A Comparison of Locality Transformations for Irregular Codes

Hwansoo Han, Chau-Wen Tseng

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

Abstract. Researchers have proposed several data and computation transformations to improve locality in irregular scientific codes. We experimentally compare their performance and present GPART, a new technique based on hierarchical clustering. Quality partitions are constructed quickly by clustering multiple neighboring nodes with priority on nodes with high degree, and repeating a few passes. Overhead is kept low by clustering multiple nodes in each pass and considering only edges between partitions. Experimental results show GPART matches the performance of more sophisticated partitioning algorithms to with 6%–8%, with a small fraction of the overhead. It is thus useful for optimizing programs whose running times are not known.

1 Introduction

As microprocessors become increasingly fast, memory system performance begins to dictate overall performance. The ability of applications to exploit locality by keeping references in cache becomes a major factor affecting performance. Computational scientists, however, are writing programs with increasingly irregular memory access patterns as they attempt to create more complex simulations. These *irregular* computations arise in several application domains. In computational fluid dynamics (CFD), meshes for modeling large problems are sparse to reduce memory and computations requirements. In n-body solvers such as those arising in astrophysics and molecular dynamics, data structures are by nature irregular because they model the positions of particles and their interactions.

Unfortunately, irregular computations have poor temporal and spatial locality because they do not repeatedly access data in memory with small constant strides. Researchers recently showed locality can be improved for irregular codes using compiler and run-time data layout and computation reordering transformations [1, 8, 27, 28]. In this paper, we form a framework for applying locality optimizations to irregular codes and experimentally evaluate the impact of a number of optimization techniques. The contributions of this paper are:

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

– Experimentally evaluate the effectiveness of several locality optimizations for a range of input data and programs. We are the first to use RCB for improving cache performance, and the first to combine RCM and METIS with computation reordering.
– Develop GPART, a new locality transformation based on hierarchical clustering which nearly matches the performance of more sophisticated partitioning algorithms, but with a small fraction of the overhead.

We begin with a refresher on existing optimization techniques, then present our new algorithm and evaluate its performance.

2 Locality Optimizations

A number of locality optimizations have been developed for irregular computations. We can decide which are needed based on application structure.

Application Structure Irregular scientific computations are typically composed of loops performing reductions such as SUM and MAX which can be safely reordered [6, 8]. For locality optimizations, it turns out the number of distinct irregular accesses made by each loop iteration can be used to choose the proper optimization algorithm.

In some codes, each iteration makes only one irregular access to each array. The NAS integer sort (IS) and sparse matrix vector multiplication found in conjugate gradient (CG) fall into this category. More often, scientific codes access two or more distinct irregular references on each loop iteration. Such codes arise in PDE solvers traversing irregular meshes or N-body simulations, when calculations are made for pair of points. In such cases loop iterations may be considered edges connecting the data accessed on that iteration.

Figure 1 shows an example of data/computation reordering for these computations. Circles represent computations (loop iterations), squares represent data, and arrows represent data accesses. Notice how computations can be viewed as edges connecting data nodes, resulting in a graph. Partitioning the graph and putting nodes in a partition close in memory can then improve spatial locality. Sorting the loop iterations can improve temporal locality further.

Lexicographical Sorting One optimization is to reorder loop iterations. If each iteration performs only one irregular access, sorting the loop iterations by data accessed results in optimal locality. Bucket sorting with cache-sized buckets can also be used, producing similar improvements with low overhead [28]. For computations with two or more irregular accesses, lexicographically sorting the loop iterations will improve temporal locality. Sorting is so effective that data meshes provided by applications writers are frequently presorted, eliminating the need to apply sorting at run time. When we perform experiments, we use presorted input meshes, and always apply lexicographical sorting after data reordering.

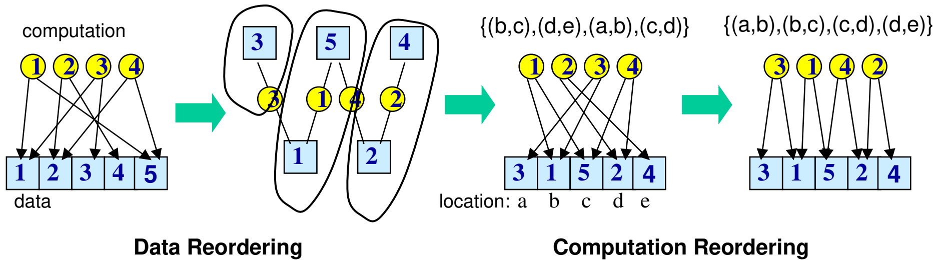

Fig. 1. Overview of Data and Computation Transformations

---

*Consecutive Packing (*CPACK*)* In addition to computation reordering, the compiler can also reorganize data layout. Ding and Kennedy proposed *consecutive packing* (CPACK), where data is moved into adjacent locations in the order they are first accessed (first-touch) by the computation [8]. CPACK can be applied with very low overhead, since it just requires one pass through the loop marking the order in which data is accessed. However, CPACK does not explicitly take into account the underlying graph or geometric structure of the data.

*Reverse Cuthill-McKee (*RCM*)* Reverse Cuthill-McKee (RCM) methods have been successfully used in sparse matrix reorganization [5, 17, 24]. RCM simply uses reverse breadth-first search (BFS) to reorder data in irregular computations, by viewing iterations as graph edges. Performance is improved with low overhead [1, 6].

*Recursive Coordinate Bisection (*RCB*)* Space-filling curves (*e.g.*, Hilbert, Morton) are continuous, non-smooth curves that pass through every point in a finite k-dimensional space [15, 27, 32]. Because interactions tend to be local, arranging data using space-filling curves reduces the distance (in memory) between two geometrically close points in space, yielding better locality [27]. When geometric coordinate information is available, sorting data and computation using space-filling curves is the locality optimization of choice, since it achieves good locality with very low overhead simply using a multidimensional bucket sort. However, geometric coordinate information is not necessarily available, and will probably require user intervention to identify. In addition, space-filling curves may not work well for unevenly distributed nodes, due to the fixed size grid.

As an alternative, we propose using recursive coordinate bisection (RCB). RCB recursively splits each dimension into two partitions by finding the median of data coordinates in that dimension. The process is recursively repeated, alternating dimension [2, 3]. After partitioning, data items are stored consecutively within each partition. Subpartitions are also arranged consecutively, creating a recursive hierarchical layout similar to Z-ORDERING, a space-filling curve for dense array layouts. RCB has higher overhead, but is likely to work well for unevenly distributed data.

*Multi-level Graph Partitioning (*METIS*)* A major limitation of RCB and space-filling curves is that geometric coordinate information is needed for each node. This information may not be available for some applications. Even if it is available, user annotations may be needed since the compiler may not be able to automatically derive coordinate information for each data item.

In comparison, graph partitioning algorithms can be applied automatically based on the loop data access pattern. Partitioning algorithms were first developed for load balancing in parallel codes [32]. Spectral partitioning methods can be effective but are computationally intensive [31]. More recently, people have employed multi-level graph partitioning algorithms encapsulated in library packages such as METIS [19, 20] to achieve good partitions with low overhead. These algorithms compute a succession of coarsened graphs (with fewer nodes) which approximate the original graph, compute a partition on the coarsened graph, and then expand back to the original graph.

3 Low-Overhead Graph Partitioning (GPART)

Previous graph partitioning algorithms were designed for load-balancing in parallel computations, VLSI circuits, and other high cost applications. As a result, greater processing time is acceptable. Experiments indicate the overhead of preprocessing for RCB and METIS are 5–40 times higher than CPACK, the lowest overhead algorithm. Since our goal is to improve cache performance at run-time for irregular applications, we desire partitioning algorithms with lower overhead. We have developed a hierarchical graph clustering algorithm (GPART) that has low overhead and produces quality ordering for cache locality.

3.1 Algorithm

The GPART algorithm performs a number of passes. On each pass, it clusters several adjacent nodes into a single partition. Each partition is then treated as a single node in the next pass, creating a hierarchical structure. An example of data partitioned by the GPART algorithm is shown in Figure 2. The GPART algorithm consists of the following steps:

1. SORT: Nodes are sorted once by their vertex degrees (number of incipient edges) from most number of neighbors to least. Sorting improves the likelihood that closely connected nodes are clustered first through multiple passes, improving the quality of the partitions produced.

2. MERGE: Visiting nodes in order of vertex degree, randomly choose partition containing neighboring node on adjacency list. If combined size is less than *pLimit* (current partition size limit), merge partitions containing the node and its chosen neighbor. Mark the neighbor as merged. Random selection

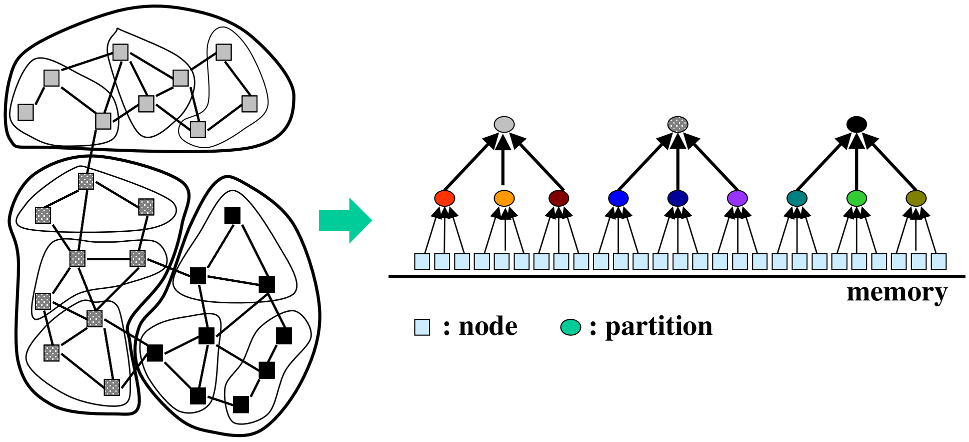

Fig. 2. Low-Overhead Graph Clustering (GPART) Example

may result in lower quality partitions than more sophisticated methods, but reduces overhead.

3. FILTER: While visiting nodes to cluster, mark neighbors as merged if they have already been merged into the partition of the current node. After visiting all nodes, marked neighbors are eliminated from the adjacency list. Fewer nodes need to be searched on each pass, speeding up future passes.

4. LOOP: Repeat STEP 2 and STEP 3, increasing the current limit on partition size (*pLimit*) by factor *growFac*, up to the maximum desired partition size (*maxLimit*). Initial *pLimit* is set to the cache line size, and *maxLimit* is set to L1 cache size divided by data accessed on each loop iteration. If *pLimit* is small, overhead increases as the algorithm iterates many times. If *pLimit* is large, the quality of partition degrades. *growFac* is experimentally set to eight.

5. LAYOUT: Finally, arrange nodes according to the resulting hierarchical layout. Nodes in a partition are recursively laid out nearby in memory, maintaining the partition hierarchy.

The full GPART algorithm is shown in Figure 3. The partitions produced by GPART are used to improve locality by generating data layouts that put neighboring nodes in nearby memory locations. The hierarchical structure in GPART is similar to that of RCB. Each partition in a level keeps all its subpartitions in lower levels close together in memory, which further improves locality. Since applications visit nodes according to the storage order due to lexicographical sorting, maintaining the hierarchical structure of the clustered graph increases the chances of visiting neighbor nodes that have already been recently accessed.

```
sort nodes by descending degree
pLimit = cacheLineSize
while (pLimit <= maxLimit) {
  for each node N (in sorted order) {
    P = partition containing N
    if (size(P) < pLimit) {
      for each neighbor M of N in adj. list {
        // pick partner in random order
        Q = partition containing M
        if (P == Q)
          mark M as merged neighbor
        else if (size(P) + size(Q) <= pLimit) {
          merge Q into P
          mark M as merged neighbor
          if (size(P) == pLimit)
            break (exit for loop)
        }
      }
    }
  }
  eliminate merged neighbors in adjacency list
  pLimit = pLimit * growFac  // new partition size
}
recursively lay out nodes according to hierarchy
```

Fig. 3. Hierarchical Graph Clustering Algorithm

These nodes are more likely to remain in cache, improving performance.

GPART is similar to the coarsening phase in METIS, but fewer clustering passes are required because more nodes are collapsed at once. Because we form clusters in a bottom-up fashion, we capture most of the locality immediately and can stop after a small number of passes. The quality of the partition produced is slightly less than that produced by METIS, but the overhead is significantly less.

We chose our parameters for GPART experimentally by performing a large number of experiments. We found that stopping after partitions reach the size of L1 cache and using a coarsening factor of eight reduced the number of passes without significantly degrading the quality of the partitions [11].

4 Experimental Evaluation

4.1 Evaluation Environment

We have begun implementation of our prototype compiler using the Stanford SUIF compiler. The prototype can identify and parallelize irregular reductions, generating parallel *pthreads* programs. The compiler can also determine where inspectors need to be inserted, based on when memory access patterns are changed. However, we have not implemented the actual inspector generation phase using techniques developed by Chaos [13, 14]. As a result, we currently insert inspectors by hand for both the sequential and parallel versions of each program.

Name	# Nodes	# Edges	Description
FOIL	144649	1074393	3D mesh of a parafoil
AUTO	448695	3314611	3D mesh of GM's Saturn
MOL1	131072	1179648	semi-uniform 3D molecule distribution (sm)
MOL2	442368	3981312	semi-uniform 3D molecule distribution (lg)

Table 1. Input Data Sets for Irregular Kernels

We used two machine configurations, DEC multiprocessor with 275MHz Alpha 21064 processors and an SGI Origin2000 with 195MHz MIPS R10000 processors. The Alpha has a 16K direct-mapped L1 cache and a 4M direct-mapped L2 cache. The R10000 has a 32K two-way set associative L1 cache and a 4M L2 cache. We also simulate cache performance using a cache simulator based on the Shade utility from Sun Microsystems.

4.2 Applications & Input Characteristics

We examine three irregular applications, IRREG, NBF, and MOLDYN. All applications contain an initialization section followed by the main computation enclosed in a sequential time-step loop. The main computation is thus repeated each iteration of the time-step loop. Statistics and timings are collected after the initialization section in order to investigate main computation execution.

IRREG is a representative of iterative partial differential equation (PDE) solvers found in computational fluid dynamics (CFD) applications. NBF is a kernel abstracted from the GROMOS molecular dynamics code [13]. MOLDYN is abstracted from the non-bonded force calculation in CHARMM, a key molecular dynamics application used at NIH to model macromolecular systems.

To test the effects of data reordering, we experiment with a variety of input data meshes described in Table 1. FOIL and AUTO are 3D meshes of a parafoil and GM Saturn automobile, respectively. MOL1 and MOL2 are small and large 3D meshes derived from semi-uniformly placed molecules of MOLDYN using a 1.5 angstrom cutoff radius. FOIL and AUTO are used for IRREG, and MOL1 and MOL2 are used for NBF and MOLDYN.

4.3 Experimental Results

In our experiments, we evaluated several locality reordering algorithms with three kernels. The different optimizations are as follows: ORIG is the original program, which corresponds to the result of computation reordering, since input meshes are presorted. RCB, METIS, and GPART apply recursive coordinate bisection, multi-level graph partitioning algorithm, and hierarchical graph clustering algorithm, respectively. After partitioning graphs, nodes in a partition are arranged nearby memory locations. RCM applies Reverse Cuthill-McKee, and CPACK arranges nodes by consecutive packing. As we mentioned earlier, computation reordering is always applied after each data reordering algorithm.

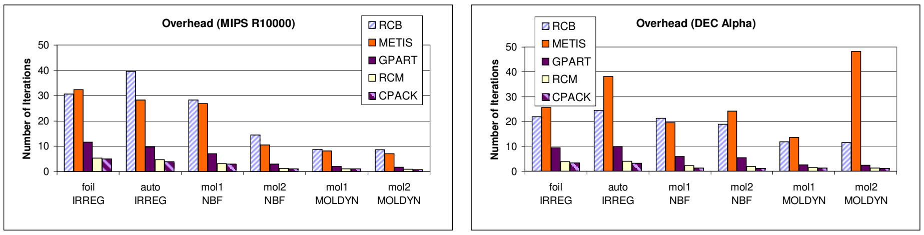

Fig. 4. Overhead of Data Reordering (relative to time-per-iteration of ORIG)

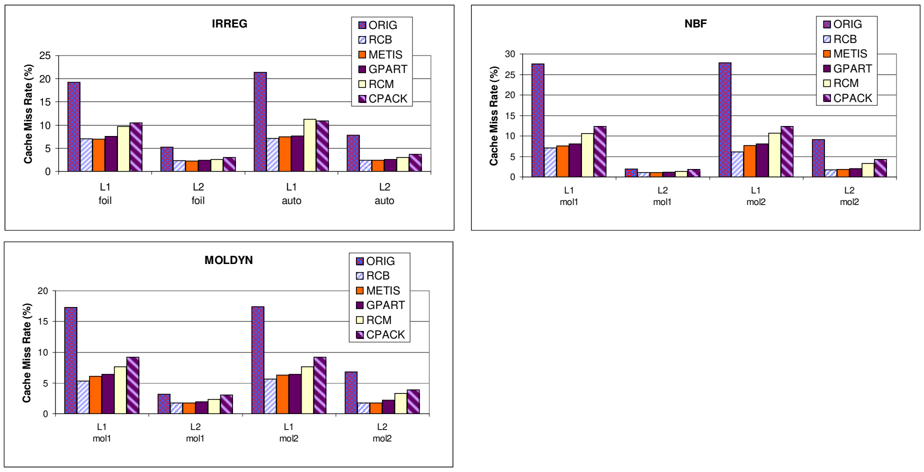

Fig. 5. L1 & L2 Cache Miss Rates (percentage of all memory references)

RCB and METIS are both configured to create partitions so that the data accessed is roughly the L1 cache size. For instance, if the 16K L1 cache holds 4K elements, and each loop iteration accesses 2 node arrays, the partition size is 2K. In GPART we start from a partition containing 4 nodes. Since each node in our applications contains an 8-byte double-precision floating point number, 4 nodes will fit into a 32-byte cache line. 8 partitions are merged into a new partition at each pass and the final partition size is determined in the same manner as RCB and METIS.

Overhead of Optimizations Figure 4 displays the costs of data reordering techniques measured relative to the time per iteration of ORIG. The overhead includes the cost to update edge structures and transform other related data structures to avoid the extra indirect accesses caused by the data reordering. The overhead also includes the cost of computation reordering.

The least expensive data layout optimization is CPACK. RCM has almost same overhead as CPACK. In comparison, partitioning algorithms have rather high overhead. RCB and METIS are quite expensive when used for cache optimizations,

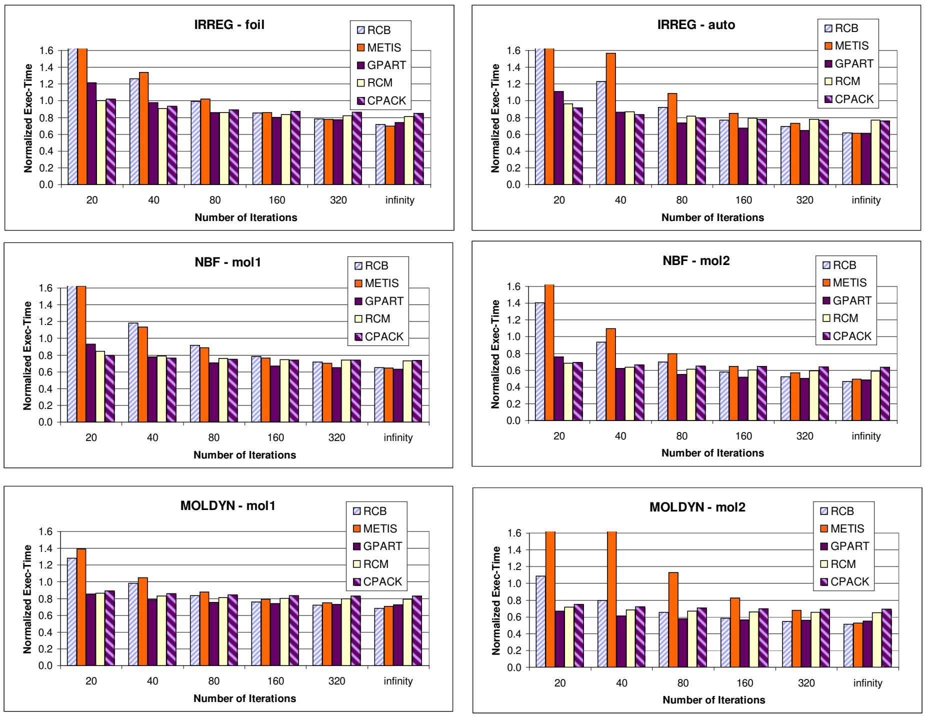

Fig. 6. Normalized Execution Times on DEC Alpha 21064 (including overhead)

---

on the order of 5–45 times higher than CPACK. The overhead of GPART is much less than METIS and RCB, but 3–5 times higher than CPACK.

Impact on Miss Rates We also simulated cache miss rates, using the cache configuration of the DEC Alpha 21064: 16K direct-mapped L1 cache with 32 byte cache lines and 4M direct-mapped L2 cache with 64 byte cache lines. Figure 5 shows L1 and L2 cache miss rates side by side for the same input. L1 cache miss rates of original, unoptimized programs are about 15%–28%. All partitioning techniques, RCB, METIS, and GPART reduce miss rates to 6%–8%, outperforming RCM and CPACK by 1%–6%. RCB and METIS achieve the best over all performance. but GPART obtains nearly the same performance. L2 cache miss rates are calculated as the percentages of all memory references, not the percentages of L2 cache references. L2 miss rates are higher for larger data sets, around 7%–8%. They are reduced by optimizations to around 2%–3%. L2 miss rates are reduced by applying partitioning techniques, outperforming RCM and CPACK. RCB and METIS show the best performance, but GPART also achieves nearly the same performance.

Impact on Performance Next we look at sequential execution times on the DEC Alpha and MIPS R10000. Figures 6 and 7 present normalized execution times

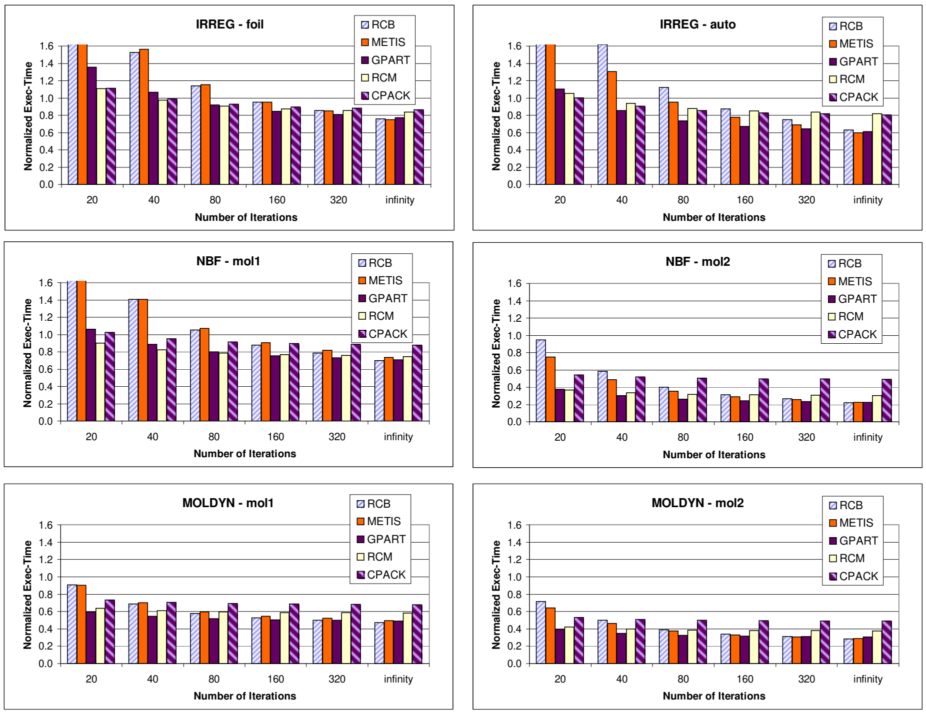

Fig. 7. Normalized Execution Times on MIPS R10000 (including overhead)

---

calculated relative to the execution time of ORIG. Results are presented for programs executing different numbers of iterations. Since results include the overhead of locality optimizations performed at the beginning of each program, performance improves as the number of iterations increases.

The set of results when the number of iterations is *infinity* represent the quality of each optimization when overhead is ignored. The results show partitioning algorithms perform better than RCM and CPACK in quality of data reordering. RCB, METIS, and GPART achieve 30%–50% improvements on the DEC Alpha and 20%–80% on the MIPS R10000. In comparison, RCM improves less than the partitioning algorithms, but a bit better than CPACK. CPACK shows the least improvement among all algorithms. We also measured parallel performance, and found locality optimizations improve performance even more than for sequential codes. On a on a 4-processor DEC Alpha multiprocessor, partitioning algorithms achieve 110%–130% improvements in speedups, while CPACK achieves only 60% improvements.

Optimization Technique Selection When applying locality optimizations in practice, we must consider overheads for each optimization. Even though RCB and METIS generate higher quality orderings, they require hundreds of iterations to be competitive with RCM or CPACK. Meanwhile, GPART begins to outperform

	Compared to ORIG				Compared to GPART			
	SGI Origin2000		DEC Alpha 21064		SGI Origin2000		DEC Alpha 21064	
	Avg	Range	Avg	Range	Avg	Range	Avg	Range
RCB	14%	-161% ~ 78%	10%	-84% ~ 54%	-35%	-152% ~ 7%	-22%	-86% ~ 8%
METIS	18%	-137% ~ 78%	-7%	-194% ~ 51%	-28%	-99% ~ 6%	-46%	-338% ~ 6%
GPART	39%	-36% ~ 77%	28%	-21% ~ 52%	-	-	-	-
RCM	35%	-11% ~ 70%	24%	-0% ~ 41%	-11%	-34% ~ 18%	-8%	-25% ~ 18%
CPACK	26%	-11% ~ 51%	22%	-2% ~ 36%	-34%	-117% ~ 18%	-11%	-32% ~ 11%

Table 2. Average improvement over ORIG and GPART

RCM and CPACK around 40 iterations due to its low overhead. RCB and METIS will be most beneficial for long running scientific applications where their high overheads can be can be amortized over many loop iterations. GPART, RCM, and CPACK will be more effective for applications with small numbers of iterations due to their low overhead.

Table 2 summarizes performance for each optimization relative to both the original performance and GPART. Results are computed as the average over the six versions of each program (running 20, 40, 80, 160, 320, and an infinite number of iterations). The range of improvements are also presented, ranging from negative values (short-running programs where optimization overhead cause degradations) to positive values (long-running programs where overheads are amortized). We see that GPART has the best overall results over a range of program executions, because it provides good performance for both short and long-running programs.

GPART has good performance compared to other optimizations, on average 11%–35% better. Under ideal conditions, GPART can improve performance from 25%–338%. Even in the worst case, the performance of GPART is close to other algorithms. For very long-running programs RCB and METIS do 6%–8% better, while for very short-running programs CPACK and RCM do 11%–18% better.

On average GPART's performance is 11% better than RCM, the next best optimization. For long-running applications GPART is up to 25% faster than RCM. Based on these results, we feel GPART is the preferred optimization in the most cases, particularly when the number of iterations executed is not known.

5 Related Work

Irregular scientific codes were first encountered in the context of message-passing systems. It was discovered these codes typically performed reduction operations which can be automatically parallelized and reordered in a compiler [25]. Researchers have investigated compiler analyses [22, 23] and ways to provide efficient run-time and compiler support [4, 29]. Libraries which can efficiently communicate data between processors using an inspector/executor approach were key to achieving good performance [7, 16, 21]. Compiler techniques for automatically generating and inserting these inspectors were developed [16, 13, 14].

Most research on improving data locality has focused on dense arrays with regular access patterns, applying either loop [26, 33] or data layout [18, 30] transformations. Locality optimizations have also been developed n the context of sparse linear algebra. The Reverse Cuthill-McGee (RCM) method [5] improves locality in sparse matrices by reordering columns using reverse breadth-first search (BFS). More sophisticated versions of the algorithm have since been developed [5, 24, 17].

A few researchers have investigated irregular scientific application. Das *et al.* investigated data/computation reordering for unstructured euler solvers [6]. They combined data reordering using RCM and lexicographical sort for computation reordering with communication optimizations to improve performance of parallel codes on an Intel iPSC/860. Al-Furaih and Ranka studied partitioning data using METIS and BFS to reorder data in irregular codes [1]. They conclude METIS yields better locality, but did not include computation reordering or account for processing overhead.

Ding and Kennedy explored applying dynamic copying (packing) of data elements based on loop traversal order, and show major improvements in performance [8]. They were able to automate most of their transformations in a compiler, using user provided information. In comparison, we show partitioning algorithms can yield better locality, albeit with higher processing overhead. Ding and Kennedy also developed algorithms for reorganizing single arrays into multi-dimensional arrays depending on their access patterns, improving the performance of many irregular applications [9]. Their technique might be useful for Fortran codes where data are often single dimension arrays, not structures as in C codes.

Mellor-Crummey *et al.* used a geometric partitioning algorithm based on space-filling curves to map multidimensional data to memory [27]. They also blocked computation using methods similar to tiling. In comparison, our graph-based partitioning techniques are more suited for compilers, since geometric coordinate information and block parameters for space-filling curves do not need to provided manually. When coordinate information is available, using RCB is better because space-filling curves cannot guarantee evenly balanced partition when data is unevenly distributed, which may cause significant performance degradation in parallel execution.

Mitchell *et al.* improved locality using bucket sorting to reorder loop iterations in irregular computations [28]. They improved the performance of two NAS applications (CG, and IS) and a medical heart simulation. Bucket sorting works only for computations containing a single irregular access per loop iteration. In comparison, we investigate more complex cases where two or more irregular access patterns exist. For simple codes lexicographic sort yields improvements similar to bucket sorting.

6 Conclusion

Our research demonstrates the benefits of improving data locality in irregular scientific applications by applying compiler and run-time transformations to partition data and computation. We develop GPART,a new partitioning algorithm based on hierarchical graph partitioning, which produces partitions which almost match the performance of the best partitioning strategies, but with much lower overhead. On experiments with a number of applications and input data sets, GPART improved performance by an average of 28% and 39% on the DEC Alpha 21064 and MIPS R10000, respectively.

Experiments also find partitioning algorithms (RCB, METIS, GPART) generate better quality ordering for cache locality than simple traversing algorithms (RCM, CPACK). Partitioning algorithms improve performance by an average of 38% on DEC Alpha 21064 and 49% on MIPS R10000, while traversing algorithms improve only 26% and 35%, respectively. These results show that as processors continue to increase in speed relative to memory, using graph partitioning to improve data layout will increase in importance since processing costs go down while benefits increase. For very large graphs, we should also obtain benefits by reducing TLB misses and paging in the virtual memory system. Locality optimizations for irregular computations can be effectively for parallel computations [10, 34]. Low-overhead transformations such as GPART will be especially useful for *adaptive* irregular applications, where optimizations may need to be reapplied after access patterns changes [12].

References

1. I. Al-Furaih and S. Ranka. Memory hierarchy management for iterative graph structures. In *Proceedings of the 12th International Parallel Processing Symposium*, Orlando, FL, April 1998.
2. M. Berger and S. Bokhari. A partitioning strategy for pdes across multiprocessors. In *Proceedings of the 1985 International Conference on Parallel Processing*, August 1985.
3. M. Berger and S. Bokhari. A partitioning strategy for non-uniform problems on multiprocessors. *IEEE Transactions on Computers*, 37(12):570–580, 1987.
4. S. Chandra and J.R. Larus. Optimizing communication in HPF programs for fine-grain distributed shared memory. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Las Vegas, NV, June 1997.
5. E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 24th National Conference of the ACM, ACM Publication P-69*, Association for Computing Machinery, NY, 1969.
6. R. Das, D. Mavriplis, J. Saltz, S. Gupta, and R. Ponnusamy. The design and implementation of a parallel unstructured euler solver using software primitives. In *Proceedings of the 30th Aerospace Sciences Meeting & Exhibit*, Reno, NV, January 1992.

7. R. Das, M. Uysal, J. Saltz, and Y.-S. Hwang. Communication optimizations for irregular scientific computations on distributed memory architectures. *Journal of Parallel and Distributed Computing*, 22(3):462–479, September 1994.

8. C. Ding and K. Kennedy. Improving cache performance of dynamic applications with computation and data layout transformations. In *Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation*, Atlanta, GA, May 1999.

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

10. H. Han and C.-W. Tseng. Improving compiler and run-time support for adaptive irregular codes. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, Paris, France, October 1998.

11. H. Han and C.-W. Tseng. Improving locality for adaptive irregular scientific codes. Technical Report CS-TR-4039, Dept. of Computer Science, University of Maryland at College Park, September 1999.

12. H. Han and C.-W. Tseng. Improving locality for adaptive irregular codes. In *Proceedings of the Thirteenth Workshop on Languages and Compilers for Parallel Computing*, White Plains, NY, August 2000.

13. R. v. Hanxleden. Handling irregular problems with Fortran D — A preliminary report. In *Proceedings of the Fourth Workshop on Compilers for Parallel Computers*, Delft, The Netherlands, December 1993.

14. R. v. Hanxleden and K. Kennedy. Give-N-Take — A balanced code placement framework. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, Orlando, FL, June 1994.

15. Y. Hu, S. L. Johnsson, and S.-H. Teng. High Performance Fortran for highly irregular problems. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Las Vegas, NV, June 1997.

16. Y.-S. Hwang, B. Moon, S. Sharma, R. Ponnusamy, R. Das, and J. Saltz. Runtime and language support for compiling adaptive irregular programs on distributed memory machines. *Software—Practice and Experience*, 25(6):597–621, June 1995.

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

18. M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. Improving locality using loop and data transformations in an integrated framework. In *Proceedings of the 31th IEEE/ACM International Symposium on Microarchitecture*, Dallas, TX, November 1998.

19. G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. In *Proceedings of the 24th International Conference on Parallel Processing*, Oconomowoc, WI, August 1995.

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

21. A. Lain and P. Banerjee. Exploiting spatial regularity in irregular iterative applications. In *Proceedings of the 9th International Parallel Processing Symposium*, Santa Barbara, CA, April 1995.
22. Y. Lin and D. Padua. On the automatic parallelization of sparse and irregular Fortran programs. In *Proceedings of the 4th Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, Pittsburgh, PA, May 1998.
23. Y. Lin and D. Padua. Compiler analysis of irregular memory accesses. In *Proceedings of the SIGPLAN '00 Conference on Programming Language Design and Implementation*, Vancouver, Canada, June 2000.
24. W. Liu and A. Sherman. Comparative analysis of the cuthill-mckee and the reverse cuthill-mckee ordering algorithms for sparse matrices. *SIAM Journal on Numerical Analysis*, 13(2):198–213, April 1976.
25. B. Lu and J. Mellor-Crummey. Compiler optimization of implicit reductions for distributed memory multiprocessors. In *Proceedings of the 12th International Parallel Processing Symposium*, Orlando, FL, April 1998.
26. K. S. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.
27. J. Mellor-Crummey, D. Whalley, and K. Kennedy. Improving memory hierarchy performance for irregular applications. In *Proceedings of the 1999 ACM International Conference on Supercomputing*, Rhodes, Greece, June 1999.
28. N. Mitchell, L. Carter, and J. Ferrante. Localizing non-affine array references. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, Newport Beach , LA, October 1999.
29. M. Rinard and P. Diniz. Commutativity analysis: A new analysis technique for parallelizing compilers. *ACM Transactions on Programming Languages and Systems*, 19(6):942–992, November 1997.
30. G. Rivera and C.-W. Tseng. Data transformations for eliminating conflict misses. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, Montreal, Canada, June 1998.
31. H. Simon. Partitioning of unstructured mesh problems for parallel processing. In *Proceedings of the Conference on Parallel Methods on Large Scale Structural Analysis and Physics Applications*. Permagon Press, 1991.
32. J. P. Singh, C. Holt, T. Totsuka, A. Gupta, and J. Hennessy. Load balancing and data locality in adaptive hierarchical n-body methods: Barnes-hut, fast multipole, and radiosity. *Journal of Parallel and Distributed Computing*, June 1995.
33. M. E. Wolf and M. Lam. A data locality optimizing algorithm. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, Canada, June 1991.
34. H. Yu and L. Rauchwerger. Adaptive reduction parallelization techniques. In *Proceedings of the 2000 ACM International Conference on Supercomputing*, Santa Fe, NM, May 2000.

This article was processed using the LaTeX macro package with LLNCS style