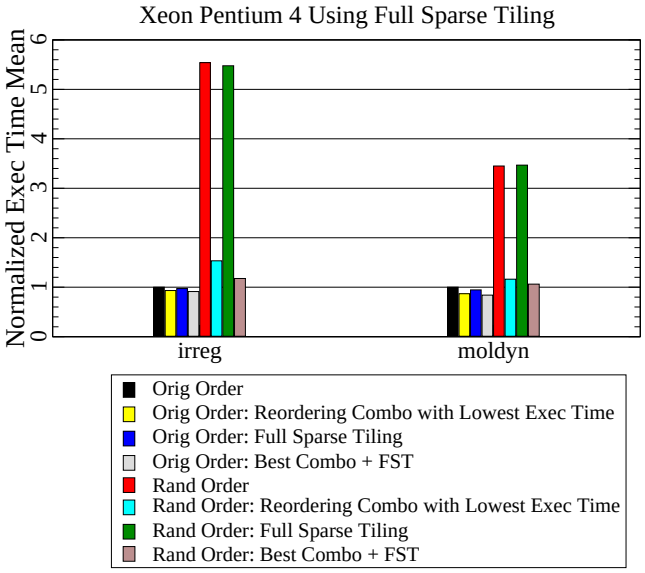

Figure 17: Results that compare the best data and iteration reordering combination for each `irreg` and `moldyn` dataset to the performance provided by full sparse tiling by itself and in composition with the best data and iteration reordering combination. Each bar represents the geometric mean of normalized execution times for the datasets in Table 1 for `irreg` and in Table 2 for `moldyn`. Random ordering indicates that the data and iterations for the datasets are randomly permuted before performing a data- and iteration-reordering strategy.

6. FUTURE WORK

For the datasets used with the `irreg` and `moldyn` benchmarks, the reordering heuristics and metrics had the most effect when the original dataset was randomly permuted before performing experiments. The principal reason is that the mesh-based and molecular datasets used in these experiments appear to have good initial orderings. The original order of the triangular and tetrahedral meshes used in the mesh quality application do benefit from data- and iteration-reordering heuristics. Although this can be estimated by performing some reorderings and comparing the resulting metrics to the original order, it would be better if some knowledge of the spatial locality and temporal locality hypergraph structures could be used to determine that a given ordering is going to perform as well or better than heuristic reorderings.

Exploring the ability of temporal and spatial locality metrics to determine when no additional reordering benefits performance would also be useful in evaluating more complex strategies. The experiments presented in this paper focus on strategies where one data-reordering heuristic is performed before one iteration-reordering heuristic. Preliminary results indicate that strategies involving multiple iterations between data and iteration reordering [34], strategies involving an iteration reordering before and after the data reordering, and reordering strategies combined in a hierarchical manner (e.g., performing CPACKIter within each temporal locality hypergraph partition) can result in even better performance. Future work includes determining whether the spatial and temporal locality hypergraph metrics predict such improved performance or whether other metrics must be taken into account. Along this same direction, we would also like to pursue possible methods for combining the spatial and temporal locality metrics to better predict relative performance.

We also want to extend the models and metrics to handle situations where there are multiple loops that can possibly share data dependencies. In a situation where there are data dependencies between loops, iteration- and data-reordering heuristics such as full sparse tiling and tile packing [34] may be performed to schedule and reorder across data dependencies. Doing so requires determining a reordering strategy for each loop involved and then determining parameters for scheduling across such loops. Figure 17 shows some preliminary experiments where the reordering strategy involves a data- and iteration-reordering combination followed by full sparse tiling which groups iterations across loops in the benchmarks into tiles. Iterating over the tiles dictates the new schedule in the executor and should improve temporal locality. Although full sparse tiling does not result in much improvement when used by itself, it is able to improve upon the best combination of data and iteration reordering when composed with those reorderings. Extending the spatial and temporal locality metrics for scheduling across loops is an open question.

The metrics and experimental results shown in this paper have focused on the improved performance of the executor. The overhead of the inspector, or algorithms that actually perform the reorderings, must be amortized depending on how often the executor will be used. In order to guide the choice of data- and iteration-reordering heuristics, it may be necessary to generate models or metrics for predicting the performance of the inspector as well. Determining whether the inspector will be amortized by the improvements experienced by the executor might require models and metrics that take architectural parameters, such as cache size and TLB size, into account.

BFS on a hypergraph uses both the spatial and temporal locality hypergraphs. It is therefore a simple extension of

the algorithm presented in Figure 10 to reorder data and iterations simultaneously. Indices for the data items and iterations are added to their respective permutation arrays as they are visited. For generality, the temporal locality hypergraph and edgelist can be replaced by primal and dual hypergraphs, respectively. If the spatial locality hypergraph (vertices are data, (hyper)edges are iterations) is primal, the resulting reordering is equivalent to a BFS data reordering followed by a CPACKIter iteration reordering. If the temporal locality hypergraph (vertices are iterations, hyperedges are data) is primal, the resulting reordering is equivalent to a BFS iteration reordering followed by a CPACK data reordering. Combining the data- and iteration-reordering phases should reduce the inspector overhead and may serve as inspiration for other heuristics that operate on the spatial and temporal locality hypergraphs simultaneously.

We describe the spatial and temporal hypergraph models with specific benchmark examples from the domains of mesh-based and molecular dynamics applications. We evaluate the ability of the metrics to select good data-reordering and iteration-reordering heuristics with benchmarks and datasets from these domains. However, the models themselves are applicable to any loop with irregular accesses. Whether or not the metrics work in the general case is an open problem.

7. CONCLUSIONS

Run-time data- and iteration-reordering transformations can significantly improve the memory system performance of irregular applications through improved spatial and temporal locality. When the overhead of performing and evaluating several reordering heuristics can be amortized over a large number of executor iterations, metrics are needed to guide the choice of reordering. The metrics must be inexpensive to compute and reasonably effective at identifying the superior reordering heuristic. We evaluate the effectiveness of a previously proposed spatial locality metric to select among various data-reordering heuristics and find that the metric is able to select a data-reordering heuristic that results in performance within 2% of that realized by the best data-reordering heuristic for experiments run on a Xeon Pentium 4, Sun-Fire-280R, and PowerPC G5. For one dataset on the Opteron 242, the performance was within 8%.

For selecting iteration-reordering heuristics, we introduce the temporal locality hypergraph model and the corresponding temporal locality metric. We evaluate the temporal locality metric on several iteration-reordering heuristics including three new strategies inspired by the temporal hypergraph model. Using the iteration reordering heuristic with the lowest temporal locality metric value results in performance that is within 10% of the performance realized after application of the best iteration-reordering heuristic. The breadth-first search for iterations (BFSIter) iteration-reordering heuristic outperforms all existing heuristics on 22 molecular dynamic datasets run with the `moldyn` benchmark. One of the four iteration reordering heuristics based on the temporal locality hypergraph (CPACKIter, BFSIter, HPart, CPACKIter-HPart) results in the best performance for each of the 17 sparse matrix datasets run with the `irreg` benchmark.

Finally, we generalize the spatial locality graph model to a spatial locality hypergraph for applications that iterate over triangles and tetrahedra. Experiments with a mesh-improving application indicate that hypergraph-based heuristics outperform edge-based heuristics.

8. ACKNOWLEDGMENTS

This work was supported in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy, Office of Science, under Contract W-31-109-Eng-38.

We thank Todd Munson (Argonne National Laboratory) for supplying and explaining the mesh quality improvement benchmark and the COSMIC project (University of Maryland) for supplying implementations of several traditional reordering heuristics and the `irreg` and `moldyn` benchmarks. We thank Gail Pieper and Boyana Norris for proofreading this paper and Alan Cox and the anonymous referees for many insightful comments and useful recommendations.

9. REFERENCES

[1] M. F. Adams. Finite element market. http://www.cs.berkeley.edu/~madams/femarket/index.html.

[2] I. Al-Furaih and S. Ranka. Memory hierarchy management for iterative graph structures. In *Proceedings of the 1st Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*, pages 298–302, March 30–April 3, 1998.

[3] B. Brooks, R. Bruccoleri, B. Olafson, D. States, S. Swaminathan, and M. Karplus. CHARMM: A program for macromolecular energy, minimization and dynamics calculations. *Journal of Computational Chemistry*, 187(4), 1983.

[4] A. C. Calder, B. C. Curtis, L. J. Dursi, B. Fryxell, G. Henry, P. MacNeice, K. Olson, P. Ricker, R. Rosner, F. X. Timmes, H. M. Tufo, J. W. Truran, and M. Zingale. High-performance reactive fluid flow simulations using adaptive mesh refinement on thousands of processors. In *Proceedings of SC2000*, 2000.

[5] P. Carloni. PDB coordinates for HIV-1 Nef binding to Thioesterase II. http://www.sissa.it/sbp/bc/-publications/publications.html.

[6] U. Catalyurek. Partitioning tools for hypergraph. http://www.cs.umd.edu/~umit/software.htm.

[7] COSMIC group, University of Maryland. COSMIC software for irregular applications. http://-www.cs.umd.edu/projects/cosmic/software.html.

[8] CUBIT Development Team. CUBIT mesh generation environment volume 1: Users manual.

[9] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 24th National Conference ACM*, pages 157–172, 1969.

[10] 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*, 32:489–496, March 1992.

[11] T. Davis. University of Florida sparse matrix collection. http://www.cise.ufl.edu/research/sparse/matrices/.

[12] C. Ding and K. Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 229–241, May 1–4, 1999.

[13] C. Ding and K. Kennedy. Inter-array data regrouping. In *Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing*, volume 1863 of *Lecture Notes in Computer Science*. Springer-Verlag, August 1999.

[14] C. Ding and Y. Zhong. Predicting whole-program locality through reuse distance analysis. In *Proceedings of the 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2003.

[15] J. Fu, A. Pothen, D. Mavriplis, and S. Ye. On the memory system performance of sparse algorithms. In *Eighth International Workshop on Solving Irregularly Struct ured Problems in Parallel*, 2001.

[16] M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some simplified NP-complete graph problems. *Theoretical Computer Science*, 1:237–267, 1976.

[17] H. Han and C. Tseng. A comparison of locality transformations for irregular codes. In *Proceedings of the 5th International Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, volume 1915 of *Lecture Notes in Computer Science*. Springer, 2000.

[18] H. Han and C.-W. Tseng. Locality optimizations for adaptive irregular scientific codes. Technical report, Department of Computer Science, University of Maryland, November 2000.

[19] L. H. Harper. Optimal assignments of numbers to vertices. *SIAM Journal*, 12(1):131–135, 1964.

[20] H. Heller. PDB coordinates for bacteriorhodopsin in a POPC membrane. http://www.lrz-muenchen.de/~heller/membrane/membrane.html.

[21] H. Heller, M. Schaefer, and K. Schulten. Molecular dynamics simulation of a bilayer of 200 lipids in the gel and in the liquid crystal phases. *J. Phys. Chem.*, 97:8343–8360, 1993.

[22] E. Im and K. Yelick. Optimizing sparse matrix computations for register reuse in sparsity. In V. N. Alexandrov, J. J. Dongarra, B. A. Juliano, R. S.Renner, and C. J. K. Tan, editors, *Computational Science - ICCS 2001*, volume 2073 of *Lecture Notes in Computer Science*, pages 127–136. Springer, May 28-30, 2001.

[23] K. London, J. Dongarra, S. Moore, P. Mucci, K. Seymour, and T. Spencer. End-user tools for application performance analysis using hardware counters. In *International Conference on Parallel and Distributed Computing Systems*, August 2001.

[24] J. Mellor-Crummey, D. Whalley, and K. Kennedy. Improving memory hierarchy performance for irregular applications. In *Proceedings of the 1999 ACM SIGARCH International Conference on Supercomputing (ICS)*, pages 425–433, June 20–25 1999.

[25] J. Mellor-Crummey, D. Whalley, and K. Kennedy. Improving memory hierarchy performance for irregular applications using data and computation reorderings. *International Journal of Parallel Programming*, 29(3):217–247, June 2001.

[26] R. Mirchandaney, J. H. Saltz, R. M. Smith, D. M. Nicol, and K. Crowley. Principles of runtime support for parallel processors. In *Proceedings of the 1988 ACM International Conference on Supercomputing (ICS)*, pages 140–152, July 1988.

[27] N. Mitchell, L. Carter, and J. Ferrante. Localizing non-affine array references. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, pages 192–202, October 12–16, 1999.

[28] N. Mitchell, L. Carter, and J. Ferrante. A modal model of memory. In V. N. Alexandrov, J. J. Dongarra, B. A. Juliano, R. S.Renner, and C. J. K. Tan, editors, *Computational Science - ICCS 2001*, volume 2073 of *Lecture Notes in Computer Science*. Springer, May 28-30, 2001.

[29] T. Munson. Mesh shape-quality optimization using the inverse mean-ratio metric. Technical Report ANL/MCS-P1136-0304, Mathematics and Computer Science Division, Argonne National Laboratory, 2004.

[30] C.-W. Ou, M. Gunwani, and S. Ranka. Architecture-independent locality-improving transformations of computational graphs embedded in k-dimensions. In *Proceedings of the International Conference on Supercomputing*, 1994.

[31] L. Rauchwerger. Run-time parallelization: Its time has come. *Parallel Computing*, 24(3–4):527–556, 1998.

[32] J. R. Shewchuk. Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In M. C. Lin and D. Manocha, editors, *Applied Computational Geometry: Towards Geometric Engineering*, volume 1148 of *Lecture Notes in Computer Science*, pages 203–222. Springer-Verlag, May 1996. From the First ACM Workshop on Applied Computational Geometry.

[33] 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*, 27(2):118–141, June 1995.

[34] M. M. Strout, L. Carter, and J. Ferrante. Compile-time composition of run-time data and iteration reorderings. In *Proceedings of the 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2003.

[35] Theoretical and Computational Biophysics Group, University of Illinois. ER-GRE and APoA1 datasets. http://www.ks.uiuc.edu/Research/namd/utilities/.

[36] H. Yu, F. Dang, and L. Rauchwerger. Parallel reductions: An application of adaptive algorithm selection. In *Proceedings of the 15th Workshop on Languages and Compilers for Parallel Computing (LCPC)*, July 2002.