Optimizing Graph Algorithms for Improved Cache Performance[*+]

Joon-Sang Park
University of California, Los Angeles, Computer Science Dept.
4732 Boelter Hall
Los Angeles, CA 90095-1596
jspark@cs.ucla.edu
(310) 206-8589

Michael Penner
University of Southern California, Dept. of Electrical Engineering
3740 McClintock Ave. EEB 205
Los Angeles, CA 90089
mipenner@alumni.usc.edu
(213) 740-9128

Viktor K. Prasanna
(corresponding author)
University of Southern California, Dept. of Electrical Engineering
3740 McClintock Ave. EEB 200C
Los Angeles, CA 90089
prasanna@usc.edu
(213) 740-4483

http://advisor.usc.edu

Index terms: Cache-Friendly Algorithms, Cache-Oblivious Algorithms, Graph Algorithms, Shortest Path, Minimum Spanning Trees, Graph Matching, Data Layout Optimizations, Algorithm Performance

---

[*] This work was done when the first author was at the University of Southern California.

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

Abstract

In this paper, we develop algorithmic optimizations to improve the cache performance of four fundamental graph algorithms. We present a cache-oblivious implementation of the Floyd-Warshall Algorithm for the fundamental graph problem of all-pairs shortest paths by relaxing some dependencies in the iterative version. We show that this implementation achieves the lower bound on processor-memory traffic of $\Omega(N^3/\sqrt{C})$, where N and C are the problem size and cache size respectively. Experimental results show that this cache-oblivious implementation shows more than 6× improvement in real execution time over that of the iterative implementation with the usual row major data layout, on three state-of-the-art architectures. Secondly, we address Dijkstra's algorithm for the single-source shortest paths problem and Prim's algorithm for Minimum Spanning Tree. For these algorithms, we demonstrate up to 2× improvement in real execution time by using a simple cache-friendly graph representation, namely adjacency arrays. Finally, we address the matching algorithm for bipartite graphs. We show performance improvements of 2× - 3× in real execution time by using the technique of making the algorithm initially work on sub-problems to generate a sub-optimal solution and then solving the whole problem using the sub-optimal solution as a starting point. Experimental results are shown for the Pentium III, UltraSPARC III, Alpha 21264, and MIPS R12000 machines with the problem sizes ranging from 1024 to 4096 vertices for the Floyd-Warshall algorithm and up to 65536 vertices for Dijkstra's algorithm, Prim's algorithm, and matching algorithm.

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 [32]. This phenomenon has resulted in severe application level performance degradation on high-end systems and has been well studied for many dense linear algebra problems like matrix multiplication and FFT [31][41][45]. A number of groups are attempting to improve performance by performing computations in memory [6][25]. Other groups are attacking the problem in software; either in the compiler through reordering instructions and prefetching [18][19][20][36] or through complex data layouts to improve cache performance [8][10][15].

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 [40]. 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 miss penalties vary from processor to processor and can cause large variations in experimental results.

The graph algorithms considered in this paper are fundamental and well known. Their importance is comparable to that of FFT to the signal processing domain and matrix multiplication to the scientific computing domain. The FFT and matrix multiplication have been isolated and studied extensively in the literature, as they are the major building blocks of solutions to problems in many areas. In a similar spirit, we study the following graph algorithms: the Floyd-Warshall algorithm, Dijkstra's algorithm, Prim's algorithm, and the augmenting path matching algorithm. The Floyd-Warshall algorithm solves the all-pairs shortest paths problem also referred to as transitive closure problem. The algorithm plays a significant role in many real-life applications: for example, in the analysis of correlated gene clusters in Bioinformatics [26]. In this problem, a graph represents the relationship among genes and identifying a gene cluster is finding a group of nodes (i.e. genes) located closely to each other in a set of graphs. As a first step of the solution, using the Floyd-Warshall algorithm, the distances between pairs of genes in graphs are computed. The Floyd-Warshall algorithm is also used in database systems for query processing and optimization [27][46], artificial intelligence for multiagent planning [4] and speech recognition [24], sensor networks for exposure and coverage calculation [23], and VLSI/CAD [14]. In Open Shortest Path First (OSPF) protocol [13], which is a widely adopted routing protocol for computer networks, each router in a network computes shortest paths to every other node using Dijsktra's algorithm based on a periodic exchange of link-states with other peer routers. Bipartite matching is the key algorithm in computing the cube operator in On-Line Analytic Processing (OLAP) databases [39]. All these applications are constrained due to massive amounts of data or real-time requirements, and in many cases, the efficiency of the underlying graph algorithm determines their overall performance.

Graph algorithms, in general, pose unique challenges to improving cache performance due to their irregular data access patterns. These challenges are significantly different from the challenges to dense linear algebra problems that are often easily handled using standard cache-friendly optimizations such

as tiling or blocking [20]. Optimizations such as tiling and data layout can be applied to some of the graph algorithms only after considering the specific details of each algorithm individually.

In the Floyd-Warshall algorithm, we are faced with data dependencies that require us to update an entire *N*x*N* array before moving on to the next iteration of the outermost loop. This data dependency from one iteration to the next makes automatic optimization through a compiler significantly more difficult than an algorithm like matrix multiplication. In Dijkstra's algorithm and Prim's algorithm, the largest data structure is the graph representation. An efficient representation, with respect to space, would be the adjacency-list representation. However, this involves pointer chasing when traversing the list. The priority queue, also used in these algorithms, has been highly optimized by various groups over the years [38]. However, in their optimizations, the update operation is often excluded, as it is not necessary in such algorithms as sorting.

The augmenting path matching algorithm for bipartite graphs (hereafter referred to as the 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. Unlike the Floyd-Warshall algorithm, the technique of recursion cannot be applied, even with clever reordering, since the search cannot be limited to a small part of the 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 focus of this paper is to develop methods for meeting these challenges. In this paper we present a number of cache-friendly optimizations to the Floyd-Warshall algorithm, Dijkstra's algorithm, Prim's algorithm, and the matching algorithm. 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. The baseline considered is a well-known implementation of the iterative Floyd-Warshall algorithm with the usual row major data layout. We

analytically show that our implementation achieves the optimal lower bound on processor memory traffic of $\Omega(N^3/\sqrt{C})$, where N and C are the problem size and cache size respectively.

For Dijkstra's algorithm and Prim's algorithm, to which tiling and recursion are not directly applicable, we use a 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 real execution time.

Finally, we discuss optimizing cache performance for the matching problem. We use the technique of making the algorithm initially work on small sized sub-problems to generate a sub-optimal solution and then solving the whole problem using the sub-optimal solution as a starting point. We show performance improvements in real execution time, in the range of 2x to 3x depending on the density of the graph. To support our claim that the performance improvement we achieve in real execution time comes mainly from improved cache performance, we also present SimpleScalar simulation results showing improved cache performance along with the experimental results.

Although this paper discusses optimizations targeting uniprocessor systems, many aspects of our optimizations are relevant to parallelization. As recursion is commonly used as a computation decomposition technique for parallelization, our recursive implementation can be used to decompose data and computation for a parallel version of the Floyd-Warshall algorithm. Given that our implementations incur minimal processor-memory traffic, parallel implementations based on our implementation will also incur minimal communication and/or sharing. Furthermore, the optimization of computation on each node is also important in achieving high performance. Our work can be directly applied to this problem.

The remainder of this paper is organized as follows: In Section 2 we briefly summarize some related work in the areas of cache optimization and compiler optimizations. In Section 3 we discuss our optimizations of graph algorithms. We discuss the optimization of the Floyd-Warshall algorithm in Section 3.1, the optimization of the single-source shortest paths problem and the minimum spanning tree problem in Section 3.2 and the optimization of the problem of bipartite matching in Section 3.3. In Section 4, we present our experimental results and finally, in Section 5, we draw conclusions.

2. 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 [44] and Sen and Chatterjee in [40]. XOR-based data layouts to eliminate cache misses have been explored by Valero and others in [15]. 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 [45]. Chatterjee et al. discuss layout optimizations for a suite of dense matrix kernels in [7]. Park and Prasanna discuss dynamic data remapping to improve cache performance for the DFT in [31]. One characteristic that all these problems share is a very regular memory accesses that are known at compile time. Another approach to improving the performance of the cache is to design cache-oblivious algorithms. This is explored by Frigo, et al. in [12], which discusses the cache performance of cache oblivious algorithms for matrix transpose, FFT, and sorting. 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.

Another area that has been studied is the area of compiler optimizations [36]. Optimizing blocked

algorithms has been extensively studied [20]. The SUIF compiler framework includes a large set of libraries including libraries for performing data dependency analysis and loop transformations such as tiling. Note that SUIF will not perform the transformations discussed in Section 3.1 without user intervention.

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 [8]. They focus 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 [21]. 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 [38], 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 [43], 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 [43]. 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 [43], 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.

3. Optimizations of Graph Algorithms

In order to improve cache performance an algorithm or application should increase data reuse, decrease cache conflicts and pollution. Cache pollution occurs 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. 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 3.1.2), or we use an adjacency array to match the access pattern of Dijkstra's and Prim's algorithm (see Section 3.2). 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 a novel recursive implementation of the Floyd-Warshall algorithm (see Section 3.1) and an implementation for the matching algorithm (see Section 3.3), which considers reducing the working set size to improve the data access pattern.

In this section we discuss our optimization techniques. In Section 3.1 we address the challenges of the Floyd-Warshall algorithm. We discuss Dijkstra's and Prim's algorithms in Section 3.2. We then discuss applying the techniques presented in these sections to the problem of the matching algorithm in Section 3.3.

The model that we use in this section 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 section 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 $C_i < N^2$ for some i where N is the problem size.

```
Floyd-Warshall(W)

1.  // Let N be the problem size of W
1.  D^0 ← W
2.  for k ← 1 to N
3.      for i ← 1 to N
4.          for j ← 1 to N
5.              d^k_ij ← min(d^(k-1)_ij, d^(k-1)_ik + d^(k-1)_kj)
6.  return D^N
```

Figure 1: Pseudo code for the Floyd-Warshall algorithm

3.1. Optimizing FW

In this section we address the challenges of the Floyd-Warshall algorithm. We start with introducing and proving the correctness of a recursive implementation for the Floyd-Warshall algorithm. We then analyze the cache performance.

```
FWI(A, B, C)

1.  // Let N be the problem size of A, B
        and C
2.  for k ← 1 to N
3.      for i ← 1 to N
4.          for j ← 1 to N
5.              a_ij ← min(a_ij, b_ik + c_kj)
```

Figure 2: Pseudo code for Floyd-Warshall algorithm with 3 arguments

We discuss that by tuning the recursive algorithm to the cache size, we can improve its performance. 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, we address the issue of data layout for both the tiled implementation and the recursive implementation.

```
FWR(A, B, C)

1.  if (base case)
2.      FWI(A, B, C);
3.  else
4.      FWR(A11, B11, C11);
5.      FWR(A12, B11, C12);
6.      FWR(A21, B21, C11);
7.      FWR(A22, B21, C12);
8.      FWR(A22, B22, C22);
9.      FWR(A21, B22, C21);
10.     FWR(A12, B12, C22);
11.     FWR(A11, B12, C21);
```

Figure 3: Pseudo code for the recursive implementation of the Floyd-Warshall algorithm

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, D^k, defined as follows: $D^k = (d^k_{ij})$ where d^k_{ij} is the weight of the shortest path

from vertex i to vertex j composed of the subset of vertices labeled 1 to k. The matrix D^0 is initialized with the original cost matrix W for the given graph G. We can think of the algorithm as composed of N steps. At each k^{th} step, we compute D^k using the data from the previous step D^{k-1}. Pseudo-code is given in Figure 1.

3.1.1. A Recursive Implementation of FW

Before presenting the recursive implementation, we introduce FWI, which is essentially the iterative Floyd-Warshall algorithm with three arguments, which are adjacency matrices. The operations on the elements of three input arguments are as shown in Figure 2. This is used as the base case for the recursive implementation. The pseudo code for the recursive implementation of the Floyd-Warshall algorithm is given in Figure 3. The initial call to the recursive algorithm passes the entire input matrix as each argument. Subsequent recursive calls pass quadrants of the input arguments as shown in Figure 3. At the first level of recursion A, B, and C all point to the given input adjacency matrix stored in the memory. At further levels of recursion, A, B, and C, can each point to the same subset or different subsets of the given input adjacency matrix. Note that in the first level of recursion A_{11}, computed and updated in the first call, is the input argument B_{11} to the second call.

Compared to ordinary matrix multiplication, the Floyd-Warshall algorithm contains additional dependencies that cannot be satisfied by a simple recursive implementation similar to that of matrix multiply. These additional dependencies are satisfied by considering the characteristic of the min operation (see Claim 1 below) and by ordering the first four recursive calls to operate on the matrix from the top left quadrant to the bottom right quadrant and the last four calls in a reverse order of the first four calls. This ordering of the recursive calls is crucial to the correctness of the final result.

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

Claim 1: Suppose d^k_{ij} is computed as

$$d^{k}_{ij} = \min(d^{k-1}_{ij}, d^{k'}_{ik} + d^{k''}_{kj}) \quad (1)$$

for $k - 1 \leq k', \leq N$, then upon termination, the Floyd-Warshall algorithm correctly computes the all-pairs shortest paths.

Proof: In the traditional Floyd-Warshall algorithm, by which we mean the algorithm shown in Figure 1, d^{k}_{ij} is computed as

$$d^{k}_{ij} = \min(d^{k-1}_{ij}, d^{k-1}_{ik} + d^{k-1}_{kj}).$$

To distinguish from the traditional Floyd-Warshall algorithm, we use t^{k}_{ij} to denote the results calculated using Equation 1, i.e.

$$t^{k}_{ij} = \min(t^{k-1}_{ij}, t^{k'}_{ik} + t^{k''}_{kj}) \quad (2)$$

for $k - 1 \leq k', k'' \leq N$ assuming that $t^{0}_{ij} = d^{0}_{ij}$ and that there is a sequence of computations (we will show that the algorithm in Figure 3 is such a sequence) that generates $t^{k'}_{ik}$ and $t^{k''}_{kj}$ prior to the computation of t^{k}_{ij}.

Now we show that for $1 \leq k \leq N$, the following inequality holds

$$t^{k}_{ij} \leq d^{k}_{ij}. \quad (3)$$

We prove this by induction.

Base case: By definition we have

$$t^{0}_{ij} = d^{0}_{ij}.$$

Induction step: Suppose $t^{k}_{ij} \leq d^{k}_{ij}$ for $k = m - 1$.

Then

$$\begin{aligned} t^{m}_{ij} &= \min(t^{m-1}_{ij}, t^{m'}_{im} + t^{m''}_{mj}) \\ &\leq \min(d^{m-1}_{ij}, t^{m'}_{im} + t^{m''}_{mj}) \\ &\leq \min(d^{m-1}_{ij}, t^{m-1}_{im} + t^{m-1}_{mj}) \\ &\leq \min(d^{m-1}_{ij}, d^{m-1}_{im} + d^{m-1}_{mj}) \\ &= d^{m}_{ij}. \end{aligned}$$

This completes the induction step, which shows that $t^k_{ij} \leq d^k_{ij}$ for $1 \leq k \leq N$.

On the other hand, since the traditional algorithm computes the *shortest* paths at termination and Equation 2 computes the length of *some* path, we have

$$d^N_{ij} \leq t^N_{ij}. \quad 4$$

From Equation 3 and 4, we have

$$d^N_{ij} = t^N_{ij}$$

which completes the proof. ■

Theorem 3.1: The recursive implementation of the Floyd-Warshall algorithm shown in Figure 3 correctly computes the shortest paths between all pairs of vertices of the input graph.

Proof sketch: Due to space limitation, we give a proof sketch. The details of the proof are given in Appendix A of the digital library version of this publication and also in [28]. The proof is by induction. For the sake of simplicity we assume that the problem size $N = 2^n$. We use FWR(A, B, C) to denote both the code segment shown in Figure 3 and the resulting matrix A after executing the code. Also FWI(A, B, C) represents both the code segment in Figure 2 and resulting matrix A of the code. D^k denotes the cost matrix containing the weights of the shortest paths in which the highest-numbered intermediate vertex is k. The initial call is FWR(D, D, D) where D is the input $N \times N$ adjacency matrix.

Base case: When the number of vertices is equal to 2, that is $n = 1$, the recursive implementation is identical to the original implementation of the Floyd-Warshall algorithm.

Induction Step: Assuming that the recursive implementation correctly computes the all-pairs shortest paths for $n = m - 1$, we need to prove it is true for the case of problem size $N = 2^m$. In the initial call, 8 recursive calls are made as shown in Figure 3.

The first call is interpreted as FWR(D^0_{11}, D^0_{11}, D^0_{11}) where D_{11} is the northwest quadrant of D. D_{11} is essentially an adjacency matrix of $N/2$ vertices. By induction hypothesis, this call correctly computes and updates D_{11} to $D^{N/2}_{11}$. 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 is FWR(D^0_{12}, $D^{N/2}_{11}$, D^0_{12}) where D^0_{12} is the northeast quadrant of D and like D_{11}, is also an adjacency matrix of $N/2$ vertices. Notice that at this point D_{11} has been updated to $D^{N/2}_{11}$ after the first recursive call whereas D_{12} still contains original values. This call is in the form of FWR(A, B, A). It can be shown (see Appendix A) that for every operation $a^k_{ij} = \min(a^{k-1}_{ij}, b^{k-1}_{ik} + a^{k-1}_{ij})$ in FWI(A, B, A), there is a corresponding operation $a^k_{ij} = \min(a^{k-1}_{ij}, b^{k-1}_{ik} + a^{k'}_{ij})$ in FWR(A, B, A) where $k' \geq k - 1$. Using Claim 1, it can be proven that the result of FWR(D^0_{12}, $D^{N/2}_{11}$, D^0_{12}) is the same as FWI(D^0_{12}, $D^{N/2}_{11}$, D^0_{12}). On the other hand, it is easy to show that FWI(D^0_{12}, $D^{N/2}_{11}$, D^0_{12}) correctly computes and generates $D^{N/2}_{12}$. This is to say FWR(D^0_{12}, $D^{N/2}_{11}$, D^0_{12}) 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 call computes $D^{N/2}_{21}$ using D^0_{21} and $D^{N/2}_{11}$, and the fourth call computes $D^{N/2}_{22}$ using D^0_{22}, $D^{N/2}_{12}$, and $D^{N/2}_{21}$ completing the computation of $D^{N/2}$. After the first four recursive calls the shortest path with intermediate vertices in the set 1 to $N/2$ for all pairs of vertices has been computed.

The second set of four recursive calls works in the same way as the first set, although in the reverse order, and completes the computation of D^N. The Proof for the second set of calls is similar to that of the first set. In this way, the recursive implementation of the Floyd-Warshall algorithm correctly computes the all-pairs shortest paths and by induction it is correct for all N. ■

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

Proof: Note that the running time of this algorithm is given by

$$T(N) = 8 * T\left(\frac{N}{2}\right) = \Theta(N^3) \qquad 5$$

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

$$D(N) = 8 * D\left(\frac{N}{2}\right) = \Theta(N^3) \quad (6)$$

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, 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:

$$D(B) = O(B^2) \quad (7)$$

By combining Equation 10 and Equation 11 it can be shown that:

$$D(N) = \frac{N^3}{B^3} * D(B) = O(\frac{N^3}{B}) \quad (8)$$

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 $(i\text{-}1)^{th}$ 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 12 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 $(m\text{-}1)^{th}$ 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 i^{th} and the $(i\text{-}1)^{th}$ level of cache will be reduced by a factor of B_i, where $B_i = O(\sqrt{C_i})$. ■

In [17], 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, the following can be shown:

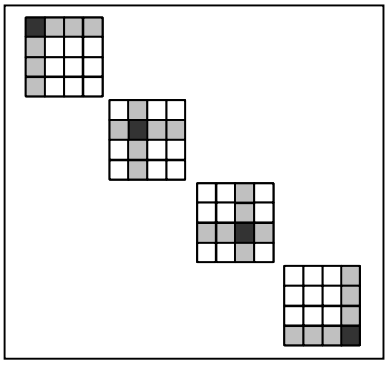

Figure 4: Tiled implementation of FW

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.

From Lemma 3.1 and Theorem 3.2 showing the upper bound on processor-memory traffic of the recursive implementation to be $O(N^3/B)$, where $B^2 = O(C)$, 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 a problem size B is reached, where B^2 is on the order of the cache size, a standard iterative implementation of the Floyd-Warshall algorithm is executed. This improvement varied from one machine to the other (see Section 4) 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 [45]. 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 [34], and outlined here.

- Use the 2:1 rule of thumb from [32] 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 13, where d is the size of one element and C is the adjusted cache size. This minimizes capacity misses.

$$3 * B^2 * d = C \qquad 9$$

3.1.2. A Tiled Implementation for FW

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 [43] using an alternate analysis. A brief description of the algorithm is as follows. Tile the problem into BxB tiles. During the b^{th} block iteration, first update the $(b,b)^{th}$ tile, then the remainder of the b^{th} row and b^{th} column, then the rest of the matrix. Figure 4 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 $(b,b)^{th}$ tile, then the grey tiles, then the white tiles. In this way we satisfy all dependencies from each b^{th} loop to the next as well as all dependencies within each b^{th} loop.

3.1.2.1 Analysis

In [43], 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 [43]. The following analysis is performed for the tiled implementation in the context of the model discussed in Section 1.

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.

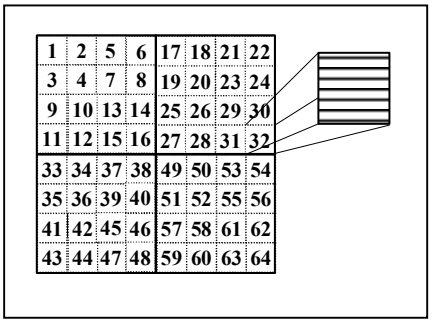

Figure 5: Z-Morton Layout

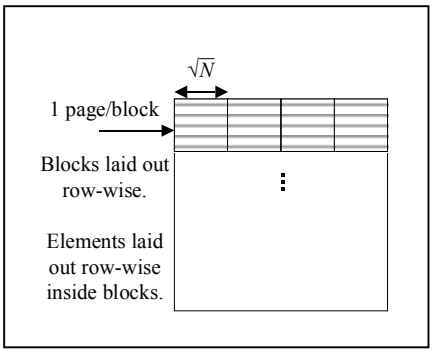

Figure 6: The Block Data Lavout

3.1.2.2 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 (see Figure 6). 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 [43] 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 [43], 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. In the experiment result section, a comparison between row-wise and block layout is given.

3.1.3. Data Layout Issues

It is 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 [7][31][34]. 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 5). See [7] for a more formal definition of the Morton ordering.

In the case of the tiled implementation, the Block Data Layout (BDL) matches the access pattern. Recall from Section 3.1.2.2 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 6). By setting the block size equal to the tile size in the tiled computation, the data layout will exactly match the data access pattern.

3.2. Optimizing the Single-Source Shortest Paths Problem and the Minimum Spanning Tree Problem

Dijkstra's algorithm is designed to solve the single-source shortest paths 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). The

pseudo-code for the algorithm is given in Figure 7. The asymptotically 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.

Prim's algorithm for the Minimum Spanning Tree problem is very similar to Dijkstra's algorithm. 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. As mentioned above, due to the structure of Dijkstra's and Prim's algorithm neither tiling nor recursion can be directly applied.

As mentioned in Section 1, the largest data structure in both Dijkstra's and Prim's algorithms 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.

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.

```
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 7: Pseudo code for Dijkstra's algorithm

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 i^{th} node to the j^{th} 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 [9] or [16] for more details regarding these common graph representations.

Consider a simple combination of these two representations [37]. For each node in the graph, there exists 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 [37].

As mentioned above, Prim's algorithm is very similar to Dijkstra's algorithm. 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. 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.

```
FindMatching(G, M)

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

Figure 8: Pseudo code for augmenting path matching algorithm

```
CacheFriendlyFindMatching(G)

1.  Partition G into g[1], g[1], …, g[p];
2.  For i = 1  to p
3.          m[i] = FindMatching(g[i], ∅);
4.  M = UnionAll(m);
5.  M = FindMatching(G, M);
6.  return M;
```

Figure 9: Pseudo code for cache friendly implementation of the matching algorithm

3.3. Optimizing Matching Algorithm for Bipartite Graph

In this section, the ideas and techniques developed in the previous sections are utilized to optimize another fundamental graph algorithm, namely matching algorithm for bipartite graphs. As discussed earlier, 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. We start with a brief description of the 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. 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 8. A more detailed explanation of this primitive matching algorithm is given in [22].

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. The augmenting path matching algorithm iteratively improves the cardinality of a matching, usually starting from an empty set, until it reaches the maximum cardinality. Our cache-friendly implementation of the matching algorithm is as follows: First, divide the input graph into sub-graphs using a certain graph-partitioning scheme and solve maximum matching problem locally for each sub-graph using the matching algorithm. Next, Union all the obtained matchings for sub-graphs to get a good matching for the complete input graph. Finally, run the algorithm on the complete input graph using the good matching as starting point. Pseudo code for this implementation is presented in Figure 9.

The main performance advantage of our implementation comes from the highly cache efficient first stage. The cache efficiency at the first stage is due to reduced working set size implying increased temporal locality. If the sizes of sub-graphs are chosen appropriately, each of which fits into the cache, it generates minimal processor-memory traffic of O($N + E$) because a single loading of each data element to cache is necessary. In the best case, that is a maximum matching is found at the first stage, our implementation only causes overall O($N + E$) processor-memory traffic as the program ends at the first stage. It is O(N) times improvement over O(N*E), the processor-memory traffic for the straight forward implementations. The size of sub-graph can be one of the tuning factors in our implementation and the heuristic used to select block size presented in the previous section can be used.

Mainly for experimental purposes, we develop and employ a simple linear time two-way partitioning algorithm. 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 matching 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.

Data level-1 cache misses (10^9)

N	Baseline	Recursive
1024	0.806	0.546
2048	6.442	4.362

Data level-2 cache misses (10^6)

N	Baseline	Recursive
1024	0.537	0.280
2048	4.294	2.232

Table 1: FWR Impl. Simulation results

4. Experimental and Simulation Results

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

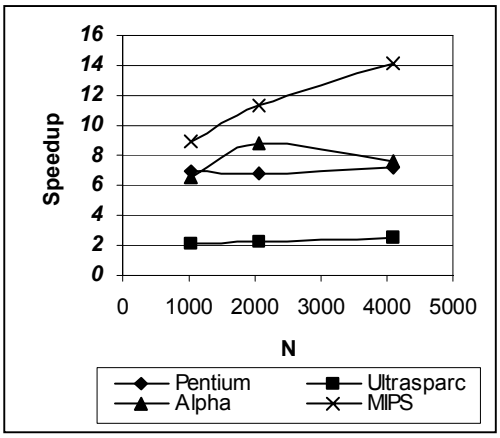

Figure 10: Speedup results for the recursive implementation of the Floyd-Warshall algorithm

Data level-1 cache performance

	Row-wise	BDL
Misses (10^9)	0.312	0.276
Miss Rate	4.82%	4.28%

Data level-2 cache performance

	Row-wise	BDL
Misses (10^6)	91.43	7.45
Miss Rate	29.11%	2.68%

Execution time (sec)

	Row-wise	BDL
SUN	283.72	201.38
P III	124.2	97.62

(N = 2048)

Table 2: Performance Comparison

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.

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

4.1. Results for Floyd-Warshall Algorithm Optimizations

The baseline used for our experiments is an implementation of the iterative Floyd-Warshall algorithm with the usual row major data layout. 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, the recursive implementation of the Floyd-Warshall algorithm was compared 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 10. Differences in performance gains between machines are expected, due to the wide variance in cache parameters and miss penalties.

Recursive Impl. Exe. Time (sec)

N	Morton Layout	Block Data Layout
2048	103.48	111.42
4096	820.45	878.89

Tiled Impl. Exe. Time (sec)

N	Morton Layout	Block Data Layout
2048	99.25	99.39
4096	779.53	780.41

Table 4: Execution time on Pentium III

Recursive Impl. Exe. Time (sec)

N	Morton Layout	Block Data Layout
2048	307.33	311.26
4096	2460.53	2488.88

Tiled Impl. Exe. Time (sec)

N	Morton Layout	Block Data Layout
2048	278.48	271.35
4096	2248.20	2184.09

Table 5: Execution time on UltraSPARC III

Cache misses (10^6)

	Linked-List	Adj. Array
Data level 1	7.04	5.62
Data level 2	3.59	1.82

(Input: 16K nodes, 0.1 density)

Table 6: Simulation results for Dijkstra's algorithm

Table 2 shows the result of comparing the tiled implementation using a row-wise layout and the block size selection used in [43] with the tiled implementation using the block data layout and our block size

selection. Simulation results show that the block size selection used in [43] 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.

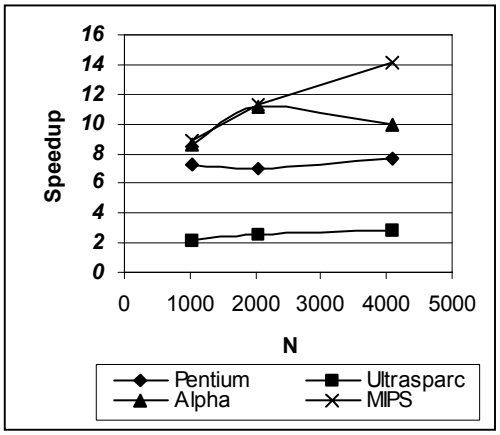

Figure 11: Speedup results for the tiled implementation of the Floyd-Warshall algorithm

A comparison for the tiled implementation using the Block Data Layout with the baseline 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 11).

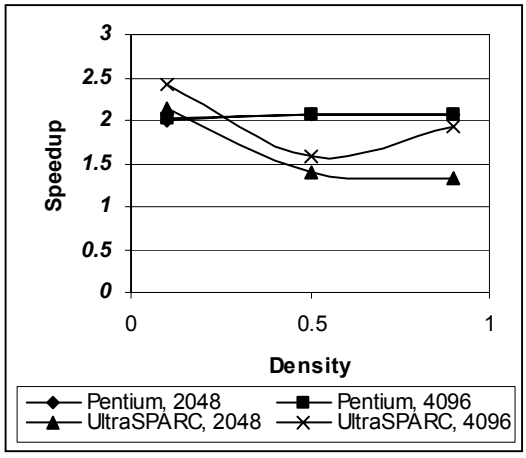

Figure 12: Speedup results for Dijkstra's algorithm

Experiments were also performed with both Z-Morton and BDL 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 data layouts have the final block laid out in contiguous memory locations, they perform equally

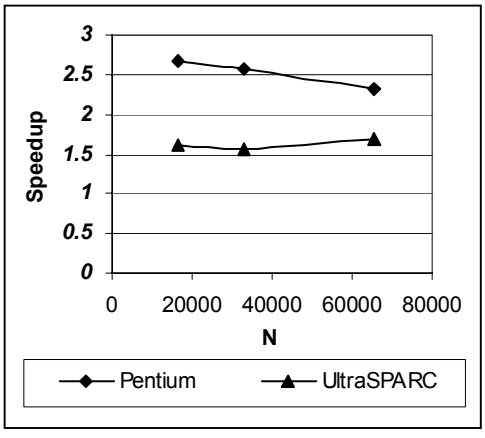

Figure 13: Speedup results for Dijkstra's algorithm

well.

Our experimental results show that the recursive implementation is slightly less efficient than the tiled implementations. This is due to inefficient coding of the recursive implementation. Both the recursive and the tiled implementations require the input matrix to be padded with values set to infinity, to meet certain criteria. The recursive implementation mandates the problem size N to be a product of the block size and a power of 2 so that the input matrix can be recursively divided into half sized sub problems until it reaches the base case. Whereas, in the tiled implementation, the problem size only has to be a multiple of the block size. Thus in some cases, more padding might be required for the recursive implementation, resulting in unnecessary computation and reduced efficiency. Efficient code should pass the padding area without any computation. We believe that if the code is efficient for both implementations, the recursive implementation will run faster. With current implementations we also see many cases when the recursive implementation works faster. For example, on UltraSPARC III if the block size is 32 or 64 and the problem size is 2048, the recursive implementation consistently runs 3% faster than the tiled implementation. Note that the recursive implementation is still preferred due to its auto blocking nature. It automatically adapts itself and is thus less sensitive to the underlying cache architecture. Whereas the process of block size selection according to the underlying cache architecture, is crucial to the tiled implementation.

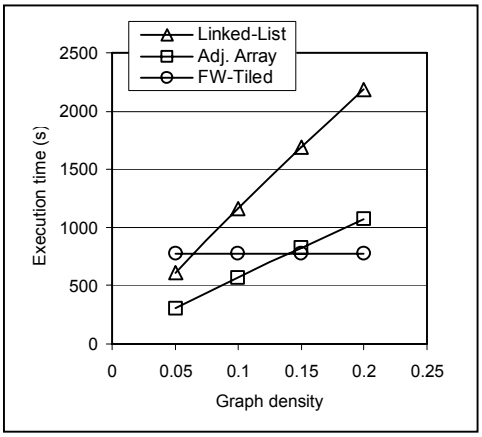

Figure 14: Dijkstra's algorithm vs. best FW on Pentium III, $N = 2048$

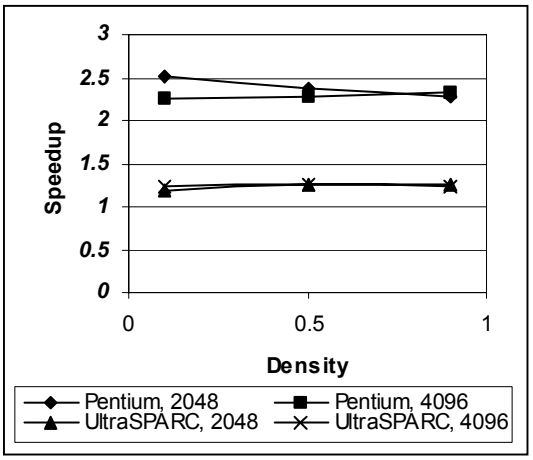

Figure 15: Speedup results for Prim's algorithm

4.2. Results for Dijkstra's Algorithm Optimization

In order to demonstrate the performance improvements using our graph representation, simulations as well as experiments on two different machines, the Pentium III and UltraSPARC III, were performed

for Dijkstra's algorithm. The simulation results 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 the memory hierarchy of these two architectures.

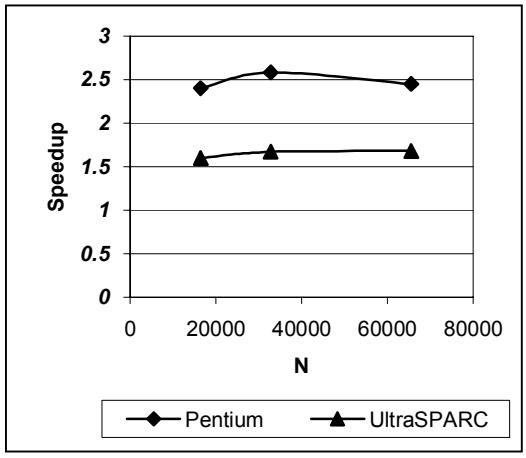

Figure 16: Speedup results for Prim's algorithm

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

Table 7: Simulation results for Prim's algorithm

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 paths 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. Figure 14 shows a comparison of the best Floyd-Warshall algorithm with Dijkstra's algorithm for sparse graphs on the Pentium III.

4.3. Results for Prim's Algorithm Optimization

As mentioned in Section 3.2, the optimizations applicable to Dijkstra's algorithm are also applicable to Prim's algorithm. Figures 15 & 16 show the result of applying the optimization to the graph representation discussed in Section 3.2 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.

The 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 similar to the results shown for Dijkstra's algorithm.

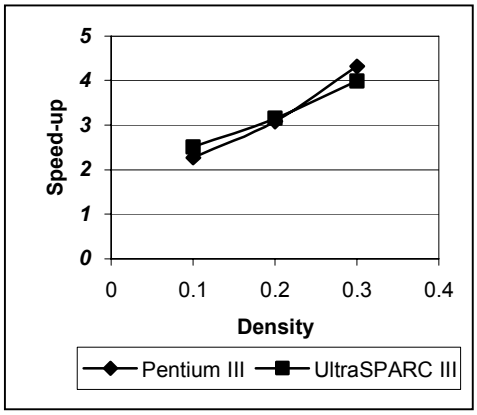

Figure 17: Speed-up vs. density results for matching algorithm

DL1 Cache Performance

	Baseline	Optimized
Accesses (10^6)	853	578
Misses (10^6)	127	32
Miss Rate	14.86%	5.56%

(Input: 8K nodes, 0.1 density)

Table 8: Simulation results for matching algorithm

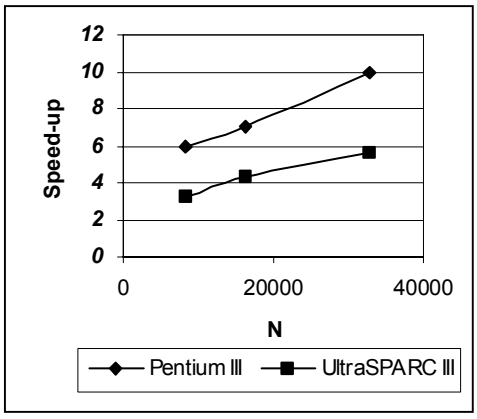

Figure 18: Best case speed-up results for graph matching

4.4. Results for Matching Algorithm Optimization

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.

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

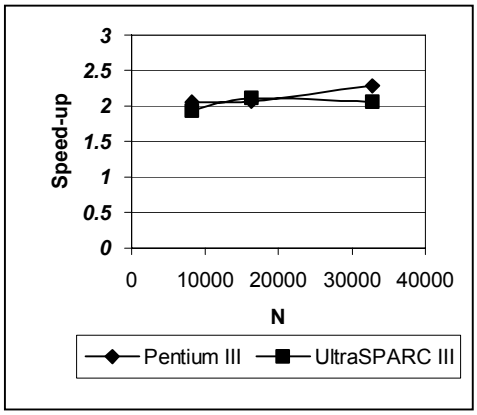

Figure 19: Average speed-up results for matching algorithm

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 dependency 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 dependency can be seen in Figure 17, 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 18. 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 19. 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. Simulations were done 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.

5. Conclusion

In this paper, we discussed using the techniques of recursion, and data layout optimization to show improved cache performance both analytically and experimentally for four fundamental 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 6x improvement in real execution time on three different architectures as well as analytically showed 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 a technique which effectively reduces the working set size, to the matching algorithm and showed 2x to 3x improvement in real execution time for randomly generated graphs and up to 10x improvement for graphs well suited to our partitioning algorithm.

In this work, we targeted four fundamental algorithms as they are of major interest to the community in terms of their applicability to a broad set of applications. Our techniques can also be beneficial to improve the performance of other graph algorithms. For example, the Bellman-Ford algorithm [9] for the shortest-paths problem visits every neighbor of a node once the node is labeled. Performance improvement can be achieved by applying our data layout optimization discussed in Section 3.2 to the Bellman-Ford algorithm, as the layout will match the data access pattern. For the same reason as the Bellman-Ford algorithm, graph traversals such as depth and breadth first search [9] and algorithms built on top of those, such as finding strongly connected components [9], can also benefit from our data layout optimization. Another example is the Ford-Fulkerson algorithm [9] for the maximum-flow problem. This algorithm shares the same structure with the matching algorithm. It iteratively finds an augmenting path; thus the optimization for the matching algorithm discussed in Section 3.3 can be directly applied to it.

As pointed out in the Introduction, our pursuit of data locality also benefits parallelization. Our sequential FW implementations and matching implementation can easily be transformed into parallel code. Since 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.

Acknowledgements

This work is part of the Algorithms for Data IntensiVe Applications on Intelligent and Smart MemORies (ADVISOR) Project [1] 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. We would like to thank Bo Hong and Gokul Govindu for helpful discussions and for their editorial assistance.

6. References

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

[2] A. Aho, J. E. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[3] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[4] M. Brenner. Multiagent Planning with Partially Ordered Temporal Plans, In *Proc. of IJCAI*, 2003.

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

[6] J. Carter, W. Hsieh, L. Stoller, M. Swanson, L. Zhang, and S. McKee. Impulse: Memory System Support for Scientific Applications. *Journal of Scientific Programming*, Vol. 7, No. 3-4, 1999.

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

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

[9] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. MIT Press, MA, 1990.

[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. Frens and D. 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, NY, October 1999.

[13] R. Gallagher and D. Bertsekas. *Data Networks*. Prentice Hall, 1987.

[14] S. Gerez. *Algorithms for VLSI Design Automation*. Wiley, 1998.

[15] 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.

[16] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithm,* Computer Science Press, MD, 1978.

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

[18] 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.

[19] 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.

[20] M. Lam, E. Rothberg, and M. 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, CA, April 1991.

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

[22] E. Lawler. *Combinatorial Optimization: Networks and Matroids*, Holt, Rhinehart, and Winston, NY, 1976.

[23] S. Meguerdichian, F. Koushanfar, G. Qu, and M. Potkonjak. Exposure in Wireless Ad Hoc Sensor Networks. In *Proc. of International Conference on Mobile Computing and Networking*, July 2001.

[24] M. Mohri and M. Riley. A Weight Pushing Algorithm for Large Vocabulary Speech Recognition. In *Proc. of the 7th European Conference on Speech Communication and Technology*, September 2001.

[25] R. Murphy and P. M. Kogge. The Characterization of Data Intensive Memory Workloads on Distributed PIM Systems, In *Proc. of Intelligent Memory Systems Workshop, ASPLOS-IX 2000*, November 2000.

[26] A. Nakaya, S. Goto, and M. Kanehisa. Extraction of Correlated Gene Clusters by Multiple Graph Comparison. *Genome Informatics*, 12, 2001.

[27] C. Papadimitriou and M. Sideri. On the Floyd-Warshall Algorithm for Logic Programs. *The Journal of Logic Programming*, Vol. 41, 1999.

[28] J. Park, M. Penner, and V. K. Prasanna. Optimizing Graph Algorithms for Improved Cache Performance. Technical Report USC-CENG 03-03, Department of Electrical Engineering, USC, November 2003.

[29] N. Park, B. Hong, and V. Prasanna. Tiling, Block Data Layout, and Memory Hierarchy Performance. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 14, No. 7, 2003.

[30] N. Park, B. Hong, and V. Prasanna. Analysis of Memory Hierarchy Performance of Block Data Layout. In *Proc. of International Conference on Parallel Processing (ICPP)*, August 2002.

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

[32] N. Park and V. Prasanna. Cache Conscious Walsh-Hadamard Transform, In *Proc. of International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, May 2001.

[33] D. Patterson and J. Hennessy. *Computer Architecture A Quantitative Approach*. 2nd Ed., Morgan Kaufmann, San Francisco, CA, 1996.

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

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

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

[37] S. Sahni. *Data Structures, Algorithms, and Applications in Java*. McGraw Hill, NY, 2000.

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

[39] S. Sarawagi, R. Agrawal, and A. Gupta. On Computing the Data Cube. Research Report 10026, IBM Almaden Research Center, San Jose, CA, 1996.

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

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

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

[43] 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.

[44] D. Weikle, S. McKee, and W. Wulf. Caches as Filters: A New Approach to Cache Analysis. In *Proc. of Grace Murray Hopper Conference*, September 2000.

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

[46] M. Yannakakis. Graph Theoretic Methods in Database Theory. In *Proc. of the ACM conference on Principles of Database Systems*, 1990.

Appendix A: The Correctness of Recursive FW

1 The Recursive Floyd-Warshall Algorithm

Traditional Iterative Algorithm:

The traditional iterative algorithm is represented as `FWI`, where given an adjacency matrix D, the corresponding all-pairs shortest path is calculated by calling `FWI(D, D, D)`.

```
FWI(A, B, C) { // A, B, C are of size N*N
  for k = 1 to N
    for i = 1 to N
      for j= 1 to N
        aij = min(aij, bik + ckj);
}
```

The Recursive Algorithm:

```
FWR(A, B, C) { // A, B, C are of size N*N
  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 {
    for k = 1 to N
      for i = 1 to N
        for j= 1 to N
          aij = min(aij, bik + ckj);
  }
}
```

In the recursive implementation, given an adjacency matrix D, the corresponding all-pairs shortest path is calculated by calling `FWR(D, D, D)`.

Clarifications: For FWI and the base case of FWR,

- if `A = B`, then the following operations will be carried out:

```
for k = 1 to N
  for i = 1 to N
    for j= 1 to N
      aij = min(aij, aik + ckj);
```

- if `A = C`, then the following operations will be carried out:

```
for k = 1 to N
  for i = 1 to N
    for j= 1 to N
      aij = min(aij, bik + akj);
```

- if `A = B = C`, then the following operations will be carried out:

```
for k = 1 to N
  for i = 1 to N
    for j= 1 to N
      aij = min(aij, aik + akj);
```

Notation 1: FWR(A, B, C) is used to denote the resulting matrix A after applying FWR to matrices A, B, and C. FWI(A, B, C) is used to denote the resulting matrix A after applying FWI to matrices A, B, and C.

Notation 2: Given two $n \times m$ matrices A and A'. If A and A' reside in memory simultaneously, $A = A'(A \neq A')$ is used to denote that A and A' are allocated to the same(different) memory location. If A and A' are not in the memory at the same time, $A = A'(A \neq A')$ is used to denote that $a_{ij} = a'_{ij}(a_{ij} \neq a'_{ij})$ for $1 \leq i \leq n$ and $1 \leq j \leq m$. Given the context, the reader can easily differentiate between these two cases.

Definition: Given two $n \times m$ matrices A and A'. $A \leq A'$ iff $a_{ij} \leq a'_{ij}$ for $1 \leq i \leq n$ and $1 \leq j \leq m$.

Claim 1: When computing the equation $d^k_{ij} = \min(d^{k-1}_{ij}, d^{k'_1}_{ik} + d^{k'_2}_{kj})$, it is sufficient for the correctness of the Floyd-Warshall algorithm that $k'_1 \geq k-1$ and $k'_2 \geq k-1$.

Lemma 1: $FWR(A, B, C) = FWI(A, B, C)$

Lemma 2: For every operation $a^k_{ij} = \min(a^{k-1}_{ij}, b_{ik} + a^{k-1}_{kj})$ in $FWI(A, B, A)$, there is a corresponding operation $a^k_{ij} = \min(a^{k-1}_{ij}, b_{ik} + a^{k'}_{kj})$ in $FWR(A, B, A)$ with $k' \geq k-1$

Lemma 3: For every operation $a^k_{ij} = \min(a^{k-1}_{ij}, a^{k-1}_{ik} + b_{kj})$ in $FWI(A, A, B)$, there is a corresponding operation $a^k_{ij} = \min(a^{k-1}_{ij}, a^{k'}_{ik} + b_{kj})$ in $FWR(A, A, B)$ with $k' \geq k-1$

Theorem 1: $FWR(A, A, A) = FWI(A, A, A)$

2 Proof of Theorem 1

The proof is by induction. N is assumed to be equal to 2^n for simplicity. Proofs for Lemmas 1, 2, and 3 will be presented after the proof of Theorem 1.

Base case: When $n = 1$, FWR(A,A,A) and FWI(A,A,A) perform exactly the same operations, therefore

$$FWR(A, A, A) = FWI(A, A, A)$$

for $n = 1$.

Induction Step: Assuming

$$FWR(A, A, A) = FWI(A, A, A)$$

for $n = m - 1$, it has to be proven true for $n = m$.

1. Comparing the following three code segments:

```
FWR(A11, A11, A11);     FWI(A11, A11, A11);     for k = 1 to N/2
FWR(A12, A11, A12);     FWI(A12, A11, A12);       for i = 1 to N
FWR(A21, A21, A11);     FWI(A21, A21, A11);         for j = 1 to N
FWR(A22, A21, A12);     FWI(A22, A21, A12);           aij = min(aij, aik+akj);
  Code segment 1          Code segment 2              Code segment 3
```

   Code segment 1 contains the first four recursive steps within FWR(A, A, A). Code segment 3 is the first half of FWI(A, A, A) where index k steps from 1 to $N/2$. Let $\hat{A}_1$, $\hat{A}_2$, and $\hat{A}_3$ denote the resulting matrices obtained after applying code segments 1, 2, and 3, respectively.

   - **Comparing $\hat{A}_2$ and $\hat{A}_3$**
     Code segment 2 is re-written as follows. Subscript I is used to denote the matrices associated with FWI. Data dependencies are indicated by super-scripts.

$$\begin{aligned} A11_I^1 &= FWI(A11_I^0, A11_I^0, A11_I^0); \\ A12_I^1 &= FWI(A12_I^0, A11_I^1, A12_I^0); \\ A21_I^1 &= FWI(A21_I^0, A21_I^0, A11_I^1); \\ A22_I^1 &= FWI(A22_I^0, A21_I^1, A12_I^1); \end{aligned}$$

     Code segment 2 is then expanded as follows:

```
1 for k = 1 to N/2
2   for i = 1 to N/2
3     for j = 1 to N/2
```

```
 4       aij = min(aij, aik + akj);
 5 for k = 1 to N/2
 6   for i = 1 to N/2
 7     for j = N/2+1 to N
 8       aij = min(aij, aik + akj);
 9 for k = 1 to N/2
10   for i = N/2+1 to N
11     for j = 1 to N/2
12       aij = min(aij, aik + akj);
13 for k = 1 to N/2
14   for i = N/2+1 to N
15     for j = N/2+1 to N
16       aij = min(aij, aik + akj);
       Code segment 2
```

Code segment 3 is re-written as below:

```
 1 for k = 1 to N/2 {
 2   for i = 1 to N/2 {
 3     for j = 1 to N/2 {
 4       aij = min(aij, aik + akj);
 5     }
 6     for j = N/2+1 to N {
 7       aij = min(aij, aik + akj);
 8     }
 9   }
10   for i = N/2+1 to N {
11     for j = 1 to N/2 {
12       aij = min(aij, aik + akj);
13     }
14     for j = N/2+1 to N {
15        aij = min(aij, aik + akj);
16     }
17   }
19 }
       Code segment 3
```

$\hat{A}_3 11$, $\hat{A}_3 12$, $\hat{A}_3 21$, and $\hat{A}_3 22$ are used to denote the northwest, northeast, southwest, and southeast quadrants of $\hat{A}_3$.

First, $A11_I^1$ and $\hat{A}_3 11$ are compared. In code segment 3, the nested loop contained in lines 3-5

and the nested loop contained in lines 6-8 act on different data. Obviously, the loop contained in lines 1-4 of code segment 2 generates the same results as the loop contained in lines 1-5 of code segment 3. This means that

$$A11_I^1 = \hat{A}_3 11 \tag{1}$$

Next $A12_I^1$ and $\hat{A}_3 12$ have to be compared. From code segments 2 and 3, it can be seen that the calculation of $A12_I^1$ and $\hat{A}_3 12$ is dependent on the calculation of $A11_I^1$ and $\hat{A}_3 11$, respectively. The following comparison shows that the dependencies are different for code segments 2 and 3.

Consider the computation of $\hat{A}_3 12$ in the loop contained in lines 1-8 of code segment 3. As can be seen from lines 3-5 in this loop, intermediate values of $\hat{A}_3 11$ are being used for computing $\hat{A}_3 12$. The computation of $A12_I^1$ is done in the loop contained in lines 5-8 of code segment 2 and uses the final values of $A11_I^1$ computed in lines 1-4.

Considering the above discussion and the fact that the final values of $A11_I^1$ are less than or equal to the intermediate values of $\hat{A}_3 11$ due to the nature of the $\min()$ operation, the following equation is obtained

$$A12_I^1 \leq \hat{A}_3 12 \tag{2}$$

On the other hand, $\hat{A}_3 12$ is the *shortest* path from vertices $\{1, ..., N/2\}$ to $\{N/2+1, ..., N\}$, with the intermediate vertices in $\{1, ..., N/2\}$. And $A12_I^1$ is just *some* path from vertices $\{1, ..., N/2\}$ to $\{N/2 + 1, ..., N\}$, with the intermediate vertices in $\{1, ..., N/2\}$. Hence we have

$$A12_I^1 \geq \hat{A}_3 12 \tag{3}$$

From (2) and (3), the following equation is obtained

$$A12_I^1 = \hat{A}_3 12 \tag{4}$$

Following similar analysis, the following equations can also be derived

$$A21_I^1 = \hat{A}_3 21 \tag{5}$$

and

$$A22_I^1 = \hat{A}_3 22 \tag{6}$$

Combining (1), (4), (5), and (6), the following equation is derived

$$\hat{A}_2 = \hat{A}_3 \tag{7}$$

- **Comparing $\hat{A_1}$ and $\hat{A_2}$**

  Code segments 1 and 2 are re-written as below and subscripts R and I are used to denote the matrices associated with FWR and FWI. Data dependencies are indicated by super-scripts. Note that $A11_R^0$ and $A11_R^1$ refer to the same sub-matrix $A11_R$. Similiarly $A11_R^0 = A11_I^0$, $A12_R^0 = A12_I^0$, $A21_R^0 = A21_I^0$, and $A22_R^0 = A22_I^0$.

$$\begin{aligned} A11_R^1 &= FWR(A11_R^0, A11_R^0, A11_R^0); & A11_I^1 &= FWI(A11_I^0, A11_I^0, A11_I^0); \\ A12_R^1 &= FWR(A12_R^0, A11_R^1, A12_R^0); & A12_I^1 &= FWI(A12_I^0, A11_I^1, A12_I^0); \\ A21_R^1 &= FWR(A21_R^0, A21_R^0, A11_R^1); & A21_I^1 &= FWI(A21_I^0, A21_I^0, A11_I^1); \\ A22_R^1 &= FWR(A22_R^0, A21_R^1, A12_R^1); & A22_I^1 &= FWI(A22_I^0, A21_I^1, A12_I^1); \\ & Code\ segment\ 1 & & Code\ segment\ 2 \end{aligned}$$

  Given the induction assumption $FWR(A, A, A) = FWI(A, A, A)$ for $N = 2^{m-1}$, the following equation is obtained:

$$A11_R^1 = A11_I^1 \tag{8}$$

  The second function call in code segment 1, $FWR(A12_R^0, A11_R^1, A12_R^0)$, is in the form of $FWR(A, B, A)$. According to Lemma 2, $FWR(A12_R^0, A11_R^1, A12_R^0)$ performs the same operations as $FWI(A12_I^0, A11_I^1, A12_I^0)$, with $k' \geq k-1$ for each of the operations. Similar to Claim 1's proof, it can be shown that $FWR(A12_R^0, A11_R^1, A12_R^0) = FWI(A12_I^0, A11_I^1, A12_I^0)$.
  Thus the following equation is obtained

$$A12_R^1 = A12_I^1 \tag{9}$$

  Similarly, it can be proven that

$$A21_R^1 = A21_I^1 \tag{10}$$

  using Lemma 3, and

$$A22_R^1 = A22_I^1 \tag{11}$$

  using Lemma 1. Combining (8), (9), (10), and (11), the following equation is obtained:

$$\hat{A_1} = \hat{A_2} \tag{12}$$

- **Comparing $\hat{A_1}$ and $\hat{A_3}$**

  Combining (12) and (7), the following equation is obtained:

$$\hat{A_1} = \hat{A_3} \tag{13}$$

2. Comparing the following three code segments:

```
FWR(A22, A22, A22);     FWI(A22, A22, A22);     for k = N/2+1 to N
FWR(A21, A22, A21);     FWI(A21, A22, A21);       for i = 1 to N
FWR(A12, A12, A22);     FWI(A12, A12, A22);         for j = 1 to N
FWR(A11, A12, A21);     FWI(A11, A12, A21);           aij = min(aij, aik+akj);
  Code segment 4          Code segment 5              Code segment 6
```

   Code segment 1 contains the last four recursive steps within FWR(A, A, A) and code segment 3 is the second half of FWI(A, A, A) where the index k steps from $N/2+1$ to N. In FWR, code segment 4 uses the data computed by code segment 1, i.e. $\hat{A}_1$. In FWI, code segment 6 uses the data computed by code segment 3, i.e. $\hat{A}_3$. Let $\hat{A}_4$, $\hat{A}_5$, and $\hat{A}_6$ denote the resulting matrices obtained after applying code segments 4, 5, and 6 respectively.

   Since $\hat{A}_1 = \hat{A}_3$ (Eq.(13)), code segments 4, 5, and 6 use data from the *same* matrix. Following the same procedure used for comparing code segments 1, 2, and 3, the following equation is derived

$$\hat{A}_4 = \hat{A}_6 \tag{14}$$

3. Eq.(14) is nothing but the following induction:

$$FWR(A, A, A) = FWI(A, A, A) \text{ for } N = 2^m \tag{15}$$

This finishes the proof for Theorem 1. ∎

3 Proof of Lemma 1

The proof is by induction. N is assumed to be equal to 2^n for simplicity.

Base case: When $n = 1$, FWR and FWI perform exactly the same operations, therefore

$$FWR(A, B, C) = FWI(A, B, C)$$

for $n = 1$.

Induction Step: Assuming

$$FWR(A, B, C) = FWI(A, B, C)$$

is true for $n = m - 1$, it has to be proven true for $n = m$.

Since matrices B and C are not updated during the computation, the FWI code can be reordered as follows without affecting the final result:

```
FWI(A11, B11, C11);
FWI(A12, B11, C12);
FWI(A21, B21, C11);
FWI(A22, B21, C12);
FWI(A22, B22, C22);
FWI(A21, B22, C21);
FWI(A12, B12, C22);
FWI(A11, B12, C21);
```

Given the induction assumption, the following equations can be obtained

$$FWR(A11, B11, C11) = FWI(A11, B11, C11)$$

...

$$FWR(A11, B12, C21) = FWI(A11, B12, C21)$$

Hence, obviously,

$$FWR(A, B, C) = FWI(A, B, C)$$

∎

4 Proof of Lemma 2 and 3

The proofs for Lemmas 2 and 3 are similar. Hence, the proof for Lemma 2 only is shown below. The proof is by induction. N is assumed to be equal to 2^n for simplicity.

Base case: When $n = 1$, FWR(A,B,A) and FWI(A,B,A) perform exactly the same operations. In other words, for every operation $a_{ij}^k = \min(a_{ij}^{k-1}, b_{ik} + a_{kj}^{k-1})$ in $FWI(A, B, A)$, there is a corresponding operation $a_{ij}^k = \min(a_{ij}^{k-1}, b_{ik} + a_{kj}^{k'})$ in $FWR(A, B, A)$ with $k' \geq k - 1$.

Induction Step: Assuming that when $n = m - 1$, for every operation $a_{ij}^k = \min(a_{ij}^{k-1}, b_{ik} + a_{kj}^{k-1})$ in $FWI(A, B, A)$, there is a corresponding operation $a_{ij}^k = \min(a_{ij}^{k-1}, b_{ik} + a_{kj}^{k'})$ in $FWR(A, B, A)$ with $k' \geq k - 1$, it has to be proven true for $n = m$.

To simplify the presentation, 'operation' will be used to refer to $a_{ij}^k = \min(a_{ij}^{k-1}, b_{ik} + a_{kj}^{k-1})$ or $a_{ij}^k = \min(a_{ij}^{k-1}, b_{ik} + a_{kj}^{k'})$ in the following discussion.

1. Comparing the following three code segments:

```
FWR(A11, B11, A11);     FWI(A11, B11, A11);     for k = 1 to N/2
FWR(A12, B11, A12);     FWI(A12, B11, A12);       for i = 1 to N
FWR(A21, B21, A11);     FWI(A21, B21, A11);         for j = 1 to N
FWR(A22, B21, A12);     FWI(A22, B21, A12);           aij = min(aij, bik+akj);
  Code segment 7          Code segment 8              Code segment 9
```

Code segment 7 contains the first four recursive steps within FWR(A, B, A). Code segment 9 is the first half of FWI(A, B, A) where the index k steps from 1 to $N/2$. Let $\hat{A_7}$, $\hat{A_8}$, and $\hat{A_9}$ denote the resulting matrices obtained after applying code segments 7, 8, and 9 respectively.

- **Comparing code segments 8 and 9**
  Code segment 8 is rewritten as follows and subscript I is used to denote the matrices associated with FWI. Data dependencies are indicated by super-scripts.

$$A11_I^1 = FWI(A11_I^0, B11, A11_I^0);$$
$$A12_I^1 = FWI(A12_I^0, B11, A12_I^0);$$
$$A21_I^1 = FWI(A21_I^0, B21, A11_I^1);$$
$$A22_I^1 = FWI(A22_I^0, B21, A12_I^1);$$

  Code segment 8 is expanded as follows:

```
 1 for k = 1 to N/2
 2   for i = 1 to N/2
 3     for j = 1 to N/2
 4       aij = min(aij, bik + akj);
 5 for k = 1 to N/2
 6   for i = 1 to N/2
 7     for j = N/2+1 to N
 8       aij = min(aij, bik + akj);
 9 for k = 1 to N/2
10   for i = N/2+1 to N
11     for j = 1 to N/2
12       aij = min(aij, bik + akj);
13 for k = 1 to N/2
14   for i = N/2+1 to N
15     for j = N/2+1 to N
16       aij = min(aij, bik + akj);
       Code segment 8
```

  Code segment 9 is re-written as below:

```
1 for k = 1 to N/2 {
2   for i = 1 to N/2 {
3     for j = 1 to N/2 {
4       aij = min(aij, bik + akj);
5     }
6     for j = N/2+1 to N {
```

```
 7        aij = min(aij, bik + akj);
 8      }
 9    }
10    for i = N/2+1 to N {
11      for j = 1 to N/2 {
12        aij = min(aij, bik + akj);
13      }
14      for j = N/2+1 to N {
15         aij = min(aij, bik + akj);
16      }
17    }
19 }
        Code segment 9
```

$\hat{A}_9 11$, $\hat{A}_9 12$, $\hat{A}_9 21$, and $\hat{A}_9 22$ are used to denote the northwest,northeast, southwest, and southeast quadrants of $\hat{A}_9$.

First, the computation of $A11_I^1$ and $\hat{A}_9 11$ are compared. In code segment 9, the nested loop contained in lines 6-8 and the nested loop contained in lines 6-8 act on different data. Obviously, the loop contained in lines 1-4 of code segment 8 performs the same operations as the loop in lines 1-5 of code segment 9.

A similiar comparison can be made between the computation of $A12_I^1$ and the computation of $\hat{A}_9 12$. Comparing the computation of $A21_I^1$ and computation of $\hat{A}_9 21$, it is obvious that, when calculating the southwest quadrant, for each operation in code segment 9, there is a corresponding operation in code segment 8. From Code segment 8 and 9, it can be seen that the calculation of $A21_I^1$ and $\hat{A}_9 21$ is dependent on the calculation of $A11_I^1$ and $\hat{A}_9 11$ respectively. The following comparison shows that the dependencies are different. The computation of $\hat{A}_9 21$ in code segment 9 is done in the nested loop in lines 10-13 using the intermediate values of $\hat{A}_9 11$ obtained from the nested loop in lines 3-5. The computation of $A21_I^1$ in code segment 8 is done in the loop contained in lines 9-12. However, code segment 2 is actually calculating $A21_I^1$ using the final values of $A11_I^1$ obtained in lines 1-4. Hence, for the computation of $A21_I^1$, for each operation in code segment 8, $k' = N/2$. Thus while calculating the southwest quadrant, for each operation in code segment 9, there is a corresponding operation in code segment 8 with $k' \geq k - 1$. A similiar approach can be used for the southeast quadrant.

- **Comparing code segments 7 and 8**

  Code segment 7 and 8 can be rewritten as below. Subscripts R and I are used to denote the matrices associated with FWR and FWI. Data dependencies are indicated by super-scripts. For example, in code segment 7, $A11_R^1$ in line 2 indicates that it has been generated by the first line.

Note that $A11_R^0$ and $A11_R^1$ refer to the same sub-matrix $A11_R$.

$$A11_R^1 = FWR(A11_R^0, B11, A11_R^0); \qquad A11_I^1 = FWI(A11_I^0, B11, A11_I^0);$$
$$A12_R^1 = FWR(A12_R^0, B11, A12_R^0); \qquad A12_I^1 = FWI(A12_I^0, B11, A12_I^0);$$
$$A21_R^1 = FWR(A21_R^0, B21, A11_R^1); \qquad A21_I^1 = FWI(A21_I^0, B21, A11_I^1);$$
$$A22_R^1 = FWR(A22_R^0, B21, A12_R^1); \qquad A22_I^1 = FWI(A22_I^0, B21, A12_I^1);$$
$$Code\ segment\ 7 \qquad\qquad Code\ segment\ 8$$

The first two function calls in code segment 7 are of the form $FWR(A, B, A)$. Given the induction assumption it can be concluded that, while computing the first two quadrants, for every operation in code segment 8, there is a corresponding operation in code segment 7 with $k' \geq k - 1$.

The last two function calls in Code segment 7 are of the form $FWR(A, B, C)$. According to Lemma 1, $FWR(A, B, C)$ performs the same operations as $FWI(A, B, C)$. Hence, it can be concluded that while computing the last two quadrants, for every operation in code segment 8, there is a corresponding operation in code segment 7 with $k' = k - 1$.

- **Comparing code segments 7 and 9**
  From the above discussions it is obvious that for every operation in code segment 9, there is a corresponding operation in code segment 7 with $k' \geq k - 1$.

2. Comparing the following three code segments:

```
FWR(A22, B22, A22);     FWI(A22, B22, A22);     for k = N/2+1 to N
FWR(A21, B22, A21);     FWI(A21, B22, A21);       for i = 1 to N
FWR(A12, B12, A22);     FWI(A12, B12, A22);         for j = 1 to N
FWR(A11, B12, A21);     FWI(A11, B12, A21);           aij = min(aij, bik+akj);
  Code segment 10         Code segment 11           Code segment 12
```

Code segment 10 consists of the last four recursive steps within FWR(A, B, A). Code segment 12 is the second half of FWI(A, B, A) where the index k steps from $N/2 + 1$ to N. In FWR, $\hat{A}_7$ computed in code segment 7 is used by code segment 10. In FWR, $\hat{A}_9$ computed in code segment 9 is used by code segment 12. Following an analysis similar to one used for comparing code segments 7 and 9, it cab be shown that for each operation in code segment 12, there is a corresponding operation in code segment 10 with $k' \geq k - 1$.

3. Code segments 7 and 10 together are $FWR(A, B, A)$ and code segmentc 9 and 12 are $FWI(A, B, A)$. From the above discussions, it can be concluded that when $n = m$, for every operation in $FWI(A, B, A)$, there is a corresponding operation in $FWR(A, B, A)$ with $k' \geq k - 1$. This completes the induction step.

∎