Title

Optimizing graph algorithms for improved cache performance

Permalink
https://escholarship.org/uc/item/9v89p5wv

Journal
IEEE Transactions on Parallel and Distributed Systems, 15(9)

ISSN
1045-9219

Authors
Park, Joon-Sang
Penner, M
Prasanna, V K

Publication Date
2004-09-01

Peer reviewed

Optimizing Graph Algorithms for Improved Cache Performance

Joon-Sang Park, *Student Member*, *IEEE*, Michael Penner, and Viktor K. Prasanna, *Fellow*, *IEEE*

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 six times the 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. Second, we address Dijkstra's algorithm for the single-source shortest paths problem and Prim's algorithm for minimum spanning tree problem. For these algorithms, we demonstrate up to two times the 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 two to three times in real execution time by using the technique of making the algorithm initially work on subproblems to generate a suboptimal solution and, then, solving the whole problem using the suboptimal solution as a starting point. Experimental results are shown for the Pentium III, UltraSPARC III, Alpha 21264, and MIPS R12000 machines.

Index Terms—Cache-friendly algorithms, cache-oblivious algorithms, graph algorithms, shortest path, minimum spanning trees, graph matching, data layout optimizations, algorithm performance.

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 [25]. 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 [24], [33], [36]. A number of groups are attempting to improve performance by performing computations in memory [4], [19]. Other groups are attacking the problem in software; either in the compiler through reordering instructions and prefetching [15], [16], [28], or through complex data layouts to improve cache performance [6], [8], [13].

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 [32]. 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 [20]. 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 [37], artificial intelligence for multiagent planning [2], and VLSI/CAD [12]. In the Open Shortest Path First (OSPF) protocol [11], 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 [31]. 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.

- *J.-S. Park is with the Computer Science Department, Univeristy of California, Los Angeles, 4732 Boelter Hall, Los Angeles, CA 90095-1596. E-mail: jspark@cs.ucla.edu.*
- *M. Penner is with the Department of Electrical Engineering, University of Southern California, 3740 McClintock Ave. EEB 205, Los Angeles, CA 90089. E-mail: mipenner@alumni.usc.edu.*
- *V.K. Prasanna is with the Department of Electrical Engineering, University of Southern California, 3740 McClintock Ave. EEB 200C, Los Angeles, CA 90089. E-mail: prasanna@usc.edu.*

Manuscript received 6 June 2002; revised 21 April 2003; accepted 18 Dec. 2003.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 116721.

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 [16]. 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 $N \times N$ 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 [30]. 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 6 × 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 2 × 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 subproblems to generate a suboptimal solution and, then, solving the whole problem using the suboptimal solution as a starting point. We show performance improvements in real execution time, in the range of two to three times depending on the density of the graph. Along with the experimental results, we also present SimpleScalar simulation results to support our claim that the performance improvement we achieve in real execution time comes mainly from improved cache performance.

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 area of cache optimization. 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 matching algorithm 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 et al. in [35] and Sen and Chatterjee in [32]. XOR-based data layouts to eliminate cache misses have been explored by Gonzalez et al. in [13]. Data layouts for improving cache performance of embedded processor applications have been explored in [8].

A number of papers have discussed the optimization of specific dense linear algebra problems with respect to cache performance. Whaley and Dongarra discuss optimizing the widely used Basic Linear Algebra Subroutines (BLAS) in [36]. Chatterjee et al. discuss layout optimizations for a suite of dense matrix kernels in [5]. Park et al. discuss dynamic data remapping to improve cache performance for the DFT in [24]. 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 [10], 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 percent and two times the 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 [28]. Optimizing blocked algorithms has been extensively studied [16]. 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 [6]. 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 [17]. However, the predictions they made were for an isolated heap, and the model they used was the *hold model*, in which the heap is static for the majority of operations. In our work, we consider Dijkstra's algorithm and Prim's algorithm in which the heap is very dynamic. In both Dijkstra's algorithm and Prim's algorithm O(N) Extract-Mins are performed and O(E) Updates are performed. Finally, in [30], Sanders discusses a highly optimized heap with respect to cache performance. He shows significant performance improvement using his *sequential heap*. The sequential heap does support Insert and Delete-min very efficiently; however, the Update operation is not supported.

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

In [34], Venkataraman et al. present a tiled implementation of the Floyd-Warshall algorithm that is essentially the same as the tiled implementation shown in this paper. In this paper, we consider a wider range of architectures and also analyze the cache performance with respect to processor memory traffic. We also consider data layouts to avoid conflict misses in the cache, which is not discussed in [34]. Due to the fact that we use a blocked data layout, we are able to relax the constraint that the blocking factor should be a multiple of the number of elements that fit into a cache line. This allows us to use a larger block size and show more speedup. In [34], they derive an upper bound on achievable speed-up of 2 for state-of-the-art architectures. Our optimizations lead to more than six times the 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 decrease 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). 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), reducing the working set size to improve 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.

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. 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.

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 $NN \times N$ matrices, D^k, defined as follows: $D^k = (d_{ij}^k)$, where d_{ij}^k 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

```
Floyd-Warshall($W$)

// Let $N$ be the problem size of $W$
$D^0 \leftarrow W$
for $k \leftarrow 1$ to $N$
    for $i \leftarrow 1$ to $N$
        for $j \leftarrow 1$ to $N$
            $d^k_{ij} \leftarrow \min(d^{k-1}_{ij}, d^{k-1}_{ik} + d^{k-1}_{kj})$
return $D^N$
```

Fig. 1. Pseudocode for the Floyd-Warshall algorithm.

given graph G. We can think of the algorithm as composed of N steps. At each $k\text{th}$ step, we compute D^k using the data from the previous step D^{k-1}. Pseudocode is given in Fig. 1.

3.1.1 A Recursive Implementation of FW

Before presenting the recursive implementation, we introduce FWI, 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 Fig. 2. This is used as the base case for the recursive implementation. The pseudocode for the recursive implementation of the Floyd-Warshall algorithm is given in Fig. 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 Fig. 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.

```
FWI($A$, $B$, $C$)

// Let $N$ be the problem size of $A$, $B$
   and $C$
for $k \leftarrow 1$ to $N$
    for $i \leftarrow 1$ to $N$
        for $j \leftarrow 1$ to $N$
            $a_{ij} \leftarrow \min(a_{ij}, b_{ik} + c_{kj})$
```

Fig. 2. Pseudocode for Floyd-Warshall algorithm with three arguments.

```
FWR($A$, $B$, $C$)

if (base case)
    FWI($A$, $B$, $C$);
else
    FWR($A_{11}$, $B_{11}$, $C_{11}$);
    FWR($A_{12}$, $B_{11}$, $C_{12}$);
    FWR($A_{21}$, $B_{21}$, $C_{11}$);
    FWR($A_{22}$, $B_{21}$, $C_{12}$);
    FWR($A_{22}$, $B_{22}$, $C_{22}$);
    FWR($A_{21}$, $B_{22}$, $C_{21}$);
    FWR($A_{12}$, $B_{12}$, $C_{22}$);
    FWR($A_{11}$, $B_{12}$, $C_{21}$);
```

Fig. 3. Pseudocode for the recursive implementation of the Floyd-Warshall algorithm.

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

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

for $k-1 \leq k', k'' \leq N$, 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 Fig. 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 (1), i.e.,

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

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 Fig. 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:

$$t^k_{ij} \leq d^k_{ij}. \tag{3}$$

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,

$$\begin{aligned} t^m_{ij} &= \min(t^{m-1}_{ij}, t^{m'}_{im} + t^{m''}_{mj}) \\ &\leq \min(d^{m-1}_{ij}, t^{m'}_{im} + t^{m4''}_{mj}) \\ &\leq \min(d^{m-1}_{ij}, t^{m-1}_{im} + t^{m-1}_{mj}) \\ &\leq \min(d^{m-1}_{ij}, d^{m-1}_{im} + d^{m-1}_{mj}) \\ &= d^m_{ij}. \end{aligned}$$

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 (2) computes the length of *some* path, we have

$$d_{ij}^{N} \leq t_{ij}^{N}. \tag{4}$$

From (3) and (4), we have $d_{ij}^{N} = t_{ij}^{N}$, which completes the proof. □

Theorem 1. *The recursive implementation of the Floyd-Warshall algorithm shown in Fig. 3 correctly computes the shortest paths between allpairs 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 (available at http://computer.org/tpds/archives.htm) and also in [21]. The proof is by induction. For the sake of simplicity, we assume that the problem size $N = 2^n$. We use $\mathrm{FWR}(A, B, C)$ to denote both the code segment shown in Fig. 3 and the resulting matrix A after executing the code. Also $\mathrm{FWI}(A, B, C)$ represents both the code segment in Fig. 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 $\mathrm{FWR}(D, D, D)$, 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, eight recursive calls are made as shown in Fig. 3.

The first call is interpreted as $\mathrm{FWR}(D_{11}^0, D_{11}^0, D_{11}^0)$, 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 $D_{11}^{N/2}$. In other words, the shortest path will be found from *i* to *j* with all intermediate vertices in the set 1 to *k*, where *i*, *j*, and *k* are in the set 1 to $N/2$.

The second call is $\mathrm{FWR}(D_{12}^0$, $D_{11}^{N/2}$, $D_{12}^0)$, where D_{12}^0 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 $D_{11}^{N/2}$ after the first recursive call, whereas D_{12} still contains original values. This call is in the form of $\mathrm{FWR}(A, B, A)$. It can be shown (see Appendix A located at http://computer.org/tpds/archives.htm) that, for every operation $a_{ij}^k = \min(a_{ij}^{k-1}, b_{ik}^{k-1} + a_{ij}^{k-1})$ in FWI (A, B, A), there is a corresponding operation $a_{ij}^k = \min(a_{ij}^{k-1}, b_{ik}^{k-1} + a^{k'}{}_{ij})$ in $\mathrm{FWR}(A, B, A)$, where $k' \geq k - 1$. Using Claim 1, it can be proven that the result of $\mathrm{FWR}(D_{12}^0, D_{11}^{N/2}, D_{12}^0)$ is the same as $\mathrm{FWI}(D_{12}^0, D_{11}^{N/2}, D_{12}^0)$. On the other hand, it is easy to show that $\mathrm{FWI}(D_{12}^0, D_{11}^{N/2}, D_{12}^0)$ correctly computes and generates $D_{12}^{N/2}$. This is to say $\mathrm{FWR}(D_{12}^0, D_{11}^{N/2}, D_{12}^0)$ 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 $D_{21}^{N/2}$ using D_{21}^0 and $D_{11}^{N/2}$, and the fourth call computes $D_{22}^{N/2}$ using D_{22}^0, $D_{12}^{N/2}$, and $D_{21}^{N/2}$ 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 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

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

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

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

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

$$D(B) = O(B^2). \tag{7}$$

By combining (10) and (11), it can be shown that:

$$D(N) = \frac{N^3}{B^3} * D(B) = O\left(\frac{N^3}{B}\right). \tag{8}$$

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

Theorem 3. *The recursive implementation reduces the traffic between the* ith *and the* $(i-1)$th *level of cache by a factor of* B_i *at each level of the memory hierarchy, where* $B_i = \mathrm{O}(\sqrt{C_i})$.

Proof. Note, first of all, that no tuning was assumed when calculating the amount of processor-memory traffic in the proof of Theorem 2. Namely, (12) holds for any *N* and any *B*, where $B = \mathrm{O}(\sqrt{C})$.

In order to prove Theorem 3, first consider the entire problem and the traffic between main memory and the mth level of cache (size C_m). By Theorem 2, the traffic will be reduced by B_m, where $B_m = \mathrm{O}(\sqrt{C_m})$. Next, consider each problem of size B_m and the traffic between the mth level of cache and the $(m-1)$th level of cache (size C_{m-1}). By replacing N in Theorem 2 by B_m, it can be shown that this traffic is reduced by a factor of B_{m-1}, where $B_{m-1} = \mathrm{O}(\sqrt{C_{m-1}})$.

This simple extension of Theorem 2 can be done for each level of the memory hierarchy and, therefore, the processor-memory traffic between the ith and the $(i-1)$th level of cache will be reduced by a factor of B_i, where $B_i = \mathrm{O}(\sqrt{C_i})$. □

In [14], 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.

Lemma 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 1 and Theorem 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 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 2 × improvement when we set the base case in FWR such that the base case would utilize more of the cache closest to the processor. Set the base case in FWR to be *B*, where B^2 is on the order of the cache size. 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 percent improvement on the Pentium III and a 2 × improvement on the UltraSPARC III.

In order to improve performance, B^2 must be chosen to be on the order of the L1 cache size. The simplest and possibly the most accurate method of choosing *B* is to experiment with various tile sizes. This is the method that the Automatically Tuned Linear Algebra Subroutines (ATLAS) project employs [36]. However, it is beneficial to find an estimate of the optimal tile size. A block size selection heuristic for finding this estimate is discussed in [26], and outlined here.

- Use the 2:1 rule of thumb from [25] 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 three 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 (13), where *d* is the size of one element and *C* is the adjusted cache size. This minimizes capacity misses.

$$3^* B^{2*} d = C. \tag{9}$$

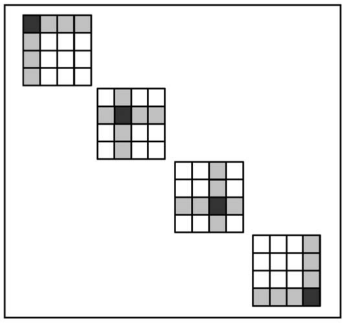


Fig. 4. Tiled implementation of FW.

3.1.2 A Tiled Implementation for FW

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

Analysis. In [34], an upper bound for any cache optimized Floyd-Warshall algorithm was shown, however, no formal analysis with respect to traffic was shown for their tiled implementation. In fact, our results show speed-ups significantly larger than the upper bound shown in [34]. The following analysis is performed for the tiled implementation in the context of the model discussed in Section 1.

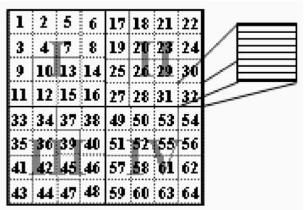
Theorem 5. *The proposed tiled implementation of the Floyd-Warshall algorithm reduces the processor-memory traffic by a factor of B, where B^2 is on the order of the cache size.*

Proof. At each block, we perform B^3 operations. There are $N/B \times N/B$ 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 1, we have the following.

Theorem 6. *The proposed tiled implementation is asymptotically optimal among all implementations of the Floyd-Warshall algorithm with respect to processor-memory traffic.*

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



1	2	5	6	17	18	21	22
3	4	7	8	19	20	23	24
9	10	13	14	25	26	29	30
11	12	15	16	27	28	31	32
33	34	37	38	49	50	53	54
35	36	39	40	51	52	55	56
41	42	45	46	57	58	61	62
43	44	47	48	59	60	63	64

Fig. 5. Z-Morton layout.

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 Fig. 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 [34] that the block size should be a multiple of the number of elements in a cache block.

As mentioned in Section 3.1, the best block size should be found experimentally, and the block size selection heuristic discussed in Section 3.1 can be used to give a rough bound on the best block size. However, when implementing the tiled implementation, it is also important to note that the search space must take into account each level of cache as well as the size of the TLB. Given these various solutions for B, the search space should be expanded accordingly. In [34], only the level-1 cache is considered, however, with an on-chip level-2 cache, often the best block size is larger than the level-1 cache. 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 [5], [24], [26]. 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 four 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 Fig. 5). See [5] 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 the section *Optimizing the Tiled Implementation* 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 Fig. 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 solves the single-source shortest paths problem 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 pseudocode for the algorithm is given in Fig. 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.

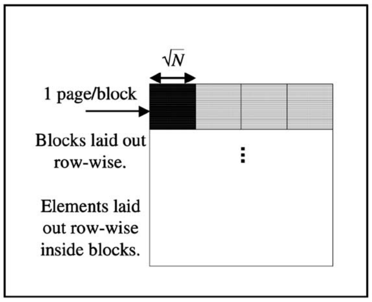

$\sqrt{N}$
1 page/block
Blocks laid out row-wise.
⋮
Elements laid out row-wise inside blocks.

Fig. 6. The block data layout.

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. 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

```
Dijkstra's($V$)

$S = \emptyset$
$Q = V[G]$
While $Q \neq \emptyset$
    $u$ = Extract-Min($Q$)
    $S = S \cup \{u\}$
    For each vertex $v \in$ Adj[$u$]
        Update $d[v]$
return $S$
```

Fig. 7. Pseudocode for Dijkstra's algorithm.

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.

There are two commonly used graph representations. The adjacency matrix is an $N \times N$ matrix, where the $(i, j)\mathrm{th}$ element of the matrix is the cost from the $i\mathrm{th}$ node to the $j\mathrm{th}$ node of the graph. This representation is of size $\mathrm{O}(N^2)$. 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, $\mathrm{O}(N + E)$. However, the fact that it is pointer based, leads to cache pollution and difficulties in prefetching. See [7] for more details regarding these common graph representations.

Consider a simple combination of these two representations [29]. 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 runtime. 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 $\mathrm{O}(N + E)$. 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 [29].

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($G$, $M$)

while (there exists an augmenting path)
{
    increase $|M|$ by one using the
        augmenting path;
}
return $M$;
```

Fig. 8. Pseudocode for augmenting path matching algorithm.

3.3 Optimizing Matching Algorithm for Bipartite Graph

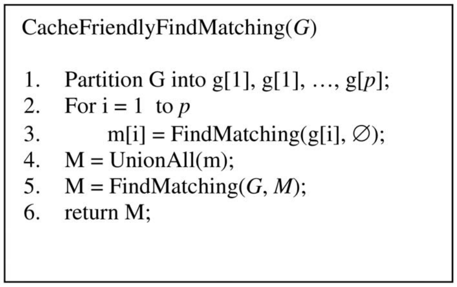
In this section, the ideas and techniques developed in the previous sections are utilized to optimize another fundamental graph algorithm, namely, matching algorithm for bipartite graphs. As discussed earlier, this algorithm shows similarities to Dijkstra's algorithm with respect to memory access in 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 $\mathrm{O}(N^* E)$. Pseudocode is given in Fig. 8. A more detailed explanation of this primitive matching algorithm is given in [18].

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 subgraphs using a certain graph-partitioning scheme and solve maximum matching problem locally for each subgraph using the matching algorithm. Next, Union all the obtained matchings for subgraphs 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. Pseudocode for this implementation is presented in Fig. 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



```
CacheFriendlyFindMatching($G$)

Partition G into g[1], g[1], …, g[$p$];
For i = 1  to $p$
    m[i] = FindMatching(g[i], $\varnothing$);
M = UnionAll(m);
M = FindMatching($G$, $M$);
return M;
```

Fig. 9. Pseudocode for cache friendly implementation of the matching algorithm.

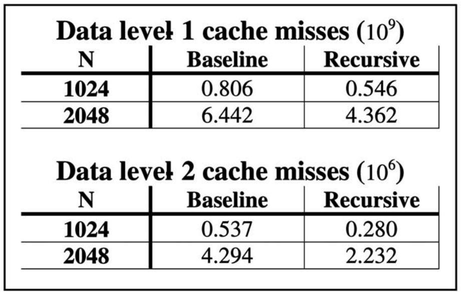
size implying increased temporal locality. If the sizes of subgraphs are chosen appropriately, each of which fits into the cache, it generates minimal processor-memory traffic of $\mathrm{O}(N+E)$ because a single loading of each data element to cache is necessary. In the best case, that is a maximum matching is found at the first stage, our implementation only causes overall $\mathrm{O}(N+E)$ processor-memory traffic as the program ends at the first stage. It is $\mathrm{O}(N)$ times improvement over $\mathrm{O}(N^*E)$, the processor-memory traffic for the straight forward implementations. The size of subgraph 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 four equal partitions, count the number of edges between each pair of partitions, and combine partitions into two partitions such that as many internal edges are created as possible.

4 Experimental and Simulation Results

We use four different architectures for our experiments. The Pentium III Xeon running Windows 2000 is a 700 MHz, four 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 two processors and 1 GB of main memory. Each processor has 64 KB of level-1 data cache and 8 MB of level-2 cache. The level-1 cache is 4-way set associative with 32 B lines and the level-2 cache is direct mapped with 64 B lines. The MIPS machine is a 300 MHz R12000, 64 processor, shared memory machine with 16 GB of main memory. Each processor has 32 KB of level-1 data cache and 8 MB of level-2 cache. The level-1 cache is 2-way set associative with 32 B lines and the level-2 cache is direct mapped with 64 B lines. The Alpha 21264 is a 500 MHz uniprocessor machine with 512 MB of main memory. It has 64 KB of level-1 data cache and 4 MB of level-2 cache. The level-1 cache is 2-way set associative with 64 B lines and the level-2 cache is direct mapped with 64 B lines. It also has an 8-element fully associative victim cache. All experiments are run on a uniprocessor or on a single node of a multiprocessor system. Unless otherwise specified simulations are performed using the SimpleScalar simulator with a 16 KB, 4-way set associative level-1 data cache and a 256 KB, 8-way set associative level-2 cache.

TABLE 1
FWR Implementation Simulation Results



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

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 percent decrease in level-1 cache misses and a 2 × decrease in level-2 cache misses for problem sizes of 1,024 and 2,048. 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 10 × improvement in overall execution time on the MIPS, roughly than 7 × improvement on the Pentium III and the Alpha, and more than 2 × improvement on the UltraSPARC III. These results are shown in Fig. 10. Differences in performance gains between machines are expected, due to the wide variance in cache parameters and miss penalties.

Table 2 shows the result of comparing the tiled implementation using a row-wise layout and the block size selection used in [34] with the tiled implementation using the block data layout and our block size selection. Simulation results show that the block size selection used in [34] optimizes the level-1 cache misses, but incurs a level-2 cache miss ratio of almost 30 percent. 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 percent to 30 percent improvement by the Block Data Layout over the row-wise data layout.

A comparison for the tiled implementation using the Block Data Layout with the baseline implementation was also performed. Simulation results for this are shown in Table 3. These results show a 2 × improvement in level-2 cache misses and a 30 percent improvement in level-1 cache misses. Experimental results show a 10 × improvement in

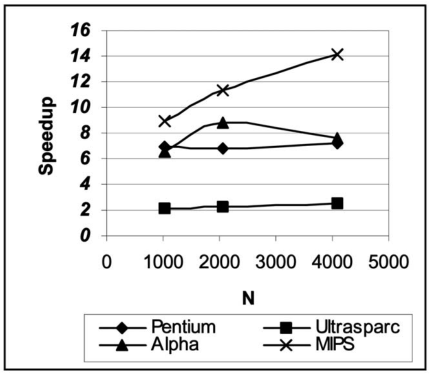


Fig. 10. Speedup results for the recursive implementation of the Floyd-Warshall algorithm.

execution time for the Alpha, better than 7 × improvement for the Pentium III and the MIPS and roughly a 3 × improvement for the UltraSPARC III (Fig. 11).

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 percent 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 most likely due to the fact that the majority of the data reuse is within the final block. Since both of these data layouts have the final block laid out in contiguous memory locations, they perform equally well.

Our experimental results show that the recursive implementation is slightly slower than the tiled implementation. 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 infinities 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 subproblems until it reaches the base case while the input problem size only has to be a multiple of the block size for the tiled implementation. Thus, in some cases, more padding is required for the recursive implementation, resulting in unnecessary computations which efficient code should avoid. We believe that if both implementations are sufficiently efficient, then the recursive implementation will run faster. Note that the recursive implementation isusually preferred for an important reason: 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.

TABLE 2
Performance Comparison

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 3
Simulation Result

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

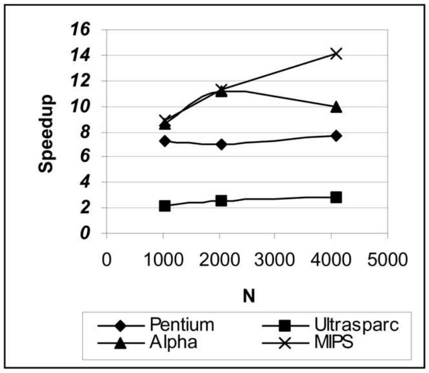


Fig. 11. Speedup results for the tiled implementation of the Floyd-Warshall algorithm.

4.2 Results for Dijkstra's Algorithm Optimization

In order to demonstrate the performance improvements using our graph representation, simulations as well as

TABLE 4
Execution Time on Pentium III

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 5
Execution Time on UltraSPARC 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

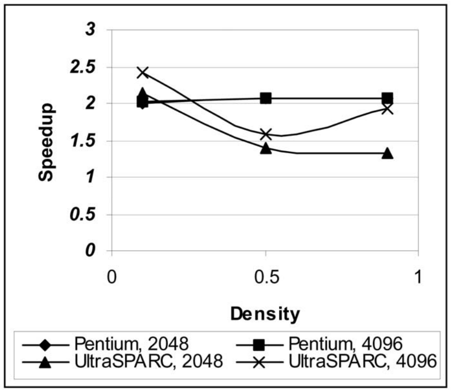


Fig. 12. Speedup results for Dijkstra's algorithm.

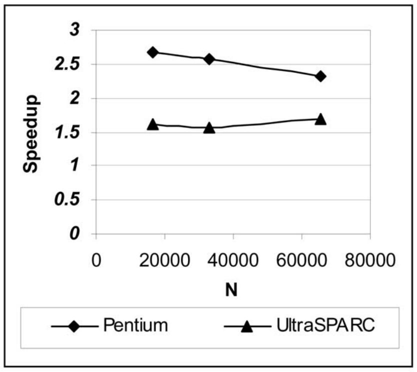


Fig. 13. Speedup results for Dijkstra's algorithm.

experiments on two different machines, the Pentium III and UltraSPARC III, were performed for Dijkstra's algorithm. The simulation results show approximately 20 percent reduction in level-1 cache misses and a 2 × 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 Fig. 12. Experiments on larger problem sizes (16K nodes up to 64K nodes) at a graph density of 10 percent are shown in Fig. 13 and again are limited by the size of main memory. All of the results show a 2 × improvement for Dijkstra's algorithm on the Pentium III and a 20 percent improvement on the UltraSPARC III. This significant difference in performance is due primarily to the difference in the memory hierarchy of these two architectures.

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. Figs. 14 and 15 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 2 × improvement on the Pentium III and 20 percent 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 percent to 90 percent as well as the large problem sizes of 16K nodes up to 64K nodes with densities of 10 percent. Simulations were also performed to verify improved cache performance. These results are shown in Table 7. They show approximately a 20 percent reduction in the number of level-1 cache misses and a 2 × reduction in the number of level-2 cache misses.

TABLE 6
Simulation Results for Dijkstra's Algorithm

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)		

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

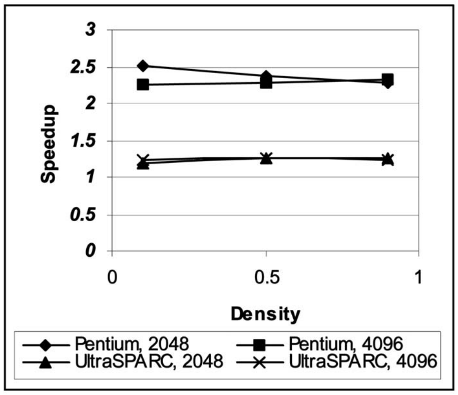


Fig. 14. Speedup results for Prim's algorithm.

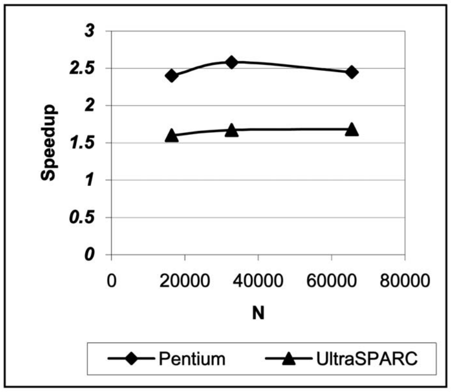


Fig. 15. Speedup results for Prim's algorithm.

in each subgraph. 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 level. For this case, the optimized implementation showed only 10 percent 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 Fig. 16, which shows the speedup vs. graph density. Results ranged from just over 2 × for graphs of 10 percent density to over 4 × for graphs of 30 percent density. In this case, the problem size was fixed at 8,192 nodes and density was limited to 30 percent by main memory. More interesting results are those shown in Fig. 17. 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 2 × 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 3 × , which indicates that the optimized implementation does improve cache performance beyond the amount reduced by the decrease in work.

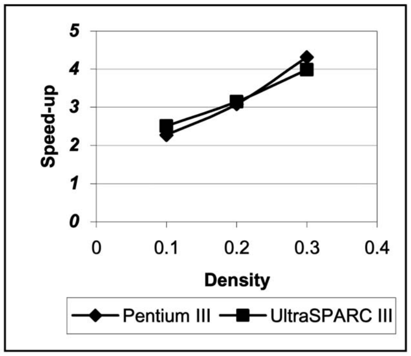


Fig. 16. Speedup versus density results for matching algorithm.

TABLE 7
Simulation 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)		

5 Conclusion

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 [7] 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 [7] and algorithms built on top of those, such as finding strongly connected components [7], can also benefit fromour data layout optimization. Another example is the Ford-Fulkerson algorithm [7] 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 needs to be added is computation and data distribution, synchronization, and communication primitives. One of our future directions will

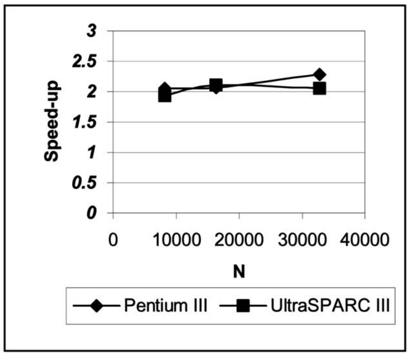


Fig. 17. Average sppedup results for matching algorithm.

TABLE 8
Simulation 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)

be to implement parallel versions of the Floyd-Warshall algorithm and matching algorithm based on the work presented in this paper.

ACKNOWLEDGMENTS

This work is part of the Algorithms for Data IntensiVe Applications on Intelligent and Smart MemORies (ADVISOR) Project [1] supported by the US Defense Advanced Research Projects Agency (DARPA) Data Intensive Systems Program under contract F33615-99-1-1483 monitored by Wright Patterson Airforce Base, in part by the US National Science Foundation under contract no. ACI-0305763, and in part by an equipment grant from Intel Corporation. The authors would like to thank Bo Hong and Gokul Govindu for helpful discussions and for their editorial assistance. This work was done while J.-S. Park was at the University of Southern California. A previous version of this paper appeared in the *Proceedings of the International Parallel and Distributed Processing Symposium*, April 2002.

REFERENCES

[1] ADVISOR Project, http://advisor.usc.edu/, 2001.

[2] M. Brenner, “Multiagent Planning with Partially Ordered Temporal Plans,” *Proc Int'l Joint Conf. Artificial Intelligence,* 2003.

[3] D. Burger and T. Austin, “The SimpleScalar Tool Set, Version 2.0,” Univ. of Wisconsin-Madison Computer Sciences Dept. Technical Report #1342, 1997.

[4] J. Carter, W. Hsieh, L. Stoller, M. Swanson, L. Zhang, and S. McKee, “Impulse: Memory System Support for Scientific Applications,” *J. Scientific Programming,* vol. 7, nos. 3-4, 1999.

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

[6] T. Chilimbi, M. Hill, and J. Larus, “Cache-Conscious Structure Layout,” *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* 1999.

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

[8] N. Dutt, P. Panda, and A. Nicolau, “Data Organization for Improved Performance in Embedded Processor Applications,” *ACM Trans. Design Automation of Electronic Systems,* vol. 2, no. 4, Oct. 1997.

[9] J. Frens and D. Wise, “Auto-Blocking Matrix-Multiplication or Tracking BLAS3 Performance from Source Code,” *Proc. Sixth ACM SIGPLAN Symp. Principles and Practice of Parallel Programming,* June 1997.

[10] M. Frigo, C.E. Leiserson, H. Prokop, and S. Ramachandran, “Cache-Oblivious Algorithms,” *Proc. 40th Ann. Symp. Foundations of Computer Science,* pp. 17-18, Oct. 1999.

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

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

[13] A. Gonzalez, M. Valero, N. Topham, and J.M. Parcerisa, “Eliminating Cache Conflict Misses through XOR-Based Placement Functions,” *Proc. 1997 Int'l Conf. Supercomputing,* July 1997.

[14] J. Hong and H. Kung, “I/O Complexity: The Red Blue Pebble Game,” *Proc. ACM Symp. Theory of Computing,* 1981.

[15] M. Kallahalla and P.J. Varman, “Optimal Prefetching and Caching for Parallel I/O Systems,” *Proc. 13th ACM Symp. Parallel Algorithms and Architectures,* 2001.

[16] M. Lam, E. Rothberg, and M. Wolf, “The Cache Performance and Optimizations of Blocked Algorithms,” *Proc. Fourth Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* Apr. 1991.

[17] A. LaMarca and R. Ladner, “The Influence of Caches on the Performance of Heaps,” *ACM J. Experimental Algorithmics,* vol. 1, 1996.

[18] E. Lawler, *Combinatorial Optimization: Networks and Matroids.* New York: Holt, Rhinehart, and Winston, 1976.

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

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

[21] J. Park, M. Penner, and V.K. Prasanna, “Optimizing Graph Algorithms for Improved Cache Performance,” Technical Report USC-CENG 03-03, Dept. of Electrical Eng., Univ. of Southern California, Nov. 2003.

[22] N. Park, B. Hong, and V. Prasanna, “Tiling, Block Data Layout, and Memory Hierarchy Performance,” *IEEE Trans. Parallel and Distributed Systems,* vol. 14, no. 7, July 2003.

[23] N. Park, B. Hong, and V. Prasanna, “Analysis of Memory Hierarchy Performance of Block Data Layout,” *Proc. Int'l Conf. Parallel Processing (ICPP),* Aug. 2002.

[24] N. Park, D. Kang, K. Bondalapati, and V. Prasanna, “Dynamic Data Layouts for Cache-Conscious Factorization of the DFT,” *Proc. Int'l Parallel and Distributed Processing Symp.,* May 2000.

[25] D. Patterson and J. Hennessy, *Computer Architecture: A Quantitative Approach,* second ed. San Francisco, Calif.: Morgan Kaufmann, 1996.

[26] M. Penner and V. Prasanna, “Cache-Friendly Implementations of Transitive Closure,” *Proc. Int'l Conf. Parallel Architectures and Compiler Techniques,* Sept. 2001.

[27] G. Rivera and C. Tseng, “Data Transformations for Eliminating Conflict Misses,” *Proc. 1998 ACM SIGPLAN Conf. Programming Language Design and Implementation,* June 1998.

[28] F. Rastello and Y. Robert, “Loop Partitioning Versus Tiling for Cache-Based Multiprocessor,” *Proc. Int'l Conf. Parallel and Distributed Computing and Systems,* 1998.

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

[30] P. Sanders, “Fast Priority Queues for Cached Memory,” *ACM J. Experimental Algorithmics,* vol. 5, 2000.

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

[32] S. Sen and S. Chatterjee, “Towards a Theory of Cache-Efficient Algorithms,” *Proc. Symp. Discrete Algorithms,* 2000.

[33] SPIRAL Project, http://www.ece.cmu.edu/~spiral/, 2004.

[34] G. Venkataraman, S. Sahni, and S. Mukhopadhyaya, “A Blocked All-Pairs Shortest-Paths Algorithm,” *Proc. Scandinavian Workshop Algorithms and Theory,* 2000.

[35] D. Weikle, S. McKee, and W. Wulf, “Caches as Filters: A New Approach to Cache Analysis,” *Proc. Grace Murray Hopper Conf.,* Sept. 2000.

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

[37] M. Yannakakis, “Graph Theoretic Methods in Database Theory,” *Proc. ACM Conf. Principles of Database Systems,* 1990.

Joon-Sang Park received the MS degree in computer science from the University of Southern California in 2001. Currently, he is pursuing the PhD degree in computer science at the University of California, Los Angeles. His research interests include parallel and distributed systems and algorithms, mobile ad hoc networks focusing on routing and MAC protocols, and wireless communication systems, especially MIMO systems. He is a student member of the IEEE.

Michael Penner received the bachelor's degree in computer engineering/computer science from the University of Southern California in 2000. He received the master's degree in computer engineering with an emphasis on computer architecture in 2002, also from the University of Southern California. While pursuing the master's degree, he performed research on improving cache performance through algorithm transformations under Professor Viktor Prasanna. Following graduation in 2002, he joined the microprocessor design group at Intel Corp. in Hillsboro, Oregon.

Viktor K. Prasanna received the BS degree in electronics engineering from the Bangalore University and the MS degree from the School of Automation, Indian Institute of Science. He received the PhD degree in computer science from the Pennsylvania State University in 1983. Currently, he is a professor in the Department of Electrical Engineering as well as in the Department of Computer Science at the University of Southern California, Los Angeles. He is also an associate member of the Center for Applied Mathematical Sciences (CAMS) at the University of Southern California. He served as the division director for the Computer Engineering Division from 1994-1998. His research interests include parallel and distributed systems, embedded systems, configurable architectures, and high-performance computing. Dr. Prasanna has published extensively and consulted for industries in the above areas. He has served on the organizing committees of several international meetings in VLSI computations, parallel computation, and high-performance computing. He is the steering cochair of the *Proceedings of the International Parallel and Distributed Processing Symposium* (merged *IEEE International Parallel Processing Symposium* (IPPS) and the *Symposium on Parallel and Distributed Processing* (SPDP)) and is the steering chair of the *International Conference on High Performance Computing* (HiPC). He serves on the editorial boards of the *Journal of Parallel and Distributed Computing* and the *Proceedings of the IEEE*. He is the editor-in-chief of the *IEEE Transactions on Computers*. He was the founding chair of the IEEE Computer Society Technical Committee on Parallel Processing. He is a fellow of the IEEE.