Optimizing Graph Algorithms for Improved Cache Performance

Joon-Sang Park, Michael Penner,
and Viktor K. Prasanna

CENG 02-01

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California 90089-2562
(213-740-4483)

Optimizing Graph Algorithms for Improved Cache Performance[*+]

Joon-Sang Park, Michael Penner, and Viktor K. Prasanna
Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California 90089-2562
{jsp, mipenner, prasanna}@usc.edu
http://advisor.usc.edu

Abstract

Graph algorithms are fundamental in a wide variety of fields, and while much focus has been on optimizing various algorithms for improved cache performance, little focus has been on the area of graph algorithms. The reasons for this are varied, but at the core is that graph algorithms pose a very different and complex challenge to improving cache performance. In this paper, we present a new recursive implementation for the fundamental graph problem of Transitive Closure, namely the Floyd-Warshall Algorithm, and prove its optimality with respect to processor-memory traffic. Using this cache-oblivious implementation we show more than a 6x *improvement in execution time on three different architectures. We also discuss the impact of data layout on cache performance in the context of a tiled implementation of the Floyd-Warshall algorithm. Secondly, we 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. Finally, we apply both the cache friendly graph representation and the basic idea of tiling to the problem of graph matching. Using these techniques we show performance improvements of* 2x – 3x. *Experimental results are shown for the Pentium III, UltraSPARC III, Alpha 21264, and MIPS R12000 machines. Problem sizes ranged from 1024 to 4096 vertices for the Floyd-Warshall algorithm and up to 65536 vertices for Dijkstra's algorithm, Prim's algorithm, and graph matching. We demonstrate improved cache performance using the SimpleScalar simulator.*

---

[*] 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.
[+] A previous version of this paper appears in *Proceedings of the International Parallel and Distributed Processing Symposium*, April 2002.

1. Introduction

The motivation for this work is what is commonly referred to as the processor-memory gap. While memory density has been growing rapidly, the speed of memory has been far outpaced by the speed of modern processors. Current latencies to memory are on the order of 100 processor cycles. This phenomenon has resulted in severe application level performance degradation on high-end systems. This problem has been well studied for many dense linear algebra problems, such as matrix multiplication and FFT (see for example, [23][32][36]). A similar problem is also present and well studied in I/O systems (see for example, [17][33]).

A number of groups are attempting to improve performance by performing computations in memory. Smart memory or processing in memory takes advantage of the high on chip bandwidth of memory to perform data intensive operations (see for example, [4][20][37]). Other groups are attacking the problem in software; either in the compiler through reordering instructions and prefetching (see for example, [16][18][27]) or through complex data layouts to improve cache performance (see for example, [6][10][13]).

Achieving better overall performance by optimizing cache performance is a difficult problem. The performance of deep memory hierarchies present in most modern processors has been shown to differ significantly from predictions based on a single level of cache. 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 the effectiveness of cache performance optimizations.

The area of graph problems is fundamental in a wide variety of fields, most notably network routing, distributed computing, and computer aided circuit design. Network routing in particular is a rapidly growing problem with the explosion of the Internet. Routing tables are growing in size and the frequency of updates is pushing the limits of current routers. 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 [9]. The focus of this research is to develop methods of meeting these challenges. A suite of data intensive kernels or stressmarks designed to stress the memory hierarchy is discussed in [21] & [22]. The transitive closure problem discussed in this paper is from the stressmark suite.

In this paper we present a number of cache-friendly optimizations to the Floyd-Warshall algorithm, Dijkstra's algorithm, Prim's algorithm, and graph matching. For the Floyd-Warshall algorithm we present a *cache-oblivious* recursive implementation that achieves more than a 6x improvement over the baseline implementation on three different architectures. We also show that by tuning the base case for the recursion, we can further improve performance by up to 2x. We also show analysis and discuss the impact of data layout on cache performance in the context of a tiled implementation of the Floyd-Warshall algorithm. While these techniques are well known for dense linear algebra problems such as matrix multiply, their application to transitive closure faces a significantly different set of challenges. Note that today's state of the art research compilers cannot generate these implementations [9].

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 the recursive and tiled 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 use a known cache-friendly graph representation. By using a data layout for the graph representation that matches the access pattern we show up to a 2x improvement in execution time.

Finally, we use the techniques discussed with respect to the Floyd-Warshall algorithm and Dijkstra's algorithm to optimize cache performance for the problem of graph matching. The algorithm we use is a primitive graph matching algorithm for bipartite graphs. We first apply the cache friendly graph representation used for Dijkstra's algorithm and Prim's algorithm, since the data access pattern to the graph is similar. We then use the idea of tiling to reduce the working set size. Performance improvements were in the range of 2x to 3x depending on the density of the graph and the quality of the partitioning done to accomplish tiling.

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 discuss optimizing the Floyd-Warshall algorithm. In Section 4 we discuss optimizing Dijkstra's algorithm. In Section 5 we apply the optimizations discussed in Section 4 to Prim's algorithm. In Section 6 we discuss applying the techniques to the problem of graph matching. Finally, in Section 7 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 - 6. In Section 2.1 we give a brief outline of the graph algorithms. Those readers comfortable with the algorithms can skip this. For more details of these algorithms see [7] or [14]. In Section 2.2 we give some background on cache-based architectures and optimizing algorithms for improved cache performance. In Section 2.3 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 is a dynamic programming algorithm, which computes a series of N, NxN matrices where D^k is the kth 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. The matrix D^0 is the original cost matrix for the given graph G. We can think of the algorithm as composed of N steps. At each kth step, we compute D^k using the data from D^{k-1} in the manner shown below for each $(i, j)^{th}$ value. Pseudo-code is given in Figure 1.

$$D^k_{(i,j)} = \min\left(D^{k-1}_{(i,j)}, D^{k-1}_{(i,k)} + D^{k-1}_{(k,j)}\right)$$

Dijkstra's algorithm is designed to solve the single-source shortest path problem. It does this by repeatedly extracting from a priority queue Q

```
Floyd-Warshall(W)

1. n ← rows[W]
2. D^(0) ← W
3. for k ← 1 to n
4.     for i ← 1 to n
5.         for j ← 1 to n
6.             d_ij^(k) ← min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
7. return D^(n)
```

Figure 1: Pseudo code for the Floyd-Warshall algorithm

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). The pseudo-code for the algorithm is given in Figure 2. The optimal implementation of Dijkstra's algorithm utilizes the Fibonacci heap and has complexity $O(N \lg(N) + E)$, although the Fibonacci heap may only be interesting in theory due to large constant factors.

```
Dijkstra's(V)

1. S = Ø
2. Q = V[G]
3. while Q ≠ Ø
4.      u = Extract-Min(Q)
5.      S = S U {u}
6.      for each vertex v ∈ Adj[u]
7.           Update d[v]
8. return S
```

Figure 2: Pseudo code for Dijkstra's algorithm

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.

```
Find_Match(G, M)

1. while (there exists an augmenting path)
2. {
3.      increase |M| by one using the augmenting
         path;
4. }

5. return M;
```

Figure 3: Pseudo code for primitive graph matching algorithm

For the sake of graph matching a subset M of E is considered a matching if no vertex is incident on more than one edge in M. A matching is considered maximal if it is not a subset of any other matching. A vertex is considered free if no edge in M is incident upon it. Using these definitions a primitive matching algorithm can be defined as follows [29]. Beginning at a free vertex use a breadth first search to find a path P from that free vertex to another free vertex alternating between edges in M and edges not in M. This is considered an augmenting path. Update the matching M by taking the symmetric difference of the edge sets of M and P. The algorithm is complete when no augmenting path can be found. The running time of this algorithm has been shown to be $O(N*E)$. Pseudo-code is given in Figure 3. A more detailed explanation of this primitive matching algorithm is given in [29].

2.2. Overview of Cache Based Architectures and Optimizing Algorithms for Improved Cache Performance

It is a well-known fact that the speed of modern processors is increasing at a rate of roughly 60% per year while the speed of memory is increasing at a rate of roughly 7% per year. This difference is often referred to as the processor-memory gap, and it causes the latency to memory as seen by the processor to increase significantly with each passing year. In order to hide this increasing latency, caches have been designed to take advantage of locality of reference; the fact that once an element is accessed there is a good chance that it and/or elements near will be accessed in the near future. The cache is much smaller than main memory and is placed much closer to the processor in terms of latency. Modern processors are including more levels of cache, each level larger in size and farther from the processor in terms of latency.

Invariably the processor will access data that is not in the cache and this will result in a cache miss. Cache misses can be categorized into one of three categories: cold misses, capacity misses, and conflict misses. A cold miss occurs the first time a data element is accessed. These misses are unavoidable. A capacity miss occurs if the working set of the application is larger than the cache. These misses can be avoided by either decreasing the working set or increasing the size of the cache. A conflict miss occurs if two or more data elements in the working set map to same place in the cache and the replacement of one results in a subsequent cache miss when that element is accessed. This type of miss can be avoided in a number of ways including improved data access patterns, improved data layout, reducing the working set, etc [24].

Two other issues that should be addressed are cache pollution and TLB misses. TLB misses are similar to cache misses except that they refer to misses in the Translation Look-aside Buffer. They can be categorized the same as cache misses and reducing them follows a similar pattern. Cache pollution is a somewhat different issue. This refers to when a cache line is brought into the cache and only a small portion of it is used before it is pushed out of the cache. A large amount of cache pollution will increase the bandwidth requirement of the application, even though the application is not utilizing more data.

Based on this discussion, the keys to improve the performance of the memory system are as follows: increase data reuse, decrease cache conflicts, and decrease cache pollution. The techniques that we use to achieve these ends can be categorized as data layout optimizations and data access pattern optimizations. In our data layout optimizations we attempt to match the data layout to an existing data access pattern. For example, we use the Block Data Layout to match the access pattern of a tiled algorithm (see Section 3), or we an adjacency array to match the access pattern of Dijkstra's algorithm and Prim's algorithm (see Section 4 & 5). In our data access pattern optimizations, we design both novel and trivial optimizations to the algorithm to improve the data access pattern. For example, we implemented both a novel tiled implementation and a novel recursive implementation of the Floyd-Warshall algorithm to improve the data access pattern.

A different approach to improving the performance of the cache is to design cache-oblivious algorithms. This is explored in by Frigo, et. al. in [12]. In this article, the algorithms do not ignore the presence of a cache, but rather they use recursion to improve performance regardless of the size or organization of the cache. By doing this, they can improve the performance of the algorithm without tuning the application to the specifics of the host machine. In our work we develop a cache-oblivious implementation of the Floyd-Warshall algorithm. One difference between this work and ours is that they assume a fully associative cache when developing and analyzing the techniques. For this reason, they do not consider any data layout optimizations to avoid cache conflicts. They assume that at some point in the recursion the problem will fit into the cache and all work done following this point will be of optimal cost. In fact we show between 20% and 2x performance improvements by optimizing what is done once we reach a problem size that fits into the cache.

2.3. Challenges

Transitive closure presents a very different set of challenges from those present in dense linear algebra problems such as matrix multiply and FFT. In the Floyd-Warshall algorithm, the operations involved are comparison and add operations. There are no floating-point operations as in matrix multiply and FFT. We are also faced with data dependences that require us to update the entire NxN array D^k before moving on to the $(k+1)^{th}$ step (see Figure 4). This data dependence 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 [9]. These challenges mean that although the computational complexity of the Floyd-Warshall algorithm is $O(N^3)$, equivalent to matrix multiply, often transitive closure displays much longer running times.

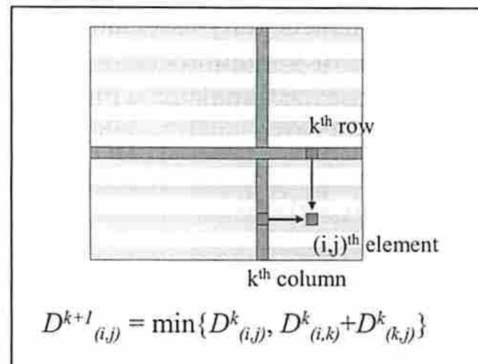

Figure 4: k^th iteration of outer loop in Floyd-Warshall Algorithm

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 primitive graph matching algorithm poses challenges that resemble challenges in both the Floyd-Warshall algorithm and Dijkstra's algorithm. As in the Floyd-Warshall algorithm, each breadth first search to find an augmenting path could examine any part or the entire input graph. Recall that the Floyd-Warshall algorithm requires updating the entire graph at each step. Unlike the Floyd-Warshall algorithm, tiling and recursion cannot be applied, even with clever reordering, since the search cannot be limited to a small part of the graph and still find a maximal matching for the entire graph. We also have the situation as in Dijkstra's algorithm where the size of the graph representation can affect performance and, although optimal with respect to size, the adjacency list representation could cause a degradation of cache performance due to pointer chasing when traversing the list.

The model that we use in this paper is that of a uniprocessor, 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 simulations are performed using the SimpleScalar simulator with a 16 KB, 4-way set associative level-1 data cache and a 256 KB, 8-way set associative level-2 cache.

2.4. Related Work

A number of groups have done research in the area of cache performance analysis and optimizations in recent years. Detailed cache models have been developed by Weikle, McKee, and Wulf in [35] and Sen and Chatterjee in[31]. XOR-based data layouts to eliminate cache misses have been explored by Valero and others in [13]. Data layouts for improving cache performance of embedded processor applications have been explored 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 [36]. Chatterjee, et. al. discuss layout optimizations for a suite of dense matrix kernels in [5]. Frigo and others discuss the cache performance of cache oblivious algorithms for matrix transpose, FFT, and sorting in [12]. Park and Prasanna discuss dynamic data remapping to improve cache performance for the DFT in [23]. 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 [27]). Optimizing blocked algorithms has been extensively studied (see for example [18]). The SUIF compiler framework includes a large set of libraries including libraries for performing data dependence analysis and loop transformations. In this context, it is important to note that SUIF does not handle the data dependences 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 [9].

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 [6]. 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 [19]. 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 [30], 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.

In [34],Venkataraman, et. al. present a tiled implementation of the Floyd-Warshall algorithm that is essentially the same as the tiled implementation shown in this paper. In this paper, we consider a

wider range of architectures and also analyze the cache performance with respect to processor memory traffic. We also consider data layouts to avoid conflict misses in the cache, which is not discussed in [34]. Due to the fact that we use a blocked data layout we are able to relax the constraint that the blocking factor should be a multiple of the number of elements that fit into a cache line. This allows us to use a larger block size and show more speedup. In [34], they derive an upper bound on achievable speed-up of 2 for state-of-the-art architectures. Our optimizations lead to more than a 6x improvement in performance on three different state-of-the-art architectures.

We have recently published work on the Floyd-Warshall algorithm in [25] that showed a 2x improvement using the Unidirectional Space Time Representation. Compared with [25], this paper represents a new approach to optimizing the Floyd-Warshall algorithm, namely recursion and tiling, 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, Prim's algorithm for the minimum spanning tree problem, and graph matching.

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 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 perform some analysis and discuss the impact of data layout on cache performance in the context of a tiled implementation of the Floyd-Warshall algorithm. Finally, in Section 3.3, we address the issue of data layout for both the tiled 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 2.3, where $C_i < N^2$ 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 [15], 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 dependence 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 [11] 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 dependences present in ordinary matrix multiplication, but also

additional dependences 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 5. 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 6. 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 6. The code similar to Figure 5 calling the min() operation is used as the base case for when the input matrices are of size 2x2.

In order to complete the proof of the correctness of the recursive implementation of the Floyd-Warshall algorithm we need the following claim.

Claim 1: When computing the following equation it is sufficient for the correctness of the Floyd-Warshall algorithm that $k' \geq k-1$.

$$D^{k}{}_{(i,j)} = \min\left(D^{k-1}{}_{(i,j)}, D^{k'}{}_{(i,k)} + D^{k'}{}_{(k,j)}\right) \tag{1}$$

Proof:

By virtue of the min operation, the values used for $D^{k'}_{(i,k)}$ & $D^{k'}_{(k,j)}$ will be $\leq$ to $D^{k-1}_{(i,k)}$ & $D^{k-1}_{(k,j)}$. Therefore, $D^{k'}_{(i,k)} + D^{k'}_{(k,j)} \leq D^{k-1}_{(i,k)} + D^{k-1}_{(k,j)}$, and $D^{k}_{(i,j)}$ using $k' \geq k-1$ will be $\leq$ the value computed using k-1. Since no values are used that are not representative of paths, there exists a path from the ith vertex to the jth vertex of cost given by Equation 1. Also, since the goal of the Floyd-Warshall algorithm is to find the shortest path, Equation 1 will give the correct final result. ■

As a final note, this does not claim that Equation 1 computes the shortest path from the ith vertex to the jth vertex using vertices up to k'. It merely computes a path from the ith vertex to the jth vertex that

Floyd-Warshall-2b2-Unrolled(W)

2. $D^{(0)} \leftarrow W$
3. $d_{11}^{(1)} \leftarrow \min(d_{11}^{(0)}, d_{11}^{(0)} + d_{11}^{(0)})$
4. $d_{12}^{(1)} \leftarrow \min(d_{12}^{(0)}, d_{11}^{(0)} + d_{12}^{(0)})$
5. $d_{21}^{(1)} \leftarrow \min(d_{21}^{(0)}, d_{21}^{(0)} + d_{11}^{(0)})$
6. $d_{22}^{(1)} \leftarrow \min(d_{22}^{(0)}, d_{21}^{(0)} + d_{12}^{(0)})$
7. $d_{22}^{(2)} \leftarrow \min(d_{22}^{(1)}, d_{22}^{(1)} + d_{22}^{(1)})$
8. $d_{21}^{(2)} \leftarrow \min(d_{21}^{(1)}, d_{22}^{(1)} + d_{21}^{(1)})$
9. $d_{12}^{(2)} \leftarrow \min(d_{12}^{(1)}, d_{12}^{(1)} + d_{22}^{(1)})$
10. $d_{11}^{(2)} \leftarrow \min(d_{11}^{(1)}, d_{12}^{(1)} + d_{21}^{(1)})$
11. return $D^{(2)}$

Figure 5: Pseudo code for the 2x2 unrolled version of the Floyd-Warshall algorithm

Floyd-Warshall-Recursive(A, B, C)

1. if (not base case) {
2. $A_{11} \leftarrow$ FWR(A_{11}, B_{11}, C_{11});
3. $A_{12} \leftarrow$ FWR(A_{12}, B_{11}, C_{12});
4. $A_{21} \leftarrow$ FWR(A_{21}, B_{21}, C_{11});
5. $A_{22} \leftarrow$ FWR(A_{22}, B_{21}, C_{12});
6. $A_{22} \leftarrow$ FWR(A_{22}, B_{22}, C_{22});
7. $A_{21} \leftarrow$ FWR(A_{21}, B_{22}, C_{21});
8. $A_{12} \leftarrow$ FWR(A_{12}, B_{12}, C_{22});
9. $A_{11} \leftarrow$ FWR(A_{11}, B_{12}, C_{21});
10. }
11. else {
12. /* run base case */
13. }
14. return A

Figure 6: Pseudo code for the recursive version of the Floyd-Warshall algorithm

is less than or equal in cost to the shortest path from the ith vertex to the jth vertex using vertices up to k-1.

Theorem 3.1: The recursive implementation of the Floyd-Warshall algorithm detailed above satisfies the dependences given by Equation 1 and correctly computes the transitive closure of the input graph.

Proof:

By definition the straightforward implementation of the Floyd-Warshall algorithm computes the outer product of the input matrix with addition replaced by minimum and multiplication replaced by addition. Subsequently, this is referred to as the FW outer product. Also, for the sake of simplicity, assume that the problem size (N) is a power of 2.

Base case:

When the number of vertices is equal to 2, the recursive implementation is identical to the original implementation of the Floyd-Warshall algorithm given in Figure 5.

Induction Step:

Assume that the recursive implementation correctly computes the FW outer product for problem sizes up to $N/2$. Then, for a problem of size N, the 8 recursive calls shown in Figure 6 will be made.

The first call, step 1, passes the Northwest quadrant as each argument. By assumption, this will correctly compute the Northwest quadrant of $D^{N/2}$. In other words, the shortest path will be found from i to j with all intermediate vertices in the set 1 to k, where i, j, and k are in the set 1 to $N/2$.

The second call, step 2, computes the Northeast quadrants of $D^{N/2}$. By Claim 1, we can use the data from the Northwest quadrant of $D^{N/2}$ instead of D^{k-1}. This step finds the shortest path from i to j with all intermediate vertices in the set 1 to k, where i and k are in the set 1 to $N/2$ and j is in the set $N/2 + 1$ to N.

In the same fashion, the third and fourth calls complete the computation of $D^{N/2}$ and after the first four recursive calls we have correctly computed the shortest path from from i to j with all intermediate vertices in the set 1 to k, where i and j are in the set 1 to N and k is in the set 1 to $N/2$.

The second set of four recursive calls works in the same way that the first set did and complete the computation of D^N, the last three using result from other quadrants of D^N instead of D^{k-1} by Claim 1. In this way, we correctly compute the shortest path from i to j, and by induction the recursive implementation of the Floyd-Warshall algorithm is correct for all N, where N is a power of 2. ■

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

Proof:

Note that the running time of this algorithm is given by

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

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

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

Consider the problem after k recursive calls. At this point the problem size is $N/2^k$. There exists some k such that $N/2^k = O(\sqrt{C})$, 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:

$$D(B) = O(B^2) \quad 4$$

By combining Equation 3 and Equation 4 it can be shown that:

$$D(N) = \frac{N^3}{B^3} * D(B) = O(\frac{N^3}{B}) \quad 5$$

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

Theorem 3.3: The recursive implementation reduces the traffic between the ith and the $(i\text{-}1)^{th}$ 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 5 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 mth 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 mth level of cache and the $(m\text{-}1)^{th}$ 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 ith and the $(i\text{-}1)^{th}$ 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 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 [36]. However, it is beneficial to find an estimate of the optimal tile size. A block size selection heuristic for finding this estimate is discussed in [25], and outlined here.

- Use the 2:1 rule of thumb from [24] to adjust the cache size to that of an equivalent 4-way set associative cache. This minimizes conflict misses since our working set consists of 3 tiles of data. Self-interference misses are eliminated by the data being in contiguous locations within each tile and cross interference misses are eliminated by the associativity.
- Choose B by Equation 6, where d is the size of one element and C is the adjusted cache size. This minimizes capacity misses.

$$3*B^2*d = C \quad (6)$$

The baseline we use for our experiments is a straightforward implementation of the Floyd-Warshall algorithm. It was shown in [25] that standard optimizations yield limited performance increases on most machines. The Simulation 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 more than 10x improvement in overall execution time on the MIPS, roughly than 7x improvement on the Pentium III and the Alpha, and more than 2x improvement on the UltraSPARC III. These results are shown in Figure 7. Differences in performance gains between machines are expected, due to the wide variance in cache parameters and miss penalties.

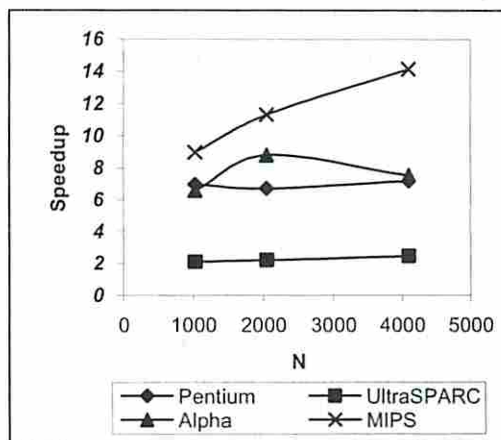

Figure 7: Speedup results for the recursive implementation of the Floyd-Warshall algorithm

Data level-1 cache misses

N	Baseline	Recursive	
1024	0.806	0.546	
2048	6.442	4.362	10^9

Data level-2 cache misses

N	Baseline	Recursive	
1024	0.537	0.280	
2048	4.294	2.232	10^6

Table 1: Simulation result

3.2. A Tiled Implementation for FW

Compiler groups have used tiling to achieve higher data reuse in looped code. Unfortunately, the data dependences 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 [9]. In order to tile the outermost loop we must cleverly reorder the tiles in such a way that satisfies data dependences from one *k*-loop to the next as well as within each *k*-loop.

Recall that Claim 1 stated that when computing Equation 1, it was sufficient that $k' \geq k-1$. Consider a special case of Claim 1 when we restrict *k'* such that $k-1 \leq k' \leq k+B-1$, where *B* is the blocking factor. This special case leads to the following tiled implementation of the Floyd-Warshall algorithm. This tiled implementation has also been derived in [34] using an alternate analysis. A brief description of the algorithm is as follows. Tile the problem into *B*x*B* tiles. During the kth block iteration, first update the $(k,k)^{th}$ tile, then the remainder of the kth row and k^{th} column, then the rest of the matrix. Figure 8 shows an example matrix tiled into a 4x4 matrix of blocks. Each block is of size *B*x*B*. 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 dependences from each kth loop to the next as well as all dependences within each kth loop.

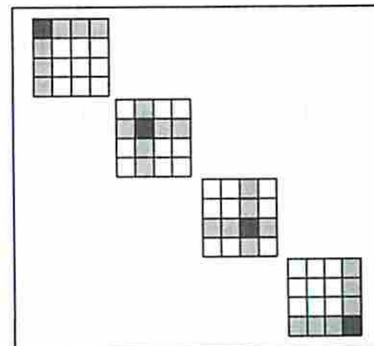

Figure 8: Tiled implementation of FW

3.2.1. Analysis. In [34], an upper bound for any cache optimized Floyd-Warshall algorithm was shown, however, no formal analysis with respect to traffic was shown for their tiled implementation. In fact

our results show speed-ups significantly larger than the upper bound shown in [34]. The following analysis is performed for the tiled implementation in the context of the model discussed in Section 2.

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. ■

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

Theorem 3.6: The proposed tiled implementation is asymptotically optimal among all implementations of the Floyd-Warshall algorithm with respect to processor-memory traffic.

3.2.2. Optimizing the Tiled Implementation. It has been shown by a number of groups that data layouts tuned to the access pattern can significantly impact cache performance and improve overall execution time. In order to match the access pattern of the tiled implementation we use the Block Data Layout (BDL). The BDL is a two level mapping that maps a tile of data, instead of a row, into contiguous memory. By setting the block size equal to the tile size in the tiled computation, the data layout will exactly match the data access pattern. By using this data layout we can also relax the restriction on block size stated in [34] that the block size should be a multiple of the number of elements in a cache block.

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. In [34], only the level-1 cache is considered, however, with an on-chip level-2 cache often the best block size is larger than the level-1 cache. Table 2 shows the result of comparing the tiled implementation using a row-wise layout and the block size selection used in [34] with the tiled implementation using the block data layout and our block size selection. Simulation results show that the

Data level-1 cache performance (10^9)

	Row-wise	BDL
Misses	0.312	0.276
Miss Rate	4.82%	4.28%

Data level-2 cache performance (10^6)

	Row-wise	BDL
Misses	91.43	7.45
Miss Rate	29.11%	2.68%

Execution time (sec), N = 2048

	Row-wise	BDL
SUN	283.72	201.38
P III	124.2	97.62

Table 2: Comparison result

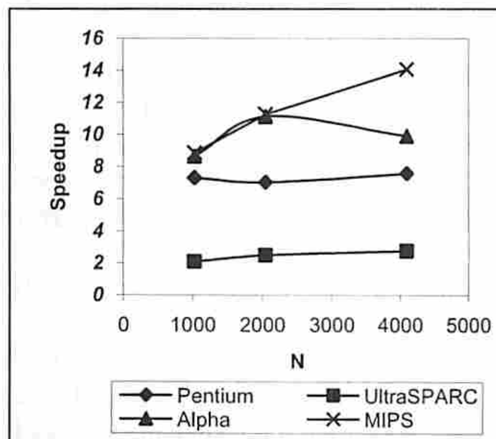

Figure 9: Speedup results for the tiled implementation of the Floyd-Warshall algorithm

Data level-1 cache misses (10^9)

N	Baseline	Tiled
1024	0.806	0.542
2048	6.442	4.326

Data level-2 cache misses (10^6)

N	Baseline	Tiled
1024	0.537	0.276
2048	4.294	2.195

Table 3: Simulation result

block size selection used in [34] optimizes the level-1 cache misses, but incurs a level-2 cache miss ratio of almost 30%. The Block Data Layout with a larger block size has roughly equal level-1 cache performance and far better level-2 cache performance. The execution times for these implementations show a 20% to 30% improvement by the Block Data Layout over the row-wise data layout.

A comparison for the tiled implementation using the Block Data Layout with the best compiler optimized implementation was also performed. Simulation results for this are shown in Table 3. 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 Figure 9).

3.3. Data Layout Issues

It is also important to consider 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 [5], [23], [25]). In the case of the recursive algorithm, the access pattern is matched by a Z-Morton data layout. The Z-Morton ordering is a recursive layout defined as follows: Divide the original matrix into 4 quadrants and lay these tiles in memory in the order NW, NE, SW, SE. Recursively divide each quadrant until a limiting condition is reach. This smallest tile is typically laid out in either row or column major fashion (see Figure 10). See [5] for a more formal definition of the Morton ordering.

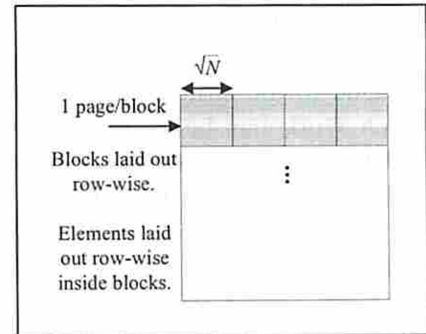

Figure 10: The Block Data Layout

In the case of the tiled implementation, the Block Data Layout (BDL) matches the access pattern. Recall from Section 3.2.2 that the BDL is a two level mapping that maps a tile of data, instead of a row, into contiguous memory. These blocks are laid out row-wise in the matrix and data is laid out row-wise within the block (see Figure 11). By setting the block size equal to the tile size in the tiled computation, the data layout will exactly match the data access pattern.

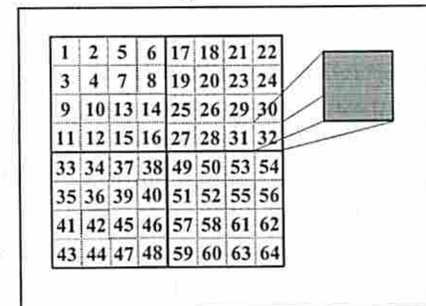

Figure 11: The Morton Layout

We experimented with both of these data layouts for each of the implementations. The results are shown in Tables 4 and 5. 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

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 4: Pentium III results

data layouts have the final block laid out in contiguous memory locations, they perform equally well.

It is also important to note that the Z-Morton data layout has a very complex index computation, which can only be hidden in a recursive algorithm. The BDL has a very simple index computation in comparison. Therefore it is significant to show that for non-recursive algorithms, the BDL performs just as well or better, while avoiding the overhead of a complex index computation.

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 5: UltraSPARC III results

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, in the problem sizes being considered, the performance of the Fibonacci heap was very poor compared with even a straightforward implementation of the heap.

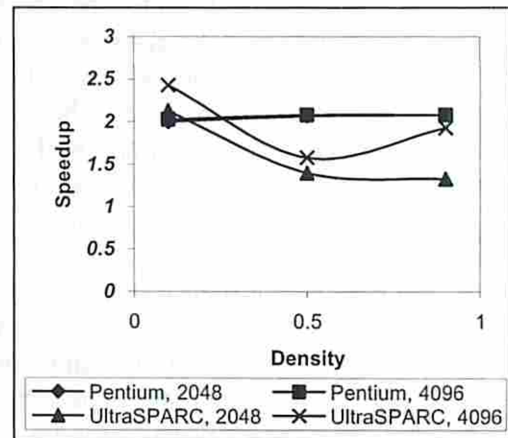

Figure 12: Speedup results for Dijkstra's algorithm

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). Also note that each element in the graph representation will be accessed exactly once. For each node extracted from the priority queue, the corresponding adjacent nodes are read and updated. All nodes will be extracted from the priority queue and no node can be extracted more than once. Therefore, the traffic as a result of the graph representation will be proportional to its size and the amount of prefetching possible. For these reasons, we focus on providing an optimization to the graph representation based on the data access pattern.

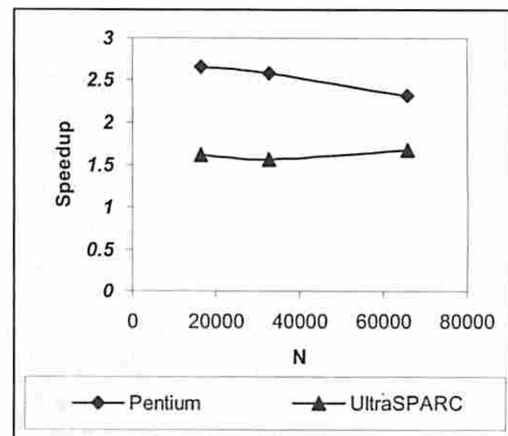

Figure 13: Speedup results for Dijkstra's algorithm

In the context of the graph representation, 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. In this way cache pollution is minimized.

There are two commonly used graph representations. The adjacency matrix is an NxN matrix, where the $(i,j)^{th}$

element of the matrix is the cost from the ith node to the jth node of the graph. 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. The adjacency list representation is a pointer-based representation where a list of adjacent nodes is stored for each node in the graph. 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 [7] or [14] for more details regarding these common graph representations.

Cache misses	Linked-List	Adj. Array
Data level 1	7.04	5.62
Data level 2	3.59	1.82
(Input: 16K nodes, 0.1 density)		(10^6)

Table 6: Simulation results

Consider a simple combination of these two representations [28]. For each node in the graph, we have an array of adjacent nodes. The size of each array is exactly the out-degree of the corresponding 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 $O(N+E)$. 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*. This graph representation is essentially the same as a graph representation discussed in [28].

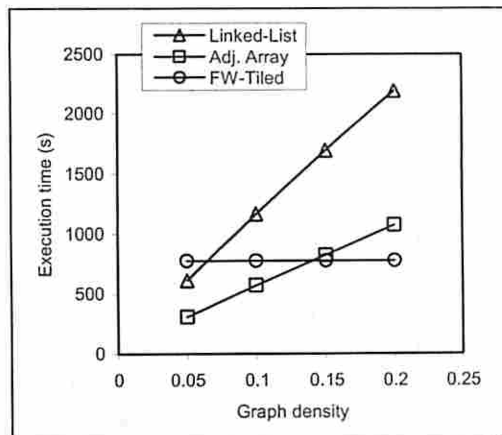

Figure 14: Dijkstra's algorithm vs. best FW on Pentium III, N = 2048

In order to demonstrate the performance improvements using our graph representation, we performed simulations as well as experiments on two different machines, the Pentium III and UltraSPARC III, for Dijkstra's algorithm. The simulations show approximately 20% reduction in level-1 cache misses and a 2x reduction in the number of level-2 cache misses (see Table 6). This is due to the reduction in cache pollution and increase in prefetching that was predicted. Due to memory limitations, experiments for all graph densities were only performed at small problem sizes, namely 2K nodes and 4K nodes. These results demonstrate improved performance using the adjacency array for all graph densities and are shown in Figure 12. Experiments on larger problem sizes (16K nodes up to 64K nodes) at a graph density of 10% are shown in Figure 13 and again are limited by the size of main memory. All of the results 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

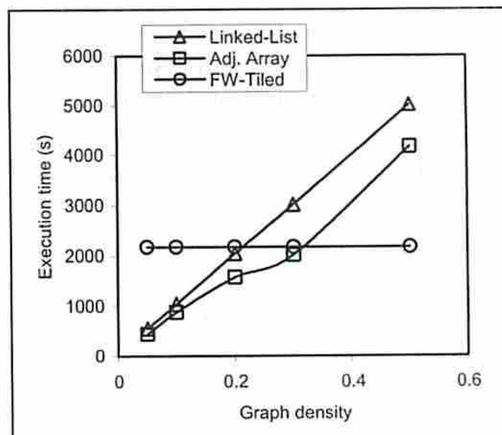

Figure 15: Dijkstra's algorithm vs. best FW on UltraSPARC III, N = 4096

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 14 & 15 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%.

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. In fact they are identical with respect to the access pattern, the difference being only in how the update operation is performed. In Dijkstra's algorithm nodes in the priority queue are updated with their distance from the source node. In Prim's algorithm nodes are updated with the shortest distance from any node already removed from the priority queue. For this reason the optimizations applicable to Dijkstra's algorithm are also applicable to Prim's algorithm. Figures 16 & 17 show the result of applying the optimization to the graph representation discussed in Section 4 to Prim's algorithm. Recall that this optimization replaces the adjacency list graph representation with the adjacency array graph representation. This representation matches the streaming access that is made to the graph and in this way minimizes cache pollution and maximizes the prefetching ability of the processor.

Our results show a 2x improvement on the Pentium III and 20% for the UltraSPARC III. This performance improvement was shown in the smaller problem sizes of 2K and 4K nodes where experiments were done for densities ranging from 10% to 90% as well as the large problem sizes of 16K nodes up to 64K nodes with densities of 10%. Simulations were also performed to verify improved cache performance. These results are shown in Table 7. They show approximately a 20% reduction in the number of level-1 cache misses and a 2x reduction in the number of level-2 cache misses. As expected, all of the results are very close to the results shown for Dijkstra's algorithm.

Cache misses	Linked-List	Adj. Array
Data level 1	7.19	5.77
Data level 2	3.59	1.82
(Input: 16K nodes, 0.1 density)		(10^6)

Table 7: Simulation results

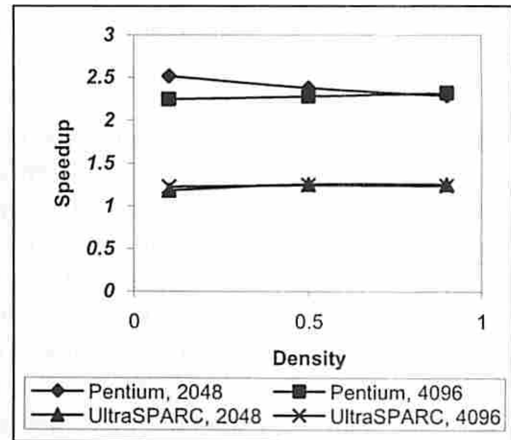

Figure 16: Speedup results for Prim's algorithm

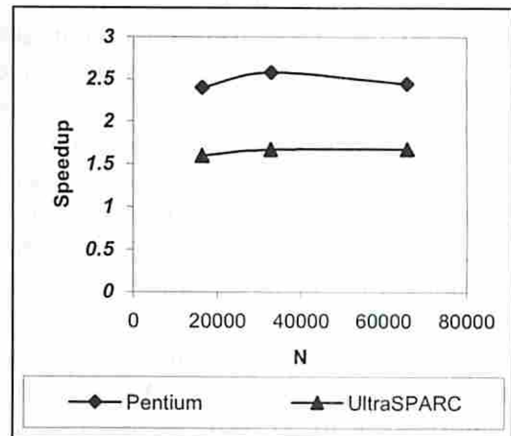

Figure 17: Speedup results for Prim's algorithm

6. Optimizing Bipartite Graph Matching

In this section, we utilize the ideas and techniques developed in the previous sections to optimize another basic graph algorithm, namely graph matching for bipartite graphs. As discussed in Section 2, this algorithm shows similarities to Dijkstra's algorithm with respect to memory access in each iteration and therefore tiling and recursion cannot be easily applied.

The first optimization that is applied is to use the adjacency arrays instead of the adjacency list. In order to find an augmenting path, a breadth first search is performed. The access pattern will then be to access all adjacent nodes to the current node. This is the same access pattern as was displayed in both Dijkstra's and Prim's algorithm.

The second optimization that is applied is intended to reduce the working set size as in tiling or recursion. As mentioned above, neither tiling nor recursion can be directly applied. What can be done is to use tiling to generate a good match as a starting point for the full problem. In this way the amount of work done when examining the complete graph will be reduced. Furthermore, the work done in the tiled steps will be cache friendly if the tiles are chosen appropriately. In order to accomplish this, first divide the graph into sub-graphs, each of which fits into the cache and find the local maximal matches. Then the local matches are combined to form a starting point for the original algorithm. Finally, the algorithm is run on the complete graph, using the match already found as a starting point, to find the maximal match.

The performance of this optimization is largely dependant on the structure and density of the graph and the partitioning chosen. Assuming a good partition, the local maximal matches will be close to a global maximal match for dense graphs due to the large number of edges present in each sub-graph. For sparse graphs, it is difficult to find a good local match and more work will be required at the global level.

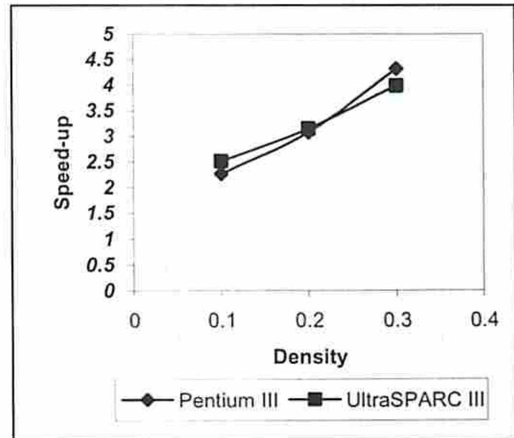

Figure 18: Speed-up vs. density results for graph matching

In order to improve the quality of the match at the local level, a very simple partitioning algorithm is employed. A basic description of this algorithm is as follows. Given a bipartite graph, the goal is to partition the edges into two groups such that the best match possible is found within each group. In order to accomplish this, as many edges as possible should have both end points in the same partition. These edges are referred to as internal edges. Arbitrarily partition the vertices into 4 equal partitions. Count the number of edges between each pair of partitions. Combine partitions into two partitions such that as many internal edges are created as possible.

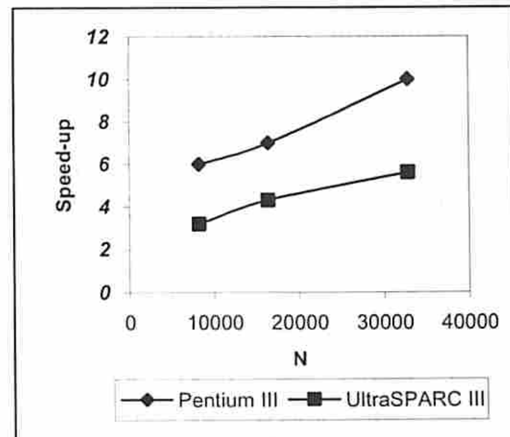

Figure 19: Best case speed-up results for graph matching

In order to support the quality of the optimization, experiments were also performed for a graph in which a worst possible graph partitioning was chosen, i.e. no matches were found at the local level. For this case, the optimized implementation showed only 10% performance degradation. The majority of experimentation was performed using randomly generated graphs in order to average out the dependence on graph partitioning. The random graphs were constructed by randomly choosing half

of the vertices to be in one partition of the bipartite graph. Edges were then created from each vertex in the partition to randomly chosen vertices not in the partition.

As expected, the performance improvement is highly dependent on the density of the graph. This dependence can be seen in Figure 18, which shows the speedup vs. graph density. Results ranged from just over 2x for graphs of 10% density to over 4x for graphs of 30% density. In this case, the problem size was fixed at 8192 nodes and density was limited to 30% by main memory. The best-case results are shown in Figure 19. For these problems, we designed the input graph such that the maximal matching is found in the tiled phase and very little work is performed on the complete graph. For these problems, results ranged from 3x up to 10x. The most interesting results are those shown in Figure 20. The input graph in this case was a randomly generated graph and the basic graph partitioning algorithm was used to improve the match found at the local level. The results shown are the average over 10 different random input graphs. The speedup shown is roughly 2x for all problem sizes. We also performed simulations to demonstrate cache performance for this case and the results are shown in Table 8. Based on the number of access to the level 1 cache, the optimized implementation is performing somewhat less work. This contributes somewhat to the decrease in the number of misses shown. However, the miss rate is also reduced by almost 3x, which indicates that the optimized implementation does improve cache performance beyond the amount reduced by the decrease in work.

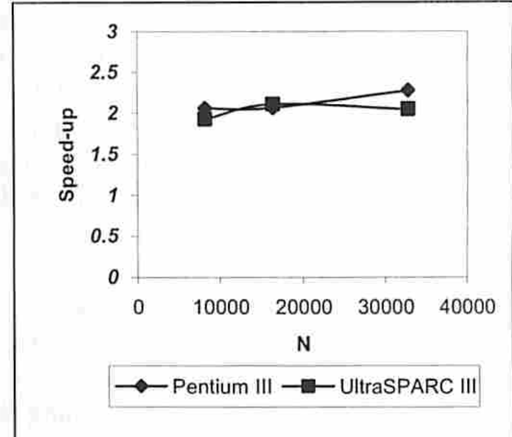

Figure 20: Average speed-up results for graph matching

DL1 Cache Performance	Baseline	Optimized
Accesses	853	578
Misses	127	32
Miss Rate	14.86%	5.56%
(Input: 8K nodes, 0.1 density)		(10^6)

Table 8: Simulation results

7. Conclusion

In the course of the research discussed in this paper, we have used the techniques of tiling, recursion, and data layout optimization to show improved cache performance both analytically and experimentally in the area of graph algorithms. The recursive implementation of the Floyd-Warshall algorithm represents a novel cache-oblivious implementation. Using this implementation as well as a tiled implementation, we have shown more than a 6x improvement in execution time on three different architectures as well as analytically showing that both implementations are optimal with respect to processor-memory traffic. We also showed significant performance improvements for Dijkstra's algorithm and Prim's algorithm using a cache friendly graph representation. Finally, we applied both the cache friendly graph representation and a tiling optimization to the problem of graph matching. These optimizations showed a 2x to 3x improvement in execution time for randomly generated graphs and up to 10x improvement for graphs well suited to our partitioning algorithm.

Tiling and recursion are also used as computation decomposition techniques for parallelization. Good parallelized code should have minimal communication and sharing between computational nodes, thus our pursuit of data locality also benefits parallelization. Our sequential FW implementations and matching implementation can easily be transformed into parallel code. Computation and data are already decomposed, what need to be added are computation and data

distribution, synchronization and communication primitives. One of our future directions will be to implement parallel versions of the Floyd-Warshall algorithm and matching algorithm based on the work presented in this paper.

Another 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 design 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.

8. 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] J.B. Carter, W.C. Hsieh, L.B. Stoller, M.R. Swanson, L. Zhang, and S.A. McKee. Impulse: Memory System Support for Scientific Applications. In the *Journal of Scientific Programming*, Vol. 7, No. 3-4, pp. 195-209, 1999.

[5] S. Chatterjee, V. V. Jain, A. R. Lebeck, S. Mundhra, and M. Thottethodi. Nonlinear Array Layouts for Hierarchical Memory Systems. *ACM Symposium on Parallel Algorithms and Architectures*, 1999.

[6] 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.

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

[8] M. Cosnard, P. Quinton, Y. Robert, and M. Tchuente (editors). *Parallel Algorithms and Architectures*. North Holland, 1986.

[9] P. Diniz. University of Southern California Information Sciences Institute, Personal Communication, March, 2001.

[10] N. Dutt, P. Panda, and A. Nicolau. Data Organization for Improved Performance in Embedded Processor Applications. *ACM Transactions on Design Automation of Electronic Systems*, Vol. 2, Number 4, October 1997.

[11] J. D. Frens and D. 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.

[12] 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.

[13] 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.

[14] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms*, Computer Science Press, Maryland, 1978.

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

[16] M. Kandemir, J. Ramanujam, and A. Choudhary. Improving Cache Locality by a Combination of Loop and Data Transformations. *IEEE Transactions on Computers*, Vol. 48, No. 2, February, 1999.

[17] M. Kallahalla and P. J. Varman. Optimal Prefetching and Caching for Parallel I/O Systems. In *Proc. of 13th ACM Symposium on Parallel Algorithms and Architectures*, 2001.

[18] 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.

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

[20] R. Murphy and P. M. Kogge. The Characterization of Data Intensive Memory Workloads on Distributed PIM Systems, *Intelligent Memory Systems Workshop, ASPLOS-IX 2000*, Boston, MA, Nov. 12, 2000.

[21] J. Mussmano. *Data-Intensive Systems Stressmark Suite*, Version 1.0, Atlantic Aerospace Electronics Corporation, August 24, 2000.

[22] J. Mussmano. DIS Benchmarking. DARPA Data Intensive Systems, Principal Investigator Meeting Presentation, Santa Fe, NM, March 26, 2002.

[23] 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.

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

[25] 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.

[26] 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.

[27] 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.

[28] S. Sahni. *Data Structures, Algorithms, and Applications in Java*. McGraw Hill, New York, 2000.

[29] H. A. B. Saip and C. L. Luchesi. Matching Algorithms for Bipartite Graphs. University of Campinas Technical Report DCC-03/93, Brazil, March, 1993.

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

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

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

[33] P. Varman, Parallel I/O Systems. *Handbook of Computer Engineering*, CRC Press, 2001.

[34] G. Venkataraman, S. Sahni, and S. Mukhopadhyaya. A Blocked All-Pairs Shortest-Paths Algorithm. *Scandinavian Workshop on Algorithms and Theory*, Lecture Notes in Computer Science, Vol. 1851, Editor: Magnus Halldorsson, Springer Verlag, 2000, 419-432.

[35] 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.

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

[37] L. Zhang, Z. Fang, M. Parker, B. K. Mathew, L. Schaelicke, J. B. Carter, W. C. Hsieh, and S. A. McKee. The Impulse Memory Controller. *IEEE Transactions on Computers*, Special Issue on Advances in High Performance Systems, November 2001.