A Blocked All-Pairs Shortest-Paths Algorithm

Gayathri Venkataraman, Sartaj Sahni, and Srabani Mukhopadhyaya

Department of Computer and Information Science and Engineering
University of Florida, Gainesville, FL 32611
{gvenkata, sahni, srabani}@cise.ufl.edu

Abstract. We propose a blocked version of Floyd's all-pairs shortest-paths algorithm. The blocked algorithm makes better utilization of cache than does Floyd's original algorithm. Experiments indicate that the blocked algorithm delivers a speedup (relative to the unblocked Floyd's algorithm) between 1.6 and 1.9 on a Sun Ultra Enterprise 4000/5000 for graphs that have between 480 and 3200 vertices. The measured speedup on an SGI O2 for graphs with between 240 and 1200 vertices is between 1.6 and 2.

Keywords: All pairs shortest paths, blocking, cache, speedup.

1 Introduction

Traditionally, algorithms are developed, analyzed, and optimized for the RAM computer model in which a computer has a single uniformly accessible memory [11]. Contemporary computers, however, have multiple levels of memory and the memory access time varies significantly from one memory level to the next. For example, contemporary Sun and SGI workstations have an L1 cache, an L2 cache, and a main memory. The L1 cache in a Sun Ultra Enterprise 4000/5000 is 16 KB, the L2 cache is 4 MB, and main memory is in excess of 100 MB. Additionally, a contemporary computer has a limited number of registers—ten to twenty. Typically, it takes 1 cycle to access data from L1 cache. When the desired data is not in L1 cache, we experience an L1 *miss* and the data is brought from L2 cache to L1 cache using 6 to 10 cycles. If the desired data is not in L2 cache either, then we experience an L2 miss and data is fetched from main memory into L2 cache at a cost of (say) 50 cycles, and from there to L1 cache. We can reduce run time by organizing our computations so as to minimize the number of L1 and L2 cache misses.

Although several theoretical models for computers with multiple-level memories have been proposed [3, ?,?], these models have not found wide application, and most of the work in the area of performance enhancement via cache optimization has been experimentally oriented. Trace driven simulators have been used to study the cache performance of a specific program running on a specific computer, determine the portions of the code or the data structures that result in a large fraction of the cache misses, and then optimize these code segments and/or data structures. Trace driven simulations have also been used to develop

analytical models of cache behavior. See [4, 15, 19, 22–24], for example, for some ways in which trace driven simulators have been used in cache performance enhancement studies.

La Marca and Ladner [13] develop a model for a single-level direct-mapped cache. They use this model to analyze the performance of binary heaps and cache-aligned d-heaps. LaMarca and Ladner [14] optimize the cache performance of several sorting methods. Their cache optimized heapsort and mergesort codes achieve a speedup of 1.85 and 1.38, respectively, when sorting 1,000,000 uniformly distributed integers on a Sprac 10 processor. Lam, Rothberg, and Wolf [12] have considered the cache performance of a blocked matrix multiply code relative to a traditional matrix multiply code. They report a speedup of 4.3 for their blocked matrix multiply code for a matrix of size 300. Sulatycke and Ghose [21] and Stewart [20] have also studied the cache performance of various matrix multilication algorithms. Stewart [20] reports that the best way to muliply the matrices A and B is to first transpose B and then use the classical three loop algorithm on A and B^T. He further reports that by simply reordering the loops from the traditional ijk order to an ikj order (i.e., interchange the second and third `for` loops in the traditional code) the code performance is about the same as when square blocks (as used in [12] are used); row blocks yield superior speedup than column blocks and ikj ordering. Note that the transpose method, ikj ordering, square blocking, and row blocking deliver speedup relative to the traditional ijk code by reducing cache misses. Stewart [20] reports a speedup of 2.7 for the transpose method relative to the ijk code; both codes were written in C and compiled using maximum compiler optimization; the matrix size was 1200, and the code was run on a SUN Ultra Enterprise 4000/5000 computer.

Al-Furaih and Ranka [5, 6] have studied cache optimization methods for sorting and unstructured iterative computations.

In this paper we propose a blocked formulation of Floyd's dynamic programming algorithm to find the lengths of the shortest paths between all pairs of vertices in a graph [11]. Blocked (or tiled) computation methods have been used before (for example, [16, ?,?,?,?,?]). Our blocked algorithm provides a speedup (relative to the unblocked algorithm) between 1.6 and 1.9 on a Sun Ultra Enterprise 4000/5000 for graphs that have between 480 and 3200 vertices. The measured speedup on an SGI O2 for graphs with between 240 and 1200 vertices is between 1.6 and 2. These speedups are comparable to the speedups cited above for cache-optimized sorting and matrix multiplication codes on Sun platforms.

In Section 2 we give Floyd's all-pairs shortest-paths algorithm. Section 3 analyzes the potential speedup benefits from reorganizing Floyd's algorithm to make better use of cache. This analysis uses data gathered using the cache simulation tool *Shade* [17]. Our blocked version of Floyd's algorithm and a correctness proof are given in Section 4. Section 5 gives measured speedup results for our blocked algorithm.

2 Floyd's All-Pairs Shortest-Paths Algorithm

Let $G = (V, E)$ be a directed graph with n vertices. Let $cost$ be the cost adjacency matrix for G. So $cost(i,i) = 0$, $1 \leq i \leq n$; $cost(i,j)$ is the length (or cost) of edge (i,j) if $(i,j) \in E(G)$ and $cost(i,j) = \infty$ if $i \neq j$ and $(i,j) \notin E(G)$.

In the all-pairs shortest-paths problem we are to determine a matrix A such that $A(i,j)$ is the length of a shortest path from i to j. When G has no cycle whose length (cost) is less than 0, the matrix A may be computed using dynamic programming [11]. Let $A^k(i,j)$ be the length of a shortest path from i to j under the constraint that the path contain no intermediate vertex whose index is more than k. It is easy to see that $A(i,j) = A^n(i,j)$. When G has no cycle with negative length, the following dynamic programming recurrence is valid:

$$A^0(i,j) = cost(i,j) \tag{1}$$

$$A^k(i,j) = \min\{A^{k-1}(i,j), A^{k-1}(i,k) + A^{k-1}(k,j)\}, k \geq 1 \tag{2}$$

Equations 1 and 2 lead to the algorithm of Figure 1 to compute A. This algorithm is known as Floyd's algorithm. It may be shown [11] that `AllPairs` computes $A^k(i,j) =$ `A[i][j]` in iteration `k` of the outermost `for` loop.

```
function AllPairs(int A, int n)
{// A[i][j] = cost(i,j) initially
 // A[i][j] equals length of shortest
 // i to j path on termination
  for (k = 1; k <= n; k++)
     for (i = 1; i <= n; i++)
        for (j = 1; j <= n; j++)
           A[i][j] = min(A[i][j],
                         A[i][k] + A[k][j]);
}
```

Fig. 1. Floyd's shortest-paths algorithm

3 Upper Bound On Attainable Speedup

We compute an upper bound on the maximum speedup attainable by rearranging the computation of Figure 1 so as to optimize cache useage. In computing this bound we assume that any rearrangement of the computation will not decrease the number of accesses made to the elements of the array `A`.

We first obtain an equation to estimate the execution/run time of Floyd's algorithm of Figure 1. The execution time of a program is given by the following equation [18]:

$$execution\ time = (CPU\ clock\ cycles + memory\ stall\ cycles) \times clock\ cycle\ time \tag{3}$$

where *memory stall cycles* is the number of cycles the CPU spends waiting for a memory reference to complete. The following equations are also from [18].

$$CPU\ clock\ cycles = CPI \times IC \tag{4}$$

$$memory\ stall\ cycles = number\ of\ L1\ misses \times L1\ miss\ penalty \tag{5}$$

$$number\ of\ L1\ misses = IC * L1\ misses\ per\ instruction \tag{6}$$

$$L1\ misses\ per\ instruction = memory\ references\ per\ instruction \times L1\ miss\ rate \tag{7}$$

where IC is the *instruction count*, CPI is the *clock cycles per instruction*, $L1$ *miss penalty* is the number of cycles the CPU waits when there is an L1 cache miss, and $L1$ *miss rate* is the number of L1 misses per memory reference.

From these equations we obtain:

$$execution\ time = (CPI \times IC + IC \times L1\ misses\ per\ instruction \times L1\ miss\ penalty) \times clock\ cycle\ time \tag{8}$$

We also see that

$$L1\ miss\ penalty = L2\ hit\ time + L2\ miss\ rate \times L2\ miss\ penalty \tag{9}$$

where $L2$ *hit time* is the number of cycles to load an L1 cache line from L2 cache and $L2\ miss\ penalty = memory\ hit\ time$ is the number of cycles needed to load an L2 cache line from main memory.

We use Equations 8 and 9 to estimate the run time of Floyd's algorithm. Since the L2 hit time and L2 miss penalty are architecture dependent and not available to us, we use typical numbers for these—the L2 hit time is assumed to be between 6 and 10 cycles and the L2 miss penalty is assumed to be 50 cycles. For the L1 misses per instruction and the L2 miss rate we use data obtained by using the cache simulator Shade on Floyd's algorithm. Table 1 gives this data.

Table 1. Cache simulator data for algorithm of Figure 1

Matrix size	L1 misses per instruction (%)	L2 miss rate (%)
480	3.950	18.42
800	4.106	19.17
1600	4.133	19.42
2400	4.826	19.64
3200	5.553	20.07

Now we obtain a lower bound on the run time of a cache optimized version of Floyd's algorithm. Substituting Equation 7 into Equation 8 and making the reasonable assumption that cache optimization will not decrease the total number of memory references (i.e., the number of memory references for the cache optimized code is at least $IC * memory\ references\ per\ instruction$ where IC and $memory\ references\ per\ instruction$ are for `AllPairs`) yields

$$\begin{aligned} execution\ time \geq\ & (CPI \times IC + \\ & IC \times memory\ references\ per \\ & instruction \\ & \times L1\ miss\ rate \times \\ & L1\ miss\ penalty) \\ & \times clock\ cycle\ time \end{aligned} \tag{10}$$

The cache simulator gives 0.35 as the memory references per instruction for `AllPairs`. Substituting 0.35 for the number of memory references per instruction and the right side of Equation 9 for the L1 miss penalty into Equation 10, we get

$$\begin{aligned} execution\ time \geq\ & (CPI \times IC + 0.35 \times IC \times \\ & L1\ miss\ rate \times (L2\ hit\ time + \\ & L2\ miss\ rate \times L2\ miss\ penalty)) \\ & \times clock\ cycle\ time \end{aligned} \tag{11}$$

We may obtain a lower bound for the L1 and L2 miss rate by determining the minimum number of L1 and L2 misses that every reorganized version of Figure 1 must make. Since we intend to declare `i`, `j`, `k`, and `n` as register variables [8], references to these variables do not access cache and so do not cause any cache misses. Therefore, we focus on cache misses attributable to the array `A`. For our analysis we use the cache characteristics of the Sun Enterprise 4000/5000 that

are shown in Table 2. By direct mapped we mean that each byte of main memory has exactly one byte of cache to which it may be mapped. The line size of a cache gives the unit of memory transfer. So in the Sun Enterprise 4000/5000 an L1 cache miss results in a 32-byte block of data being transferred from L2 cache into L1 cache. The transferred block is one-half of an L2 line.

Table 2. Cache characteristics of the Sun Enterprise 4000/5000

Cache	Associativity	Cache size	Line size
L1	Direct mapped	16KB	32 bytes
L2	Direct mapped	4MB	64 bytes

For the analysis we assume that A is an integer array and that each integer is 4 bytes. Since Floyd's algorithm accesses each of the n^2 elements of A, all n^2 elements of A must get to L1 cache at some time. Each L1 cache miss brings in exactly 32 bytes of data (i.e., 8 elements of A). Therefore, the number of L1 cache misses is at least $n^2/8$. By a similar reasoning, the number of L2 cache misses is at least $n^2/16$. Further, Floyd's algorithm makes $3n^3$ read accesses to A (i.e., in the right side of the `min` statement of Figure 1) and n^3 write accesses (the left side of the `min` statement). We note that when the `min` statement of Figure 1 is coded as an `if` statement, write accesses are made only when the new `a[i][j]` value is smaller than the old one. In this case the number of write accesses ranges from 0 to n^3. To keep the analysis simple, we use n^3 as the write access count. The total number of accesses to A (read and write) is $4n^3$. Therefore,

$$\begin{aligned}\text{L1 miss rate} &= \text{L1 misses per } A \text{ reference}\\ &\geq n^2/8/(4n^3) = 1/(32n)\end{aligned} \tag{12}$$

$$\begin{aligned}\text{L2 miss rate} &= \text{L2 misses per } A \text{ reference}\\ &\geq n^2/16/(4n^3) = 1/(64n)\end{aligned} \tag{13}$$

The equality between the miss rate and the misses per A reference follows from our assumption that variables other than A will be register variables and so all memory references are to elements of A. Since we assume that cache optimization does not reduce the number of A references, these bounds apply to all cache optimized versions of `AllPairs`.

Substituting the bounds of Equations 12 and 13 into Equation 11, we get the following lower bound on the run time of a cache optimized version of Floyd's algorithm.

$$execution\ time \geq (CPI \times IC +$$

$$0.35 \times IC \times 1/(32n) \times$$
$$(L2\ hit\ time +$$
$$1/(64n) \times L2\ miss\ penalty)$$
$$\times clock\ cycle\ time \quad (14)$$

Dividing Equation 8 by Equation 14 yields an upper bound on the speedup obtainable by optimizing cache utilization. Figure 2 plots this upper bound when CPI ranges between 1 and 2, L2 hit time ranges from 6 to 10 cycles, and L2 miss penalty is 50 cycles. The L1 misses per instruction and the L2 miss rate are taken from Table 1. Figure 2 gives the maximum speedup we can get by optimizing the cache usage of Floyd's algorithm on typical computers that have a two-level cache.

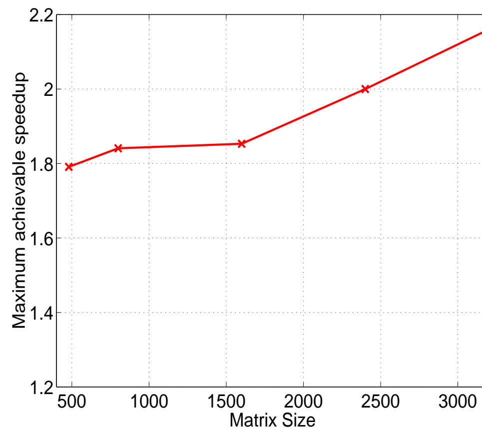


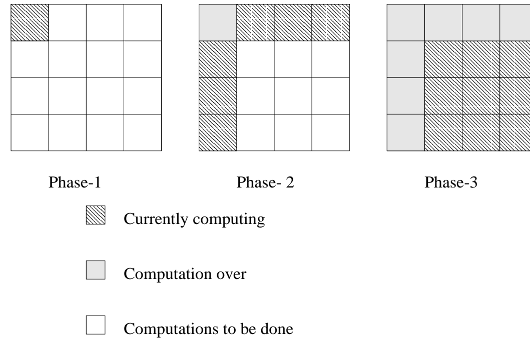
Fig. 2. Maximum achievable speedup for different matrix sizes

4 Blocked Version of Floyd's Algorithm

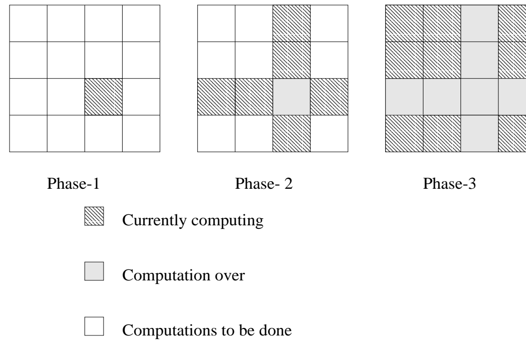
4.1 The Algorithm

We partition the cost adjacency matrix into submatrices of size $B \times B$. B is called the *blocking factor*. Although this is not necessary, we assume, for simplicity, that B divides n. Our blocked version of Floyd's algorithm (Figure 1) will perform B iterations of the outermost loop of Figure 1 on each $B \times B$ block of A before advancing to the next B iterations. It is convenient to think of each set of B iterations as divided into three phases. (Note that our implementation does not actually preform the computation in the three phase order described below.) For example, in phase 1 of the first set of B iterations, Equation 2 is used to

compute $D^k = A^k$, $1 \leq k \leq B$ for the elements in the top left block, block (1,1). Since these B iterations access only the A elements within block (1,1), we say that block (1,1) is a self-dependent block in the first B iterations.



(a) Phases when (1,1) is the self-dependent block



(b) Phases when block (t, t) is the self-dependent block

Fig. 3. Blocks computed in each phase

In phase 2 of the first B iterations a modified Equation 2 is used to compute D^k, $1 \leq k \leq B$ for the remaining blocks $(1, *)$ and $(*, 1)$ that are on the same row or column as the self-dependent block. For the remaining (1,*) blocks the modified Equation 2 is

$$D^k(i,j) = \min\{D^{k-1}(i,j), D^B(i,k) + D^{k-1}(k,j)\}, k \geq 1 \tag{15}$$

where $D^0(i,j) = A^0(i,j)$. For the remaining (*,1) blocks the modified Equation 2 is

$$D^k(i,j) = \min\{D^{k-1}(i,j), D^{k-1}(i,k) + D^B(k,j)\}, k \geq 1 \qquad (16)$$

In phase 3 D^k, $1 \leq k \leq B$ is computed for the remaining blocks (i.e., for blocks that are not on the same row or column as the self-dependent block). This computation is done using Equation 17.

$$D^k(i,j) = \min\{D^{k-1}(i,j), D^B(i,k) + D^B(k,j)\}, k \geq 1 \qquad (17)$$

Phase 3 is followed by the next round of B iterations. These are also done in three phases. This time block (2,2) is the self-dependent block. D^k, $B < k \leq 2B$ are computed for the self-dependent block in phase 1 using the equation

$$D^k(i,j) = \min\{D^{k-1}(i,j), D^{k-1}(i,k) + D^{k-1}(k,j)\} \qquad (18)$$

In phase 2 D^k, $B < k \leq 2B$ are computed for the remaining blocks that are on the same row or column as the self-dependent block and in phase 3 D^k, $B < k \leq 2B$ is computed for the blocks that are not on the same row or column as the self-dependent block. The phase 2 computation uses the following equation for the (2,*) blocks

$$D^k(i,j) = \min\{D^{k-1}(i,j), D^{2B}(i,k) + D^{k-1}(k,j)\} \qquad (19)$$

The (*,2) blocks use the following equation

$$D^k(i,j) = \min\{D^{k-1}(i,j), D^{k-1}(i,k) + D^{2B}(k,j)\} \qquad (20)$$

and the phase 3 blocks use the equation

$$D^k(i,j) = \min\{D^{k-1}(i,j), D^{2B}(i,k) + D^{2B}(k,j)\} \qquad (21)$$

The following equations are used to compute the $(t,*)$, $(*,t)$, and phase 3 blocks, respectively.

$$D^k(i,j) = \min\{D^{k-1}(i,j), D^{tB}(i,k) + D^{k-1}(k,j)\} \qquad (22)$$

$$D^k(i,j) = \min\{D^{k-1}(i,j), D^{k-1}(i,k) + D^{tB}(k,j)\} \qquad (23)$$

$$D^k(i,j) = \min\{D^{k-1}(i,j), D^{tB}(i,k) + D^{tB}(k,j)\} \qquad (24)$$

4.2 Correctness of Blocked Algorithm

The $D^k(i,j)$ values computed by the blocked algorithm are not necessarily the same as the $A^k(i,j)$ values computed by the unblocked algorithm. For example, when $B = 4$, the unblocked algorithm computes $A^1(4,7) = \min\{A^0(4,7), A^0(4,1) + A^0(1,7)\}$, whereas the blocked algorithm computes $D^1(4,7) = \min\{D^0(4,7), D^4(4,1) + D^0(1,7)\} = \min\{A^0(4,7), D^4(4,1) + A^0(1,7)\}$. Since $D^4(4,1) = A^4(4,1)$ is $\leq A^0(4,1)$, $D^1(4,7) \leq A^1(4,7)$.

To establish the correctness of the blocked algorithm we must show that $D^n(i,j) = A^n(i,j)$ for all i and j. That is, even though $D^k(i,j)$ and $A^k(i,j)$ may not be equal for $k < n$, the values agree in the end when $k = n$. Actually we will show that A and D agree at the end of each set of B iterations That is, $D^k(i,j) = A^k(i,j)$ for all i and j whenever k is a multiple of B. Hence $D^n(i,j) = A^n(i,j)$ for all i and j.

Let $k = qB$. The proof is by induction on q. We may show that $D^k(i,j) = A^k(i,j)$ for all i and j for $0 \leq q \leq n/B$. The proof is omitted from this version of the paper.

4.3 Optimal Blocking Factor

When computing the D values in a block during any round (i.e., an iteration of the outermost loop) of function `BoundedAllPairs`, at most three blocks are active. The computation for the self-dependent block accesses elements only in the self-dependent block. So during the self-dependent block computation only 1 block is active. The computation for a block R that is on the same row or column as the self-dependent block acceses elements in R as well as elements in the self-dependent block. Therefore, 2 blocks are active during the computation for R. For a block R that is not on the same row or column as the self dependent block, `BlockedAllPairs` accesses elements from 3 blocks—block R, the block that is in the same row as the self-dependent block and the same column as R, and the block that is in the same column as the self-dependent block and in the same row as R. Therefore, L1 cache misses are minimized by choosing the largest block size B such that 3 block loads of the array D fit into L1 cache. Suppose that the elements of D are 4-byte integers and that our L1 cache capacity is C bytes and that each L1 cache line is S bytes. We must choose B to be the largest integer such that $3B^2 * 4 \leq C$ (equivalently, $B \leq \sqrt{C/12}$) and B is a multiple of $S/4$. The second requirement is necessary as the smallest unit of data brought into L1 cache is S bytes and these S bytes are contiguous bytes of memory.

For the Sun Ultra Enterprise 4000/5000 $C = 16K$ and $S = 32$. Therefore, the blocking factor should be the largest integer that is $\leq \sqrt{C/12} = 37$ and is a multiple of $32/4 = 8$. That is, we should use $B = 32$ as the blocking factor. For the SGI O2 $C = 32K$ and $S = 32$. The optimal blocking factor for the SGI O2 is the largest integer that is $\leq \sqrt{C/12} = 52$ and is a multiple of $32/4 = 8$. This optimal blocking factor is 48.

5 Experimental Results

The speedup of our blocked shortest paths algorithm relative to the standard unblocked algorithm was measured by programming the two algorithms in C++ (the g++ compiler with optimization option o5 was used) and running the two programs on on a Sun Ultra Enterprise 4000/5000 and an SGI O2. Both programs were compiled using the highest-level of compiler optimization possible.

We first present the results for the SUN Ultra Enterprise. Figure 4 gives the measured speedups for different blocking factors and different n. As predicted by our analysis, the otimal blocking factor is 32 for all n.

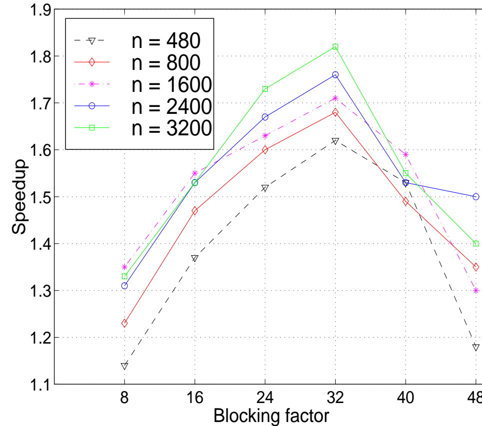


Fig. 4. Speedup of `BlockedAllPairs` on a Sun Ultra Enterprise

Figure 5 compares the speedup obtained by `BlockedAllPairs` and the maximum speedup possible by optimizing cache utilization. The curve for maximum possible speedup is that of Figure 2.

The speedup obtained by `BlockedAllPairs` is fairly close to the maximum possible. One reason we do not achieve the predicted maximum speedup is that the total instruction count for `BlockedAllPairs` is more than that for `AllPairs`. Recall that in determining the maximum speedup curve of Figure 2 we assumed that the instruction count for the cache optimized algorithm is the same as that of `AllPairs`.

Figure 6 gives the $L1$ *misses per instruction* for the unblocked and blocked versions of Floyd's algorithm. The data for this figure were obtained using the cache simulator Shade. As expected the blocked code shows better cache utilization.

Table 3 shows the cache details for the SGI O2 computer and Figure 7 shows the speedup obtained by the blocked algorithm on an SGI O2. Except for one anomaly, maximum speedup is obtained when the blocking factor is the predicted optimal factor of 48.

6 Conclusion

We have developed a blocked version of Floyd's all-pairs shortest-paths algorithm. Experimental results show that the blocked version obtains speedups close to the maximum possible for a cache optimized version of Floyd's algorithm.

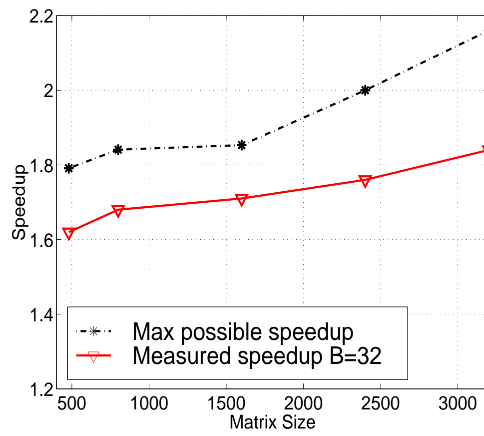


Fig. 5. Measured and maximum possible speedup

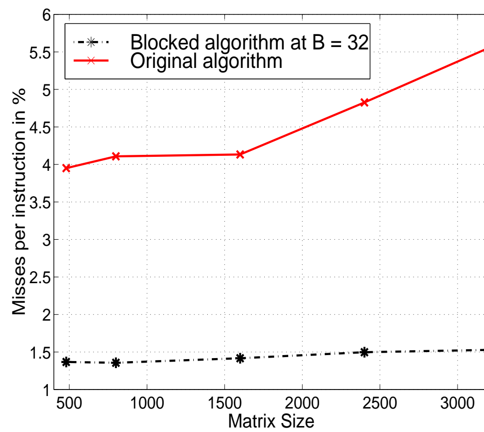


Fig. 6. Misses per instruction for unblocked and blocked algorithms

Table 3. Cache configuration of SGI

Cache type	Cache size
L1	32KB
L2	1MB

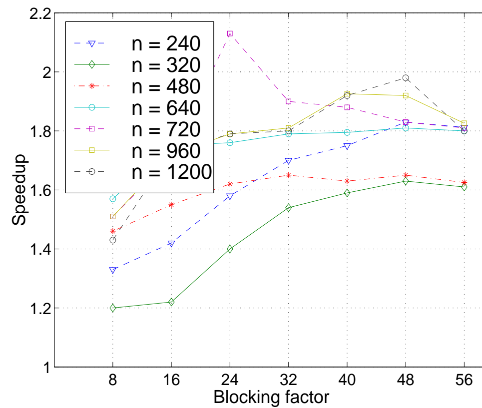


Fig. 7. Speedup obtained by `BlockedAllPairs` on an SGI O2

References

1. W. AbuSufah, D. J. Kuck, and D. H. Lawrie. Automatic program transformation for virtual memory computers. In *Proc. of the 1979 National Computer Conference*, pages 969–974, New York, 1979.
2. A. Aggarwal, K. Chandra, and M. Snir. A model for hierarchical memory. In *The 19th Annual ACM Symposium on Theory of Computing*, pages 305–314, NewYork, 1987.
3. A. Aggarwal, K. Chandra, and M. Snir. Hierarchical memory with block transfer. In *The 28th Annual IEEE Symposium on Foundations of Computer Science*, pages 204–216, LosAngeles, CA, 1987.
4. A. Aggarwal, M. Horowitz, and Hennessey. An analytical cache model. *The ACM Transactions on Computer Systems*, 7(2):184–215, 1989.
5. I. Al-Furaih and S. Ranka, Memory hierarchy management for iterative graph structures. *Proc. 12th International Parallel Processing Symposium '98. (IPPS98), Orlando, Florida.*
6. I. Al-Furaih and S. Ranka, Ibraheem Al-Furaih and Sanjay Ranka, " Practical Algorithms for Internal and External Sorting", *Proc. the Second International Conference on Parallel and Distributed Computing and Networks (PDCN'98), Brisbane, Australia, 14-16 December 1998.*
7. B. Alpern, L. Carter, E. Feig, and T. Selker. The uniform memory hierarchy model of computation. *Algorithmica*, 12(2-3):72–109, 1994.
8. D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. In *In Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, White Plains, New York, 1990.
9. D. Gannon and W. Jalby. The influence of memory hierarchy on algorithm organization: Programming FFTs on a vector multiprocessor. *In The Characteristics of Parallel Algorithms, MIT Press, Cambridge*, 1987.
10. G. H. Golub and C. F. Van Loan. Matrix Computations. *Johns Hopkins University Press, Baltimore*, 1989.

11. E. Horowitz, S. Sahni, and S. Rajasekaran. *Computer Algorithms.* Computer Science Press, New York, 1998.
12. M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. *ACM*, 26:63–74, 1991.
13. A. LaMarca and R. E. Ladner. The influences of caches on the performance of heaps. *The ACM Journal of Experimental Algorithms*, 1(4), 1996.
14. A. LaMarca and R. E. Ladner. The influences of caches on the performance of sorting. In *The ACM-SIAM Symposium on Discrete Algortihms*, pages 370–379, New Orleans, Louisiana, 5-7 January, 1997.
15. M. Martonosi, A. Gupta, and T. Anderson. Memspy: analyzing memory system bottlenecks in programs. In *Proceedings of the 1992 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 1–12, Newport, Rhode Island, 1992.
16. A. C. McKeller and E. G. Coffman. The organization of matrices and matrix operations in a paged multiprogramming environment. *CACM*, 12(3):153–165, 1969.
17. Sun Microsytems. Introduction to Shade. Manual, Sun Microsytems, Mountain View, CA, 1998.
18. D. Patterson and J. L. Hennessy. *Computer Architecture: A Quantitative Analysis.* Morgan Kaufmann, San Mateo, CA, 1996.
19. J. P. Singh, H. S. Stone, and D. F. Thiebaut. A model of workloads and its use in miss-rate prediction for fully assocaitive caches. *IEEE Transactions on Computers*, 41(7):811–825, 1992.
20. Larry Stewart. Programming to Optimize Cache Memory on the SUN Ultrasparc-IIi Processor. Master's thesis, University of Florida, Gainesville, FL, April 1999.
21. P. Sulatycke and K. Ghose, Caching-efficient multithreaded fast multiplication of sparse matrices. *Proceedings 12the International Parallel Processing Symposium*, 117–123, 1998.
22. O. Temam, C. Fricker, and W. Jalby. Cache interference phenomena. In *Proceedings of the 1994 ACM SIGMETRICS: Conference on Measurement and Modelling of Computer Systems*, pages 261–271, Nashville, Tennessee, 1994.
23. O. Temam, C. Fricker, and W. Jalby. Influence of cross-interference on blocked loops: A case study with matrix-vector multiply . *The ACM Transactions on Programming Languages and Systems*, 17(4):561–575, 1994.
24. H. Wen and J. L. Baer. Efficient trace driven simulation methods for cache performance analysis. *The ACM Transactions on Computer Systems*, 9(3):222–241, 1991.
25. M. E. Wolf and M. S. Lam. A data locality optimizing. In *In Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 30–44, Toronto, Ontario, Canada, 1991.