

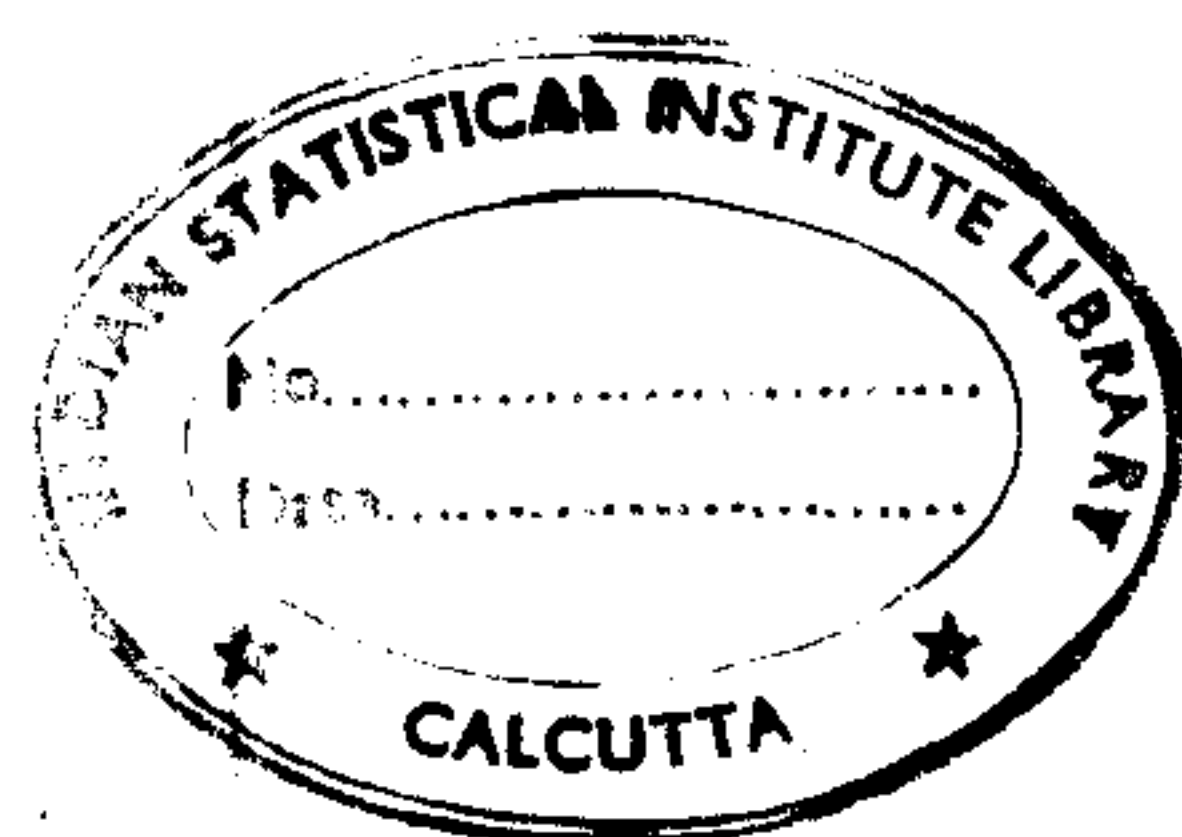
# **CACHE CONSCIOUS ALGORITHMS FOR ALL-PAIRS SHORTEST-PATHS PROBLEM**

By

**Chinmay Mahata**

**Dr. Srabani Mukhopad<sup>h</sup>yaya**  
Research Associate

**Dr. Krisnendu Mukhopad<sup>h</sup>yaya**  
Associate Professor




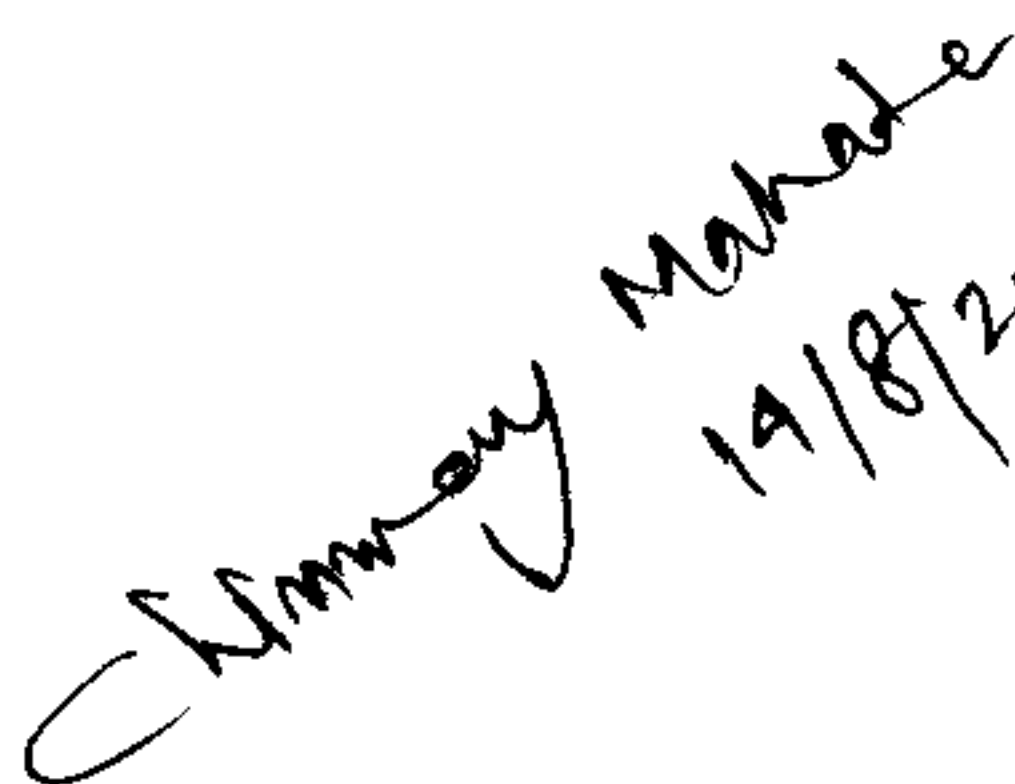
**Advanced Computing and Microelectronics Unit  
Indian Statistical Institute  
Calcutta 700 035**

# CERTIFICATE OF APPROVAL

This is to certify that this dissertation titled **Cache Conscious Algorithm for All-Pairs Shortest-Paths Problem** submitted by **Chinmay Mahata** towards partial fulfillment of the requirements for the degree of *M.Tech. (Computer Science)* at the Indian Statistical Institute, Calcutta, embodies the work carried under our supervision. His work is satisfactory.

Dr. Srabani Mukhopadhyaya  
Date:

  
Dr. Krisnendu Mukhopadhyaya  
Date: 27-07-2000

  
Chinmay Mahata  
19/8/2000.

(External Examiner)

# ACKNOWLEDGEMENTS

I am deeply indebted to the Advanced Computing and Microelectronics Unit, Indian Statistical Institute, Calcutta and to my guides Dr. Srabani Mukhopadhyaya and Dr. Krisnendu Mukhopadhyaya, for their constant encouragement and valuable guidance throughout this dissertation work.

I would like to acknowledge the members of A.C.M.U. of ISI, for offering their valuable resources , advice and help for completion for this work. I would like to thank Mr Subhasis Banerjee for providing me with many valuable suggestions. I would also like to thank Mr Arijit Bishnu for his help.

I take this opportunity to thank all my classmates.

ISI, Calcutta  
July, 2000.

Chinmay Mahata

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Role of Cache on Optimizing Algorithms . . . . .	3
1.2	Caches . . . . .	5
<b>2</b>	<b>Floyd's All-Pairs Shortest-Paths Algorithm</b>	<b>7</b>
2.1	Unoptimized Version of Floyd's Algorithm . . . . .	7
2.2	Upper Bound On Attainable Speedup . . . . .	8
<b>3</b>	<b>Overview of Related Works</b>	<b>12</b>
3.1	Related Works on Optimization of Cache Performance . . . . .	12
3.2	Previous Works on Floyd's Algorithm . . . . .	13
<b>4</b>	<b>Blocked Version of Floyd's Algorithm</b>	<b>15</b>
4.1	The Algorithm . . . . .	15
4.2	Optimal Blocking Factor . . . . .	19
<b>5</b>	<b>Implementation</b>	<b>20</b>
5.1	Design and Evaluation Methodology . . . . .	20
5.2	Algorithm . . . . .	21
5.3	Experimental Results . . . . .	22

# Chapter 1

## Introduction

### 1.1 Role of Cache on Optimizing Algorithms

Traditional algorithm design and analysis has, for the most part, ignored caches. Algorithms are developed, analyzed, and optimized for the RAM computer model in which a computer has a single uniformly accessible memory. Contemporary computers, however, have multiple levels of memory and the memory access time varies significantly from one memory level to the next. So, ignoring cache behavior can result in misleading conclusions regarding an algorithm's performance.

Contemporary SUN workstation has an L1 cache, L2 cache and a main memory (Figure 1). 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 about 50 cycles, and from there to L1 cache. Since the introduction of caches, main memory has continued to grow slower relative to processor cycle time. Cache miss penalties have grown to the point where good overall performance cannot be achieved without good cache performance. as a consequence of this change in computer architectures, algorithms that have been designed to minimize instruction count may not achieve the performance of algorithms that take into account both instruction count and cache performance. We can reduce run time by organizing our computations so as to minimize the number of L1 and L2 cache misses.

In our work 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 directed graph. In the process we develop some simple analytic techniques that enable us to predict the memory performance of these

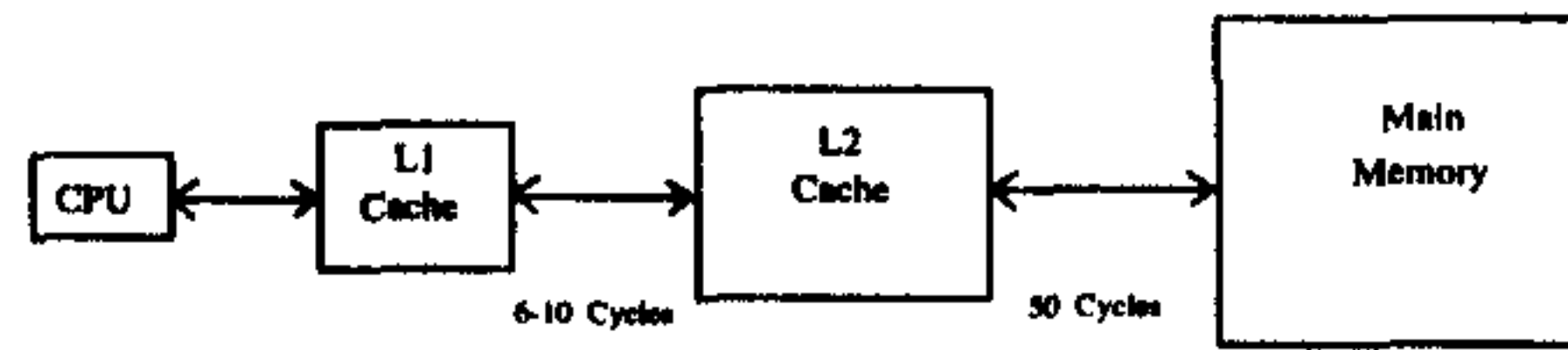


Figure 1: A Computer With Two Levels of Cache

algorithms in terms of cache misses. Cache misses cannot be analyzed precisely due to a number of factors such as variations in process scheduling and the operating system's virtual to physical page-mapping policy. In addition, the memory behaviour of an algorithm may be too complex to analyze completely. For these reasons the analyses we present are only approximate and must be validated empirically. In this paper, our experimental cache performance data is gathered using trace-driven cache simulation tool *Shade*. Cache simulations have the benefit that they are easy to run and the results are accurate.

## 1.2 Caches

In order to speed up memory accesses, small high speed memories called *caches* are placed between the processor and the main memory. Accessing the cache is typically much faster than accessing main memory. Unfortunately, since caches are smaller than main memory they can hold only a small subset of its contents. Memory accesses first consult the cache to see if it contains the desired data. If the data is found in the cache, the main memory need not be consulted and the access is considered to be a cache *hit*. If the data is not in the cache it is considered a *miss*, and the data must be loaded from main memory. On a miss, the block containing the accessed data is loaded into the cache in the hope that data in the same block will be accessed again in the future. The *hit ratio* is a measure of cache performance and is the total number of hits divided by the total number of accesses.

The major design parameters of caches are:

- **Capacity**, which is the total number of bytes that the cache can hold.
- **Block size**, which is the number of bytes that are loaded from and written to memory at a time.
- **Associativity**, which indicates the number of different locations in the cache where a particular block can be loaded. In an *N*way setassociative cache, a particular block can be loaded in *N* different cache locations. *Directmapped* caches have an associativity of one, and can load a particular block only in a single location. *Fully associative* caches are at the other extreme and can load blocks anywhere in the cache.
- **Replacement policy**, which indicates the policy for deciding which block to remove from the cache when a new block is loaded. For the *directmapped* cache the replacement policy is simply to remove the block currently residing in the required location of the cache.

In most modern machines, more than one caches are placed between the processor and main memory. These hierarchies of caches are configured with the smallest, fastest cache next to the processor and the largest, slowest cache next to main memory. The largest miss penalty is typically incurred with the cache closest to main memory and this cache is usually *directmapped*. We will assume that the cache parameters, block size and capacity, are known to the programmer.



High cache hit ratios depend on a program's stream of memory references exhibiting locality. A program exhibits *temporal locality* if there is a good chance that an accessed data item will be accessed again in the near future. A program exhibits *spatial locality* if there is good chance that subsequently accessed data items are located near each other in memory. Most programs tend to exhibit both kinds of locality and typical hit ratios are greater than 90%. With a 90% hit ratio, cutting the number of cache misses in half has the effect of raising hit ratio to 95%. This may not seem like a big improvement, but with miss penalties on the order of 100 cycles, normal programs will exhibit speedups approaching 2:1 in execution time. Accordingly, our design techniques will attempt to improve both the temporal and spatial locality of the Floyd's all-pair shortest-paths algorithm.

Cache misses are often categorized into *compulsory*, *capacity*, and *conflict* misses. *Compulsory* misses are those that occur when a block is first accessed and brought into the cache. *Capacity* misses are those caused by the fact that more blocks are accessed than can fit all at one time in the cache. *Conflict* misses are those that occur because two or more blocks that map to the same location in the cache are accessed. In this dissertation we address techniques to reduce the number of both *capacity* and *conflict* misses for the Floyd's all-pair shortest-paths algorithm.



## Chapter 2

# Floyd's All-Pairs Shortest-Paths Algorithm

### 2.1 Unoptimized Version of Floyd's Algorithm

Let  $G = (V, E)$  be a directed graph with  $n$  vertices. Let  $cost$  be the cost adjacency matrix for  $G$ . So

$$\begin{aligned} cost(i, i) &= 0, 1 \leq i \leq n ; \\ cost(i, j) &= \text{length (or cost) of edge } (i, j), \text{ if } (i, j) \in E(G) ; \\ &= \infty, \text{ if } i \neq j \text{ and } (i, j) \notin E(G). \end{aligned}$$

In the all-pairs shortest-paths problem we are to 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 is less than 0, the matrix  $A$  may be computed using dynamic programming. Let  $A^k(i, j)$  be the length of a shortest path from  $i$  to  $j$  under the constraint that the path contains 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:

$$A^0(i, j) = cost(i, j) \tag{2.1}$$

$$A^k(i, j) = \min\{A^{k-1}(i, j), A^{k-1}(i, k) + A^{k-1}(k, j)\}, k \geq 1 \tag{2.2}$$

Equations 2.1 and 2.2 lead to the algorithm of figure 2 to compute  $A$ . This algorithm is known as Floyd's algorithm. It may be shown that AllPairs makes  $n$  passes through the matrix  $A$ . During each pass, we bring in the entire matrix. The floyd's algorithm is an  $O(n^3)$  time algorithm for finding all-pairs shortest paths.

```

function AllPairs (int A, int n)
{
    /* A[i][j] = cost(i, j) initially */
    /* A[i][j] is the length of shortest i to j path on termination */
    /* n = the number of vertices in the graph */

    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]);
}

```

Figure 2: Floyd's shortest-paths algorithm

## 2.2 Upper Bound On Attainable Speedup

We compute an upper bound on the maximum speedup attainable by rearranging the computation of figure 2 so as to optimize cache usage. In computing this bound we assume that any rearrangement of the computation will not decrease the no 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 2.

The execution time of a program is given by the following equation:

$$\text{execution time} = (\text{CPU clock cycles} + \text{memory stall cycles}) \times \text{clock cycle time} \quad (2.3)$$

Where *memory stall cycles* is the number of cycles of the CPU spends waiting for a memory reference to complete.

We also know the following equations:

$$\text{CPU clock cycles} = \text{CPI} \times \text{IC} \quad (2.4)$$

$$\text{memory stall cycles} = \text{number of L1 misses} \times \text{L1 miss penalty} \quad (2.5)$$

$$\text{number of L1 misses} = IC \times \text{L1 misses per instruction} \quad (2.6)$$

$$\text{L1 misses per instruction} = \text{memory references per instruction} \times \text{L1 miss rate} \quad (2.7)$$

Where

*IC* : instruction count,

*CPI* : clock cycles per instruction,

*L1 miss penalty* : number of cycles the CPU waits when there is an L1 cache miss and

*L1 miss rate* : number of L1 misses per memory reference.

From these equations we get :

$$\begin{aligned} \text{execution time} = & (CPI \times IC + IC \times \text{L1 misses per instruction} \times \text{L1 miss penalty}) \\ & \times \text{clock cycle time} \end{aligned} \quad (2.8)$$

We also know that

$$\text{L1 miss penalty} = \text{L2 hit time} + \text{L2 miss rate} \times \text{L2 miss penalty} \quad (2.9)$$

Where

*L2 hit time* : number of cycles to load an L1 cache line from L2 cache and

*L2 miss penalty* : number of cycles needed to load an L2 cache line from main memory.

We use equations 2.8 and 2.9 to estimate the run time of Floyd's algorithm.

Now we can obtain a lower bound on the run time of a cache optimized version of Floyd's algorithm. Substituting equation 2.7 into equation 2.8 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  $IC \times \text{memory references per instruction}$  where *IC* and *memory references per instructions* are for AllPairs) yields

$$\begin{aligned} \text{execution time} \geq & (CPI \times IC + IC \times \text{memory references per instruction} \\ & \times \text{L1 miss rate} \times \text{L1 miss penalty}) \times \text{clock cycle time} \end{aligned} \quad (2.10)$$

Substituting the right side of equation 2.9 for the L1 miss penalty into equation 2.10, we get

$$\text{execution time} \geq (CPI \times IC + \text{memory references per instruction} \times IC \times L1 \text{ miss rate} \times (L2 \text{ hit time} + L2 \text{ miss rate} \times L2 \text{ miss penalty})) \times \text{clock cycle time} \quad (2.11)$$

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 Floyd's algorithm must take.

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  $3n^2$  read access to  $A$  (i.e., in the right side of the `min` statement of figure 2 ) and  $n^3$  write accesses (the left side of the `min` statement). We note that when the `min` statement of figure 2 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. So the total number of accesses to  $A$  (read and write) is  $4n^3$ . Therefore,

$$L1 \text{ miss rate} = L1 \text{ misses per } A \text{ reference} \geq n^2/8/(4n^3) = 1/(32n) \quad (2.12)$$

$$L2 \text{ miss rate} = L2 \text{ misses per } A \text{ reference} \geq n^2/16/(4n^3) = 1/(64n) \quad (2.13)$$

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 2.11, we get the following lower bound on the run time of a cache optimized version of Floyd's algorithm.

$$\text{execution time} \geq (CPI \times IC + \text{memory references per instruction} \times IC \times 1/(32n) \times (L2 \text{ hit time} + 1/(64n) \times L2 \text{ miss penalty})) \times \text{clock cycle time} \quad (2.14)$$

Dividing Equation 2.8 by Equation 2.14 yields the following bound on the speedup obtainable by optimizing cache utilization.

$$\text{speedup} \leq \frac{CPI + L1 \text{ misses per instruction} \times (L2 \text{ hit time} + L2 \text{ miss rate} \times L2 \text{ miss penalty})}{CPI + \text{memory references per instruction} \times 1/(32n) \times (L2 \text{ hit time} + 1/(64n) \times L2 \text{ miss penalty})} \quad (2.15)$$

## Chapter 3

# Overview of Related Works

### 3.1 Related Works on Optimization of Cache Performance

La Marca and Ladner [2] developed a model for a single-level direct-mapped cache. They use this model to analyse the performance of binary heaps and cache-aligned  $d$ -heaps. Table 1 gives the speedups exhibited by heapsort when using a cache-aligned 4-heap vs. a binary heap. These speedups are from [2] and are for sorting 1,000,000 uniformly distributed 32-bit integers. The reported speedup generally increases as the number of elements to be sorted increases.

Table 1: Speedup of cache-aligned 4-heap relative to a binary heap

Machine	Speedup
Pentium 90	1.70
Power PC	1.62
Alphastation 250	1.46
Sparc 20	1.38

LaMarca and Ladner [3] optimized the cache performance of several sorting methods. Table 2 gives the measured speedups for sorting 1,000,000 uniformly distributed integers using cache optimized versions of heap sort and merge sort versus traditional implementations of these sort methods.



Table 2: Speedups for cache-optimized heap sort and merge sort

Machine	Speedup	
	Memory-tuned heapsort	Tiled mergesort
Sparc 10	1.86	1.38
Power PC	1.65	1.10
Pentium PC	1.75	1.15
Dec Alpha station 250	1.58	1.90
Dec Alpha 3000/400	1.62	1.32

Lam, Rothberg and Wolf [9] have considered the cache performance of a blocked matrix multiply code relative to a traditional matrix multiply code. Table 3 gives the measured speedup of their blocked matrix multiply code for a matrix of size 300.

Table 3: Speedup of blocked matrix multiplication

Machine	Speedup
DECStation 3100	4.3
IBM RS/6000	3.0

Al-Furaih and Ranka [5, 8] have studied cache optimization methods for sorting and unstructured iterative computations.

## 3.2 Previous Works on Floyd's Algorithm

Venkataraman, Sahni and Mukhopadhyaya [8] proposed 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 [4]. Their blocked (or tiled) 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 been 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 2.0.

The Sun and SGI workstations, they used, 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 the main memory is in excess of 100 MB. For the analysis of their blocked algorithm they used data gathered using the cache simulation tool *Shade* [10, 11].

Since the L2 hit time and L2 miss penalty are architecture dependent and not available, they



used 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 they used data obtained by using the cache simulator SHADE on Floyd's algorithm. The cache simulator gave 0.35 as the memory references per instruction.

For the analysis they used the cache characteristics of the SUN Enterprise 4000/5000 that are shown in Table 4.

Table 4: Cache characteristics of the Sun Enterprise 4000/5000

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

In their blocked version of Floyd's algorithm they used 32 as the blocking factor  $B$  to get the maximum speedup on a Sun Ultra Enterprise 4000/5000 for graphs that have been between 480 and 3200 vertices.

Table 5: Speedup of cache-optimized (blocked) All-Pairs Shortest-Path Problem

No. of vertices in the graph ( $n$ )	Speedup (approx.)
$n = 480$	1.62
$n = 800$	1.68
$n = 1600$	1.70
$n = 2400$	1.76
$n = 3200$	1.82

The speedups they obtained is fairly close to the maximum possible.

# Chapter 4

## Blocked Version of Floyd's Algorithm

### 4.1 The Algorithm

We partition the cost adjacency matrix into submatrices of size  $B \times B$ , where  $B$  is called the *blocking factor*. Although this is not necessary, we assume, for simplicity, that  $B$  divides  $n$ . Figure 3(a) shows a *blocked*  $16 \times 16$  matrix, the blocking factor is  $B = 4$ .

block(1,1)	block(1,2)	block(1,3)	block(1,4)
block(2,1)	block(2,2)	block(2,3)	block(2,4)
block(3,1)	block(3,2)	block(3,3)	block(3,4)
block(4,1)	block(4,2)	block(4,3)	block(4,4)

Figure 3(a) : Blocked Matrix

Our blocked version of Floyd's algorithm will perform  $B$  iterations of the outermost loop of Figure 2 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.

In phase 1 of the first set of  $B$  iterations, Equation 2.2 is used to compute  $D^k = A^k, 1 \leq k \leq B$  for the elements in block (1, 1), we say that block (1, 1) is a self-dependent block in the first  $B$  iterations.

In phase 2 of the first  $B$  iterations a modified Equation 2.2 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 2.2 is

$$D^k(i, j) = \min\{D^{k-1}(i, j), D^B(i, k) + D^{k-1}(k, j)\}, k \geq 1 \quad (4.1)$$

where  $D^0(i, j) = A^0(i, j)$ .

For the remaining  $(*, 1)$  blocks the modified Equation 2 is

$$D^k(i, j) = \min\{D^{k-1}(i, j), D^{k-1}(i, k) + D^B(k, j)\}, k \geq 1 \quad (4.2)$$

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 as

$$D^k(i, j) = \min\{D^{k-1}(i, j), D^B(i, k) + D^B(k, j)\}, k \geq 1 \quad (4.3)$$

Phase 3 is followed by the next round of  $B$  iterations. These are also done 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

$$D^k(i, j) = \min\{D^{k-1}(i, j), D^{k-1}(i, k) + D^{k-1}(k, j)\} \quad (4.4)$$

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

$$D^k(i, j) = \min\{D^{k-1}(i, j), D^{2B}(i, k) + D^{k-1}(k, j)\} \quad (4.5)$$

The  $(*, 2)$  blocks use the following equation

$$D^k(i, j) = \min\{D^{k-1}(i, j), D^{k-1}(i, k) + D^{2B}(k, j)\} \quad (4.6)$$

And the phase 3 blocks use the equation

$$D^k(i, j) = \min\{D^{k-1}(i, j), D^{2B}(i, k) + D^{2B}(k, j)\} \quad (4.7)$$

Figure 3 shows the blocks computed in each phase when block  $(t, t)$  is the self-dependent block. The following equations are used to compute the  $(t, *)$ ,  $(*, t)$  and phase 3 blocks, respectively.

$$D^k(i, j) = \min\{D^{k-1}(i, j), D^{tB}(i, k) + D^{k-1}(k, j)\} \quad (4.8)$$

$$D^k(i, j) = \min\{D^{k-1}(i, j), D^{k-1}(i, k) + D^{tB}(k, j)\} \quad (4.9)$$

$$D^k(i, j) = \min\{D^{k-1}(i, j), D^{tB}(i, k) + D^{tB}(k, j)\} \quad (4.10)$$

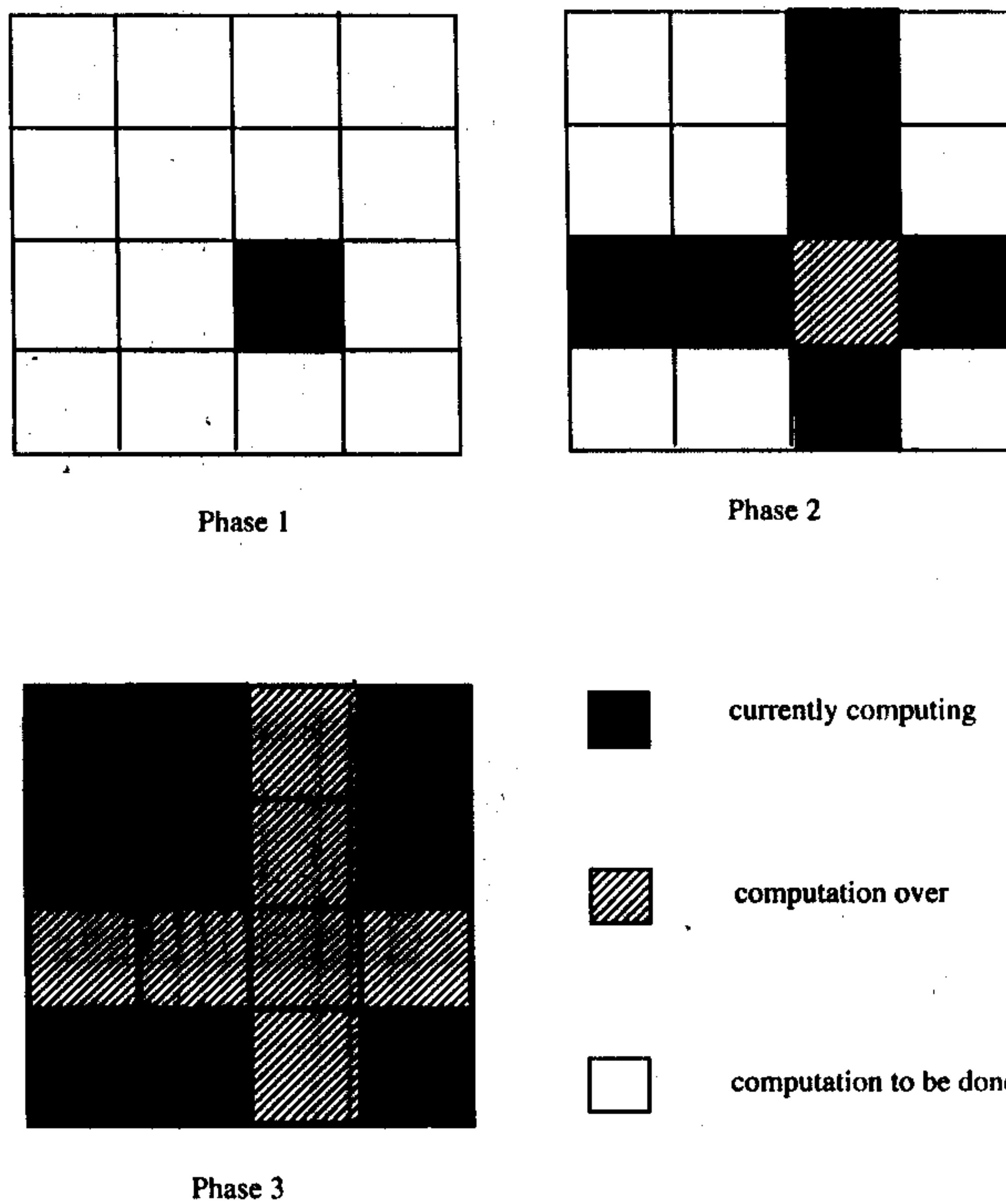


Figure 3: Blocks computed in each phase

## 4.2 Optimal Blocking Factor

When computing the  $D$  values in a block during any round (i.e., an iteration of the outermost loop) of function **BlockedAllPairs**, 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  $M$  that is on the same row or column as the self-dependent block accesses elements in  $M$  as well as elements in the self-dependent block. Therefore, 2 blocks are active during the computation for  $M$ .

For a block  $M$  that is not on the same row or column as the self-dependent block, **BlockedAllPairs** accesses elements from 3 blocks—block  $M$ , the block that is in the same row as the self-dependent block and the same column as  $M$ , and the block that is in the same column as the self-dependent block and in the same row as  $M$ .

Therefore, L1 cache misses are minimized by choosing the largest block size  $B$  such that 3 blocks of the array  $D$  can fit into L1 cache. Suppose that the elements of  $D$  are 4-byte integers and the L1 cache capacity is  $C$  bytes and each L1 cache line is  $S$  bytes long. We must choose  $B$  to be the largest integer such that  $3B^2 \times 4 \leq C \Rightarrow B \leq \sqrt{C/12}$  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 SUN Ultra Workstations  $C = 16K$  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.

# Chapter 5

## Implementation

### 5.1 Design and Evaluation Methodology

When spatial and temporal locality can be improved at no extra cost it should always be done. In this dissertation, however, we develop techniques for improving locality even when it results in an increase in the total number of executed instructions. This represents a significant departure from traditional design and optimization methodology. We take this approach in order to show how large an impact cache performance can have on overall performance. Interestingly, many of the design techniques are not particularly new. Some have already been used in optimizing compilers, in algorithms which use external storage devices, and in parallel algorithms.

As mentioned earlier we focus on three measures of performance: instruction count, cache misses, and overall performance in terms of execution time. All of the dynamic instruction counts and cache simulation results were measured using SHADE[10, 11].

*Shade* is an instruction-set simulator and custom trace generator. Application programs are executed and traced under the control of a user-supplied trace analyzer. To reduce communication costs, *Shade* and the analyzer are run in the same address space. To further improve performance, code which simulates and traces the application is dynamically generated and cached for reuse.

Here we tried to develop the algorithm in such a manner that it exhibits *temporal locality* i.e., there is a good chance that an accessed data item will be accessed again in the near future. We tried to increase the hit ratio as much as possible. This may not seem like a big improvement, but with miss penalties on the order of 100 cycles, normal programs will exhibit significant speedups. Accordingly, our design techniques will attempt to improve the temporal locality of the Floyd's



all-pair shortest-paths algorithm.

## 5.2 Algorithm

**Input :** The directed graph with  $n$  vertices i.e., The cost matrix  $A(i,j)$ .

**Output :** The shortest-path between any two vertices i.e., The matrix  $D(i, j)$  where  $(i,j)$ th entry is the shortest path between the  $i$ th and the  $j$ th vertices.

```
function BlockedAllPairs (int A, int n, int B)
{
    /* A[i][j] = cost(i, j) initially,
    A[i][j] is the length of shortest i to j path on termination,
    n = the number of vertices in the graph,
    B = the blocking factor. */

    for (pass = 0; pass < n; pass +B)
    {
        /* self-dependent block (r, r) */
        for (k = pass; k < pass +B; k++)
            for (all i and j in the self-dependent block)
                A[i][j] = min(A[i][j], A[i][k] + A[k][j]);

        /* remaining blocks */
        do the following for remaining blocks, one block at a time and in the order:
            phase 2 blocks which are in the same row as the block (r, r) and
            the direction of computation is from left to right;
            phase 2 blocks which are in the same column as the block (r, r) and
            the direction of computation is from bottom to top;
            phase 3 blocks and the direction of computation is starting at top-left
            or top-right corner, then in a snake like manner.

        for (k = pass; k < pass +B; k++)
```

```

    for (all i and j in the current block)
        A[i][j] = min(A[i][j], A[i][k] + A[k][j]);
    }
}

```

Figure 4: Blocked Version of Floyd's All-Pairs Shortest-Paths Algorithm

### 5.3 Experimental Results

Since we intend to declare  $i$ ,  $j$ ,  $k$  as register variables, 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 have used the different cache characteristics. And we took matrices of different sizes.

Along with our algorithm we also implemented the algorithm given by G. Venkataraman, S. Sahni, and S. Mukhopadhaya in their paper [8]. We have studied a comparison between them.

Table 6: Cache characteristics (L2 includes L1).

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

We used the cache characteristics of Table 6 and got the following results for the two algorithms which are in the Table 7 and Table 8 for  $n = 480$  and in Table 9 Table 10 for  $n = 800$ :

Table 7: Output of our Algorithm given by Figure 4 when  $n = 480$ .

Cache	I-misses	D-misses
L1	10345380	27596567
L2	11825	2232688

Table 8: Output of the Algorithm given by [8] when  $n = 480$ .

Cache	I-misses	D-misses
L1	12148426	29909072
L2	1873	1078267

Table 9: Output of our Algorithm given by Figure 4 when  $n = 800$ .

Cache	I-misses	D-misses
L1	45518361	91715827
L2	1829	6188253

Table 10: Output of the Algorithm given by [8] when  $n = 800$ .

Cache	I-misses	D-misses
L1	48534963	96728129
L2	1888	6819678

Next we took the cache characteristics given by Table 11 bellow. And similarly got the outputs for different matrix size ( $n$ ) in the Tables 12-15 :

Table 11: Cache characteristics (L2 includes L1).

Cache type	Associativity	Replace. algo	Cache size	Line size
L1-Instruction	Direct mapped	N/A	16KB	32 bytes
L1-Data	Fully-Associative	LRU	16KB	32 bytes
L2	Direct mapped	N/A	4MB	64 bytes

Table 12: Output of our Algorithm given by Figure 4 when  $n = 320$ .

Cache	I-misses	D-misses
L1	439037	1690620
L2	1286	998570

Table 13: Output of the Algorithm given by [8] when  $n = 320$ .

Cache	I-misses	D-misses
L1	661185	1903571
L2	1325	484866

Table 14: Output of our Algorithm given by Figure 4 when  $n = 480$ .

Cache	I-misses	D-misses
L1	952504	4253515
L2	1588	2232576

Table 15: Output of the Algorithm given by [8] when  $n = 480$ .

Cache	I-misses	D-misses
L1	1464211	4731940
L2	1636	1078164

Next we took the cache characteristics given by Table 16 bellow. And similarly got the outputs for different matrix size ( $n$ ) in the Tables 17-20 :

Table 16: Cache characteristics (L2 includes L1).

Cache type	Associativity	Replace. algo	Cache size	Line size
L1	4-way	LRU	16KB	32 bytes
L2	Direct mapped	N/A	4MB	64 bytes

Table 17: Output of our Algorithm given by Figure 4 when  $n = 320$ .

Cache	I-misses	D-misses
L1	26250	2215459
L2	1281	320378

Table 18: Output of the Algorithm given by [8] when  $n = 320$ .

Cache	I-misses	D-misses
L1	43949	1977504
L2	1327	833864

Table 19: Output of our Algorithm given by Figure 4 when  $n = 480$ .

Cache	I-misses	D-misses
L1	33343	481779
L2	1830	709358

Table 20: Output of the Algorithm given by [8] when  $n = 480$ .

Cache	I-misses	D-misses
L1	45280	4273124
L2	1876	1863702

Next we took the cache characteristics given by Table 21 bellow. And similarly got the outputs for the matrix size  $n = 480$  in the Tables 22-23 :

Table 21: Cache characteristics (L2 does NOT include L1).

Cache type	Associativity	Replace. algo	Cache size	Line size
L1	Direct mapped	N/A	16KB	32 bytes
L2	4-way	LRU	4MB	64 bytes

Table 22: Output of our Algorithm given by Figure 4 when  $n = 480$ .

Cache	I-misses	D-misses
L1	10354846	27619414
L2	1000	2231754

Table 23: Output of the Algorithm given by [8] when  $n = 480$ .

Cache	I-misses	D-misses
L1	12151588	29925376
L2	1056	1077348

So from the Tables above we can see that we are getting better performances in the L1 cache both for the instructions and for the data. But in the L2 cache we are getting better performances for instructions but not for the data except one cache characteristics in the Table 16.

# Bibliography

- [1] D. Patterson and J.L. Hennessy, *Computer Architecture: A Quantitative Analysis*. Morgan Kaufmann, San mateo, CA, 1996.
- [2] A. LaMarca and R.E. Ladner, "The influences of caches on the performance of heaps". *The ACM Journal of Experimental Algorithms*, 1(4),1996.
- [3] A. LaMarca and R.E. Ladner, "The influences of caches on the performance of sorting". *The ACM-SIAM Symposium on Discrete Algorithms*, pages 370–379, New Orleans, Louisiana, 5-7 January, 1997.
- [4] E. Horowitz, S. Sahni, and S. Rajasekaran, *Computer Algorithms*. Computer Science Press, New York, 1998.
- [5] I. Al-Furaih and S. Ranka, "Memory hierarchy management for iterative graph structures". *Proc. 12th International Parallel Processing Symposium 1998. (IPPS98)*, Orlando, Florida.
- [6] I. Al-Furaih and S. 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] T.H. Cormen, C.E. Leiserson, and R.L. Rivest, *Introduction to Algorithms*, McGraw-Hill, Massachusetts, 1994.
- [8] G. Venkataraman, S. Sahni, and S. Mukhopadhyaya, "A Blocked All-Pairs Shortest-Paths Algorithm", *Proc. 7th Scandinavian Workshops on Algorithm Theory (SWAT2000)*, July 5-7, 2000, Bergen, Norway.
- [9] M.S. Lam, E.E. Rothberg and M.E. Wolf, *The cache performance and optimizations of blocked algorithms*, ACM, 26:63-74, 1991.

- [10] Introduction to Shade V5.33A, *Sun Microsystems Laboratories Inc*, Mountain View, CA 94043. 1998.
- [11] Shade User's Manual V5.33A, *Sun Microsystems Laboratories Inc*, Mountain View, CA 94043. 1998.