Rescheduling for Locality in Sparse Matrix Computations

Michelle Mills Strout, Larry Carter, and Jeanne Ferrante

University of California, San Diego

Abstract. In modern computer architecture the use of memory hierarchies causes a program's data locality to directly affect performance. Data locality occurs when a piece of data is still in a cache upon reuse. For dense matrix computations, loop transformations can be used to improve data locality. However, sparse matrix computations have non-affine loop bounds and indirect memory references which prohibit the use of compile time loop transformations. This paper describes an algorithm to tile at runtime called *serial sparse tiling*. We test a runtime tiled version of sparse Gauss-Seidel on 4 different architectures where it exhibits speedups of up to 2.7. The paper also gives a static model for determining tile size and outlines how overhead affects the overall speedup.

1 Introduction

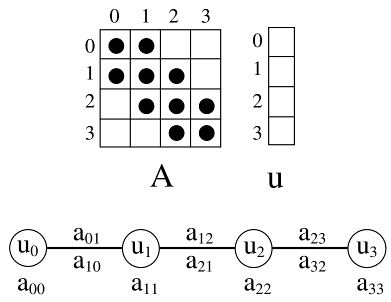
In modern computer architecture the use of memory hierarchies causes a program's data locality to directly affect performance. Data locality occurs when a piece of data is still in the cache upon reuse. This paper presents a technique for tiling sparse matrix computations in order to improve the data locality in scientific applications such as Finite Element Analysis.

The Finite Element Method (FEM) is a numerical technique used in scientific applications such as Stress Analysis, Heat Transfer, and Fluid Flow. In FEM the physical domain being modeled is discretized into an unstructured grid or mesh (see figure 3). FEM then generates simultaneous linear equations that describe the relationship between the unknowns at each node in the mesh. Typical unknowns include temperature, pressure, and xy-displacement. These equations are represented with a sparse matrix A and vectors $\boldsymbol{u}$ and $\boldsymbol{f}$ such that $A\boldsymbol{u} = \boldsymbol{f}$.

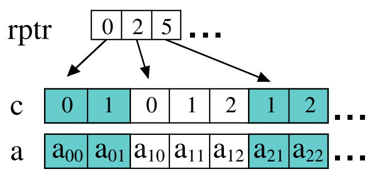
Conjugate Gradient, Gauss-Seidel and Jacobi are all iterative methods for solving simultaneous linear equations. They solve for $\boldsymbol{u}$ by iterating over the sparse matrix A a constant number of times, converging towards a solution. The iteratively calculated value of a mesh node unknown u_j depends on the values of other unknowns on the same node, the unknowns associated with adjacent nodes within the mesh, and the non-zeros/coefficients in the sparse matrix which relate those unknowns. Typically the sparse matrix is so large that none of the values used by one calculation of u_j remain in the cache for future iterations on u_j, thus the computation exhibits poor data locality.

For dense matrix computations, compile time loop transformations such as tiling or blocking [12] can be used to improve data locality. However, since sparse matrix computations operate on compressed forms of the matrix in order to avoid storing zeros, the loop bounds are not affine and the array references include indirect memory references such as $a[c[i]]$. Therefore, straightforward application of tiling is not possible. In this paper, we show how to extend tiling via runtime reorganization of data and rescheduling of computation to take advantage of the data locality in such sparse matrix computations.

Specifically, we reschedule the sparse Gauss-Seidel computation at runtime. First we tile the iteration space and then generate a new schedule and node numbering which allows each tile to be executed atomically. Typically the numbering of the nodes in the mesh is arbitrary, therefore, renumbering the nodes and maintaining the Gauss-Seidel partial order on the new numbering allows us to still use the convergence theorems for Gauss-Seidel. The goal is to select the tile size so that the tile only touches a data subset, which fits into cache.



(a) Logical data associations



(b) Actual CSR storage format

Fig. 1. Data associated with Mesh

To illustrate, we look at an example of how one would tile the Gauss-Seidel computation on a one-dimensional mesh. Figure 1(a) shows how we can visualize what data is associated with each node in the mesh. The unknown values being iteratively updated are associated with the nodes, [1] and the coefficients representing how the unknowns relate are associated with the edges and nodes. However, keep in mind that the matrix is stored in a compressed format like compressed sparse row (see figure 1(b)) to avoid storing the zeros.

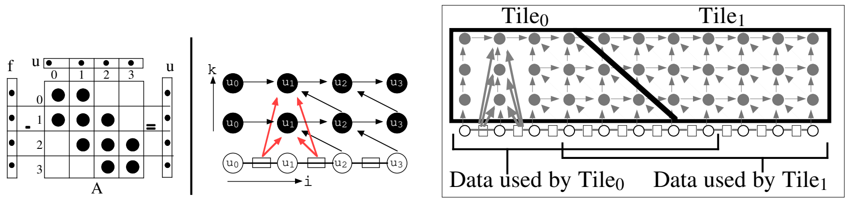
The pseudo-code for Gauss-Seidel is shown below. The outermost loop iterates over the entire sparse matrix generated by solving functions on the mesh. We refer to the i iterator as the *convergence iterator*. The j loop iterates over the rows in the sparse matrix. [2] The k loop which is implicit in the summations

[1] In this example there is only one unknown per mesh node.
[2] There is one row in the matrix for each unknown at each mesh node.

iterates over the unknowns which are related to u_j, with $a_{jk}u_k^{(i)}$ and $a_{jk}u_k^{(i-1)}$ only being computed when a_{jk} is a non-zero value.

for $i = 1, 2, ..., T$
  for $j = 1, 2, ..., R$
    $u_j^{(i)} = (1/a_{jj})(f_j - \sum_{k=1}^{j-1} a_{jk}u_k^{(i)} - \sum_{k=j+1}^{R} a_{jk}u_k^{(i-1)})$

The Gauss-Seidel computation can be visualized with the iteration space graph shown in figure 2. Each black iteration point [3], $< i, v >$, represents the computations for all $u_j^{(i)}$ where u_j is an unknown associated with mesh node v and i is the convergence iteration. The initial values associated with a 1D mesh are shown in white. The arrows represent data dependences [4] that specify when an initial value or a value generated by various iteration points is used by other iteration points. We refer to each set of computation for a particular value of i within the iteration space as a *layer*. Figure 2 contains three layers of computation over a mesh.



(a) Original Computation

(b) Divided into 2 Tiles

Fig. 2. Gauss-Seidel Iteration Space Graph

Notice that the sparse matrix values associated with the edges adjacent to a particular mesh node v are reused in each computation layer. However, the mesh is typically so large that upon reuse the matrix entries are no longer in the cache. To improve the computation's data locality, we reschedule it based on a tiling like the one shown in figure 2(b). The resulting schedule executes all of the iteration points in one tile before continuing on to the next tile; in other words, each tile is executed atomically. By choosing an appropriate tile size the data used by each tile will fit into cache for all instances of $< i, v >$ within the tile and therefore improve the data locality of the computation.

In Section 2 we present the algorithm which tiles and reschedules Gauss-Seidel at runtime. Then in section 3 we give experimental results which show

[3] We use the term iteration point for points in the iteration space graph and node for points in the mesh.

[4] Some dependences are omitted for clarity.

that improving the data locality does improve code performance. We also outline the affect of overhead and how to select tile sizes. Finally, we present some related work and conclusions.

2 Tiling Sparse Computations

In order to tile the iteration space induced by the convergence iteration over the mesh, we partition the mesh and then grow tiles backwards through the iteration space based on the seed partitions. Figure 3 shows the iteration space for a 2D mesh with each layer drawn separately. Edges show the connectivity of the underlying mesh. We use the resulting tiling to reschedule the computation and renumber the nodes in the mesh. Since tiles depend on results calculated by neighboring tiles, the tiles must be executed in a partial order which respects those dependences.

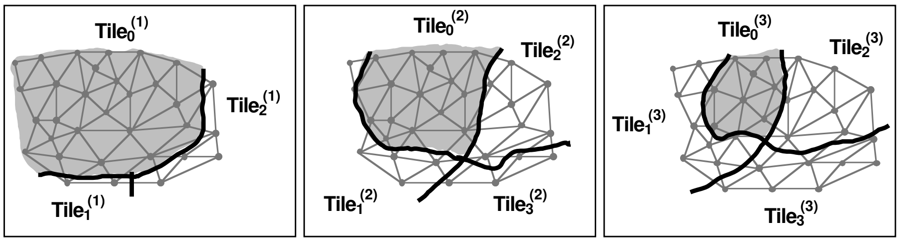


Fig. 3. Tile layers for $Tile_0$, $Tile_1$, $Tile_2$, and $Tile_3$. The tile layers for $Tile_0$ are shaded.

We refer to the runtime tiling of sparse matrix computations as *sparse tiling*. This paper describes and implements a *serial sparse tiling*, in that the resulting schedule is serial. Douglas et al. [3] describe a *parallel sparse tiling* for Gauss-Seidel. They partition the mesh and then grow tiles forward through the iteration space (in the direction of the convergence iterator) in such a way that the tiles do not depend on one another and therefore can be executed in parallel. After executing the tiles resulting from parallel sparse tiling, it is necessary to execute a fill-in stage which finishes all the iteration points not included in the tiles. Future work includes determining when to use a serial sparse tiling or a parallel sparse tiling based on the target architecture and problem size.

Both sparse tiling strategies follow the same overall process at runtime.

1. **Partition** the mesh
2. **Tile** the iteration space induced by the partitioned mesh
3. **Reschedule** the computation
4. **Execute** the new schedule

The next sub-sections describe each part of the process for the serial sparse tiling strategy which we have developed.

2.1 Partition

Although graph partitioning is an NP-Hard problem [4], there are many heuristics used to get reasonable graph partitions. We use the Metis [8] software package to do the partitioning at runtime on the mesh. The partitioning algorithm in Metis has a complexity of $O(|E|)$ where $|E|$ is the number of edges in the mesh [9].

2.2 Tiling

Recall the iteration space for sparse Gauss-Seidel shown in figure 2 where each iteration point represents values being generated for the unknowns on the associated mesh node v at convergence iteration i. A tile within this space is a set of layers, one per each instance of the convergence iterator i. Each tile layer computes the values for a subset of mesh nodes. The final layer of a tile (see the last layer in figure 3) corresponds to the nodes in one partition, p, of the mesh. The tile layers for earlier convergence iterations are formed by adding or deleting iteration points from the seed partition to allow atomic execution of the tile without violating any data dependences.

To describe the sparse serial tiling algorithm for sparse Gauss-Seidel we use the following terminology. The mesh can be represented by a graph $G(V, E)$ consisting of a set of nodes V and edges E. An **iteration point**, $< i, v >$, represents the computation necessary at convergence iteration i for the unknowns associated with node v. A **tile**, $Tile_p$, is a set of iteration points that can be executed atomically. Each tile is designated by an integer identifier p, which also represents the execution order of the tiles. A **tile layer**, $Tile_p^{(i)}$, includes all iteration points within tile p being executed at convergence iteration i.

The tiling algorithm generates a function θ that returns the identifier for the tile which is responsible for executing the given iteration point, $\theta(< i, v >)$: $\{1, .., T\}$ x $V \rightarrow \{0, ..., (m-1)\}$, where m is the number of tiles. $Tile_0$ will execute all iteration points with $\theta(< i, v >) = 0$, $Tile_1$ will execute all iteration points with $\theta(< i, v >) = 1$, etc. A **tile vector**, $\Theta(v) =< \theta(< 1, v >), ..., \theta(< T, v >) >$, stores tile identifiers for all the tiles which will be executing iteration points for a specific node in the mesh.

The algorithm shown below gives all nodes a legal tile vector. It takes as input the *part* function, $part(v) : V \rightarrow \{0, ..., (m-1)\}$, which is the result of the mesh partitioning. The *part* function specifies a partition identifier for each mesh node. Recall that we will be growing one tile for each seed partition. The first step in the algorithm is to initialize all tile vectors so that each iteration point is being executed by the tile being grown from the associated mesh node's partition in the mesh. $Worklist(T)$ is then initialized with all nodes. The loop then grows the tiles backward from $i = T$ by adding and removing iteration points as needed in order to maintain the data dependences. A detailed explanation of this loop is omitted due to space constraints.

Algorithm $\textsc{AssignTileVector}(part, T, V, E)$

$\forall v \in V, \Theta(v) = < part(v), part(v), ..., part(v) >$
$Worklist = \{<v,w> \mid \exists\ (v,w) \in E \text{ with } \theta(<T,v>) < \theta(<T,w>)\}$
for $i = T$ downto 2
    for $<v,w> \in Worklist$
        $\forall q$ st. $1 \leq q \leq (i-1)$,
            $\theta(<q,w>) \longleftarrow min(\theta(<i,v>), \theta(<i-1,w>))$
    for $<v,w> \in Worklist$ in order by $\theta(<i-1,w>)$
        $\forall q$ st. $1 \leq q \leq (i-1)$,
            $\theta(<q,v>) \longleftarrow min(\theta(<i-1,v>), \theta(<i-1,w>))$
    $Worklist = Worklist \bigcup$
        $\{<v,w> \mid \exists\ (v,w) \in E \text{ with } \theta(<i-1,v>) < \theta(<i-1,w>)\}$

An upper bound on the complexity of this algorithm is $O(T^2|E|)$ where $|E|$ is the number of edges in the mesh and T is the number of convergence iterations the Gauss-Seidel algorithm will perform.

2.3 Renumbering and Rescheduling

The mesh nodes are renumbered in lexicographical order of their corresponding tile vectors. The lexicographical order insures that the resulting schedule will satisfy the Gauss-Seidel partial order on the new numbering. We schedule all the computations in $Tile_p$ before any in $Tile_{p+1}$, and within a tile we schedule the computations by layer and within a layer.

2.4 Execute Transformed Computation

Finally, we rewrite the sparse Gauss-Seidel computation to execute the new schedule. The new schedule indicates which iteration points should be executed for each tile at each convergence iteration.

3 Experimental Results for Gauss-Seidel

To evaluate the possible benefits of our approach, we compare the performance of the Gauss-Seidel routine in the finite element package FEtk [6] with a runtime tiled and rescheduled version of the same algorithm. For input, we use the sparse matrices generated for a nonlinear elasticity problem on 2D and 3D bar meshes. We generate different problem sizes by using FEtk's adaptive refinement. The rescheduled code runs on an Intel Pentium III, an IBM Power3 node on the Blue Horizon at the San Diego Supercomputer Center, a Sun UltraSparc-IIi, and a DEC Alpha 21164.

When not taking overhead into account the new schedule exhibits speedups between 0.76 (a slowdown) and 2.7 on the four machines, see figure 4. Next we describe the simple static model used for selecting the partition size - the main tuning parameter for the new schedule. Finally we outline the effect overhead will have on the overall speedup.

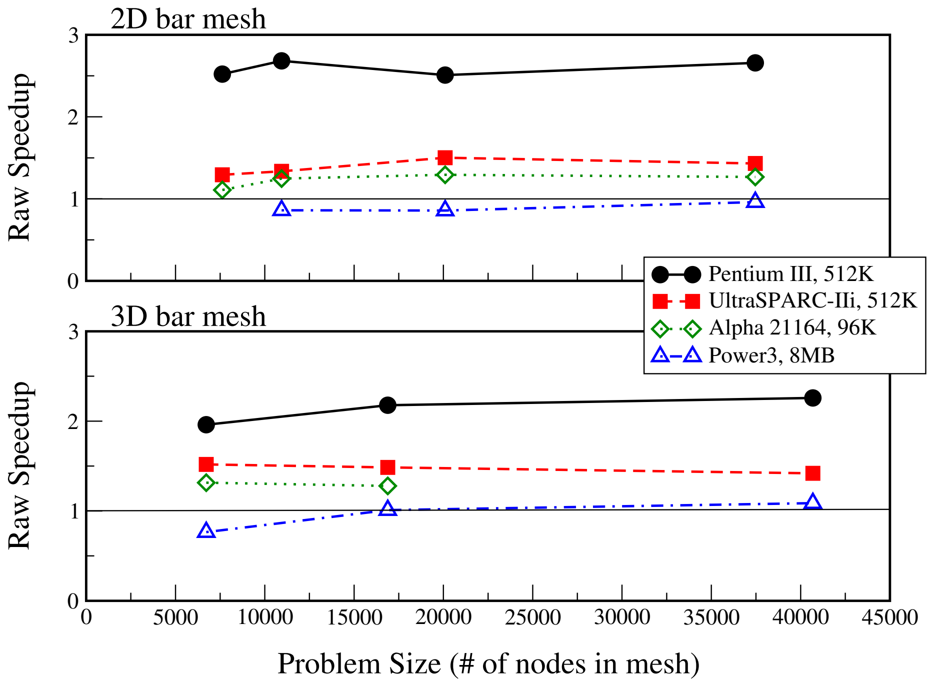


Fig. 4. Speedups over FEtk's Gauss-Seidel for 2D and 3D bar mesh without adding overhead. The partition size was selected to fit into the L2 cache on each machine whose sizes are shown in the legend.

3.1 Partition Size Selection

Before tiling and rescheduling at runtime the available parameters are the number of nodes in the mesh, the number of unknowns per vertex, the number of convergence iterations, and the cache size of the target architecture. Using this information we want to determine which partition size will generate tiles which fit into a level of cache and therefore improve performance.

In Gauss-Seidel for each unknown at each mesh node we iteratively compute $w_j = f_j - \sum_{k>i} a_{jk} * u_j$ and $u_j = (w_j - \sum_{k<j} a_{jk} * u_j)/a_{jj}$. Using K as the average number of neighbors each node has in the mesh and d as the number of

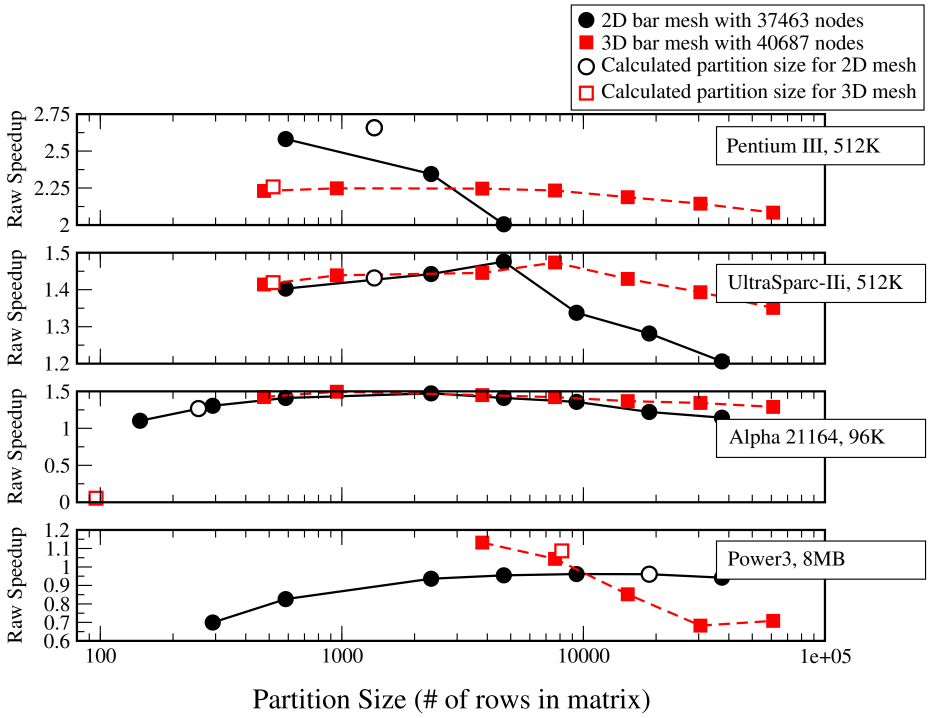


Fig. 5. How the partitions size affects speedup for 5 convergence iterations on 4 different machines. The outlined symbols represent partitions sizes calculated by the model.

unknowns per mesh node, the computation will use $3 * d$ scalars for the vectors u, w, and f and $K * d^2$ associated non-zeros from the sparse matrix A while updating the unknowns for each mesh node. If we assume compressed sparse row (CSR) storage format then the amount of memory needed by N mesh nodes is $Mem(N) = N * K * d^2 * sizeof(double) + N * K * d^2 * sizeof(int) + 3 * N * d * sizeof(double)$. In all of the experiments we solve for N such that only half the memory in the L2 cache of the machine is utilized.

Figure 4 shows the speedups on different mesh sizes when the partition sizes are selected in this manner. When compared to the speedups over a variety of partitions sizes the calculated partition sizes do reasonably well, see figure 5, except on the Alpha. This is probably due to the small L2 size on the Alpha and the existence of a 2MB L3 cache.

3.2 Overhead

Figures 4 and 5 show the speedups obtained by the tiled and rescheduled code over FEtk's implementation of Gauss-Seidel without taking overhead into account. It is important to look at the speedups without overhead because Gauss-

Seidel can be called multiple times on the same mesh within an algorithm like Multigrid. Therefore, even though overhead might make rescheduling not beneficial for one execution of the Gauss-Seidel computation, when amortized over multiple calls to the computation we get an overall speedup.

By looking at the calculated partition sizes resulting in the highest (2.68) and lowest [5] (1.11) speedups we see that the tiled and rescheduled version of Gauss-Seidel would need to be called between 5 to 27 times in order to observe an overall speedup. However, there were partition sizes not calculated by our simple static model which resulted in an overall speedup with only 1 call to the tiled and rescheduled Gauss-Seidel. For example, on a 3D mesh with $N = 40,687$ an overall speedup of 1.17 is observed even when the overhead cost is included. This indicates that the tradeoff between raw speedup and overhead must be considered when calculating partition sizes.

4 Related Work

Douglas et al. [3] does tiling on the iteration space graph resulting from unstructured grids in the context of the Multigrid algorithm using Gauss-Seidel as a smoother. They achieve overall speedups up to 2 with 2D meshes containing 3983, 15679, and 62207 nodes on an SGI O2. They are able to reschedule their tiles in parallel and then finish the remaining computation with a serial backtracking step. Our technique also tiles the Gauss-Seidel iteration space, but we execute our tiles serially in order to satisfy dependences between tiles. Also, we do not require a backtracking step which exhibits poor data locality. These two tiling algorithms are instances of a general class of temporal locality transformations which we will refer to as sparse tiling.

Mitchell et al [11] describe a compiler optimization which operates on non-affine array references in code. The use of sparse data structures causes indirect array references which are a type of non-affine array reference. Also, Eun-Jin Im [7] describes a code generator called SPARSITY which generates cache-blocked sparse matrix-vector multiply. Both of these techniques improve spatial and temporal locality on the vectors $\boldsymbol{u}$ and $\boldsymbol{f}$ when dealing with the system $A\boldsymbol{u} = \boldsymbol{f}$. However, they do not improve the temporal locality on the sparse matrix, because in their rescheduled code the entire sparse matrix is traversed each convergence iteration. Other work which looks at runtime data reorganization and rescheduling includes Demmel et al. [1], Han and Tseng[5], Ding and Kennedy [2], and Mellor-Crummey et al.[10].

5 Conclusion

Runtime tiling is possible with unstructured iteration spaces, and we show it can improve the data locality and therefore the performance of Gauss-Seidel. Specifically we present an algorithm for generating a serial sparse tiling for Gauss-Seidel. We also describe a simple static model for selecting partition sizes from

[5] Ignoring the Power3 results because speedup was less than 1.

which the tiles are grown. Future work includes improving the model used to calculate partition sizes so that the tradeoff between overhead and raw speedup is taken into account. Also, the performance model needs to determine when to use a serial sparse tiling or a parallel sparse tiling based on the target architecture and problem size.

References

1. James W. Demmel, Stanley C. Eisenstat, John R. Gilbert, Xiaoye S. Li, and Joseph W. H. Liu. A supernodal approach to sparse partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 20(3):720–755, July 1999.
2. Chen Ding and Ken Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 229–241, Atlanta, Georgia, May 1–4, 1999.
3. Craig C. Douglas, Jonathan Hu, Markus Kowarschik, Ulrich Rüde, and Christian Weiss. Cache Optimization for Structured and Unstructured Grid Multigrid. *Electronic Transaction on Numerical Analysis*, pages 21–40, February 2000.
4. Michael R. Garey, David S. Johnson, and L. Stockmeyer. Some simplified *NP*-complete graph problems. *Theoretical Computer Science*, 1:237–267, 1976.
5. Hwansoo Han and Chau-Wen Tseng. Efficient compiler and run-time support for parallel irregular reductions. *Parallel Computing*, 26(13–14):1861–1887, December 2000.
6. Michael Holst. Fetk = the finite element toolkit. http://www.fetk.org.
7. Eun-Jin Im. *Optimizing the Performance of Sparse Matrix-Vector Multiply*. Ph.d. thesis, University of California, Berkeley, May 2000.
8. George Karypis and Vipin Kumar. *Metis: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and computing Fill-Reducing Orderings of Sparse Matrices Version 4.0*, 1998.
9. George Karypis and Vipin Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48(1):96–129, 10 January 1998.
10. John Mellor-Crummey, David Whalley, and Ken Kennedy. Improving memory hierarchy performance for irregular applications. In *Proceedings of the 1999 Conference on Supercomputing*, ACM SIGARCH, pages 425–433, N.Y., June 20–25 1999. ACM Press.
11. Nicholas Mitchell, Larry Carter, and Jeanne Ferrante. Localizing non-affine array references. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT '99)*, pages 192–202, Newport Beach, California, October 12–16, 1999. IEEE Computer Society Press.
12. Michael J. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.