Evaluation of Hierarchical Mesh Reorderings

Michelle Mills Strout[†], Nissa Osheim[†], David Rostron[†], Paul D. Hovland[‡], Alex Pothen[§]

† Colorado State University, ‡ Argonne National Laboratory, § Purdue University

Abstract. Irregular and sparse scientific computing programs frequently experience performance losses because of inefficient use of the memory system in most machines. Previous work has shown that, for a graph model, performing a partitioning and then reordering within each partition (hierarchical reordering) improves performance. More recent work has shown that reordering heuristics based on a hypergraph model result in better reorderings than those based on a graph model. This paper studies the effects of hierarchical reordering strategies within the hypergraph model. In our experiments, the reorderings are applied to the nodes and elements of tetrahedral meshes, which are used as input to a mesh optimization application. This application includes computations such as loops over the elements in the mesh and sparse matrix multiplication based on the structure of the mesh. We show that cache performance degrades over time with consecutive packing, but not with breadth-first ordering, and that hierarchical reorderings involving hypergraph partitioning followed by consecutive packing or breadth-first orderings in each partition improve overall execution time.

1 Introduction

Irregular scientific computing applications often achieve less than 10% of peak performance on current high performance computation systems, whereas on some systems dense matrix multiply can achieve more than 90% of peak performance [9]. This gap in performance between dense (and regular) computations and sparse (and irregular) computations has been called the "sparse matrix gap" [2]. The sparse matrix gap can be attributed primarily not to poor scaling but to poor single-processor performance because of irregular memory references.

Within the context of this paper, we focus on irregular memory references due to indirect array addressing, which occurs in many applications such as partial differential equation solvers, molecular dynamics simulations, finite element analysis, mesh manipulation applications, and computations involving sparse matrix data structures. Figure 1 shows a loop with indirect array references that traverses triangular elements in a mesh. Each array entry in `data` could contain multiple fields such as the x and y coordinates of the corresponding vertex. Indirect memory references such as `data[n1[i]]` can exhibit poor data locality, and therefore cause performance problems. Improving data locality in irregular applications has been shown to improve parallel performance even more than serial performance [10, 12, 20, 15].

Previous research has studied various heuristic data and computation reorderings for improving data locality [1, 17, 7, 12, 16, 27, 24], but automatic determination of the

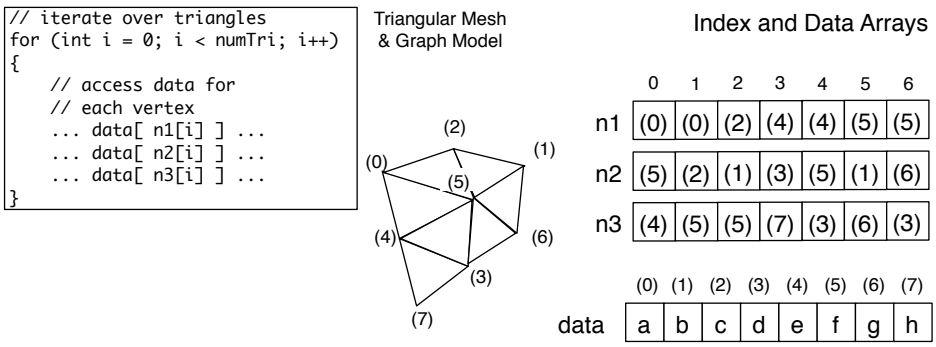


Fig. 1. Loop iterating over triangular elements and example triangular mesh and index arrays that represent the mesh topology. For each triangle i, the indices for the three vertices are stored in $n1[i]$, $n2[i]$, $n3[i]$.

reordering strategy that results in the best performance improvement remains an open problem. Toward solving this problem, this paper contributes a performance study of how hypergraph partitioning combined with a low overhead consecutive packing reordering [7] or a high performance hypergraph breadth-first ordering [24] affects the execution of a mesh optimization algorithm. In the context of Figure 1, a data reordering involves reordering the entries in the `data` array. Each iteration of the loop in Figure 1 is a computation. Figure 1 shows an example triangular mesh for use with the code in Figure 1, where the data associated with one triangle is visited at each iteration of the loop. The data for each node in the mesh, (0), (1), etc., is stored in the corresponding entry in the *data* array. The *index* arrays store the mesh topology. An iteration, or computation, reordering involves rearranging the values in the index arrays and logically corresponds to permuting the order that triangles in the mesh are visited. Both the data and computation reordering problems can be modeled as the minimal linear arrangement problem, which is NP-complete.

Previous work on data reordering for irregular applications observed that *hierarchical*, or *hybrid*, heuristics can result in a 5 to 10% performance improvement over local heuristics alone [1]. Hierarchical heuristics perform a graph partitioning and then use a local reordering heuristic within each partitioning. The hierarchical technique proposed in [1] entails a graph partitioning , followed by a breadth-first ordering within each partition.

More recent research showed that local reordering heuristics (e.g., consecutive packing and breadth-first ordering) based on the hypergraph model perform up to 30% better than those based on a graph model in computations when three or more pieces of data are accessed within each iteration of the loop [24]. A hypergraph model groups any number of nodes into hyperedges. For the example in Figure 1, each triangle could be represented with a hyperedge.

This paper studies the effect of hierarchical reordering in concert with a hypergraph-model based consecutive packing (Hyper-CPACK) or breadth-first ordering (Hyper-BFS) heuristics. We hypothesize that the performance of a computation when the data has been reordered using consecutive packing will degrade over the course of the computation. We also hypothesize that performing a hypergraph partitioning followed

by a consecutive packing within each partition will improve performance because of the degradation within a partition having less effect than degradation over the full computation. For a breadth-first ordering based on a hypergraph model (Hyper-BFS), we hypothesize some improvement with hierarchical reordering, but not much because previous work [18] has shown that a breadth-first ordering using a hypergraph model achieves a high percentage of the memory bandwidth limit.

This paper makes the following contributions:

- algorithms for hierarchical reordering within the hypergraph model for both consecutive packing and breadth-first orderings within each partition;
- experimental results showing that cache performance degrades over time with consecutive packing and not with breadth-first on a hypergraph; and
- experimental results showing that hierarchical reordering on the hypergraph prevents the performance degradation since the local reorderings are performed within partitions.

2 Heuristics Using the Hypergraph Model

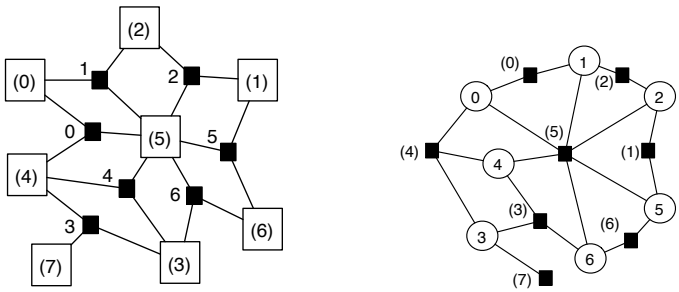
The distinguishing feature of hypergraphs are hyperedges, which are capable of connecting any number of nodes. Figure 2(a) shows the hypergraph that models relationships between the nodes in the mesh in Figure 1. In the figure, the square vertices with parenthesized numbers directly correspond to nodes in the mesh. The filled-in squares are hyperedges connecting all the vertices of the element the hyperedge represents. Figure 2(b) is a dual hypergraph for the hypergraph in Figure 2(a). Hyper-BFS and the hierarchical version of Hyper-BFS use both the hypergraph and the dual hypergraph.

We use various combinations of six orderings for the experiments in this paper: original order, Hyper-BFS, Hyper-CPack, Hyper-Pack, HierBFS (Hierarchical BFS), and HierCPack (Hierarchical consecutive packing). We know from earlier work that heuristics based on hypergraph models outperform graph models [24]; we know hierarchical orderings are better capable of limiting the growth of the working set, therefore in this work we hypothesize and show that combining hierarchical orders with hypergraph models do well.

Hypergraph Consecutive Packing (Hyper-CPack): One heuristic that most naturally generalizes to a hypergraph model is consecutive packing. The consecutive packing heuristic [7] packs the data associated with nodes in the hypergraph in the order that they are visited within the computation. For the example in Figure 1, that means visiting the triangles in their given order and packing data for nodes as each node is seen in that order. It is critical that the tetrahedrons should be ordered well in this scheme, which is why our baseline mesh orderings start with tetrahedrons lexicographically sorted by the nodes they contain. Consecutive packing is commonly used because of its low overhead and reasonable resulting performance improvement.

Hyper-CPack on the example in Figure 1 first orders the nodes of triangle 0: (0), (4), (5); then triangle 1: (2); and so on. The following data ordering results: (0), (4), (5), (2), (1), (3), (7), (6). The nodes in the first elements are guaranteed to be near each other, but the nodes in the later elements can be spread out because some of them have already been placed by earlier elements in which they are included. Therefore we hypothesize that the performance will deteriorate for the elements/iterations near the end of the ordering.

Hypergraph Breadth-First (Hyper-BFS): Hyper-BFS [24] also operates on the hypergraph. It starts at the first node and performs a breadth-first traversal, placing



(a) Hypergraph (b) Dual hypergraph

Fig. 2. The hypergraph and dual hypergraph models of the relationship between data and computation. The small black squares represent hyperedges. Numbers in parentheses represent nodes in the original mesh and the other numbers represent triangles in the original computation.

the nodes in the order the traversal visits them. Other heuristics for selecting the start node for the traversal exist [8], but our experience has not indicated that the starting node significantly affects the final performance. Hyper-BFS on the hypergraph visits all neighboring nodes that are part of the same hyperedge, before going on to other neighboring nodes. When Hyper-BFS orders a neighboring node, it orders all currently unordered nodes that are part of the same element before it orders other neighboring nodes. For example, in Figure 1, a Hyper-BFS starting at node (0) could have the ordering (0), (4), (5), (2), (3), (7), (1), (6).

Hypergraph Partitioning (Hyper-Part): Hypergraph partitioning decomposes the nodes of a hypergraph into disjoint sets. A reordering heuristic based on hypergraph partitioning then orders the nodes by partition. We have not done an extensive study, but differences between mesh partition quality given by various partitioners do not appear to have a significant effect our results. This is probably due to the fact that we are using the partitioners for single core data locality and not for parallelization. We use PaToH [5] as our hypergraph partitioner. If a hypergraph partitioner is used alone, the nodes and elements within each partition are left in their original order. If used as part of a hierarchical reordering, the local reordering is used within each partition created by a hypergraph partitioner. For these experiments, we set the size of the partitions so that the memory accesses of the computations for each partition fit into a fraction of the L2 cache (1/2 to be exact), which is a heuristic that has been used previously [22].

Hierarchical Consecutive Packing (HierCPACK): The hierarchical reordering heuristics, which are the new contributions of this paper, start with a hypergraph partitioning of the nodes and then perform a local reordering of the nodes within each partition. Hierarchical consecutive packing proceeds by visiting each hyperedge in the hypergraph model in the order that the hyperedges will be visited during run-time computations and packing the nodes in those hyperedges on a per partition basis. The algorithm visits all hyperedges in order and then maintains one packing lists of nodes for each partition. The final ordering concatenates all of the packing lists.

As an example, for the hypergraph in Figure 2(a) assume two partitions with nodes (0), (1), (2), and (5) in the first partition and nodes (3), (4), (6), (7) in the second partition. An in-order visit to the hyperedges (represented as black filled squares) will result in the node ordering (0), (5), (2), (1) for the first partition and (4), (3), (7), (6) for the second partition. Concatenating the orders for the partitions will result in the following ordering: (0), (5), (2), (1), (4), (3), (7), (6).

Hierarchical Breadth-First (HierBFS): Hierarchical Breadth-First also uses a hypergraph partitioning for hierarchical reordering. A breadth-first traversal over the nodes in each partition provides the local ordering. As with the non-hierarchical breadth-first ordering over a hypergraph, both the primal and dual hypergraphs are used to perform this reordering.

The algorithm first selects a root node for the breadth-first traversal from each partition. Next, it loops though the partitions, and for each partition it uses a queue data structure to perform a breadth-first traversal of the nodes based on adjacent hyperedges. Then, it loops through all unvisited neighbors in all the hyperedges and adds those to the new ordering and a queue. When it has searched through all the hyperedges for the root node, which can be found by accessing the dual hypergraph, it repeats the process for the next node in the queue. This process continues until all nodes in the partition have been added to the new ordering or it runs out of nodes in the queue. If the queue runs out before all the nodes in the partition have been visited, it searches the nodes for one in this partition that has not yet been visited and uses it as a new root node.

To illustrate, assume the nodes in the example in Figure 2(a) are split into two partitions: the first partition containing nodes (0), (1), (2), and (5), and the second partition containing nodes (3), (4), (6), (7). The root node for the first partition would be node (5). Assuming the hyperedges adjacent to (5) are visited in order, a breadth-first ordering of the first partition would be (5), (0), (2), (1). The breadth-first ordering for the second partition would be (7), (3), (4), (6).

3 Experimental Results

We test the efficacy of the data and iteration/element reorderings by reordering real mesh data sets and feeding the reordered meshes into the FeasNewt mesh-quality optimization benchmark [18]. FeasNewt optimizes the quality of the tetrahedra by adjusting the coordinates of the internal mesh vertices. Higher-quality tetrahedra improve the accuracy and speed of computations or simulations using a discretization method. This approach does not change the topology of the mesh or the external shape.

FeasNewt has calculations and memory access patterns similar to those found in many scientific computing applications [18]. FeasNewt's gradient evaluation, Hessian computation, and a sparse matrix-vector product take the majority of its execution time. The gradient evaluation and Hessian computations iterate over mesh elements while the matrix-vector product operates on a sparse matrix with a row for each mesh node with non-zero blocks for higher-numbered neighboring mesh nodes. Although FeasNewt iteratively optimizes the mesh quality until convergence is reached, none of the reorderings used in this study alter the number of convergence iterations.

For input, we use six irregular tetrahedral meshes modeling different physical entities and from varying mesh generators (see Table 1). The sources of the meshes include tetrahedral volume mesh generated using TetGen [23] using surface meshes from the INRIA Gamma team research database [13] (INRIA and TetGen), meshes generated

Table 1. Mesh data-set information.

Mesh	# Nodes	# Elements	Size in MB	Comments
1-001.mesh	120,399	725,258	25.4	INRIA and TetGen
dna.mesh	185,823	938,168	33.3	INRIA and TetGen
ductbig.mesh	177,887	965,759	31.5	CUBIT
gear.mesh	285,640	1,595,392	58.3	CUBIT
sf2.mesh	378,747	2,067,739	61.6	CMU UMS
ucol.mesh	477,977	1,955,366	67.0	BioMesh

using CUBIT [6], a mesh of the San Fernando Valley in Southern California from the CMU Unstructured Mesh Suite [19] (CMU UMS), and a mesh of parts of the circulatory system, generated as part of BioMesh Project [4], courtesy Chaman Singh Verma at Argonne. For the original ordering (baseline), the mesh node ordering provided by a mesh generator is used and all mesh elements are lexicographically sorted.

Our experiments were performed on a quiescent HP-xw9300 with 2 GB of memory and dual 64-bit AMD Opteron 250 2.4 GHz processors with 128 KB L1 cache and 1 MB L2 cache per processor. The code is single-threaded and therefore uses only one of the processors.

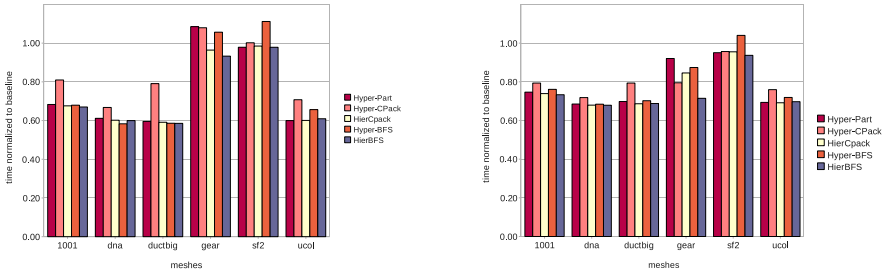
3.1 Effect of Hierarchical Reordering

Hierarchical data reordering improves performance over local reordering strategies alone (Hyper-CPack and Hyper-BFS), although for Hyper-BFS the improvement is minimal. Execution times for the full FeasNewt benchmark and the Hessian computation only are shown in Figures 3(a) and 3(b), respectively. The execution times are normalized to the execution time for the benchmark when the original ordering is used and shown for each mesh and reordering strategy. The hierarchical reordering strategies show the most improvement for most of the meshes for full FeasNewt benchmark execution times. In some cases, the performance improvement over the original mesh ordering is 40%.

One observation for Figure 3(a) is that although the hierarchical reordering clearly improves over the local ordering consecutive packing, the hierarchical reorderings do not significantly improve over a global ordering based on hypergraph partitioning. One exception to this is the gear mesh, where hierarchical reorderings are the only reorderings that do not cause a slowdown. Possible future work is determining whether hierarchical reorderings can be "proven" safe from the standpoint of never causing slowdown.

3.2 Fine-Grained Cache Miss Results

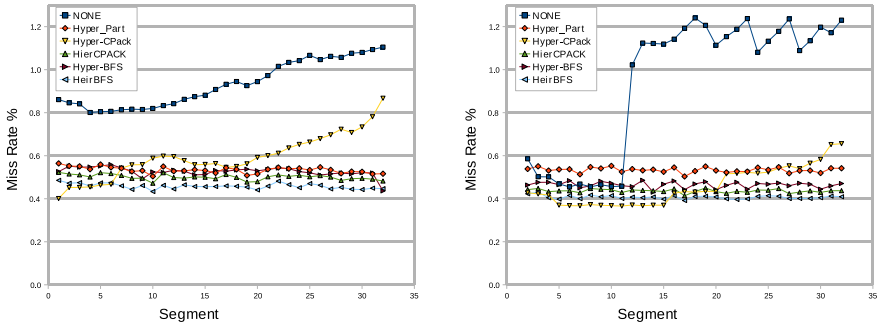
We now demonstrate why hierarchical reordering improves over a consecutive packing alone. The performance due to a consecutive packing degrades over time, and hierarchical reordering evens out the performance benefits by doing consecutive packing within localized partitions. We observe this degradation by using a novel approach to studying the effect of data reordering. Specifically, we break the loop over tetrahedrons



(a) Normalized execution times for FeasNewt benchmark.

(b) Normalized execution times for Hessian computation only.

Fig. 3. Normalized execution times for various data reorderings on a number of input meshes. All data reorderings are followed by Hyper-CPack iteration reordering.



(a) ductbig

(b) dna

Fig. 4. L1 miss rates by segment over a run. All data reorderings are followed by a Hyper-CPACK iteration reordering.

in the Hessian computation into equal-sized segments and record the cache miss rates for each segment. This approach exposes the data ordering quality as the computation progresses through the iteration ordering.

We use PAPI [14] to instrument FeasNewt to record L1 cache, L2 cache, and TLB hits and misses at 32 regular intervals, hereafter called segments, in the Hessian computation. The Hessian computation is performed multiple times per outer convergence loop. We therefore record a weighted running average for each segment's measurements. Execution times for these segments as well as the overall benchmark execution time were recorded. We present the minimum execution time from three runs.

To determine if Hyper-CPack and Hyper-BFS degrade over time and if hierarchical reordering with HierCPack and HierBFS stop this degradation, we observe the L1 cache miss rate by segment over the runs. Figure 4 shows L1 cache miss rates for the 32 segments of the Hessian loop. The graphs show the data reorderings for the ductbig and dna. The trends seen in ductbig and dna are typical of the trends seen in the other

meshes, except for sf2. sf2 does not show much improvement in the cache miss rate over the segments, because it appears to already be well ordered.

Hyper-CPack does show the expected degradation over time, and HierCPack levels out this degradation and reduces the the overall cache misses as hypothesized (See the triangles pointing down and the triangles pointing up in Figure 4). HierCPack can eliminate the performance degradation of Hyper-CPACK. Unlike Hyper-CPACK, Hyper-BFS does not show a degradation. Nonetheless, HierBFS still has consistently lower miss rates than Hyper-BFS and marginally better performance.

4 Related Work

Making decisions among all of the reordering heuristics is an open problem. Some work has done comparison among subsets [21, 12, 20, 16]. However, since such comparisons might be relevant only to the specific benchmarks and datasets in the study, no clear winner exists. Typically, an ordering is selected due to results from earlier work on similar problems, or the desire to keep reordering costs low. Other work has used metrics to select among various reorderings without comparing execution time, with some success [24].

Many earlier studies on memory system performance of irregular codes have focused on reordering for data locality and other optimizations for sparse matrix-vector multiplication [25–27, 20]. The mesh optimization benchmark [18] used in our experiments includes a symmetric, blocked sparse matrix multiply as well as iteration over a large tetrahedral mesh data structure. We observe that orderings based on a model of computation over the mesh data structure also improve the performance of the sparse matrix-vector multiply. Techniques specific to sparse matrices such as register tiling [27] might lead to even further performance benefits in the mesh optimization benchmark.

Our work differs from previous research in that the effect of data reordering on execution time and the memory hierarchy is explored at a finer granularity and in the context of multiple real datasets for a single benchmark. Previous work [11] has looked at the degradation of performance as the relationship between nodes in a molecular dynamics application changes. The granularity that we look at is smaller, since we focus on segments of one sweep over the mesh.

This paper studies the detailed differences between two local reordering heuristics and a hypergraph partitioning heuristic coupled with a local reordering heuristic. Al Furaih and Ranka [1] showed experimentally that hierarchical reorderings within a graph model improve performance, and later research has used some form of partitioning followed by a reordering within each partition [10, 3]. This paper provides a similar basis for performing hierarchical reordering in reorderings based on hypergraph models. Selecting between hierarchical reorderings on a graph model versus a hypergraph model remains an open problem, but we hypothesize that, based on previous comparisons between the two models [24], hierarchical reordering on the hypergraph model will prevail.

5 Conclusion

Reordering the data and computation within irregular applications is important for improved data locality and performance. This paper presents new hierarchical heuristics based on a hypergraph model of the data reuse between computations. The new

heuristics, hierarchical consecutive packing and hierarchical breadth-first, depend on a hypergraph partitioning followed by local reorderings within each partition. Our results show that hierarchical consecutive packing does improve performance in comparison with consecutive packing alone. More detailed experiments show that consecutive packing degrades in performance later in the ordering. When partitioning is done before the consecutive packing, each partition degrades separately and the overall degradation is not as severe. Hierarchical reordering does not improve significantly over a breadth-first ordering of the nodes in a hypergraph. Based on hardware counter results, we conclude that a breadth-first reordering on the hypergraph model does not result in the same degradation as consecutive packing and therefore does not benefit much from the grouping provided by hypergraph partitioning.

6 Acknowledgements

This work was supported by the CSCAPES Institute (DE-AC02-06CH11357 at Argonne) funded through DOE's SciDAC program. We thank Kevin Depue and Jinghua Fu for their programming efforts that contributed to this paper. We thank Gail Pieper for proofreading a draft of this paper.

References

1. 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.
2. Anonymous (The High End Crusader). U.S. has sparse matrix gap. In *HPCC Week*, pages 7–8, November 23 1998.
3. A.-H. A. Badawy, A. Aggarwal, D. Yeung, and C.-W. Tseng. Evaluating the impact of memory system performance on software prefetching and locality optimizations. In *International Conference on Supercomputing*, pages 486–500, 2001.
4. BioMesh Project, an all-hex meshing strategy for bifurcation geometries. http://wwwunix.mcs.anl.gov/~csverma/BioMesh/biomesh.html.
5. U. Catalyurek and C. Aykanat. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):673–693, 1999.
6. CUBIT, Geometry and Mesh Generation Toolkit. http://cubit.sandia.gov/.
7. C. Ding and K. Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 229–241, May 1999.
8. 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.
9. K. Goto and R. A. van de Geijn. Anatomy of a high-performance matrix multiplication. *ACM Transactions on Mathematical Software*, 34(3), 2008.
10. W. D. Gropp, D. K. Kaushik, D. E. Keyes, and B. F. Smith. Performance modeling and tuning of an unstructured mesh CFD application. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, 2000.

11. H. Han, G. Rivera, and C.-W. Tseng. Software support for improving locality in scientific codes. In *8th Workshop on Compilers for Parallel Computers (CPC'2000)*, Aussois, France, January 2000.
12. 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 LCNS 1915. Springer, 2000.
13. INRIA Gamma team research database. http://www-c.inria.fr/gamma/gamma.php.
14. 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.
15. M. J. Martin, D. E. Singh, and J. Tourino. Exploiting locality in the run-time parallelization of irregular loops. In *International Conference on Parallel Processing (ICPP)*, August 18-21 2002.
16. 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, 2001.
17. N. Mitchell, L. Carter, and J. Ferrante. Localizing non-affine array references. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 192–202, October 1999.
18. T. S. Munson and P. D. Hovland. The FeasNewt benchmark. In *The IEEE International Symposium on Workload Characterization (IISWC 2005)*, October 2005.
19. D. R. O'Hallaron and J. R. Shewchuk. CMU Unstructured Mesh Suite. http://www.cs.cmu.edu/~quake/meshsuite.html.
20. L. Oliker, X. Li, P. Husbands, and R. Biswas. Effects of ordering strategies and programming paradigms on sparse matrix computations. *SIAM Review*, 44(3):373–393, 2002.
21. C. 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*, 1995.
22. V. K. Pingali, S. A. McKee, W. C. Hseih, and J. B. Carter. Computation regrouping: restructuring programs for temporal data cache locality. In *Proceedings of the 16th International Conference on Supercomputing*, pages 252–261, 2002.
23. H. Si. TetGen, a quality tetrahedral mesh generator and three-dimensional delaunay triangulator. http://tetgen.berlios.de/.
24. M. M. Strout and P. D. Hovland. Metrics and models for reordering transformations. In *Proceedings of the The Second ACM SIGPLAN Workshop on Memory System Performance (MSP)*, pages 23–34, June 2004.
25. V. E. Taylor. Sparse matrix computations: implications for cache designs. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, pages 598–607, 1992.
26. S. Toledo. Improving the memory-system performance of sparse-matrix vector multiplication. *IBM Journal of Research and Development*, 41(6):711–725, 1997.
27. R. Vuduc, J. W. Demmel, K. A. Yelick, S. Kamil, R. Nishtala, and B. Lee. Performance optimizations and bounds for sparse matrix-vector multiply. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, pages 1–35, 2002.