Memory Hierarchy Management for Iterative Graph Structures [*]

Ibraheem Al-Furaih [†]
Syracuse University

Sanjay Ranka
University of Florida

Abstract

The increasing gap in processor and memory speeds has forced microprocessors to rely on deep cache hierarchies to keep the processors from starving for data. For many applications, this results in a wide disparity between sustained and peak achievable speed. Applications need to be tuned to processor and memory system architectures for cache locality, memory layout and data prefetch and reuse.

In this paper we investigate optimizations for unstructured iterative applications in which the computational structure remains static or changes only slightly through iterations. Our methods reorganize the data elements to obtain better memory system performance without modifying code fragments.

Our experimental results show that the overall time can be reduced significantly using our optimizations. Further, the overhead of our methods is small enough that they are applicable even if the computational structure does not substantially change for tens of iterations.

1. Introduction

The increasing gap between processor speed and memory speed has forced microprocessors to rely on deep cache hierarchies. Locality of data reference is necessary to achieve good performance.

Previous work on exploiting the above memory hierarchy has focused on regular structures and scalars. Carr [2] discussed applying transformations such as scalar replacement, unroll and jam, loop interchange and blocking to optimize the memory performance. Bacon et al. [1] reviewed some of the important techniques that can be used by optimizing compilers for both sequential and parallel machines.

For many irregular problems the computational structure can be effectively represented by a graph in which nodes represent data elements and the edges represent interaction between the data elements. In this paper we concentrate on the problem of how to reorganize the data to enable good performance in the presence of memory hierarchy for iterative applications in which the computational structure (represented by an interaction graph) remains static or changes only slightly through iterations. In such applications the data locality does not change much through iterations which enables us to reorganize the data only once or after tens of iterations.

We present several methods for data reorganization and present performance results. Our results are promising and show that performance can be improved by up to a factor of two in many cases. Further, many of these methods are very general and can be potentially developed as a runtime library which can be used by compilers.

The rest of the paper is organized as follows. In section 2 we briefly describe the different types of target interaction graphs. In section 3 and 4 we provide optimization methods for single and coupled interaction graphs respectively. Experimental results are presented in Section 5. We conclude in Section 6.

2. Interaction Graphs

A graph in which nodes represent data elements and edges represent the interaction between them is called an *Interaction graph*. An interaction graph G is defined as the pair $G = (V, E)$, where V is a collection of nodes and E is a list of edges between these nodes. $|V|$ refers to the number of nodes in the graph, similarly $|E|$ is the number of edges. A graph G is sparse if $|E|$ is much less than $|V|^2$.

Nodes are identified by their numbers, each node has a number between 1 and $|V|$ and node i is stored next to $i+1$ in memory. We will use $Adj[u]$ to refer the set of all nodes adjacent to u. In this paper we will look at two type of Interaction graphs, single graphs and coupled graphs. The computational structure of applications such as iterative unstructured grid solvers can be represented using a single *Interaction graph*. Other applications, such as Particle-in-cell simulation can be represented as coupled graphs. It requires two data structures corresponding to the particles and cells

respectively. The interaction of the elements of two data structures can be divided into interaction within the elements of each structure (two graph that represents the interaction between the two structures is called a *Coupled graph*.

Our goal was to derive methods which will reorganize the data elements of a graph to improve temporal and spatial locality. This allows to obtain good performance in the presence of memory hierarchies. Further, our goal was to achieve this without modifying the code fragments. We were interested in investigating methods which map the data elements of a graph at hand to another isomorphic graph in which neighboring nodes will be stored adjacently in memory and will be accessed for computation in this order.

3. Single Graphs

The computational structure of several applications can be represented using a single interaction graph. To achieve this we first create a *Mapping Table (MT)* of size $|v|$. $MT[i]$ provides the new location for node i. For simplicity we will assume two levels of memory hierarchy (main memory and cache). Our methods can be generalized to larger number of levels in the memory hierarchy.

By ensuring that neighboring nodes are stored adjacently and accessing them in this order we can achieve good performance even in the presence of memory hierarchy.

For most practical applications the interaction graph is sparse. Several representations can be used to represent this sparse interaction graph. They require $O(|E| + |V|)$ storage. In the adjacency list representation every edge is stored twice one in each of its two end points adjacency list. This duplication can be reduced by using a *compact adjacency list* representation were we impose an index order on the nodes and list the edge only once with the node with lower index.

We briefly describe four different algorithms for obtaining the mapping table, and compare the speedups achieved using these mappings and the time required for data reorganization using each method.

1. **Graph partitioning:** One way of achieving the temporal locality is by partitioning the interaction graph into several partitions such that the size of each partition is less than the size of the cache. Elements within a partition will be mapped to proximate indices and accesses will result in a low probability of cache misses. We used METIS [5] to partition the graph. The number of partitions P was chosen such that $\frac{GraphSize}{P} < CS * \alpha$, where CS is the cache size. α is typically less than 1 and can be adjusted to observe performance for different size partitions which can fit the cache. The nodes within a partition were assigned indices within a consecutive interval (of size equal to the number of partitions). This can be shown to require $O(|V| \log |V|)$ time without the cost of the graph partitioning algorithm.

2. **Breadth first search:** BFS algorithm starts from a root node and then visit all its neighboring nodes. This process is repeated recursively until all nodes have been visited. The time requirements for BFS tree is proportional to $O(|E| + |V|)$. The nodes are indexed using the breadth first order. This layers the interaction graph into several layers. The intent is that if nodes of three consecutive layers can be stored in cache then this will result in low cache misses.

3. **Hybrid approach:** In the hybrid approach we combine the benefits of both the Graph Partitioning as well as the Breadth First Search approaches. We first use graph partitioning to partition the nodes into a small number of partitions followed by layering the nodes within a partition by using a Breadth First search. All the nodes within a partition are assigned consecutive indices within a interval of size proportional to the partition size. Further, within a partition the indices are assigned based on the layering produced by Breadth First Search. This algorithm takes $O(|E| + |V|)$ without the cost of $METIS$.

4. **Connected components:** For large graphs, application of the BFS algorithm may result in large number of nodes to be assigned to the same layer. If the size of the cache is smaller than the size of nodes in consecutive layers, it will result in a large number of cache misses. This problem can be rectified by using a single tree bisection algorithm for deriving connected components as proposed by Dagum [3]. Using this algorithm, we first construct a BFS based spanning tree. We then compute a weight function for every node. For node v this is the size of the subtree rooted at v. We proceed by picking a node which has a weight that is just smaller than the size of the cache. We decompose the tree using these nodes and assign nodes within a subtree to consecutive intervals of indices.

For many applications, physical coordinate information about the nodes are available. These can be used for reorganization of the data elements. Several such methods based on Hilbert curves and Z-curves have been used in the literature [7].

4. Coupled Graphs

The computational structure of several applications such as Particle-in-cell simulation can be represented as coupled

graphs. This application requires two data structures corresponding to the particles and cells respectively. The interaction of the elements of two data structures can be divided into interaction within the elements of each data structure and the interaction between them (coupling).

In the following we will present two general methods for reordering the two graphs. We will discuss how to apply these techniques to the Particles-In-Cell problem in a later section.

1. **Independent reordering:** In this method, we reorder each of the two graphs independently of the other. Any of the algorithms we presented earlier for single graphs can be used. Each graph will be reordered according to the interaction between its nodes only.

2. **Coupled reordering:** We reorder each of the graphs using its own interactions as well as interactions with other subgraphs. There are several ways of constructing this graph. Additional nodes may be introduced to incorporate nodes of the other graphs. Figure 1 gives an example of a coupled graph for reordering the particles for the PIC problem.

The nodes of the coupled graph are the union of the particles and the grid points. The edges are obtained by adding four edges between every particle and the four grids points that represent the corners of the cell containing the particle.
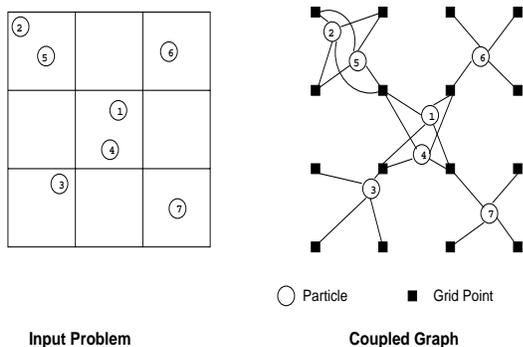

○ Particle   ■ Grid Point

Input Problem   Coupled Graph

Figure 1. An Example of a Coupled Graph

Any of the methods described in the previous section can be used to perform the reordering of this coupled graph.

5. Experimental Results

We have chosen two representative applications for demonstrating the performance of our methods. Static unstructured grid for single graphs and the 3-D Particles in cell simulation (PIC) algorithm for coupled graphs.

In this section, we present our experimental results on a sun UltraSPARC-I model 170 with $128MBytes$ main memory, $16KBytes$ data cache, $16KBytes$ instruction cache, $512KBytes$ external cache, and a cache line size of $64Bytes$.

5.1. Static unstructured grid

We executed a Laplace solver code for a large number of computational grids. These grids were obtained from Finite Element Group at the AHPCRC in the University of Minnesota. Representative results on a few grids are presented in this section due to space limitations.

The program execution can be divide into four phases: reading the input file (input time), generating the mapping table using one of the several algorithms (preprocessing time), reordering the data using the generated mapping table (reordering time), executing the code fragment for one iteration (execution time).

In the following we present experimental results for speedups obtained for these graphs in their original orderings. Figure 2 presents the speedups obtained using different reordering algorithms ignoring preprocessing and reordering times on the original graphs, in this graph $GP(X)$ is for graph partitioning (X is the number of partitions); BFS is for Breadth First Search; $BFS(X)$ is for graph partitioning and then running BFS in every partition (X is the number of partitions) and finally $CC(X)$ is for the connected components algorithm (X is the size of subtrees). For large graphs the speedups obtained are significant. Speedups of up to 1.75 are achieved for some graphs. However, the performance improvements for most of the reordering algorithms are comparable. The best results were obtained by using graph partitioning followed by a breadth first search within each of the partitions.

To remove any inherent locality present in the input graphs, we also evaluated the performance deterioration for these graphs after randomization of the initial node ordering, we found out that performance deteriorates significantly due to this randomization. This deterioration can be as large as 50% of the overall time. Thus, our methods can provide speedups of between two to three over randomized orderings.

Figure 3 shows the time taken for preprocessing by different algorithms for 144.graph which has 144649 nodes and 1074393 edges. The BFS algorithm has substantially lower costs and the speedups achieved are comparable to other techniques. This makes it a useful practical algorithm even in cases when the computational structure does not change substantially for as few as ten iterations. Including all preprocessing costs, the BFS algorithm only needs 6 iterations to achieve better overall time than a non-optimized algorithm.
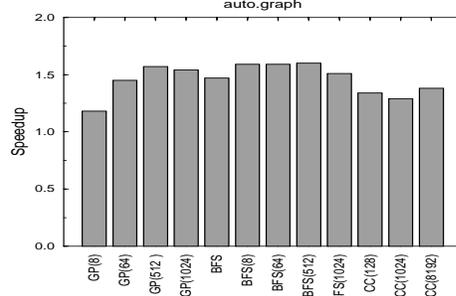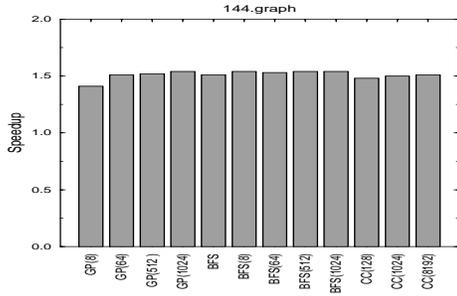
Figure 2. Speedups obtained for different data reordering methods
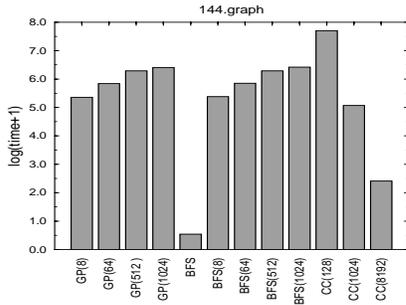

Figure 3. Preprocessing costs for 144.graph

5.2. Particle-in-cell Problem

The PIC problem has two data structures, a particles array and a $3 - D$ mesh grid array. The PIC algorithm simulates the movement of the particles for a sequence of time steps. Each time step consists of four phases: scatter, field solve, gather and push.

The scatter and the gather phases are the two that require interaction between the two data structures. Two particles in the same cell need to contribute to the same grid point. We can exploit this to reorder the particle array to improve locality of access in the scatter and gather phases.

Decyk and de Boer [4] have discussed data reorganization techniques for optimizing the PIC code. These are based on sorting the particles along one of the coordinate axes. Other optimizations for improving locality of access have also been described. They demonstrated that by sorting particles along one of the axes, one can obtain speedups of 12% to 32% depending on the machine used.

Since particles move during the simulation, it is expensive to reorder the particles at every time step. We assume that this reorganization is done every k number of iterations. The optimal choice of k depends on the distribution of particles. Several strategies for deciding when to reorder have been discussed in the literature [6].

We have experimented with the following data reorderings

1. **Independent reordering** This method reorders the two structures independently of the other. For the $3-D$ grid array, since it is a regular structure and does not change through iterations, we store it normally using either column major or row major ordering [1]. To reorder the particles, we can sort them along one of the coordinate axes as was suggested by Decyk and de Boer [4].

   We can also sort using a Hilbert space filling curve transformation [7] which requires calculation of the Hilbert index for every particle. This index is then used to sort the particles. These mapping have been shown to achieve good locality of access. Instead of executing the Hilbert algorithm on the particles, we can execute it only once on the grid at time of initialization and then assign an index to every cell. This can be used to assign an index to every particle.

2. **Coupled reordering** We first create a coupled graph in which the nodes are the particles and the mesh grid points, and we have an edge between every particle and the eight corner grid points corresponding to the cell in which the particle currently resides (Figure 1). Using a BFS algorithm on this graph, we obtain the index order of all the particles. This algorithm is referred to as BFS3 for the presentation of the execution times.

   Creating the coupled graph and applying BFS is expensive. A possible optimization is to run this algorithm for the underlying mesh and then use it to determine the indices of the particles. This requires creating the coupled graph as well as execution of the BFS algorithm only once. This index can then be used by particles during the simulation for reordering. This algorithm is referred to as BFS2 for the presentation of the execution times.

---

[1] Better performance may be achieved by blocking. However, we have not explored this. This scheme may have additional overhead associated with address translation of the mesh array for the field solve, gather and scatter phases

Similar results can also be obtained by using a couple graph which consists of the mesh plus the diagonal edges connecting pairs of diagonally opposite vertices of a cell. This algorithm is referred to as BFS1 for the presentation of the execution times.

Only the scatter and the gather phases will have performance improvements due to optimizations discussed above as these are the only phases that involve interaction between the particles array and the mesh array. Figure 4 shows the time per iteration for running the different algorithms on for an 8k mesh for different number of particles (The field solve phase takes only a very small fraction of the time, and can't be seen on the graph). All the different variant of the BFS and the Hilbert based algorithms achieved similar results.
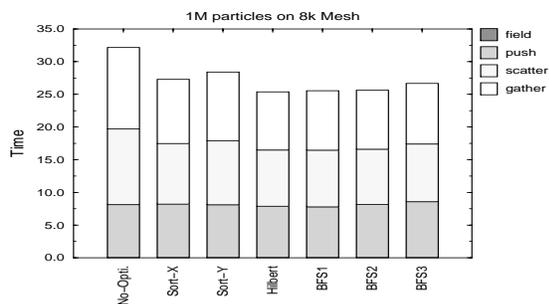

Figure 4. Execution times for PIC application

The time required by scatter and gather phases using BFS and Hilbert was about $25 - 30\%$ less than the corresponding time without any reordering. Use of locality along multiple dimensions (as achieved by Hilbert or BFS) results in approximate 10% more reduction of time as compared to sorting along only one dimension. The interesting feature of BFS algorithms is that they can be potentially incorporated in a compiler by using a runtime library to perform data reorganization without having explicit knowledge of the underlying particle geometry information.

The cost of all the algorithms are comparable except BFS3 which requires a factor of three larger amount of time,table 1. Since these data reorganization algorithms will only be used every few hundred iterations, small differences in the cost of BFS and Hilbert over sorting based will be more than offsetted by their improved performance.

Method	Iterations	Method	Iterations
Sort on X	3.34	Sort on Y	4.54
Hilbert curve	3.14	BFS1	3.14
BFS2	3.52	BFS3	10.03

Table 1. Iterations required to achieve better performance by data reordering.

6. Conclusions

In this paper, we presented methods for reorganize the data elements to obtain better memory system performance without modifying code fragments for a large class of unstructured applications. Our experimental results show that the overall time can be reduced significantly using our optimizations. Further, the overhead of our methods is small enough that they are applicable even if the computational structure does not substantially change for a few tens of iterations. These methods are general enough that they can be used to develop a runtime library which can be used by a compiler for performing these optimizations. The BFS algorithm achieved excellent speedups with very low cost compared to the other algorithms and should be the algorithm of choice for most applications.

7. Acknowledgments

References

[1] D. Bacon, S. Graham, and O. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–419, Dec. 1994.

[2] S. Carr. *Memory-Hierarchy Management*. PhD thesis, RICE Univ., July 1994.

[3] L. Dagum. Automatic partitioning of unstructured grids into connected components. In *Supercomputing*, pages 95–101. IEEE, Nov. 1993.

[4] V. Decyk and A. de Boer. Optimization of particle-in-cell codes on risc processors. *Computers in Physics*, 10(290), 1996.

[5] G. Karypis and V. Kumar. *METIS, Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 2.0*. Computer Science Dept., Univ. of Minnesota, Minneapolis, MN 55454, Aug. 1995.

[6] D. Nicol and J. Saltz. Dynami remapping of parallel computations with varying resource demands. *IEEE Trans. on Computers*, 37(9):1073–1087, Sept. 1988.

[7] C. Ou and S. Ranka. Fast and parallel mapping algorithms for irregular problems. *J. Supercomputing*, 1993.