Optimizing Graph Algorithms for Improved Cache Performance*

Joon-Sang Park, Michael Penner, and Viktor K Prasanna
University of Southern California
{jsp, mipenner, prasanna} @usc.edu
http://advisor.usc.edu

Abstract

Tiling has long been used to improve cache performance. Recursion has recently been used as a cache-oblivious method of improving cache performance. Both of these techniques are normally applied to dense linear algebra problems. We develop new implementations by means of these two techniques for the fundamental graph problem of Transitive Closure, namely the Floyd-Warshall Algorithm, and prove their optimality with respect to processor-memory traffic. Using these implementations we show up to 10x improvement in execution time. We also address Dijkstra's algorithm for the single-source shortest-path problem and Prim's algorithm for Minimum Spanning Tree, for which neither tiling nor recursion can be directly applied. For these algorithms, we demonstrate up to a 2x improvement by using a cache friendly graph representation. Experimental results are shown for the Pentium III, UltraSPARC III, Alpha 21264, and MIPS R12000 machines using problem sizes between 1024 and 4096 vertices. We demonstrate improved cache performance using the Simplescalar simulator.

1. Introduction

The topic of cache performance has been well studied in recent years. It has been clearly shown that the amount of processor-memory traffic is the bottleneck for achieving high performance in many applications [4][21]. While cache performance has been well studied, much of the focus has been on dense linear algebra problems, such as matrix multiplication and FFT [4][9][15][24]. All of these problems possess very regular access patterns that are known at compile time. In this paper, we take a different approach to this topic by focusing on some fundamental graph problems.

Optimizing cache performance to achieve better overall performance is a difficult problem. Modern microprocessors are including deeper and deeper memory hierarchies to hide the cost of cache misses. The performance of these deep memory hierarchies has been shown to differ significantly from predictions based on a single level of cache [21]. Different miss penalties for each level of the memory hierarchy as well as the TLB also play an important role in the effectiveness of cache-friendly optimizations. These penalties vary among processors and cause large variations in execution time.

The area of graph problems are fundamental in a wide variety of fields, most notably network routing, distributed computing, and computer aided circuit design. Graph problems pose unique challenges to improving cache performance due to their irregular data access patterns. These challenges often cannot be handled using standard cache-friendly optimizations [7]. The focus of this research is to develop methods of meeting these challenges.

In this paper we present a number of optimizations to the Floyd-Warshall algorithm, Dijkstra's algorithm, and Prim's algorithm. For the Floyd-Warshall algorithm we present a recursive implementation that achieves a 6x improvement over the baseline implementation. We also show that by tuning the base case for the recursion, we can further improve performance by up to 2x. We also show a novel approach to tiling for the Floyd-Warshall algorithm that achieves performance very close to that of the recursive implementation. Note that today's state of the art research compilers cannot generate this implementation [7].

There are some natural combinations of implementation and data layout that decrease overhead costs, such as index computation, and yield performance advantage. In this paper, we show that our implementations of the Floyd-Warshall algorithm perform roughly equal with either the Morton layout or the Block Data Layout.

For Dijkstra's algorithm and Prim's algorithm, to which tiling and recursion are not directly applicable, we present a cache-friendly graph representation. By matching the data layout of the representation to the access pattern we show up to a 2x improvement in execution time.

The remainder of this paper is organized as follows: In Section 2 we give the background needed and briefly summarize some related work in the areas of cache optimization and compiler optimizations. In Section 3 we

* Supported by the US DARPA Data Intensive Systems Program under contract F33615-99-1-1483 monitored by Wright Patterson Airforce Base and in part by an equipment grant from Intel Corporation.

discuss optimizing the Floyd-Warshall algorithm. In Section 4 we discuss optimizing Dijkstra's algorithm and the graph representation. In Section 5 we apply the optimizations discussed in Section 4 to Prim's algorithm. In Section 6 we draw conclusions.

2. Background and Related Work

In this section we give the background information required in our discussion of various optimizations in Section 3 - 5. In Section 2.1 we give a brief outline of the graph algorithms. For more details of these algorithms see [6]. In Section 2.2 we discuss some of the challenges that are faced in making the transitive closure problem cache-friendly. We also discuss the model that we use to analyze cache performance and the four architectures that we use for experimentation throughout the paper. Finally, in Section 2.3 we give some information regarding other work in the fields of cache analysis, cache-friendly optimizations, and compiler optimizations and how they relate to our work.

2.1. Overview of Key Graph Algorithms

For the sake of discussion, suppose we have a directed graph G with N vertices labeled 1 to N and E edges. The Floyd-Warshall algorithm (FW) is a dynamic programming algorithm, which computes a series of N, NxN matrices where D^k is the k^{th} matrix and is defined as follows: $D^k_{(i,j)}$ = shortest path from vertex i to vertex j composed of the subset of vertices labeled 1 to k.

Dijkstra's algorithm is designed to solve the single-source shortest path problem. It does this by repeatedly extracting from a priority queue Q the nearest vertex u to the source, given the distances known thus far in the computation (Extract-Min operation). Once this nearest vertex is selected, all vertices v that neighbor u are updated with a new distance from the source (Update operation).

Prim's algorithm for Minimum Spanning Tree is very similar to Dijkstra's algorithm for the single-source shortest path problem. In both cases a root node or source node is chosen and all other nodes reside in the priority queue. Nodes are extracted using an Extract-min operation and all neighbors of the extracted vertex are updated. The difference in Prim's algorithm is that nodes are updated with the weight of the edge from the extracted node instead of the weight from the source or root node.

2.2. Challenges

Transitive closure presents a very different set of challenges from those present in dense linear algebra problems such as matrix multiply and FFT. These challenges are highlighted here and discussed in more detail in [17]. In the Floyd-Warshall algorithm, the operations involved are comparison and add operations. Also, we are faced with unique data dependencies between iterations of the outer loop. This data dependency from one k^{th} loop to the next eliminates the ability of any commercial or research compiler to improve data reuse. We have explored using the SUIF research compiler and found that it cannot perform the optimizations discussed in Section 3 without user provided knowledge of the algorithm [7]. These challenges mean that often transitive closure displays much longer running times than other problems equivalent in complexity.

In Dijkstra's algorithm and Prim's algorithm, the largest data structure is the graph representation. An optimal representation, with respect to space, would be the adjacency-list representation. However, this involves pointer chasing when traversing the list. The priority queue has been highly optimized by various groups over the years. Unfortunately, the update operation is often excluded, as it is not necessary in such algorithms as sorting. The asymptotically optimal implementation that considers the update operation is the Fibonacci heap. Unfortunately this implementation includes large constant factors and did not perform well in our experiments.

The model that we use in this paper is that of a uni-processor, cache-based system. We refer to the cache closest to the processor as L_1 with size C_1, and subsequent levels as L_i with size C_i. Throughout this paper we refer to the amount of *processor-memory traffic*. This is defined as the amount of traffic between the last level of the memory hierarchy that is smaller than the problem size and the first level of the memory hierarchy that is larger than or equal to the problem size. In most cases we refer to these as cache and memory respectively. Finally, we assume an internal TLB with a fixed number of entries.

We use four different architectures for our experiments. The Pentium III Xeon running Windows 2000 is a 700 MHz, 4 processor shared memory machine with 4 GB of main memory. Each processor has 32 KB of level-1 data cache and 1 MB of level-2 cache on-chip. The level-1 cache is 4-way set associative with 32 B lines and the level-2 cache is 8-way set associative with 32 B lines. The UltraSPARC III machine is a 750 MHz SUN Blade 1000 shared memory machine running Solaris 8. It has 2 processors and 1 GB of main memory. Each processor has 64 KB of level-1 data cache and 8 MB of level-2 cache. The level-1 cache is 4-way set associative with 32 B lines and the level-2 cache is direct mapped with 64 B lines. The MIPS machine is a 300 MHz R12000, 64 processor, shared memory machine with 16 GB of main memory. Each processor has 32 KB of level-1 data cache and 8 MB of level-2 cache. The level-1 cache is 2-way set associative with 32 B lines and the

level-2 cache is direct mapped with 64 B lines. The Alpha 21264 is a 500 MHz uniprocessor machine with 512 MB of main memory. It has 64 KB of level-1 data cache and 4 MB of level-2 cache. The level-1 cache is 2-way set associative with 64 B lines and the level-2 cache is direct mapped with 64 B lines. It also has an 8 element fully-associative victim cache. All experiments are run on a uniprocessor or on a single node of a multiprocessor system. Unless otherwise specified the Simplescalar simulations are done using 16 KB of level-1 data cache and 256 KB of level-2 cache parameters.

2.3. Related Work

A number of groups have done research in the area of cache performance analysis in recent years. Detailed cache models have been developed by Weikle, McKee, and Wulf in [23] and Sen and Chatterjee in [21]. XOR-based data layouts to eliminate cache misses have been explored by Valero and others in [10].

A number of papers have discussed the optimization of specific dense linear algebra problems with respect to cache performance. Whaley and others discuss optimizing the widely used Basic Linear Algebra Subroutines (BLAS) in [24]. Chatterjee and Sen discuss a cache efficient matrix transpose in [4]. Frigo and others discuss the cache performance of cache oblivious algorithms for matrix transpose, FFT, and sorting in [9]. Park and Prasanna discuss dynamic data remapping to improve cache performance for the DFT in [15]. One characteristic that all these problems share is a very regular memory accesses that are known at compile time.

Another area that has been studied is the area of compiler optimizations (see for example [19]). Optimizing blocked algorithms has been extensively studied (see for example [12]). The SUIF compiler framework includes a large set of libraries including libraries for performing data dependency analysis and loop transformations. In this context, it is important to note that SUIF does not handle the data dependencies present in the Floyd-Warshall algorithm in a manner that improves the processor-memory traffic. It will not perform the transformations discussed in Section 3 without user intervention [7].

Although much of the focus of cache optimization has been on dense linear algebra problems, there has been some work that focuses on irregular data structures. Chilimbi et. al. discusses making pointer-based data structures cache-conscious in [5]. He focuses on providing structure layouts to make tree structures cache-conscious. LaMarca and Ladner developed analytical models and showed simulation results predicting the number of cache misses for the heap in [13]. However, the predictions they made were for an isolated heap, and the model they used was the *hold model*, in which the heap is static for the majority of operations. In our work, we consider Dijkstra's algorithm and Prim's algorithm in which the heap is very dynamic. In both Dijkstra's algorithm and Prim's algorithm O(N) Extract-Mins are performed and O(E) Updates are performed. Finally in [20], Sanders discusses a highly optimized heap with respect to cache performance. He shows significant performance improvement using his *sequential heap*. The sequential heap does support Insert and Delete-min very efficiently, however the Update operation is not supported.

In the presence of the Update operation, the asymptotically optimal implementation of the priority queue, with respect to time complexity, is the Fibonacci heap. This implementation performs O(N*lg(N) + E) operations in both Dijkstra's algorithm and Prim's algorithm. In our experiments the large constant factors present in the Fibonacci heap caused it to perform very poorly. For this reason, we focus our work on the graph representation and the interaction between the graph representation and the priority queue.

We have recently published work on the Floyd-Warshall algorithm in [17] that showed a 2x improvement using the Unidirectional Space Time Representation. Compared with [17], this paper represents a new approach to optimizing the Floyd-Warshall algorithm, namely recursion and a novel tiled implementation, which gives up to an additional 3x improvement in execution time. Further, we expand our scope of algorithms to include Dijkstra's algorithm for the single source shortest path problem and Prim's algorithm for the minimum spanning tree problem.

3. Optimizing FW

In this section we address the challenges of the Floyd-Warshall algorithm. In Section 3.1 we introduce and prove the correctness of a recursive implementation for the Floyd-Warshall algorithm. We also analyze the cache performance and show experimental results for this implementation compared with the baseline. We also show that by tuning the recursive algorithm to the cache size, we can improve its performance by up to 2x. In Section 3.2, we present a novel tiled implementation of the Floyd-Warshall algorithm. Finally, in Section 3.3, we address the issue of data layout for both the blocked implementation and the recursive implementation.

Throughout this section we make the following assumptions. We assume a directed graph with N vertices and E edges. We assume the cache model described in Section 2.2, where $Ci < N^2$ for some i and the TLB size is much less than N. To experimentally validate our approaches and their analysis, the actual problem sizes that we are working with are between 1024 and 4096 nodes ($1024 \leq N \leq 4096$). Each data element is 8 bytes.

Many processors currently on the market have in the range of 16 to 64 KB of level-1 cache and between 256 KB and 4 MB of level-2 cache. Many processors have a TLB with approximately 64 entries and a page size of 4 to 8 KB.

In [11], it was shown that the lower bound on processor-memory traffic was $\Omega(N^3/\sqrt{C})$ for the usual implementation of matrix multiply. By examining the data dependency graphs for both matrix multiplication and the Floyd-Warshall algorithm, it can be shown that matrix multiplication reduces to the Floyd-Warshall algorithm with respect to processor-memory traffic. Therefore, we have the following:

Lemma 3.1: The lower bound on processor-memory traffic for the Floyd-Warshall algorithm, given a fixed cache size C, is $\Omega(N^3/\sqrt{C})$, where N is the number of vertices in the input graph.

3.1. A Recursive Implementation of FW

As stated earlier, recursive implementations have recently been used to increase cache performance. It was stated in [8] that recursive implementations perform automatic blocking at every level of the memory hierarchy. To the authors' knowledge, there does not exist a recursive implementation of the Floyd-Warshall algorithm. The reason for this, is that the Floyd-Warshall algorithm not only contains all the dependencies present in ordinary matrix multiplication, but also additional dependencies that can not be satisfied by the simple recursive implementation of matrix multiply. What is shown here is a novel recursive implementation of the Floyd-Warshall algorithm. We also prove the correctness of the implementation and show analytically that it reaches the asymptotically optimal amount of processor memory traffic.

In order to design a recursive implementation of the Floyd-Warshall algorithm, first examine the standard implementation when applied to a 2x2 matrix. The standard implementation loops over the variables k, i, and j from 1 to N. When the 2x2 case is unrolled we have the code shown in Figure 1a. Notice that 8 calls are made to the min() operation and each call requires 3 data values from the matrix. This is converted into a recursive program by replacing the call to the min() function with a recursive call. Instead of passing 3 data values, we pass 3 sub-matrices corresponding to quadrants of the input matrix. This code is shown in Figure 1b. The initial call to the recursive algorithm passes the entire input matrix as each argument. Subsequent calls pass quadrants of their input arguments as shown in Figure 1b. The code similar to Figure 1a calling the min() operation is used as the base case for when the input matrices are of size 2x2.

Theorem 3.1: The recursive implementation of the Floyd-Warshall algorithm detailed above satisfies all dependencies in the Floyd-Warshall algorithm and computes the correct result.

For a proof of Theorem 3.1, namely the correctness of the recursive implementation of the Floyd-Warshall algorithm see [14].

Theorem 3.2: The recursive implementation reduces the processor-memory traffic by a factor of B, where $B = \Theta(\sqrt{C})$.

Proof:

Note that the running time of this algorithm is given by

$$T(N) = 8 * T\left(\frac{N}{2}\right) = \Theta(N^3) \qquad \textbf{1}$$

Define the amount of processor memory traffic by the function $D(x)$. Without considering cache, the function behaves exactly as the running time.

$$D(N) = 8 * D\left(\frac{N}{2}\right) = \Theta(N^3) \qquad \textbf{2}$$

Consider the problem after k recursive calls. At this point the problem size is $N/2^k$. There exists some k such that $N/2^k = \Theta(\sqrt{C})$, where C = cache size. For simplicity we set $B = N/2^k$. At this point, all data will fit in the cache and no further traffic will occur for recursive calls below this point. Therefore:

$$D(B) = O(B^2) \qquad \textbf{3}$$

By combining Equation 2 and Equation 3 it can be shown that:

$$D(N) = \frac{N^3}{B^3} * D(B) = O(\frac{N^3}{B}) \qquad \textbf{4}$$

Therefore, the processor-memory traffic is reduced by a factor of B. ■

Theorem 3.3: The recursive implementation reduces the traffic between the i^{th} and the $(i\text{-}1)^{th}$

```
Floyd-Warshall (A) {
    A11 = min(A11, A11+A11);
    A12 = min(A12, A11+A12);
    A21 = min(A21, A21+A11);
    A22 = min(A22, A21+A12);
    A22 = min(A22, A22+A22);
    A21 = min(A21, A22+A21);
    A12 = min(A12, A12+A22);
    A11 = min(A11, A12+A21);
}
```

```
FWR (A, B, C) {
  if (not base case) {
    FWR(A11, B11, C11);
    FWR(A12, B11, C12);
    FWR(A21, B21, C11);
    FWR(A22, B21, C12);
    FWR(A22, B22, C22);
    FWR(A21, B22, C21);
    FWR(A12, B12, C22);
    FWR(A11, B12, C21);
  }
  else {
    /* run base case */
  }
}
```

Figure 1, a&b: Recursive implementation of FW.

level of cache by a factor of B_i at each level of the memory hierarchy, where $B_i = O(\sqrt{C_i})$.

Proof:

Note first of all, that no tuning was assumed when calculating the amount of processor-memory traffic in the proof of Theorem 3.2. Namely, Equation 3 holds for any N and any B where $B = O(\sqrt{C})$.

In order to prove Theorem 3.3, first consider the entire problem and the traffic between main memory and the m^{th} level of cache (size C_m). By Theorem 3.2, the traffic will be reduced by B_m where $B_m = O(\sqrt{C_m})$. Next consider each problem of size B_m and the traffic between the m^{th} level of cache and the $(m-1)^{th}$ level of cache (size C_{m-1}). By replacing N in Theorem 3.2 by B_m, it can be shown that this traffic is reduced by a factor of B_{m-1} where $B_{m-1} = O(\sqrt{C_{m-1}})$.

This simple extension of Theorem 3.2 can be done for each level of the memory hierarchy, and therefore the processor-memory traffic between the i^{th} and the $(i-1)^{th}$ level of cache will be reduced by a factor of B_i, where $B_i = O(\sqrt{C_i})$. ■

Finally, recall from Lemma 3.1 that the lower bound on processor-memory traffic for the Floyd-Warshall algorithm is given by $\Omega(N^3/\sqrt{C})$, where C is the cache size. Also recall from Theorem 3.2 the upper bound on processor-memory traffic that was shown for the recursive implementation was $O(N^3/B)$, where $B^2 = O(C)$. Given this information we have the following Theorem.

Theorem 3.4: Our recursive implementation is asymptotically optimal among all implementations of the Floyd-Warshall algorithm with respect to processor-memory traffic.

As a final note in the recursive implementation, we show up to 2x improvement when we set the base case such that the base case would utilize more of the cache closest to the processor. Once we reached a problem size B, where B^2 is on the order of the cache size, we execute a standard iterative implementation of the Floyd-Warshall algorithm. This improvement varied from one machine to the other and is due to the decrease in the overhead of recursion. It can be shown that the number of recursive calls in the recursive algorithm is reduced by a factor of B^3 when we stop the recursion at a problem of size B. A comparison of full recursion and recursion stopped at a larger block size showed a 30% improvement on the Pentium III and a 2x improvement on the UltraSPARC III.

In order to further improve performance, B^2 must be chosen to be on the order of the L1 cache size. The simplest and possibly the most accurate method of choosing B is to experiment with various tile sizes. This is the method that the Automatically Tuned Linear Algebra Subroutines (ATLAS) project employs [24]. However, it is beneficial to find an estimate of the optimal block size. In order to get an estimate we used the block size selection heuristic for finding this estimate discussed in [14].

The baseline we use for our experiments is a straightforward implementation of the Floyd-Warshall algorithm. It was shown in [17] that standard optimizations yield limited performance increases on most machines. The Simplescalar results in Table 1 for the recursive implementation show a 30% decrease in level-1 cache misses and a 2x decrease in level-2 cache misses for problem sizes of 1024 and 2048. In order to verify the improvements on real machines, we compare the recursive implementation of the Floyd-Warshall algorithm with the baseline. For these experiments the best block size was found experimentally. The results show a 10x improvement in overall execution time on the Alpha, better than 7x improvement on the Pentium III and the MIPS, and almost a 3x improvement on the UltraSPARC III. These results are shown in Figures 3 - 6. Differences in performance gains between machines are expected, due to the wide variance in cache parameters and miss penalties.

3.2. A Tiled Implementation for FW

Compiler groups have used tiling to achieve higher data reuse in looped code. Unfortunately, the data dependencies from one k-loop to the next in the Floyd-Warshall algorithm make it impossible for current compilers including research compilers to perform 3 levels of tiling. In order to tile the outermost loop we must cleverly reorder the tiles in such a way that satisfies data dependencies from one k-loop to the next as well as within each k-loop.

Consider the following tiled implementation of the Floyd-Warshall algorithm. Tile the problem into BxB tiles. During the k^{th} block iteration, first update the $(k,k)^{th}$

Data level-1 cache misses

N	Baseline	Recursive
1024	0.806	0.546
2048	6.442	4.362

(billions)

Data level-2 cache misses

N	Baseline	Recursive
1024	0.537	0.280
2048	4.294	2.232

(millions)

Table 1: Simplescalar result

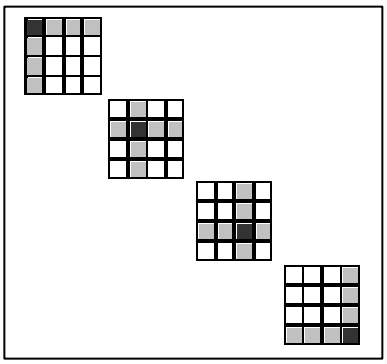


Figure 2: Tiled implementation of FW.

tile, then the remainder of the k^{th} row and k^{th} column, then the rest of the matrix. Figure 2 shows an example matrix tiled into a 4x4 matrix of blocks. Each block is of size BxB. During each outermost loop, we would update first the black tile representing the $(k,k)^{th}$ tile, then the grey tiles, then the white tiles. In this way we satisfy all dependencies from each k^{th} loop to the next as well as all dependencies within each k^{th} loop.

Theorem 3.5: The proposed tiled implementation of the Floyd-Warshall algorithm reduces the processor-memory traffic by a factor of B where B^2 is on the order of the cache size.

Proof sketch: At each block we perform B^3 operations. There are N/B x N/B blocks in the array and we pass through each block N/B times. This gives us a total of N^3 operations. In order to process each block we require only $3*B^2$ elements. This gives us a total of N^3/B total processor-memory traffic. ■

Data level-1 cache misses

N	Baseline	Tiled	
1024	0.806	0.542	
2048	6.442	4.326	10^9

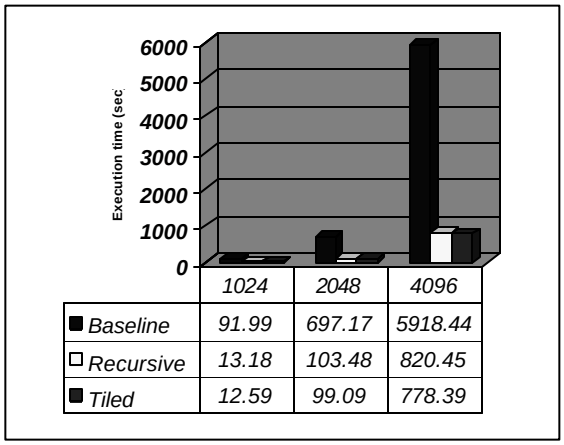
Data level-2 cache misses

N	Baseline	Tiled	
1024	0.537	0.276	
2048	4.294	2.195	10^6

Table 2: Simplescalar result

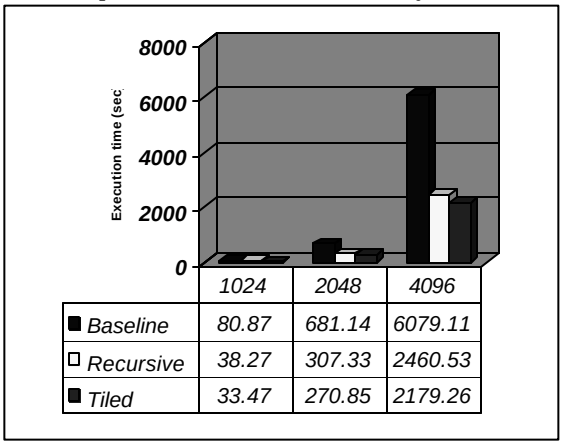
Given this upper bound on traffic for the tiled implementation and the lower bound shown in Lemma 3.1, we have.

Theorem 3.6: The proposed tiled implementation is asymptotically optimal among all implementations of the Floyd-Warshall



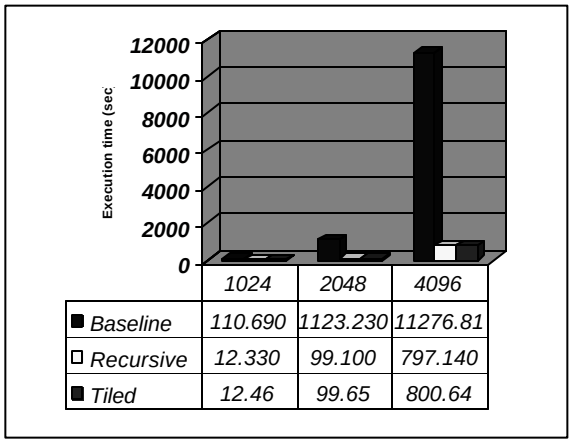
	1024	2048	4096
Baseline	91.99	697.17	5918.44
Recursive	13.18	103.48	820.45
Tiled	12.59	99.09	778.39

Figure 3: Pentium III results.



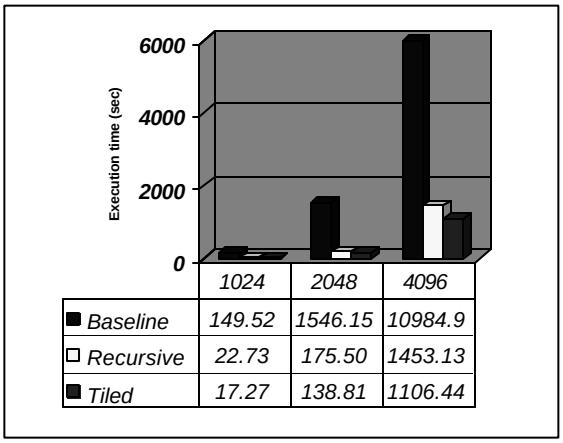
	1024	2048	4096
Baseline	80.87	681.14	6079.11
Recursive	38.27	307.33	2460.53
Tiled	33.47	270.85	2179.26

Figure 4: UltraSPARC III results.



	1024	2048	4096
Baseline	110.690	1123.230	11276.81
Recursive	12.330	99.100	797.140
Tiled	12.46	99.65	800.64

Figure 5: MIPS results.



	1024	2048	4096
Baseline	149.52	1546.15	10984.9
Recursive	22.73	175.50	1453.13
Tiled	17.27	138.81	1106.44

Figure 6: Alpha results.

Recursive Implementation

N	Morton Layout	Block Data Layout	
2048	103.48	111.42	
4096	820.45	878.89	(sec)

Tiled Implementation

N	Morton Layout	Block Data Layout	
2048	99.25	99.39	
4096	779.53	780.41	(sec)

Table 3: Pentium III results.

Recursive Implementation

N	Morton Layout	Block Data Layout	
2048	307.33	311.26	
4096	2460.53	2488.88	(sec)

Tiled Implementation

N	Morton Layout	Block Data Layout	
2048	278.48	271.35	
4096	2248.20	2184.09	(sec)

Table 4: Ultrasparc III results.

algorithm with respect to processor-memory traffic.

When implementing the tiled implementation of the Floyd-Warshall algorithm, it is important to use the best possible block size. As mentioned in Section 3.1, the best block size should be found experimentally, and the block size selection heuristic discussed in Section 3.1 can be used to give a rough bound on the best block size. However, when implementing the tiled implementation, it is also important to note that the search space must take into account each level of cache as well as the size of the TLB. Given these various solutions for B the search space should be expanded accordingly.

Simplescalar results for the tiled implementation are shown in Table 2. These results show a 2x improvement in level-2 cache misses and a 30% improvement in level-1 cache misses. Experimental results show a 10x improvement in execution time for the Alpha, better than 7x improvement for the Pentium III and the MIPS and roughly a 3x improvement for the UltraSPARC III (See Figures 3 - 6).

3.3. Data Layout Issues

It is also important to consider the data layout when implementing any algorithm. It has been shown by a number of groups that data layouts tuned to the data access pattern of the algorithm can reduce both TLB and cache misses (see for example [15], [18]). In the case of the recursive algorithm, the access pattern is matched by a Z-Morton data layout. See [4] for a definition of the Morton ordering.

In the case of the tiled implementation, the Block Data Layout (BDL) matches the access pattern. The BDL is a two level mapping that maps a tile of data, instead of a row, into contiguous memory. See [14] for a definition of the BDL. By setting the block size equal to the tile size in the tiled computation, the data layout will exactly match the data access pattern.

We experimented with both of these data layouts for each of the algorithms. The results are shown in Tables 3 and 4. All of the execution times were within 15% of each other with the Z-Morton data layout winning slightly for the recursive implementation and the BDL winning slightly for the tiled implementation. The fact that the Z-Morton was slightly better for the recursive implementation and likewise the BDL for the tiled implementation was exactly as expected, since they match the data access pattern most closely. The closeness of the results is mostly likely due to the fact that the majority of the data reuse is within the final block. Since both of these data layouts have the final block laid out in contiguous memory locations, they perform equally well.

4. Optimizing the Single-Source Shortest Path Problem

Due to the structure of Dijkstra's algorithm neither tiling nor recursion can be directly applied. Much work has been done to generate cache friendly implementations of the heap, however, the update operation has not been considered in great detail (see section 2.3). In the presence of the update operation, the Fibonacci heap represents the asymptotically optimal implementation with respect to time complexity. Unfortunately the performance of the Fibonacci heap was very poor compared with even a straightforward implementation of the heap [14].

As mentioned in Section 2, the largest data structure is the graph representation. This structure will be of size $O(N+E)$, where E can be as large as N^2 for dense graphs. In contrast, the priority queue, the other data structure involved, will be of size $O(N)$. For these reasons, we focus primarily on optimizing the graph representation and on eliminating the cache conflicts between the graph representation and the priority queue.

One difficulty we face when optimizing the graph representation is the access pattern. In Dijkstra's algorithm each element in the representation is accessed

exactly once. For each node that is extracted from the heap, the corresponding list of adjacent nodes is read from the graph representation. Once each node is extracted from the heap, the computation is complete. In this context, we can take advantage of two things. The first is prefetching. Modern processors perform aggressive prefetching in order to hide memory latencies. The second is to optimize at the cache line level. In this case, a single miss would bring in multiple elements that would subsequently be accessed and result in cache hits. This is known as minimizing cache pollution.

There are two commonly used graph representations. This representation is of size $O(N^2)$. It has the nice property that elements are accessed in a contiguous fashion and therefore, cache pollution will be minimized and prefetching will be maximized. However, for sparse graphs, the size of this representation is inefficient. Each node in the list includes the cost of the edge from the given node to the adjacent node. This representation has the property of being of optimal size for all graphs, namely $O(N+E)$. However, the fact that it is pointer based, leads to cache pollution and difficulties in prefetching. See [6] for more details.

Consider a simple combination of these two representations. For each node in the graph, we have a corresponding array of adjacent nodes. The size of this array is exactly the out-degree of the given node. There are simple methods to construct this representation when the out-degree is not known until run time. For this representation, the elements at each point in the array look similar to the elements stored in the adjacency list. Each element must store both the cost of the path and the index of the adjacent node. Since the size of each array is exactly the out-degree of the corresponding node, the size of this representation is exactly $O(N+E)$. This makes it optimal with respect to size. Also, since the elements are stored in arrays and therefore in contiguous memory locations, the cache pollution will be minimized and prefetching will be maximized. Subsequently this representation will be referred to as the *adjacency array representation*.

Cache miss rates

	Linked-List	Adj. Array
D-Level 1	0.2936	0.2622
D-Level 2	0.4242	0.3545

(DL1:16k, DL2:256k, Input: 2048 nodes, 0.9 density)

Table 5: Simplescalar results.

In order to demonstrate the performance improvements using our graph representation, we performed Simplescalar simulations as well as experiments on two different machines, the Pentium III and UltraSPARC III, for Dijkstra's algorithm. The Simplescalar simulations show a significant improvement in level-2 cache miss rate for the adjacency array representation compared with the adjacency list representation (see Table 5). This is due to the reduction

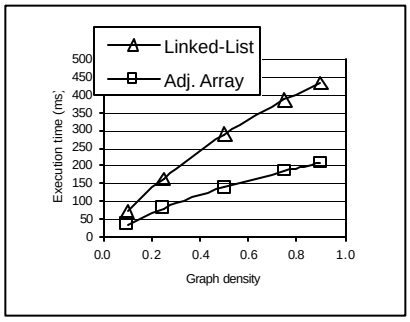


Figure 7: Dijkstra's alg. on Pentium III, *N* = 2048

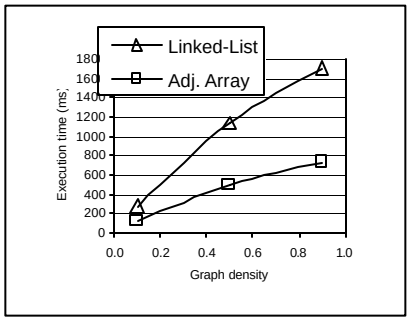


Figure 8: Dijkstra's alg. on Pentium III, *N* = 4096

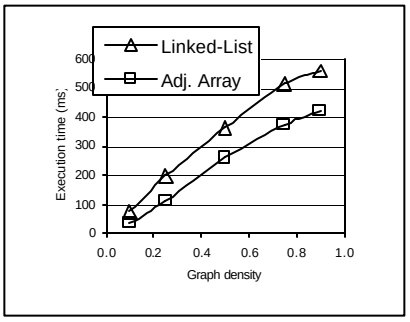


Figure 9: Dijkstra's alg. on UltraSPARC III, *N* = 2048

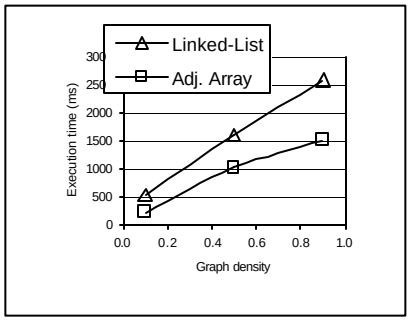


Figure 10: Dijkstra's alg. on UltraSPARC III. *N* = 4096

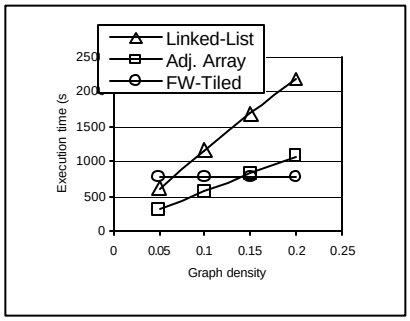


Figure 11: Dijkstra's algorithm vs. best FW on Pentium III, *N* = 2048

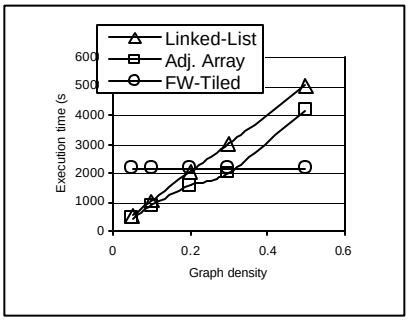


Figure 12: Dijkstra's algorithm vs. best FW on UltraSPARC III, *N* = 2048

N	Linked-List	Adj. Array	Adj. Array w/ coloring
512 K	13.57	10.24	10.41
1 M	14.369	13.879	14.864
2 M	32.324	32.108	31.419
4 M	71.908	69.796	66.531

(sec)

Table 6: Dijkstra's algorithm on UltraSPARC III.

in cache pollution and increase in prefetching that was predicted. The experimental results also demonstrate improved performance. Figures 7 - 10 show a 2x improvement for Dijkstra's algorithm on the Pentium III and a 20% improvement on the UltraSPARC III. This significant difference in performance is due primarily to the difference in the memory hierarchy of these two architectures.

A second comparison to observe is between the Floyd-Warshall algorithm and Dijkstra's algorithm for sparse graphs, i.e. edge densities less than 20%. For these graphs, Dijkstra's algorithm is more efficient for the all pairs shortest path problem. By using the adjacency array representation of the graph in Dijkstra's algorithm, the range of graphs over which Dijkstra's algorithm outperforms the Floyd-Warshall algorithm can be increased. Figures 11 & 12 show a comparison of the best Floyd-Warshall algorithm with Dijkstra's algorithm for sparse graphs. On the Pentium III, we were able to increase the range for Dijkstra's algorithm from densities up to 5% to densities up to 20%. On the UltraSPARC III we increased the range from densities up to 20% to densities up to 30%.

This first set of experiments was done using the problem sizes used in Section 3 in order to compare the Floyd-Warshall algorithm with Dijkstra's algorithm for the all pairs shortest path problem. Also, with these problem sizes, the cache is much larger than the data set size and cache conflicts are not a significant problem. If the problem set is much larger, the conflict between the graph representation and the priority queue should be considered. In our data layout the priority queue is placed in memory such that it maps only to the top half of the cache. The graph representation is placed in memory such that it maps only to the bottom half of the cache. In this way the conflicts between the graph representation and the priority queue will be eliminated. It should be noted that this scheme may increase the cache conflicts within the priority queue, since it is a frequently accessed data structure.

Experimental results for problem sizes larger than the cache size are shown in Table 6. For these problem sizes the performance gains were somewhat smaller than the gains for smaller problems. This is most likely due to the fact that the priority queue is much larger and causes more traffic. Also, due to memory limitations, the experiments were run for very sparse graphs, $E = O(N)$. The adjacency array graph representation should give more performance improvements for denser graphs, simply because the number of adjacent nodes will be greater. In the case of 512 K nodes, the experiment had a higher percentage of edges and showed better performance improvements.

5. Optimizing the Minimum Spanning Tree Problem

As mentioned in Section 2, Prim's algorithm for minimum spanning tree is very similar to Dijkstra's algorithm for the single source shortest path problem. For this reason the optimizations applicable to Dijkstra's algorithm are applicable to Prim's algorithm. Figures 13 - 16 show the result of applying the optimization to the graph representation discussed in Section 4 to Prim's algorithm. Recall that this was an optimization to the graph representation replacing the adjacency list representation with the adjacency array representation.

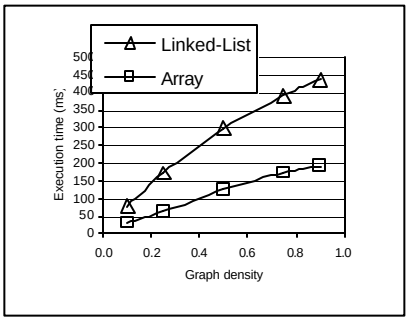


Figure 13: Prim's alg. on Pentium III. *N* = 2048

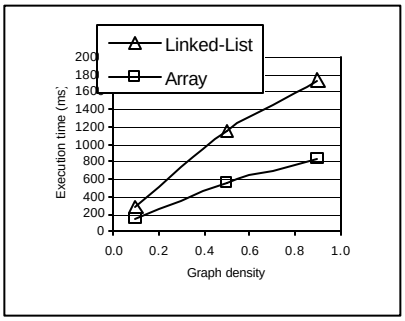


Figure 14: Prim's alg. on Pentium III. *N* = 4096

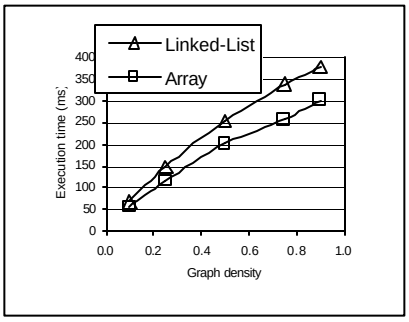


Figure 15: Prim's alg. on UltraSPARC III. *N* = 2048

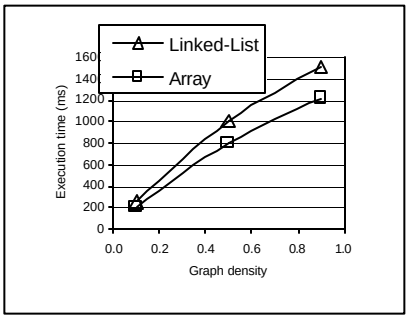


Figure 16: Prim's alg. on UltraSPARC III, *N* = 2048

Our results show a 2x improvement on the Pentium III and 20% for the UltraSPARC III. These results are for problem sizes 2048 and 4096. This result is very similar to the results we saw for the same comparison in Dijkstra's algorithm. Recall that our Simplescalar results for Dijkstra's algorithm showed an improvement in the level-2 cache misses. Based on the similarity between Dijkstra's algorithm and Prim's algorithm, we could expect similar cache performance for Prim's algorithm.

6. Conclusion

Using various optimizations for graph algorithms, we have showed a 3x to 10x improvement for the Floyd-Warshall algorithm and a 20% to 2x improvement for Dijkstra's algorithm and Prim's algorithm. Our optimizations to the Floyd-Warshall algorithm represent a novel recursive implementation as well as a novel tiled implementation of the algorithm. For Dijkstra's algorithm and Prim's algorithm, we presented a cache-friendly graph representation that gave significant performance improvements.

One area for future work is the optimization of the priority queue in Dijkstra's algorithm and Prim's algorithm. As mentioned, the Fibonacci heap is the asymptotically optimal implementation for priority queue in the presence of the update operation, however, due to large constant factors, it performed poorly in experiments.

This work is part of the Algorithms for Data IntensiVe Applications on Intelligent and Smart MemORies (ADVISOR) Project at USC [1]. In this project we focus on developing algorithmic techniques for mapping applications to architectures. Through this we understand and create a framework for application developers to exploit features of advanced architectures to achieve high performance.

7. References

[1] ADVISOR Project. http://advisor.usc.edu/.

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley Publishing Company, Menlo Park, California, 1974.

[3] D. Burger and T. M. Austin. The SimpleScalar Tool Set, Version 2.0. University of Wisconsin-Madison Computer Sciences Department Technical Report #1342, June, 1997.

[4] S. Chatterjee and S. Sen. Cache Efficient Matrix Transposition. In *Proc. of International Symposium on High Performance Computer Architecture*, January 2000.

[5] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-Conscious Structure Layout. In *Proc. of ACM SIGPLAN Conference on Programming Language Design and Implementation*, May 1999.

[6] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, Massachusetts, 1990.

[7] P. Diniz. USC ISI, Personal Communication, March, 2001.

[8] Jeremy D. Frens and David S. Wise. Auto-blocking matrix-multiplication or tracking BLAS3 performance from source code. In *Proc. of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, June 1997.

[9] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-Oblivious Algorithms. In *Proc. of 40th Annual Symposium on Foundations of Computer Science*, 17-18, New York, NY, USA, October, 1999.

[10] A. Gonzalez, M. Valero, N. Topham, and J. M. Parcerisa. Eliminating Cache Conflict Misses through XOR-Based Placement Functions. In *Proc. of 1997 International Conference on Supercomputing*, Vienne, Austria, July, 1997.

[11] J. Hong and H. Kung. I/O Complexity: The Red Blue Pebble Game. In *Proc. of ACM Symposium on Theory of Computing*, 1981.

[12] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. In *Proc. of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, California, April 1991.

[13] A. LaMarca and R. E. Ladner. The Influence of Caches on the Performance of Heaps. *ACM Journal of Experimental Algorithmics*, 1, 1996.

[14] J. Park, M. Penner, and V. K. Prasanna. Optimizing Graph Algorithms for Improved Cache Performance. Technical Report USC-CENG 02-01, Department of Electrical Engineering, USC, January 2002.

[15] N. Park, D Kang, K Bondalapati, and V. K. Prasanna. Dynamic Data Layouts for Cache-conscious Factorization of the DFT. In *Proc. of International Parallel and Distributed Processing Symposium*, May 2000.

[16] D. A. Patterson and J. L. Hennessy. *Computer Architecture A Quantitative Approach*. 2nd Ed., Morgan Kaufmann Publishers, Inc., San Francisco, California, 1996.

[17] M. Penner and V. K. Prasanna. Cache-Friendly Implementations of Transitive Closure. In *Proc. of International Conference on Parallel Architectures and Compiler Techniques*, Barcelona, Spain, September 2001.

[18] G. Rivera and C. Tseng. Data Transformations for Eliminating Conflict Misses. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Montreal, Canada, June 1998.

[19] F. Rastello and Y. Robert. Loop Partitioning Versus Tiling for Cache-Based Multiprocessor. In *Proc. of International Conference Parallel and Distributed Computing and Systems*, Las Vegas, Nevada, 1998.

[20] P. Sanders. Fast Priority Queues for Cached Memory. *ACM Journal of Experimental Algorithmics*, 5, 2000.

[21] S. Sen, S. Chatterjee. Towards a Theory of Cache-Efficient Algorithms. In *Proc. of Symposium on Discrete Algorithms*, 2000.

[22] SPIRAL Project. http://www.ece.cmu.edu/~spiral/.

[23] D. A. B. Weikle, S. A. McKee, and Wm.A. Wulf. Caches As Filters: A New Approach To Cache Analysis. In *Proc. of Grace Murray Hopper Conference*, September 2000.

[24] R. C. Whaley and J. J. Dongarra. Automatically Tuned Linear Algebra Software. *High Performance Computing and Networking*, November 1998.