Cache-Oblivious Algorithms

by

Harald Prokop

Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY.

June 1999

© Massachusetts Institute of Technology 1999.
All rights reserved.

ENG
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
JUL 1 5 1999
LIBRARIES
WITHDRAWN FROM MIT LIBRARIES

Author [illegible]
Department of Electrical Engineering and Computer Science
May 21, 1999

Certified by [illegible]
Charles E. Leiserson
Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by [illegible]
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

Cache-Oblivious Algorithms

by
Harald Prokop

*Submitted to the
Department of Electrical Engineering and Computer Science
on May 21, 1999 in partial fulfillment of the
requirements for the degree of Master of Science.*

Abstract

This thesis presents "cache-oblivious" algorithms that use asymptotically optimal amounts of work, and move data asymptotically optimally among multiple levels of cache. An algorithm is ***cache oblivious*** if no program variables dependent on hardware configuration parameters, such as cache size and cache-line length need to be tuned to minimize the number of cache misses.

We show that the ordinary algorithms for matrix transposition, matrix multiplication, sorting, and Jacobi-style multipass filtering are not cache optimal. We present algorithms for rectangular matrix transposition, FFT, sorting, and multipass filters, which are asymptotically optimal on computers with multiple levels of caches. For a cache with size Z and cache-line length L, where $Z = \Omega(L^2)$, the number of cache misses for an $m \times n$ matrix transpose is $\Theta(1 + mn/L)$. The number of cache misses for either an n-point FFT or the sorting of n numbers is $\Theta(1 + (n/L)(1 + \log_Z n))$. The cache complexity of computing n time steps of a Jacobi-style multipass filter on an array of size n is $\Theta(1 + n/L + n^2/ZL)$. We also give an $\Theta(mnp)$-work algorithm to multiply an $m \times n$ matrix by an $n \times p$ matrix that incurs $\Theta(m + n + p + (mn + np + mp)/L + mnp/L\sqrt{Z})$ cache misses.

We introduce an "ideal-cache" model to analyze our algorithms, and we prove that an optimal cache-oblivious algorithm designed for two levels of memory is also optimal for multiple levels. We further prove that any optimal cache-oblivious algorithm is also optimal in the previously studied HMM and SUMH models. Algorithms developed for these earlier models are perforce ***cache-aware***: their behavior varies as a function of hardware-dependent parameters which must be tuned to attain optimality. Our cache-oblivious algorithms achieve the same asymptotic optimality on all these models, but without any tuning.

Thesis Supervisor: Charles E. Leiserson
Title: Professor of Computer Science and Engineering

Acknowledgments

I am extremely grateful to my advisor Charles E. Leiserson. He has greatly helped me both in technical and nontechnical matters. Without his insight, suggestions, and excitement, this work would have never taken place. Charles also helped with the write-up of the paper on which this thesis is based. It is amazing how patiently Charles can rewrite a section until it has the quality he expects.

Most of the work presented in this thesis has been a team effort. I would like to thank those with whom I collaborated: Matteo Frigo, Charles E. Leiserson, and Sridhar Ramachandran. Special thanks to Sridhar who patiently listened to all my (broken) attempts to prove that cache-oblivious sorting is impossible.

I am privileged to be part of the stimulating and friendly environment of the Supercomputing Technologies research group of the MIT Laboratory of Computer Science. I would like to thank all the members of the group, both past and present, for making it a great place to work. Many thanks to Don Dailey, Phil Lisiecki, Dimitris Mitsouras, Alberto Medina, Bin Song, and Volker Strumpen.

The research in this thesis was supported in part by the Defense Advanced Research Projects Agency (DARPA) under Grant F30602-97-1-0270 and by a fellowship from the Cusanuswerk, Bonn, Germany.

Finally, I want to thank my family for their love, encouragement, and help, which kept me going during the more difficult times.

HARALD PROKOP
Cambridge, Massachusetts
May 21, 1999

Contents

SECTION 1

Introduction

Resource-oblivious algorithms that nevertheless use resources efficiently offer advantages of simplicity and portability over resource-aware algorithms whose resource usage must be programmed explicitly. In this thesis, we study cache resources, specifically, the hierarchy of memories in modern computers. We exhibit several "cache-oblivious" algorithms that use cache as effectively as "cache-aware" algorithms.

Before discussing the notion of cache obliviousness, we introduce the (Z, L) ***ideal-cache model*** to study the cache complexity of algorithms. This model, which is illustrated in Figure 1-1, consists of a computer with a two-level memory hierarchy consisting of an ideal (data) cache of Z words and an arbitrarily large main memory. Because the actual size of words in a computer is typically a small, fixed size (4 bytes, 8 bytes, etc.), we shall assume that word size is constant; the particular constant does not affect our asymptotic analyses. The cache is partitioned into ***cache lines,*** each consisting of L consecutive words that are always moved together between cache and main memory. Cache designers typically use $L > 1$, banking on spatial locality to amortize the overhead of moving the cache line. We shall generally assume in this thesis that the cache is ***tall***:

$$Z = \Omega(L^2)\,, \tag{1.1}$$

which is usually true in practice.

The processor can only reference words that reside in the cache. If the referenced word belongs to a line already in cache, a ***cache hit*** occurs, and the word is

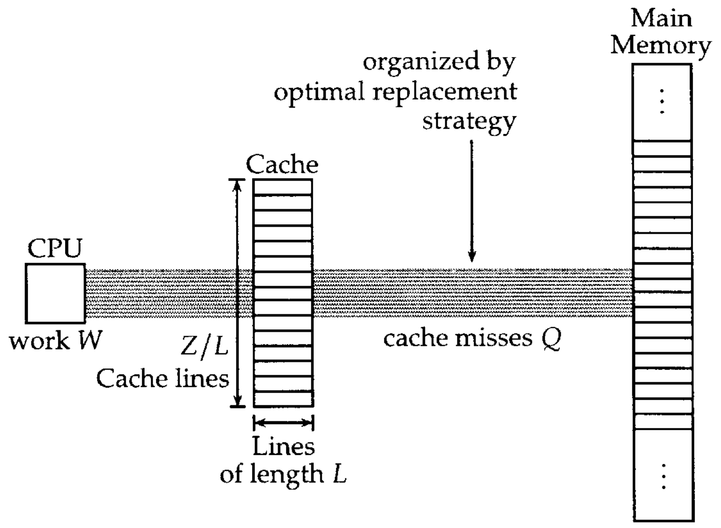


Figure 1-1: The ideal-cache model

delivered to the processor. Otherwise, a ***cache miss*** occurs, and the line is fetched into the cache. The ideal cache is ***fully associative*** [24, Ch. 5]: Cache lines can be stored anywhere in the cache. If the cache is full, a cache line must be evicted. The ideal cache uses the optimal off-line strategy of replacing the cache line whose next access is farthest in the future [7], and thus it exploits temporal locality perfectly.

An algorithm with an input of size n is measured in the ideal-cache model in terms of its ***work complexity*** $W(n)$—its conventional running time in a RAM model [4]—and its ***cache complexity*** $Q(n; Z, L)$—the number of cache misses it incurs as a function of the size Z and line length L of the ideal cache. When Z and L are clear from context, we denote the cache complexity as simply $Q(n)$ to ease notation.

We define an algorithm to be ***cache aware*** if it contains parameters (set at either compile-time or runtime) that can be tuned to optimize the cache complexity for the particular cache size and line length. Otherwise, the algorithm is ***cache oblivious***. Historically, good performance has been obtained using cache-aware algorithms, but we shall exhibit several cache-oblivious algorithms for fundamental problems that are asymptotically as efficient as their cache-aware counterparts.

To illustrate the notion of cache awareness, consider the problem of multiplying two $n \times n$ matrices A and B to produce their $n \times n$ product C. We assume that the three matrices are stored in row-major order, as shown in Figure 2-1(a). We further assume that n is "big," i.e., $n > L$, in order to simplify the analysis. The conventional way to multiply matrices on a computer with caches is to use a ***blocked*** algorithm [22, p. 45]. The idea is to view each matrix M as consisting of $(n/s) \times (n/s)$ submatrices M_{ij} (the blocks), each of which has size $s \times s$, where s is a tuning parameter. The following algorithm implements this strategy:

```
BLOCK-MULT$(A, B, C, n)$
1  for $i \leftarrow 1$ to $n/s$
2      do for $j \leftarrow 1$ to $n/s$
3             do for $k \leftarrow 1$ to $n/s$
4                    do ORD-MULT$(A_{ik}, B_{kj}, C_{ij}, s)$
```

where ORD-MULT(A, B, C, s) is a subroutine that computes $C \leftarrow C + AB$ on $s \times s$ matrices using the ordinary $O(s^3)$ algorithm (see Section 7.1). (This algorithm assumes for simplicity that s evenly divides n. In practice, s and n need have no special relationship, which yields more complicated code in the same spirit.)

Depending on the cache size of the machine on which BLOCK-MULT is run, the parameter s can be tuned to make the algorithm run fast, and thus BLOCK-MULT is a cache-aware algorithm. To minimize the cache complexity, we choose s as large as possible such that the three $s \times s$ submatrices simultaneously fit in cache. An $s \times s$ submatrix is stored on $\Theta(s + s^2/L)$ cache lines. From the tall-cache assumption (1.1), we can see that $s = \Theta(\sqrt{Z})$. Thus, each of the calls to ORD-MULT runs with at most $Z/L = \Theta(s + s^2/L)$ cache misses needed to bring the three matrices into the cache. Consequently, the cache complexity of the entire algorithm is $\Theta(n + n^2/L + (n/\sqrt{Z})^3(Z/L)) = \Theta(n + n^2/L + n^3/L\sqrt{Z})$, since the algorithm must read n^2 elements, which reside on $\lceil n^2/L \rceil$ cache lines.

The same bound can be achieved using a simple cache-oblivious algorithm that requires no tuning parameters such as the s in BLOCK-MULT. We present such an algorithm, which works on general rectangular matrices, in Section 2. The problems of computing a matrix transpose and of performing an FFT also succumb to remarkably simple algorithms, which are described in Section 3. Cache-oblivious sorting poses a more formidable challenge. In Sections 4 and 5, we present two sorting algorithms, one based on mergesort and the other on distribution sort, both of which are optimal. Section 6 compares an optimal recursive algorithm with an "ordinary" iterative algorithm, both of which compute a multipass filter over one-dimensional data. It also provides some brief empirical results for this problem. In Section 7, we show that the ordinary algorithms for matrix transposition, matrix multiplication, and sorting are not cache optimal.

The ideal-cache model makes the perhaps-questionable assumption that memory is managed automatically by an optimal cache replacement strategy. Although the current trend in architecture does favor automatic caching over programmer-specified data movement, Section 8 addresses this concern theoretically. We show that the assumptions of two hierarchical memory models in the literature, in which memory movement is programmed explicitly, are actually no weaker than ours. Specifically, we prove (with only minor assumptions) that optimal cache-oblivious algorithms in the ideal-cache model are also optimal in the hierarchical memory model (HMM) [1] and in the serial uniform memory hierarchy (SUMH) model [5, 42]. Section 9 discusses related work, and Section 10 offers some concluding remarks.

Many of the results in this thesis are based on a joint paper [21] coauthored by Matteo Frigo, Charles E. Leiserson, and Sridhar Ramachandran.

SECTION 2

Matrix multiplication

This section describes and analyzes an algorithm for multiplying an $m \times n$ matrix by an $n \times p$ matrix cache-obliviously using $\Theta(mnp)$ work and incurring $\Theta(m + n + p + (mn + np + mp)/L + mnp/L\sqrt{Z})$ cache misses. These results require the tall-cache assumption (1.1) for matrices stored in row-major layout format, but the assumption can be relaxed for certain other layouts. We also show that Strassen's algorithm [38] for multiplying $n \times n$ matrices, which uses $\Theta(n^{\log_2 7})$ work, incurs $\Theta(1 + n^2/L + n^{\log_2 7}/L\sqrt{Z})$ cache misses.

The following algorithm extends the optimal divide-and-conquer algorithm for square matrices described in [9] to rectangular matrices. To multiply an $m \times n$ matrix A by an $n \times p$ matrix B, the algorithm halves the largest of the three dimensions and recurs according to one of the following three cases:

$$AB = \begin{pmatrix} A_1 \\ A_2 \end{pmatrix} B = \begin{pmatrix} A_1 B \\ A_2 B \end{pmatrix} , \tag{2.1}$$

$$AB = \begin{pmatrix} A_1 & A_2 \end{pmatrix} \begin{pmatrix} B_1 \\ B_2 \end{pmatrix} = A_1 B_1 + A_2 B_2 , \tag{2.2}$$

$$AB = A \begin{pmatrix} B_1 & B_2 \end{pmatrix} = \begin{pmatrix} AB_1 & AB_2 \end{pmatrix} . \tag{2.3}$$

In case (2.1), we have $m \geq \max\{n, p\}$. Matrix A is split horizontally, and both halves are multiplied by matrix B. In case (2.2), we have $n \geq \max\{m, p\}$. Both matrices are split, and the two halves are multiplied. In case (2.3), we have $p \geq \max\{m, n\}$. Matrix B is split vertically, and each half is multiplied by A. For square matrices, these three cases together are equivalent to the recursive multiplication

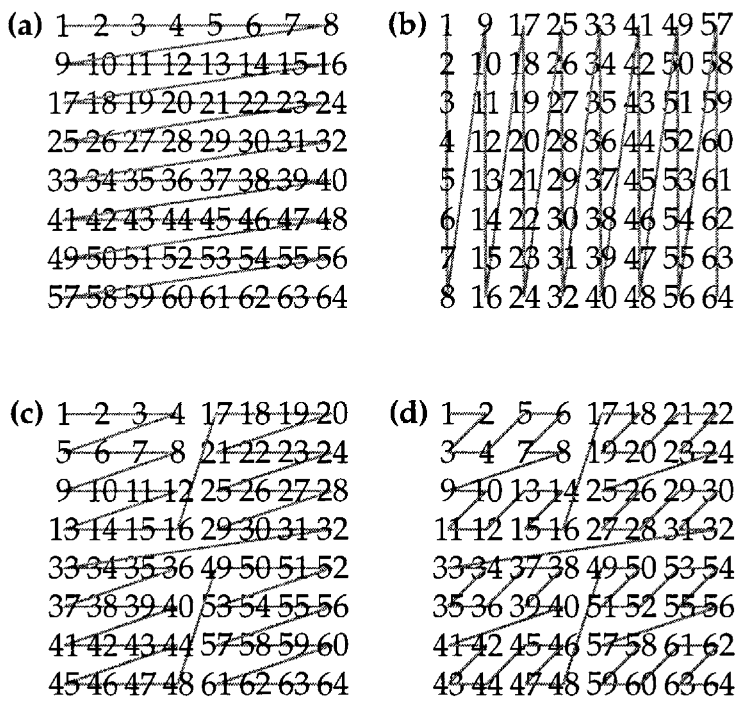


Figure 2-1: Layout of a 16×16 matrix in **(a)** row major, **(b)** column major, **(c)** 4×4-blocked, and **(d)** bit-interleaved layouts.

algorithm described in [9]. The base case occurs when $m = n = p = 1$, in which case the two elements are multiplied and added into the result matrix.

Although this straightforward divide-and-conquer algorithm contains no tuning parameters, it uses cache optimally. To analyze the algorithm, we assume that the three matrices are stored in row-major order, as shown in Figure 2-1(a). Intuitively, the cache-oblivious divide-and-conquer algorithm uses the cache effectively, because once a subproblem fits into the cache, its smaller subproblems can be solved in cache with no further cache misses.

Theorem 1 *The cache-oblivious matrix multiplication algorithm uses $\Theta(mnp)$ work and incurs $\Theta(m + n + p + (mn + np + mp)/L + mnp/L\sqrt{Z})$ cache misses when multiplying an $m \times n$ by an $n \times p$ matrix.*

Proof. It can be shown by induction that the work of this algorithm is $\Theta(mnp)$. To analyze the cache misses, let α be a constant sufficiently small that three submatrices of size $m' \times n'$, $n' \times p'$, and $m' \times p'$, where $\max\{m', n', p'\} \leq \alpha\sqrt{Z}$, all fit completely in the cache. We distinguish the following four cases cases depending on the initial size of the matrices.

Case I: $m, n, p > \alpha\sqrt{Z}$.

This case is the most intuitive. The matrices do not fit in cache, since all dimensions are "big enough." The cache complexity of matrix multiplication can be described by the recurrence

$$Q(m,n,p) \leq \begin{cases} \Theta((mn+np+mp)/L) & \text{if } (mn+np+mp) \leq \alpha Z\,, \\ 2Q(m/2,n,p)+O(1) & \text{otherwise and if } m \geq n \text{ and } m \geq p\,, \\ 2Q(m,n/2,p)+O(1) & \text{otherwise and if } n > m \text{ and } n \geq p\,, \\ 2Q(m,n,p/2)+O(1) & \text{otherwise}\,. \end{cases} \tag{2.4}$$

The base case arises as soon as all three submatrices fit in cache. The total number of lines used by the three submatrices is $\Theta((mn+np+mp)/L)$. The only cache misses that occur during the remainder of the recursion are the $\Theta((mn+np+mp)/L)$ cache misses required to bring the matrices into cache. In the recursive cases, when the matrices do not fit in cache, we pay for the cache misses of the recursive calls, which depend on the dimensions of the matrices, plus $O(1)$ cache misses for the overhead of manipulating submatrices. The solution to this recurrence is $Q(m,n,p) = \Theta(mnp/L\sqrt{Z})$.

Case II: ($m \leq \alpha\sqrt{Z}$ and $n,p > \alpha\sqrt{Z}$) OR ($m \leq \alpha\sqrt{Z}$ and $n,p > \alpha\sqrt{Z}$) OR ($p \leq \alpha\sqrt{Z}$ and $m,n > \alpha\sqrt{Z}$).

Here, we shall present the case where $m \leq \alpha\sqrt{Z}$ and $n,p > \alpha\sqrt{Z}$. The proofs for the other cases are only small variations of this proof. The multiplication algorithm always divides n or p by 2 according to cases (2.2) and (2.3). At some point in the recursion, both are small enough that the whole problem fits into cache. The number of cache misses can be described by the recurrence

$$Q(m,n,p) \leq \begin{cases} \Theta(1+n+np/L+m) & \text{if } n,p \in [\alpha\sqrt{Z}/2, \alpha\sqrt{Z}]\,, \\ 2Q(m,n/2,p)+O(1) & \text{otherwise and if } n \geq p\,, \\ 2Q(m,n,p/2)+O(1) & \text{otherwise}\,. \end{cases}$$

The solution to this recurrence is $\Theta(np/L + mnp/L\sqrt{Z})$.

Case III: ($n,p \leq \alpha\sqrt{Z}$ and $m > \alpha\sqrt{Z}$) OR ($m,p \leq \alpha\sqrt{Z}$ and $n > \alpha\sqrt{Z}$) OR ($m,n \leq \alpha\sqrt{Z}$ and $p > \alpha\sqrt{Z}$).

In each of these cases, one of the matrices fits into cache, and the others do not. Here, we shall present the case where $n,p \leq \alpha\sqrt{Z}$ and $m > \alpha\sqrt{Z}$. The other cases can be proven similarly. The multiplication algorithm always divides m by 2 according to case (2.1). At some point in the recursion, m is in the range $\alpha\sqrt{Z}/2 \le m \le \alpha\sqrt{Z}$, and the whole problem fits in cache. The number cache misses can be described by the recurrence

$$Q(m,n) \le \begin{cases} \Theta(1+m) & \text{if } m \in [\alpha\sqrt{Z}/2, \alpha\sqrt{Z}] \,, \\ 2Q(m/2,n,p) + O(1) & \text{otherwise} \,; \end{cases}$$

whose solution is $Q(m,n,p) = \Theta(m + mnp/L\sqrt{Z})$.

Case IV: $m,n,p \le \alpha\sqrt{Z}$.

From the choice of α, all three matrices fit into cache. The matrices are stored on $\Theta(1 + mn/L + np/L + mp/L)$ cache lines. Therefore, we have $Q(m,n,p) = \Theta(1 + (mn + np + mp)/L)$. $\square$

We require the tall-cache assumption (1.1) in these analyses, because the matrices are stored in row-major order. Tall caches are also needed if matrices are stored in column-major order (Figure 2-1(b)), but the assumption that $Z = \Omega(L^2)$ can be relaxed for certain other matrix layouts. The $s \times s$-blocked layout (Figure 2-1(c)), for some tuning parameter s, can be used to achieve the same bounds with the weaker assumption that the cache holds at least some sufficiently large constant number of lines. The cache-oblivious bit-interleaved layout (Figure 2-1(d)) has the same advantage as the blocked layout, but no tuning parameter need be set, since submatrices of size $\Theta(\sqrt{L} \times \sqrt{L})$ are cache-obliviously stored on one cache line. The advantages of bit-interleaved and related layouts have been studied in [18] and [12, 13]. One of the practical disadvantages of bit-interleaved layouts is that index calculations on conventional microprocessors can be costly.

For square matrices, the cache complexity $Q(n) = \Theta(n + n^2/L + n^3/L\sqrt{Z})$ of the cache-oblivious matrix multiplication algorithm is the same as the cache complexity of the cache-aware BLOCK-MULT algorithm and also matches the lower bound by Hong and Kung [25]. This lower bound holds for all algorithms that execute the $\Theta(n^3)$ operations given by the definition of matrix multiplication

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} \,.$$

No tight lower bounds for the general problem of matrix multiplication are known.

By using an asymptotically faster algorithm, such as Strassen's algorithm [38] or one of its variants [45], both the work and cache complexity can be reduced. When multiplying $n \times n$ matrices, Strassen's algorithm, which is cache oblivious, requires only 7 recursive multiplications of $n/2 \times n/2$ matrices and a constant number of matrix additions, yielding the recurrence

$$Q(n) \le \begin{cases} \Theta(1+n+n^2/L) & \text{if } n^2 \le \alpha Z\,, \\ 7Q(n/2) + O(n^2/L) & \text{otherwise}\,; \end{cases} \tag{2.5}$$

where α is a sufficiently small constant. The solution to this recurrence is $\Theta(n + n^2/L + n^{\log_2 7}/L\sqrt{Z})$.

Summary

In this section we have used the ideal-cache model to analyze two algorithms for matrix multiplication. We have described an efficient cache-oblivious algorithm for rectangular matrix multiplication and analyzed the cache complexity of Strassen's algorithm.

SECTION 3

Matrix transposition and FFT

This section describes an optimal cache-oblivious algorithm for transposing an $m \times n$ matrix. The algorithm uses $\Theta(mn)$ work and incurs $\Theta(1 + mn/L)$ cache misses. Using matrix transposition as a subroutine, we convert a variant [44] of the "six-step" fast Fourier transform (FFT) algorithm [6] into an optimal cache-oblivious algorithm. This FFT algorithm uses $O(n \lg n)$ work and incurs $O(1 + (n/L)(1 + \log_Z n))$ cache misses.

The problem of matrix transposition is defined as follows. Given an $m \times n$ matrix stored in a row-major layout, compute and store A^T into an $n \times m$ matrix B also stored in a row-major layout. The straightforward algorithm for transposition that employs doubly nested loops incurs $\Theta(mn)$ cache misses on one of the matrices when $mn \gg Z$, which is suboptimal.

Optimal work and cache complexities can be obtained with a divide-and-conquer strategy, however. If $n \geq m$, we partition

$$A = (A_1 \; A_2) \, , \quad B = \begin{pmatrix} B_1 \\ B_2 \end{pmatrix} .$$

Then, we recursively execute $\textsc{Transpose}(A_1, B_1)$ and $\textsc{Transpose}(A_2, B_2)$. Alternatively, if $m > n$, we divide matrix A horizontally and matrix B vertically and likewise perform two transpositions recursively. The next two theorems provide upper and lower bounds on the performance of this algorithm.

Theorem 2 *The cache-oblivious matrix-transpose algorithm involves $\Theta(mn)$ work and incurs $\Theta(1 + mn/L)$ cache misses for an $m \times n$ matrix.*

Proof. That the algorithm uses $\Theta(mn)$ work can be shown by induction. For the cache analysis, let $Q(m,n)$ be the cache complexity of transposing an $m \times n$ matrix. We assume that the matrices are stored in row-major order, the column-major case having a similar analysis.

Let α be a constant sufficiently small that two submatrices of size $m' \times n'$ and $n' \times m'$, where $\max\{m',n'\} \le \alpha L$, fit completely in the cache. We distinguish the following three cases.

Case I: $\max\{m,n\} \le \alpha L$.

Both the matrices fit in $O(1) + 2mn/L$ lines. From the choice of α, the number of lines required is at most Z/L, which implies $Q(m,n) = \Theta(1 + mn/L)$.

Case II: $m \le \alpha L < n$ OR $n \le \alpha L < m$.

For this case, we assume without loss of generality that $m \le \alpha L < n$. The case $n \le \alpha L < m$ is analogous. The transposition algorithm divides the greater dimension n by 2 and performs divide-and-conquer. At some point in the recursion, n is in the range $\alpha L/2 \le n \le \alpha L$, and the whole problem fits in cache. Because the layout is row-major, at this point the input array has n rows, m columns, and it is laid out in contiguous locations, thus requiring at most $O(1 + nm/L)$ cache misses to be read. The output array consists of nm elements in m rows, where in the worst case every row lies on a different cache line. Consequently, we incur at most $O(m + nm/L)$ for writing the output array. Since $n \ge \alpha L/2$, the total cache complexity for this base case is $O(1+m)$.

These observations yield the recurrence

$$Q(m,n) \le \begin{cases} \Theta(1+m) & \text{if } n \in [\alpha L/2, \alpha L]\,, \\ 2Q(m,n/2) + O(1) & \text{otherwise}\,; \end{cases}$$

whose solution is $Q(m,n) = \Theta(1 + mn/L)$.

Case III: $m, n > \alpha L$.

As in Case II, at some point in the recursion, both n and m fall in the interval $[\alpha L/2, \alpha L]$. The whole problem then fits into cache, and it can be solved with at most $O(m + n + mn/L)$ cache misses.

The cache complexity thus satisfies the recurrence

$$Q(m,n) \le \begin{cases} \Theta(m+n+mn/L) & \text{if } m,n \in [\alpha L/2, \alpha L]\,, \\ 2Q(m/2,n) + O(1) & \text{if } m \ge n\,, \\ 2Q(m,n/2) + O(1) & \text{otherwise}\,; \end{cases}$$

whose solution is $Q(m,n) = \Theta(1 + mn/L)$. □

Theorem 3 *The cache-oblivious matrix-transpose algorithm is asymptotically optimal.*

Proof. For an $m \times n$ matrix, the matrix-transposition algorithm must write to mn distinct elements, which occupy at least $\lceil mn/L \rceil = \Omega(1 + mn/L)$ cache lines. $\square$

As an example application of the cache-oblivious transposition algorithm, the rest of this section describes and analyzes a cache-oblivious algorithm for computing the discrete Fourier transform of a complex array of n elements, where n is an exact power of 2. The basic algorithm is the well-known "six-step" variant [6, 44] of the Cooley-Tukey FFT algorithm [15]. By using the cache-oblivious transposition algorithm, however, we can make the FFT cache oblivious, and its performance matches the lower bound by Hong and Kung [25].

Recall that the ***discrete Fourier transform (DFT)*** of an array X of n complex numbers is the array Y given by

$$Y[i] = \sum_{j=0}^{n-1} X[j]\omega_n^{-ij} , \tag{3.1}$$

where $\omega_n = e^{2\pi\sqrt{-1}/n}$ is a primitive nth root of unity, and $0 \leq i < n$.

Many known algorithms evaluate Equation (3.1) in time $O(n \lg n)$ for all integers n [17]. In this thesis, however, we assume that n is an exact power of 2, and compute Equation (3.1) according to the Cooley-Tukey algorithm, which works recursively as follows. In the base case where $n = O(1)$, we compute Equation (3.1) directly. Otherwise, for any factorization $n = n_1 n_2$ of n, we have

$$Y[i_1 + i_2 n_1] = \sum_{j_2=0}^{n_2-1} \left[\left(\sum_{j_1=0}^{n_1-1} X[j_1 n_2 + j_2]\omega_{n_1}^{-i_1 j_1} \right) \omega_n^{-i_1 j_2} \right] \omega_{n_2}^{-i_2 j_2} . \tag{3.2}$$

Observe that both the inner and outer summations in Equation (3.2) are DFT's. Operationally, the computation specified by Equation (3.2) can be performed by computing n_2 transforms of size n_1 (the inner sum), multiplying the result by the factors $\omega_n^{-i_1 j_2}$ (called the ***twiddle factors*** [17]), and finally computing n_1 transforms of size n_2 (the outer sum).

We choose n_1 to be $2^{\lceil (\lg n)/2 \rceil}$ and n_2 to be $2^{\lfloor (\lg n)/2 \rfloor}$. The recursive step then operates as follows:

1. Pretend that the input is a row-major $n_1 \times n_2$ matrix A. Transpose A in place, i.e., use the cache-oblivious algorithm to transpose A onto an auxiliary array B, and copy B back onto A. (If $n_1 = 2n_2$, consider the matrix to be made up of records containing two elements.)

2. At this stage, the inner sum corresponds to a DFT of the n_2 rows of the transposed matrix. Compute these n_2 DFT's of size n_1 recursively. Observe that, because of the previous transposition, we are transforming a contiguous array of elements.

3. Multiply A by the twiddle factors, which can be computed on the fly with no extra cache misses.

4. Transpose A in-place, so that the inputs to the next stage are arranged in contiguous locations.

5. Compute n_1 DFT's of the rows of the matrix, recursively.

6. Transpose A in-place so as to produce the correct output order.

It can be proven by induction that the work complexity of this FFT algorithm is $O(n \lg n)$. We now analyze its cache complexity. The algorithm always operates on contiguous data, by construction. In order to simplify the analysis of the cache complexity, we assume a tall cache, in which case each transposition operation and the multiplication by the twiddle factors require at most $O(1 + n/L)$ cache misses. Thus, the cache complexity satisfies the recurrence

$$Q(n) \leq \begin{cases} O(1+n/L), & \text{if } n \leq \alpha Z \,, \\ n_1 Q(n_2) + n_2 Q(n_1) + O(1+n/L) & \text{otherwise} \,; \end{cases} \tag{3.3}$$

for a sufficiently small constant α chosen such that a subproblem of size αZ fits in cache. This recurrence has solution

$$Q(n) = O\left(1 + (n/L)\left(1 + \log_Z n\right)\right) \,,$$

which is asymptotically optimal for a Cooley-Tukey algorithm, matching the lower bound by Hong and Kung [25] when n is an exact power of 2. As with matrix multiplication, no tight lower bounds for cache complexity are known for the general problem of computing the DFT.

Summary

In this section, we have described an optimal cache-oblivious algorithm for FFT. The basic algorithm is the well-known "six-step" variant [6, 44] of the Cooley-Tukey FFT algorithm [15]. By using an optimal cache-oblivious transposition algorithm, however, we can make the FFT cache oblivious, and its performance matches the lower bound by Hong and Kung [25].

SECTION 4

Funnelsort

Although it is cache oblivious, algorithms like familiar two-way merge sort (see Section 7.3) are not asymptotically optimal with respect to cache misses. The Z-way mergesort mentioned by Aggarwal and Vitter [3] is optimal in terms of cache complexity, but it is cache aware. This section describes a cache-oblivious sorting algorithm called "funnelsort." This algorithm has an asymptotically optimal work complexity $\Theta(n \lg n)$, as well as an optimal cache complexity $\Theta(1 + (n/L)(1 + \log_Z n))$ if the cache is tall. In Section 5, we shall present another cache-oblivious sorting algorithm based on distribution sort.

Funnelsort is similar to mergesort. In order to sort a (contiguous) array of n elements, funnelsort performs the following two steps:

1. Split the input into $n^{1/3}$ contiguous arrays of size $n^{2/3}$, and sort these arrays recursively.

2. Merge the $n^{1/3}$ sorted sequences using a $n^{1/3}$-merger, which is described below.

Funnelsort differs from mergesort in the way the merge operation works. Merging is performed by a device called a ***k*-merger**, which inputs k sorted sequences and merges them. A k-merger operates by recursively merging sorted sequences that become progressively longer as the algorithm proceeds. Unlike mergesort, however, a k-merger stops working on a merging subproblem when the merged output sequence becomes "long enough," and it resumes working on another merging subproblem.

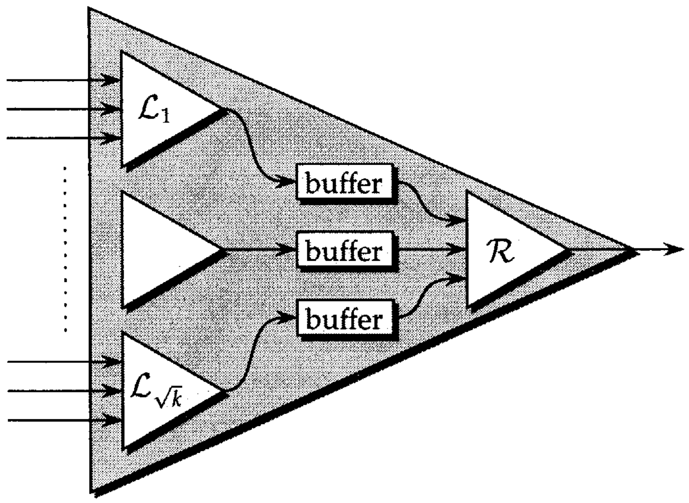


Figure 4-1: Illustration of a k-merger. A k-merger is built recursively out of $\sqrt{k}$ left $\sqrt{k}$-mergers $\mathcal{L}_1, \mathcal{L}_2, \ldots, \mathcal{L}_{\sqrt{k}}$, a series of buffers, and one right $\sqrt{k}$-merger $\mathcal{R}$.

Since this complicated flow of control makes a k-merger a bit tricky to describe, we explain the operation of the k-merger pictorially. Figure 4-1 shows a representation of a k-merger, which has k sorted sequences as inputs. Throughout its execution, the k-merger maintains the following invariant.

Invariant *The invocation of a k-merger outputs the first k^3 elements of the sorted sequence obtained by merging the k input sequences.*

A k-merger is built recursively out of $\sqrt{k}$-mergers in the following way. The k inputs are partitioned into $\sqrt{k}$ sets of $\sqrt{k}$ elements, and these sets form the input to the $\sqrt{k}$ left $\sqrt{k}$-mergers $\mathcal{L}_1, \mathcal{L}_2, \ldots, \mathcal{L}_{\sqrt{k}}$ in the left part of the figure. The outputs of these mergers are connected to the inputs of $\sqrt{k}$ ***buffers***. Each buffer is a FIFO queue that can hold $2k^{3/2}$ elements. Finally, the outputs of the buffers are connected to the $\sqrt{k}$ inputs of the right $\sqrt{k}$-merger $\mathcal{R}$ in the right part of the figure. The output of this final $\sqrt{k}$-merger becomes the output of the whole k-merger. The reader should notice that the intermediate buffers are overdimensioned. In fact, each buffer can hold $2k^{3/2}$ elements, which is twice the number $k^{3/2}$ of elements output by a $\sqrt{k}$-merger. This additional buffer space is necessary for the correct behavior of the algorithm, as will be explained below. The base case of the recursion is a k-merger with $k = 2$, which produces $k^3 = 8$ elements whenever invoked.

A k-merger operates recursively in the following way. In order to output k^3 elements, the k-merger invokes $\mathcal{R}$ $k^{3/2}$ times. Before each invocation, however, the k-merger fills all buffers that are less than half full, i.e., all buffers that contain less than $k^{3/2}$ elements. In order to fill buffer i, the algorithm invokes the corresponding left merger $\mathcal{L}_i$ once. Since $\mathcal{L}_i$ outputs $k^{3/2}$ elements, the buffer contains at least $k^{3/2}$ elements after $\mathcal{L}_i$ finishes.

In order to prove this result, we need three auxiliary lemmata. The first lemma bounds the space required by a k-merger.

Lemma 4 *A k-merger can be laid out in $O(k^2)$ contiguous memory locations.*

Proof. A k-merger requires $O(k^2)$ memory locations for the buffers, plus the space required by the $\sqrt{k}$-mergers. The space $S(k)$ thus satisfies the recurrence

$$S(k) \leq (\sqrt{k}+1)S(\sqrt{k}) + O(k^2) ,$$

whose solution is $S(k) = O(k^2)$. □

It follows from Lemma 4, that a problem of size $\alpha\sqrt{Z}$ can be solved in cache with no further cache misses, where α is a sufficiently small constant.

In order to achieve the bound on the number $Q(n)$ of cache misses, it is important that the buffers in a k-merger be maintained as circular queues of size k. This requirement guarantees that we can manage the queue cache-efficiently, in the sense stated by the next lemma.

Lemma 5 *Performing r insert and remove operations on a circular queue causes $O(1 + r/L)$ cache misses if four cache lines are available for the buffer.*

Proof. Associate the two cache lines with the head and tail of the circular queue. The head- and tail-pointers are kept on two seperate lines. Since the replacement strategy is optimal, it will keep the frequently accessed pointers in cache. If a new cache line is read during an insert (delete) operation, the next $L-1$ insert (delete) operations do not cause a cache miss. The result follows. □

Define Q_M to be the number of cache misses incurred by a k-merger. The next lemma bounds the number of cache misses incurred by a k-merger.

Lemma 6 *On a tall cache, one invocation of a k-merger incurs*

$$Q_M(k) = O\left(k + k^3/L + k^3 \log_Z k/L\right)$$

cache misses.

Proof. There are two cases: either $k \leq \alpha\sqrt{Z}$ or $k > \alpha\sqrt{Z}$.

Assume first that $k \leq \alpha\sqrt{Z}$. By Lemma 4, the data structure associated with the k-merger requires at most $O(k^2) = O(Z)$ contiguous memory locations. By the choice of α the k-merger fits into cache. The k-merger has k input queues, from which it loads $O(k^3)$ elements. Let r_i be the number of elements extracted from the ith input queue. Since $k \le \alpha\sqrt{Z}$ and $L = O(\sqrt{Z})$, there are at least $Z/L = \Omega(k)$ cache lines available for the input buffers. Lemma 5 applies, whence the total number of cache misses for accessing the input queues is

$$\sum_{i=1}^{k} O(1 + r_i/L) = O(k + k^3/L) .$$

Similarly by Lemma 5, the cache complexity of writing the output queue is at most $O(1 + k^3/L)$. Finally, for touching the $O(k^2)$ contiguous memory locations used by the internal data structures, the algorithm incurs at most $O(1 + k^2/L)$ cache misses. The total cache complexity is therefore

$$\begin{aligned} Q_M(k) &= O\left(k + k^3/L\right) + O\left(1 + k^2/L\right) + O\left(1 + k^3/L\right) \\ &= O\left(k + k^3/L\right) \end{aligned}$$

completing the proof of the first case.

Assume now that $k > \alpha\sqrt{Z}$. In this second case, we prove by induction on k that whenever $k > \alpha\sqrt{Z}$, we have

$$Q_M(k) \le (ck^3 \log_Z k)/L - A(k) , \tag{4.1}$$

for some constant $c > 0$, where $A(k) = k(1 + (2c\log_Z k)/L) = o(k^3)$. The lower-order term $A(k)$ does not affect the asymptotic behavior, but it makes the induction go through. This particular value of $A(k)$ will be justified later in the analysis.

The base case of the induction consists of values of k such that $\sqrt{\alpha}Z^{1/4} < k \le \alpha\sqrt{Z}$. (It is not sufficient to just consider $k = \Theta(\sqrt{Z})$, since k can become as small as $\Theta(Z^{1/4})$ in the recursive calls.) The analysis of the first case applies, yielding $Q_M(k) = O\left(k + k^3/L\right)$. Because $k^2 > \alpha\sqrt{Z} = \Omega(L)$ and $k = \Omega(1)$, the last term dominates, and $Q_M(k) = O\left(k^3/L\right)$ holds. Consequently, a large enough value of c can be found that satisfies Inequality (4.1).

For the inductive case, let $k > \alpha\sqrt{Z}$. The k-merger invokes the $\sqrt{k}$-mergers recursively. Since $\sqrt{\alpha}Z^{1/4} < \sqrt{k} < k$, the inductive hypothesis can be used to bound the number $Q_M(\sqrt{k})$ of cache misses incurred by the submergers. The right merger $\mathcal{R}$ is invoked exactly $k^{3/2}$ times. The total number l of invocations of left mergers is bounded by $l < k^{3/2} + 2\sqrt{k}$. To see why, consider that every invocation of a left merger puts $k^{3/2}$ elements into some buffer. Since k^3 elements are output and the buffer space is $2k^2$, the bound $l < k^{3/2} + 2\sqrt{k}$ follows.

Before invoking $\mathcal{R}$, the algorithm must check every buffer to see whether it is empty. One such check requires at most $\sqrt{k}$ cache misses, since there are $\sqrt{k}$

buffers. This check is repeated exactly $k^{3/2}$ times, leading to at most k^2 cache misses for all checks.

These considerations lead to the recurrence

$$Q_M(k) \le \left(2k^{3/2} + 2\sqrt{k}\right) Q_M(\sqrt{k}) + k^2 .$$

Application of the inductive hypothesis yields the desired bound Inequality (4.1), as follows.

$$\begin{aligned}
Q_M(k) &\le \left(2k^{3/2} + 2\sqrt{k}\right) Q_M(\sqrt{k}) + k^2 \\
&\le 2\left(k^{3/2} + \sqrt{k}\right) \left[\frac{ck^{3/2}\log_Z k}{2L} - A(\sqrt{k})\right] + k^2 \\
&\le (ck^3\log_Z k)/L + k^2\left(1 + (c\log_Z k)/L\right) \\
&\qquad - \left(2k^{3/2} + 2\sqrt{k}\right) A(\sqrt{k}) .
\end{aligned}$$

If $A(k) = k(1 + (2c\log_Z k)/L)$ (for example), we get

$$\begin{aligned}
Q_M(k) &\le (ck^3\log_Z k)/L + k^2\left(1 + (c\log_Z k)/L\right) \\
&\qquad - \left(2k^{3/2} + 2\sqrt{k}\right) \sqrt{k}\left(1 + (2c\log_Z \sqrt{k})/L\right) \\
&\le (ck^3\log_Z k)/L + k^2\left(1 + (c\log_Z k)/L\right) \\
&\qquad - \left(2k^2 + 2k\right)\left(1 + (c\log_Z k)/L\right) \\
&\le (ck^3\log_Z k)/L - \left(k^2 + 2k\right)\left(1 + (c\log_Z k)/L\right) \\
&\le (ck^3\log_Z k)/L - A(k)
\end{aligned}$$

and Inequality (4.1) follows. □

It can be proven by induction that the work complexity of funnelsort is $O(n \lg n)$. The next theorem gives the cache complexity of funnelsort.

Theorem 7 *Funnelsort sorts n elements incurring at most $Q(n)$ cache misses, where*

$$Q(n) = O\left(1 + (n/L)\left(1 + \log_Z n\right)\right) .$$

Proof. If $n \le \alpha Z$ for a small enough constant α, then the funnelsort data structures fit into cache. To see why, observe that only one k-merger is active at any time. The biggest k-merger is the top-level $n^{1/3}$-merger, which requires $O(n^{2/3}) < O(n)$ space. The algorithm thus can operate in $O(1 + n/L)$ cache misses.

If $n > \alpha Z$, we have the recurrence

$$Q(n) = n^{1/3}Q(n^{2/3}) + Q_M(n^{1/3}) .$$

By Lemma 6, we have $Q_M(n^{1/3}) = O\left(n^{1/3} + n/L + (n\log_Z n)/L\right)$.

With the hypothesis $Z = \Omega(L^2)$, we have $n/L = \Omega(n^{1/3})$. Moreover, we also have $n^{1/3} = \Omega(1)$ and $\lg n = \Omega(\lg Z)$. Consequently, $Q_M(n^{1/3}) = O\left((n\log_Z n)/L\right)$ holds, and the recurrence simplifies to

$$Q(n) = n^{1/3}Q(n^{2/3}) + O\left((n\log_Z n)/L\right) \ .$$

The result follows by induction on n. □

This upper bound matches the lower bound stated by the next theorem, proving that funnelsort is cache-optimal.

Theorem 8 *The cache complexity of any sorting algorithm is*

$$Q(n) = \Omega\left(1 + (n/L)\left(1 + \log_Z n\right)\right) \ .$$

Proof. Aggarwal and Vitter [3] show that there is an $\Omega\left((n/L)\log_{Z/L}(n/Z)\right)$ bound on the number of cache misses made by any sorting algorithm on their "out-of-core" memory model, a bound that extends to the ideal-cache model. By applying the tall-cache assumption $Z = \Omega(L^2)$, we have

$$\begin{aligned} Q(n) &\geq a(n/L)\log_{Z/L}(n/Z) \\ &\geq a(n/L)\lg(n/Z)/(\lg Z - \lg L) \\ &\geq a(n/L)\lg(n/Z)/\lg Z \\ &\geq a(n/L)\lg n/\lg Z - a(n/L) \ . \end{aligned}$$

It follows that $Q(n) = \Omega((n/L)\log_Z n)$. The theorem can be proven by combining this result with the trivial lower bounds of $Q(n) = \Omega(1)$ and $Q(n) = \Omega(n/L)$. □

Corollary 9 *The cache-oblivious Funnelsort is asymptotically optimal.*

Proof. Follows from Theorems 8 and 7. □

Summary

In this section we have presented an optimal cache-oblivious algorithm based on mergesort. Funnelsort uses a device called a k-merger, which inputs k sorted sequences and merges them in "chunks". It stops when the merged output becomes "long enough" to resume work on another subproblem. Further, we have shown that any sorting algorithm incurs at least $\Omega\left(1 + (n/L)\left(1 + \log_Z n\right)\right)$ cache misses. This lower bound is matched by both our algorithms.

SECTION 5

Distribution sort

In this section, we describe a cache-oblivious optimal sorting algorithm based on distribution sort. Like the funnelsort algorithm from Section 4, the distribution-sorting algorithm uses $O(n \lg n)$ work to sort n elements, and it incurs

$$\Theta(1 + (n/L)(1 + \log_Z n))$$

cache misses if the cache is tall. Unlike previous cache-efficient distribution-sorting algorithms [1, 3, 30, 42, 44], which use sampling or other techniques to find the partitioning elements before the distribution step, our algorithm uses a "bucket-splitting" technique to select pivots incrementally during the distribution.

Given an array A (stored in contiguous locations) of length n, the cache-oblivious distribution sort sorts A as follows:

1. Partition A into $\sqrt{n}$ contiguous subarrays of size $\sqrt{n}$. Recursively sort each subarray.
2. Distribute the sorted subarrays into $q \le \sqrt{n}$ buckets $B_1, B_2, \ldots, B_q$ of size n_1, $n_2, \ldots, n_q$, respectively, such that for $i = 1, 2, \ldots, q-1$, we have

    1. $\max\{x \mid x \in B_i\} \le \min\{x \mid x \in B_{i+1}\}$,
    2. $n_i \le 2\sqrt{n}$.

    (See below for details.)
3. Recursively sort each bucket.
4. Copy the sorted buckets back to array A.

A stack-based memory allocator is used to exploit spatial locality. A nice property of stack based allocation is that memory is not fragmented for problems of small size. So if the space complexity of a procedure is S, only $O(1 + S/L)$ cache misses are made when $S \leq Z$, provided the procedure accesses only its local variables.

Distribution step

The goal of Step 2 is to distribute the sorted subarrays of A into q buckets B_1, $B_2, \ldots, B_q$. The algorithm maintains two invariants. First, each bucket holds at most $2\sqrt{n}$ elements at any time, and any element in bucket B_i is smaller than any element in bucket B_{i+1}. Second, every bucket has an associated ***pivot***, a value which is greater than all elements in the bucket. Initially, only one empty bucket exists with pivot ∞. At the end of Step 2, all elements will be in the buckets and the two conditions (a) and (b) stated in Step 2 will hold.

The idea is to copy all elements from the subarrays into the buckets cache efficiently while maintaining the invariants. We keep state information for each subarray and for each bucket. The state of a subarray consists of an index *next* of the next element to be read from the subarray and a bucket number *bnum* indicating where this element should be copied. By convention, $bnum = \infty$ if all elements in a subarray have been copied. The state of a bucket consists of the bucket's pivot and the number of elements currently in the bucket.

We would like to copy the element at position *next* of a subarray to bucket *bnum*. If this element is greater than the pivot of bucket *bnum*, we would increment *bnum* until we find a bucket for which the element is smaller than the pivot. Unfortunately, this basic strategy has poor caching behavior, which calls for a more complicated procedure.

The distribution step is accomplished by the recursive procedure DISTRIBUTE. DISTRIBUTE(i, j, m) distributes elements from the ith through $(i + m - 1)$th subarrays into buckets starting from B_j. Given the precondition that each subarray $r = i, i + 1, \ldots, i + m - 1$ has its $bnum[r] \geq j$, the execution of DISTRIBUTE(i, j, m) enforces the postcondition that $bnum[r] \geq j + m$. Step 2 of the distribution sort invokes DISTRIBUTE$(1, 1, \sqrt{n})$. The following is a recursive implementation of DISTRIBUTE:

```
DISTRIBUTE($i, j, m$)
1  if $m = 1$
2     then COPYELEMS($i, j$)
3     else DISTRIBUTE($i, j, m/2$)
4          DISTRIBUTE($i + m/2, j, m/2$)
5          DISTRIBUTE($i, j + m/2, m/2$)
6          DISTRIBUTE($i + m/2, j + m/2, m/2$)
```

In the base case (line 1), the subroutine COPYELEMS(i, j) copies all elements from subarray i that belong to bucket j. If bucket j has more than $2\sqrt{n}$ elements after the insertion, it can be split into two buckets of size at least $\sqrt{n}$. For the splitting operation, we use the deterministic median-finding algorithm [16, p. 189] followed by a partition. The next lemma shows that the median-finding algorithm uses $O(m)$ work and incurs $O(1 + m/L)$ cache misses to find the median of an array of size m. (In our case, we have $m \geq 2\sqrt{n} + 1$.) In addition, when a bucket splits, all subarrays whose *bnum* is greater than the *bnum* of the split bucket must have their *bnum*'s incremented. The analysis of DISTRIBUTE is given by the following two lemmata.

Lemma 10 *The median of m elements can be found cache-obliviously using $O(m)$ work and incurring $O(1 + m/L)$ cache misses.*

Proof. See [16, p. 189] for the linear-time median finding algorithm and the work analysis. The cache complexity is given by the same recurrence as the work complexity with a different base case.

$$Q(m) = \begin{cases} O(1 + m/L) & \text{if } m \leq \alpha Z\,, \\ Q(\lceil m/5 \rceil) + Q(7m/10 + 6) + O(1 + m/L) & \text{otherwise}\,, \end{cases}$$

where α is a sufficiently small constant. The result follows. □

Lemma 11 *Step 2 uses $O(n)$ work, incurs $O(1 + n/L)$ cache misses, and uses $O(n)$ stack space to distribute n elements.*

Proof. In order to simplify the analysis of the work used by DISTRIBUTE, assume that COPYELEMS uses $O(1)$ work. We account for the work due to copying elements and splitting of buckets separately. The work of DISTRIBUTE on m subarrays is described by the recurrence

$$T(m) = 4T(m/2) + O(1)\,.$$

It follows that $T(m) = O(m^2)$, where $m = \sqrt{n}$ initially.

We now analyze the work used for copying and bucket splitting. The number of copied elements is $O(n)$. Each element is copied exactly once and therefore the work due to copying elements is also $O(n)$. The total number of bucket splits is at most $\sqrt{n}$. To see why, observe that there are at most $\sqrt{n}$ buckets at the end of the distribution step, since each bucket contains at least $\sqrt{n}$ elements. Each split operation involves $O(\sqrt{n})$ work and so the net contribution to the work is $O(n)$. Thus, the total work used by DISTRIBUTE is $W(n) = O(T(\sqrt{n})) + O(n) + O(n) = O(n)$.

For the cache analysis, we distinguish two cases. Let α be a sufficiently small constant such that the stack space used by sorting a problem of size αZ, including the input array, fits completely into cache.

Case I: $n \leq \alpha Z$.

The input and the auxiliary space of size $O(n)$ fit into cache using $O(1 + n/L)$ cache lines. Consequently, the cache complexity is $O(1 + n/L)$.

Case II: $n > \alpha Z$.

Let $R(m,d)$ denote the cache misses incurred by an invocation of the subroutine DISTRIBUTE(i,j,m) that copies d elements from m subarrays to m buckets. We again account for the splitting of buckets separately. We first prove that R satisfies the following recurrence:

$$R(m,d) \leq \begin{cases} O(L + d/L) & \text{if } m \leq \alpha L\,, \\ \sum_{1 \leq i \leq 4} R(m/2, d_i) & \text{otherwise}\,, \end{cases} \tag{5.1}$$

where $\sum_{1 \leq i \leq 4} d_i = d$.

First, consider the base case $m \leq \alpha L$. An invocation of DISTRIBUTE(i,j,m) operates with m subarrays and m buckets. Since there are $\Omega(L)$ cache lines, the cache can hold all the auxiliary storage involved and the currently accessed element in each subarray and bucket. In this case there are $O(L + d/L)$ cache misses. The initial access to each subarray and bucket causes $O(m) = O(L)$ cache misses. The cache complexity for copying the d elements from one set of contiguous locations to another set of contiguous locations is $O(1 + d/L)$, which completes the proof of the base case. The recursive case, when $m > \alpha L$, follows immediately from the algorithm. The solution for Recurrence 5.1 is $R(m,d) = O(L + m^2/L + d/L)$.

We still need to account for the cache misses caused by the splitting of buckets. Each split causes $O(1 + \sqrt{n}/L)$ cache misses due to median finding

(Lemma 10) and partitioning of $\sqrt{n}$ contiguous elements. An additional $O(1+\sqrt{n}/L)$ misses are incurred by restoring the cache. As proven in the work analysis, there are at most $\sqrt{n}$ split operations.

By adding $R(\sqrt{n}, n)$ to the complexity of splitting, we conclude that the total cache complexity of the distribution step is $O(L + n/L + \sqrt{n}(1 + \sqrt{n}/L)) = O(n/L)$.

□

Theorem 12 *Distribution sort uses $O(n \lg n)$ work and incurs $O(1 + (n/L)(1 + \log_Z n))$ cache misses to sort n elements.*

Proof. The work done by the algorithm is given by

$$W(n) = \sqrt{n}W(\sqrt{n}) + \sum_{i=1}^{q} W(n_i) + O(n) ,$$

where $q \leq \sqrt{n}$, each $n_i \leq 2\sqrt{n}$, and $\sum_{i=1}^{q} n_i = n$. The solution to this recurrence is $W(n) = O(n \lg n)$.

The space complexity of the algorithm is given by

$$S(n) \leq S(2\sqrt{n}) + O(n) .$$

Each bucket has at most $2\sqrt{n}$ elements, thus the recursive call uses at $S(2\sqrt{n})$ space and the $O(n)$ term comes from Step 2. The solution to this recurrence is $S(n) = O(n)$.

The cache complexity of distribution sort is described by the recurrence

$$Q(n) \leq \begin{cases} O(1+n/L) & \text{if } n \leq \alpha Z , \\ \sqrt{n}Q(\sqrt{n}) + \sum_{i=1}^{q} Q(n_i) + O(1+n/L) & \text{otherwise} , \end{cases}$$

where α is a sufficiently small constant such that the stack space used by a sorting problem of size αZ, including the input array, fits completely in cache. The base case $n \leq \alpha Z$ arises when both the input array A and the contiguous stack space of size $S(n) = O(n)$ fit in $O(1+n/L)$ cache lines of the cache. In this case, the algorithm incurs $O(1+n/L)$ cache misses to touch all involved memory locations once. In the case where $n > \alpha Z$, the recursive calls in Steps 1 and 3 cause $Q(\sqrt{n}) + \sum_{i=1}^{q} Q(n_i)$ cache misses and $O(1+n/L)$ is the cache complexity of Steps 2 and 4, as shown by Lemma 11. The theorem now follows by solving the recurrence. □

Corollary 13 *The cache-oblivious distribution sort algorithm is asymptotically optimal.*

Proof. Follows from Theorems 8 and 12. □

Summary

In this section, we have presented another optimal cache-oblivious sorting algorithm, which is based on distribution sort. All previous cache-efficient distribution sort algorithms [1, 3, 30, 42, 44] are cache aware, since they are designed for caching models where the data is moved explicitly. They usually use a sampling processes to find the partitioning elements before the distribution step. Our algorithm finds the pivots incrementally during the distribution.

SECTION 6

Jacobi multipass filter

This section compares an optimal recursive algorithm with a more straightforward iterative algorithm, both which compute a multipass filter over one-dimensional data. When computing n generations on n elements, both algorithms use $\Theta(n^2)$ work. The iterative incurs $\Theta(n^2/L)$ cache misses, if the data does not fit into the cache, where the recursive algorithm incurs only $\Theta(1+n/L+n^2/ZL)$ cache misses which we prove to be cache optimal. We also provide some brief empirical results for this problem. The recursive algorithm executes in less than 70% of the time of the iterative algorithm for problem sizes that do not fit in L2-cache

Consider the problem of a computing a multipass filter on an array A of size n, where a new value $A_i^{(\tau+1)}$ at generation $\tau+1$ is computed from values at the previous step τ according to some update rule. A typical update function is

$$A_i^{(\tau+1)} \leftarrow \left(A_{i-1}^{(\tau)} + A_i^{(\tau)} + A_{i+1}^{(\tau)}\right)/3\,. \tag{6.1}$$

Applications of multipass filtering include the Jacobi iteration for solving heat-diffusion equations [31, p. 673] and the simulation of lattice gases with cellular automata. These applications usually deal with multidimensional data, but here, we shall explore the one-dimensional case for simplicity, even though caching effects are often more pronounced with multidimensional data.

```
JACOBI-ITER($A$)
1  $n \leftarrow$ length of $A$
2  for $i \leftarrow 1$ to $n/2$
3      do for $j \leftarrow 1$ to $n$    $\triangleright$ Generation $2i$
4             do $tmp[j] \leftarrow (A[(j-1) \bmod n] + A[j] + A[(j+1) \bmod n])/3$
5         for $j \leftarrow 1$ to $n$    $\triangleright$ Generation $2i+1$
6             do $A[j] \leftarrow (tmp[(j-1) \bmod n] + tmp[j] + tmp[(j+1) \bmod n])/3$
```

Figure 6-1: Iterative implementation of n-pass Jacobi update on array A with n elements.

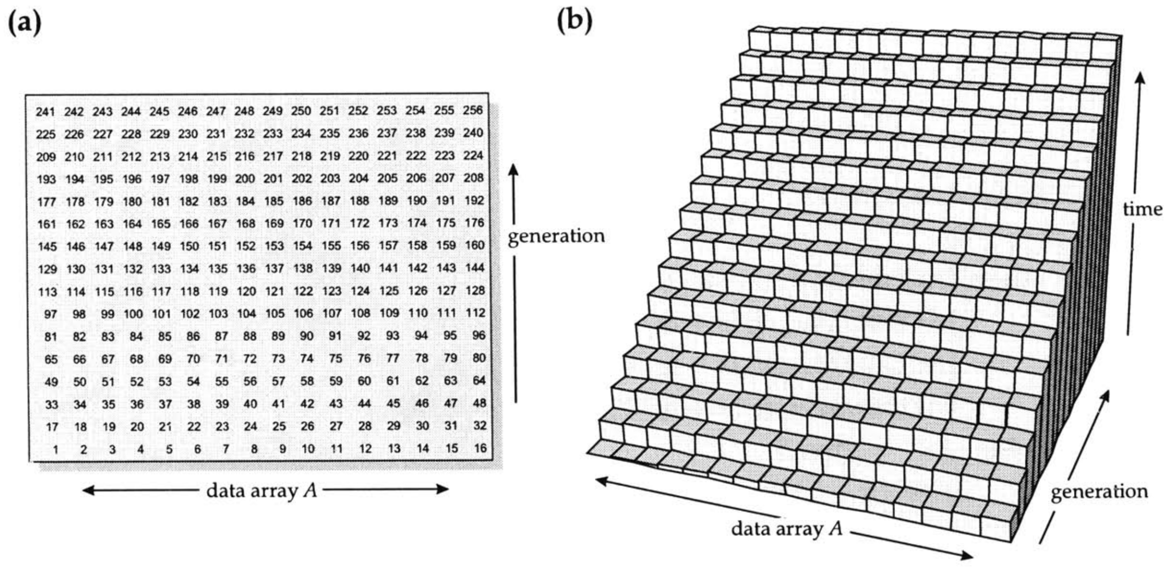
6.1 *Iterative algorithm*

We first analyze the cache complexity of the straightforward implementation JACOBI-ITER of the update rule given in Equation (6.1). We show that this algorithm, shown in Figure 6-1, uses $\Theta(n)$ temporary storage and performs $\Theta(n^2)$ memory accesses for an array of size n. If the array of size n does not fit into cache, the total number of cache misses is $\Theta(n^2/L)$.

To illustrate the order of updates of JACOBI-ITER on input A of size n, we view the computation of n generations of the multipass as a two-dimensional ***trace matrix*** $\mathcal{T}$ of size $n \times n$. One dimension of $\mathcal{T}$ is the offset in the input array and the other dimension is the "generation" of the filtered result. The value of element $\mathcal{T}_{4,2}$ is the value of array element $A[2]$ at the 4th generation of the iterative algorithm. One row in the matrix represents the updates on one element in the array. The trace matrix of the iterative algorithm on a data array of size 16 is shown in Figure 6-2. The height of a bar represents the ordering of the updates, where the higher bars are updated later. The bigger the difference in the height of two bars, the further apart in time are their updates. If the height of a bar is not much bigger than the height of the bar directly in front of it, it is likely that the element is still in cache and a hit occurs. The height differences between two updates to the same element in the iterative algorithm are all equal. Either the updates are close enough together that all updates are cache hits, or they are too far apart, and all updates are cache misses.

Theorem 14 *The* JACOBI-ITER *algorithm uses $\Theta(n^2)$ work when computing n generations on an array of size n.* JACOBI-ITER *incurs $\Theta(1+n/L)$ cache misses if the data fits into cache, and it incurs $\Theta(n^2/L)$ cache misses if the array does not fit into cache.*

Proof. Since there are two nested loops, each of which performs n iterations, the work is $\Theta(n^2)$.



241	242	243	244	245	246	247	248	249	250	251	252	253	254	255	256
225	226	227	228	229	230	231	232	233	234	235	236	237	238	239	240
209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	224
193	194	195	196	197	198	199	200	201	202	203	204	205	206	207	208
177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	192
161	162	163	164	165	166	167	168	169	170	171	172	173	174	175	176
145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	160
129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	144
113	114	115	116	117	118	119	120	121	122	123	124	125	126	127	128
97	98	99	100	101	102	103	104	105	106	107	108	109	110	111	112
81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96
65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80
49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64
33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48
17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Figure 6-2: Ordering of updates of JACOBI-ITER on an array with 16 elements. **(a)** The trace matrix with the order of updates from 1 to 256. **(b)** A bar-graph illustrating the updates, where the height of a bar represents the ordering of updates and the smallest bar is updated first.

We now analyze the cache misses of JACOBI-ITER on a (Z, L) ideal cache. Let α be a constant sufficiently small that αZ elements fit into a cache of size Z. As long as the array and the temporary storage fit into cache, e.g., $n \leq \alpha Z$, the algorithm performs well. The cache complexity is only $\Theta(1 + n/L)$ since the $O(n)$ elements are read (in order) only once.

If the array has size $n \geq \alpha Z$, however, then it does not all fit in cache at one time. The optimal replacement strategy can keep at most $O(Z)$ elements in cache. Thus, per iteration we have $\Omega(n/L - Z)$ updates which are cache misses. Consequently, the total number of cache misses is $\Theta(n)\Omega(n/L - Z) = \Omega(n^2/L)$ for the n iterations.

The algorithm can be optimized to use only $O(1)$ temporary storage and avoid the modulo computation, but the number of cache misses remains at $\Theta(n^2/L)$. $\square$

6.2 *Recursive algorithm*

In this section, we present an optimal recursive algorithm to compute an n-pass Jacobi filter. This cache-oblivious algorithm JACOBI-REC is sketched in Figure 6-3, where the input size is a power of 2.[1] We prove that the work used by JACOBI-

[1]The algorithm for the general case is slightly more complicated.

```
JACOBI△$(A, n, s, w, \tau)$
1  if $w > 2$
2    then JACOBI△$(A, n, s, w/2, \tau)$
3         JACOBI△$(A, n, (s + w/2), w/2, \tau)$
4         JACOBI▽$(A, n, (s + w/4) + 1, w/2, \tau)$
5         JACOBI△$(A, n, (s + w/4), w/2, \tau + w/4)$
6    else $p \leftarrow \tau \bmod 2$
7         $q \leftarrow (\tau + 1) \bmod 2$
8         $A[p][s \bmod n] \leftarrow$
9             $(A[q][(s-1) \bmod n] + A[q][s \bmod n] + A[q][(s+1) \bmod n])/3$
10        $A[p][(s+1) \bmod n] \leftarrow$
11            $(A[q][s \bmod n] + A[q][(s+1) \bmod n] + A[q][(s+2) \bmod n])/3$

JACOBI▽$(A, n, s, w, \tau)$
1  if $w > 2$
2    then JACOBI▽$(A, n, s + w/4, w/2, \tau)$
3         JACOBI△$(A, n, s + w/4, w/2, \tau + w/4)$
4         JACOBI▽$(A, n, s, w/2, \tau + w/4)$
5         JACOBI▽$(A, n, s + w/2, w/2, \tau + w/4)$

JACOBI-REC$(A)$
1  $n \leftarrow$ length of $A$
2  JACOBI△$(A, n, 0, n, 0)$
3  JACOBI▽$(A, n, n/2, n, 0)$
4  JACOBI△$(A, n, n/2, n, n/2)$
5  JACOBI▽$(A, n, 0, n, n/2)$
```

Figure 6-3: The recursive implementation of the multipass filter on array A of size n, where n is a power of 2. The algorithm uses two auxiliary subroutines JACOBI△(A, n, s, w, τ) and JACOBI▽(A, n, s, w, τ). The input is in $A[0]$ and $A[1]$ is the $O(n)$ auxiliary space. The parameters s and w specify the position and size of the computed triangle and τ is the generation of the lowest level of the triangle. JACOBI-REC(A) is the initial call.

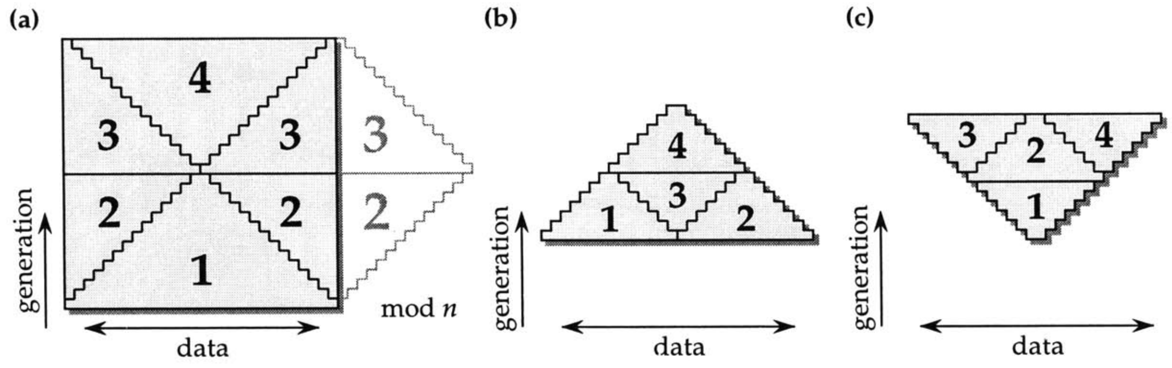
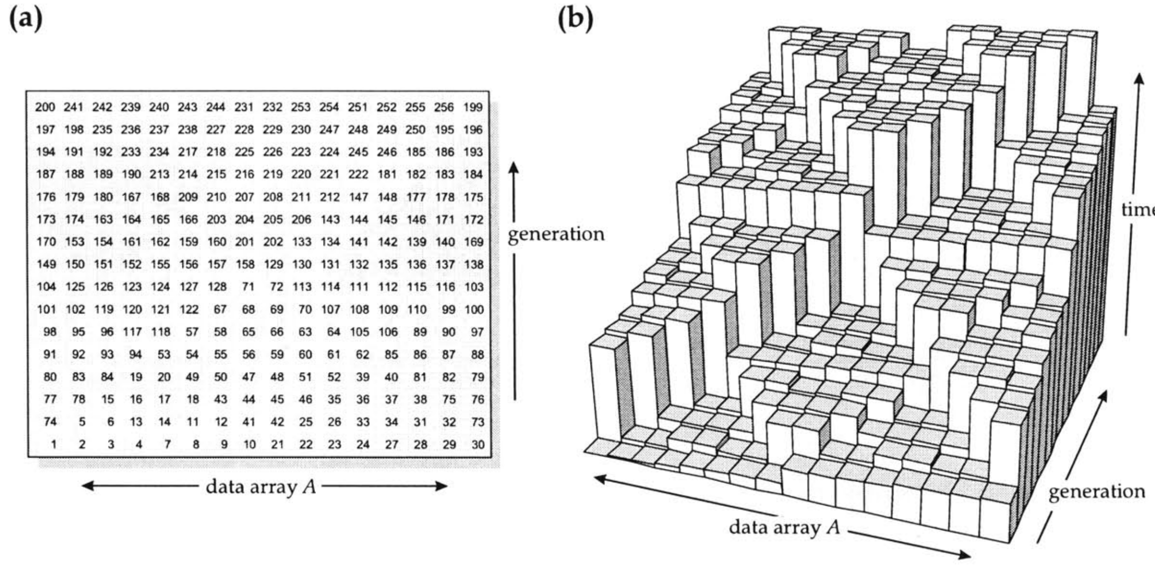


Figure 6-4: **(a)** Decomposition of trace matrix in four triangles by JACOBI-REC. Triangles 2 and 3 "wrap around" since array positions are computed modulo n. **(b)** shows the decomposition used by JACOBI$\triangle$ and **(c)** shows the decomposition used by JACOBI∇.

REC is $\Theta(n^2)$ and the cache complexity is $O(1 + n/L + n^2/ZL)$, even if the problem does not fit into cache, which is a factor of Z fewer cache misses than the iterative method.

In order to simplify the description, we describe the recursive algorithm as if the whole trace matrix would be computed. It turns out that in practice one auxiliary array of size n suffices to compute the n steps on an array of size n.

The divide-and-conquer algorithm divides the trace matrix into 4 triangles, which are recursively divided into smaller triangles, as shown in Figure 6-4(a). Two auxiliary functions JACOBI$\triangle$ and JACOBI∇ are used to implement the recursion. JACOBI$\triangle(A, n, s, w, \tau)$ computes an "upper triangle" of the trace matrix A of size n, where the base of the triangle has size w and starts at s with generation τ. It recursively computes up to $w/2$ generations ahead as shown in Figure 6-4(b). Analogously, JACOBI∇ computes a lower triangle recursively as shown in Figure 6-4(c). The resulting trace matrix for an array of size 16 is shown in Figure 6-5. It illustrates the locality of the recursive algorithms. The triangles of the decomposition are clearly visible. Depending on the cache size, triangles of different size fit entirely into cache, which are then computed without any further cache misses. Although JACOBI-REC computes the elements in different order than JACOBI-ITER, it computes exactly the same values as JACOBI-ITER.



200	241	242	239	240	243	244	231	232	253	254	251	252	255	256	199
197	198	235	236	237	238	227	228	229	230	247	248	249	250	195	196
194	191	192	233	234	217	218	225	226	223	224	245	246	185	186	193
187	188	189	190	213	214	215	216	219	220	221	222	181	182	183	184
176	179	180	167	168	209	210	207	208	211	212	147	148	177	178	175
173	174	163	164	165	166	203	204	205	206	143	144	145	146	171	172
170	153	154	161	162	159	160	201	202	133	134	141	142	139	140	169
149	150	151	152	155	156	157	158	129	130	131	132	135	136	137	138
104	125	126	123	124	127	128	71	72	113	114	111	112	115	116	103
101	102	119	120	121	122	67	68	69	70	107	108	109	110	99	100
98	95	96	117	118	57	58	65	66	63	64	105	106	89	90	97
91	92	93	94	53	54	55	56	59	60	61	62	85	86	87	88
80	83	84	19	20	49	50	47	48	51	52	39	40	81	82	79
77	78	15	16	17	18	43	44	45	46	35	36	37	38	75	76
74	5	6	13	14	11	12	41	42	25	26	33	34	31	32	73
1	2	3	4	7	8	9	10	21	22	23	24	27	28	29	30

Figure 6-5: Ordering of updates of JACOBI-REC on an array with 16 elements. **(a)** The trace matrix with the order of updates from 1 to 256. **(b)** A bar-graph illustrating the updates, where the height of a bar represents the ordering of updates and the smallest bar is updated first.

Theorem 15 *The recursive* JACOBI-REC *algorithm involves* $\Theta(n^2)$ *work and incurs* $\Theta(1 + n/L + n^2/ZL)$ *cache misses when computing n generations on n elements.*

Proof. To simplify the analysis, we assume that n is an exact power of 2.[2] The work of JACOBI-REC can be described by three recurrences:

$$\begin{aligned} W(n) &= 2W_{\triangle}(n/2) + 2W_{\triangledown}(n/2) + O(1) \,, \\ W_{\triangle}(n) &= 3W_{\triangle}(n/2) + W_{\triangledown}(n/2) + O(1) \,, \\ W_{\triangledown}(n) &= 3W_{\triangledown}(n/2) + W_{\triangle}(n/2) + O(1) \,; \end{aligned}$$

where $W_{\triangle}$ and $W_{\triangledown}$ are the work used by the recursive procedures JACOBI$\triangle$ and JACOBI$\triangledown$. The solution for the total work is $W(n) = \Theta(n^2)$, which is the same as the work of the iterative algorithm.

The number $Q(n)$ of cache misses incurred by a subproblem of size n is described by three recurrences:

$$\begin{aligned} Q(n) &= 2Q_{\triangle}(n/2) + 2Q_{\triangledown}(n/2) + O(1) \,; \\ Q_{\triangle}(n) &\le \begin{cases} \Theta(1 + n/L) & \text{if } n \le \alpha Z \,, \\ 3Q_{\triangle}(n/2) + Q_{\triangledown}(n/2) + O(1) & \text{otherwise} \,; \end{cases} \end{aligned}$$

[2]The results can be extended, but the analysis is somewhat more complicated.

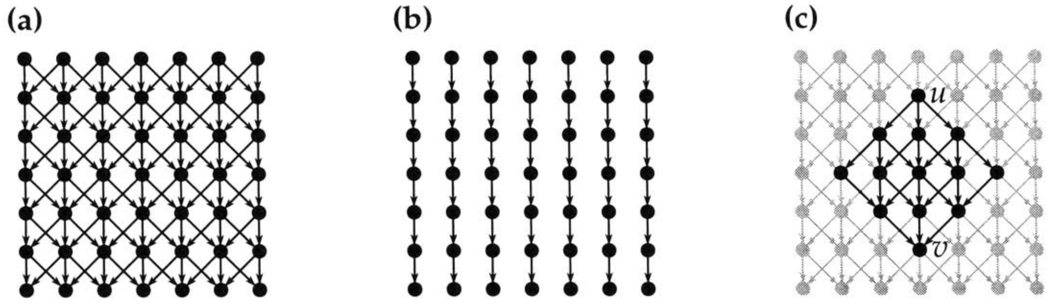


Figure 6-6: Computational dag of JACOBI-ITER. **(a)** complete subgraph of size 9×9, **(b)** its decomposition into lines, and **(c)** diamond-shaped subdag of width $\Omega(d)$ that is enclosed by two nodes u and v of distance $d = 4$.

$$Q_{\bigtriangledown}(n) \leq \begin{cases} \Theta(1 + n/L) & \text{if } n \leq \alpha Z\,, \\ 3Q_{\bigtriangledown}(n/2) + Q_{\triangle}(n/2) + O(1) & \text{otherwise}\,; \end{cases}$$

where $Q_{\triangle}$ and $Q_{\triangle}$ are the cache misses of the two recursive procedures JACOBI$\triangle$ and JACOBI$\bigtriangledown$, and α is a sufficiently small constant. The base case occurs when the two arrays fit into the cache. Solving these recurrences, we obtain $Q(n) = \Theta(1 + n/L + n^2/ZL)$ cache misses. $\square$

6.3 *Lower bound*

Finally, we prove that the number of cache misses for this problem is lower bounded by $\Omega(1 + n/L + n^2/ZL)$, which implies that the recursive algorithm JACOBI-REC is indeed optimal.

We can use the red-blue pebble game technique described by Hong and Kung [25] to lower-bound the number of cache misses incurred by any algorithm computing n generations of an Jacobi-multipass filter on n elements. Hong and Kung use properties of the ***computation dag*** (directed acyclic graph) G given by a computation to lower-bound the number of cache misses on a two-level memory. Nodes in a computation dag represent operations, and edges, the data-flow of the algorithm. Nodes with no incoming edges are input and nodes with no outgoing edges are output. Figure 6.3(a) shows a subgraph of the computation dag given by JACOBI-ITER. A vertex-disjoint path from inputs to outputs will be called ***lines***. The decomposition of Figure 6.3(a) into lines is shown in in Figure 6.3(b). The number of cache misses can be lower-bounded by the ***information speed function*** $F_G(d)$ of a dag G, which is defined as follows.

For any two vertices u and v on the same line that are at least d apart, there are $F_G(d)$ vertices in the dag G satisfying two properties:

1. None of these vertices belongs to the same line.
2. Each of these vertices belongs to a path connecting u and v.

In the dag given by JACOBI-ITER, for example, two nodes u and v enclose a diamond-shaped subdag of width $\Omega(d)$, where d is the distance of u and v, as shown in Figure 6.3(c).

We can obtain lower bounds on the cache complexity Q using the following lemma which is proven in [25].

Lemma 16 *Suppose G is a computation dag where all inputs can reach all outputs through vertex-disjoint paths, and its information speed function is $\Omega(F_G(d))$. If $F_G(d)$ is monotonically increasing, and $F_G^{-1}(d)$ exists, then the number of cache misses required to execute G is*

$$Q = \Omega(K/F_G^{-1}(Z)) ,$$

where K is the total number of vertices on the vertex-disjoint paths or lines. □

We use Lemma 16 to prove a lower bound on the cache complexity of any algorithm computing n generations of a Jacobi multipass filter on n elements by finding an upper bound on F_G^{-1}.

Theorem 17 *Any scheduling of the computation dag induced by the* JACOBI-ITER *algorithm on an array of size n incurs $\Omega(1 + n/L + n^2/LZ)$ cache misses.*

Proof. This theorem can be proven by applying three lower bounds:

1. Suppose that $L = 1$. We can lower-bound the cache complexity using Lemma 16. Consider the subnetwork of the dag of JACOBI-ITER that includes only one third of the edges, as shown in Figure 6.3(b). The subnetwork has n lines with $K = \Theta(n^2)$ vertices. The information speed function is $F(d) = \Omega(d)$, since a diamond-shaped subdag of width $\Omega(d-2)$ is enclosed by two nodes as illustrated in Figure 6.3(c) for $d = 4$. Therefore $F^{-1}(d) = O(d)$ and the resulting lower bound for Q is $Q(n) = \Omega(n^2/Z)$.

   At most L data items are moved into cache when a cache miss occurs. Thus, a first lower bound for $L > 1$ is

   $$Q(n) = \Omega(n^2/ZL) .$$

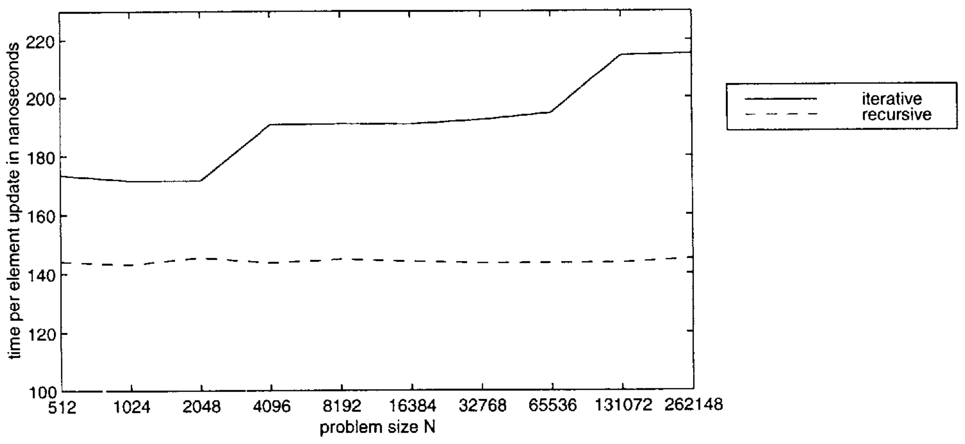


Figure 6-7: Plot of update time per element per generation for optimized iterative and recursive implementations of a multipass filter on a 167-MHz UltraSparc with 16kB L1-cache and 512kB L2-cache.

2. The algorithm must read all $\Theta(n)$ inputs, which reside on $\Omega(n/L)$ cache lines. This yields the second lower bound of $\Omega(n/L)$.

3. The third lower bound is the trivial lower bound of $Q(n) = \Omega(1)$.

By combining these lower bounds we get $Q(n) = \Omega(1 + n/L + n^2/LZ)$. $\square$

6.4 *Experimental results*

We now compare optimized implementations of the iterative and the recursive algorithms for the simple update rule given in Equation (6.1). (The iterative algorithm uses only 2 temporary variables, and the recursive implementation uses a "unfolded" [18] base case.) Figure 6-7 shows a plot of the update time per element per generation for the two versions on a 167-MHz Sun UltraSparc with 16kB L1-cache and 512kB L2-cache. The update time for the recursive algorithm is not only faster than the iterative algorithm, it is also nearly constant, whereas the iterative implementation slows down with every new level of the memory hierarchy. For arrays that do not fit in L2-cache, the recursive implementation executes in less than 70% of the time of the iterative version. The gain can be even higher for out-of-core algorithms, because disk bandwidth is considerably less than memory or cache bandwidths.

Summary

In this section, we have presented an optimal recursive algorithm to compute a multipass filter over one-dimensional data. We compared its cache complexity to a iterative algorithm and gave some brief empirical results for this problem. The recursive algorithm executes in less than 70% of the time of the iterative algorithm on problems that do not fit in $L2$-cache. The technique presented here can be extended to multidimensional stencil-filters. I expect that the advantage of the cache-oblivious algorithm on the multidimensional data will prove to be even greater.

SECTION 7

Cache complexity of ordinary algorithms

This section analyzes the cache complexity of the "ordinary" algorithms for matrix transposition, matrix multiplication, and sorting. Although optimal in the random-access machine model [4] and cache oblivious, these algorithms are not asymptotically optimal with respect to cache misses. We first prove that the number of cache misses of algorithms with a "regular" complexity bound (as defined later) is asymptotically the same even if least-recently-used (LRU) is used instead of optimal replacement. We then show that the standard iterative algorithm to transpose a matrix incurs $\Omega(n^2)$ cache misses on a $n \times n$ matrix matching the trivial upper bound of one cache miss per time step. The ordinary iterative algorithm to multiply two $n \times n$ matrices incurs $\Omega(n^3)$ cache misses, which is also the worst possible asymptotic behavior for an $O(n^3)$-work algorithm. Many "ordinary" algorithms for sorting exit. We pick mergesort and prove that its cache complexity is $\Omega((n/L)\lg(n/Z))$ when sorting an array of n elements, which is a factor of $\Theta(\lg Z)$ away from optimal.

The ideal-cache model is well suited for algorithm design and upper-bound analyses. This comes in part from the optimal replacement strategy employed by the ideal-cache.

Lower-bounding the cache complexity of an algorithm with optimal replacement is somewhat hard, since it must be proven that the optimal replacement strategy will do. For upper bounds, we can pick any replacement strategy we want and the optimal replacement will perform as least as well as our arbitrary strategy. However, for lower bounds we must be more careful. We usually do not know which line the optimal replacement strategy would replace. The following analysis shows that the optimal and omniscient replacement strategy used by an ideal cache can be simulated efficiently by the LRU replacement strategy. The LRU strategy replaces the cache line whose most recent access was earliest among all lines in the associativity set. In fact, for algorithms with a "regular" complexity bound, LRU and optimal replacement yield the same asymptotic bounds. We define a cache complexity bound $Q(n; Z, L)$ to be ***regular*** if

$$Q(n; Z, L) = O(Q(n; 2Z, L)) . \tag{7.1}$$

Lemma 18 *Consider an algorithm that causes $Q^*(n; Z, L)$ cache misses on a problem of size n using a (Z, L) ideal cache. Then, the same algorithm incurs $Q(n; Z, L) \leq 2Q^*(n; Z/2, L)$ cache misses on a (Z, L) cache that uses LRU replacement.*

Proof. Sleator and Tarjan [37] have shown that the cache misses on a (Z, L) cache using LRU replacement is $(Z/(Z - Z^* + 1))$-competitive with optimal replacement on a (Z^*, L) ideal if both caches start with an empty cache. It follows that the number of misses on a (Z, L) LRU-cache is at most twice the number of misses on a $(Z/2, L)$ ideal-cache. □

Corollary 19 *For algorithms with regular cache complexity bounds, the asymptotic number of cache misses is the same for LRU and optimal replacement.*

Proof. This corollary follows directly from Lemma 18 and the regularity condition. □

The same argument extends to a variety of other replacement strategies [11], including:

flush when full: Whenever there is a cache miss and there is no space left in the cache, evict all lines currently in the cache (call this action a "flush").

clock replacement: An approximation to LRU in which a single "use bit" replaces the implicit (time of last access) timestamp of LRU.

first-in, first-out: Replace the line that has been in the fast memory longest.

random: Whenever a cache miss occurs, evict a page chosen randomly and uniformly among all fast memory pages.

We shall use Corollary 19 in the following lower-bound proofs and assume that the cache is handled by LRU to simplify our analyses. If an algorithm analyzed with LRU is regular, then the optimal strategy must also be regular. Therfore, according to Corollary 19 the bound derived with the LRU analysis applies to the ideal cache model as well.

7.1 *Matrix multiplication*

In this section, we analyze the straightforward iterative algorithm for matrix multiplication. We prove that it causes $\Omega(n^3)$ cache misses when the $n \times n$ matrices are stored in row-major order and do not fit in cache. We further show that even if the matrices are stored in the order in which they are used and do not fit in cache, the number of cache misses is at least $\Omega(n^3/L)$, compared to $\Theta(n^3/L\sqrt{Z})$ for an the cache-optimal $\Theta(n^3)$-work algorithm presented in Section 2.

The simplest way to compute the product of two matrices is to evaluate the formula

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

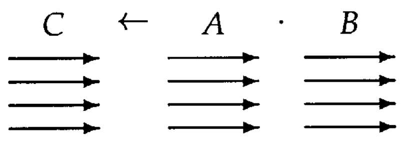
directly, as in the following program:

```
ORD-MULT($A, B, C, n$)
1  for $i \leftarrow 1$ to $n$
2      do for $j \leftarrow 1$ to $n$
3             do $c_{ij} \leftarrow 0$
4                for $k \leftarrow 1$ to $n$
5                    do $c_{ij} \leftarrow c_{ij} + a_{ik} b_{kj}$
```

Theorem 20 *The* ORD-MULT *algorithm for matrix multiplication uses $\Theta(n^3)$ work when multiplying $n \times n$ matrices that do not fit in cache. It incurs $\Omega(n^3)$ cache misses, when the matrices are stored in row-major order. Even if the matrices are stored in the order in which they are used,* ORD-MULT *incurs $\Omega(n^3/L)$ cache misses, which is a factor of $\Theta(\sqrt{Z})$ from optimal.*

Proof. Analyzing the work of ORD-MULT(A, B) is straightforward. Since there are three nested loops, each of which performs n iterations, the work is $\Theta(n^3)$. Since the algorithm cannot access more than $O(1)$ elements in constant time, $O(n^3)$ is also an upper bound on the number of cache misses for this algorithm.

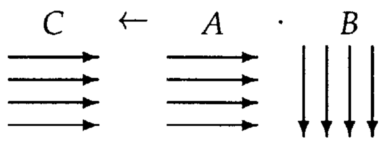
First, we assume that both matrices are stored in row-major order (Figure 2-1(a)):



The number of cache misses can be lower-bounded by counting only the misses caused by reading matrix B. In the ith iteration of the outer loop, the elements of the ith row of C are computed. While i is fixed the inner two loops iterate over all n^2 values of k and j, reading all elements of matrix B column by column.

Assuming that the matrix does not fit in cache, e.g. $n \gg Z/L$, the LRU replacement strategy overwrites lines of matrix B before they can be reused. Therefore, the number of misses on matrix B to compute one element of matrix C is $\Omega(n)$. Since C contains n^2 elements, the algorithm causes $\Omega(n^3)$ cache misses. It follows from Corollary 19 that even with an optimal replacement strategy, ORD-MULT incurs $\Omega(n^3)$ cache misses.

ORD-MULT(A, B) does not exhibit good cache behavior. Accesses to the same element, or at least to the same cache line, are far apart. The spatial locality of the memory accesses can be improved by changing the memory layout of the matrices: The previous analysis showed that the accesses to matrix B alone cause $\Theta(n^3)$ cache misses. The problem is that matrix B is stored in row-major order, but accessed columnwise. Assume that the memory layout for B is changed from row-major to column-major order (Figure 2-1(b)):



Now, both matrices are accessed in the order in which they are stored. As long as the cache can provide a single line for each of A, B, and C, for each cache miss, the following $L-1$ accesses are cache hits. Hence, the number of cache misses is $\Theta(n^3/L)$, which is a factor of $\Theta(L)$ improvement over the previous algorithm. This improvement comes with the disadvantage that the matrices are not stored uniformly and it is still a factor of $\Theta(\sqrt{Z})$ away from the cache-optimal algorithms shown in Sections 1 and 2. □

7.2 *Matrix transposition*

In this section, we argue that the iterative algorithm for matrix transposition causes $\Omega(n^2)$ cache misses on a $n \times n$ matrix, when the matrix is stored in row- or column-major order (Figure 2-1(a,b)). This is a factor of $\Theta(L)$ more cache misses than the cache-optimal algorithm presented in Section 3.

The ordinary algorithm for matrix transposition walks through the matrix row by row and swaps elements:

```
ORD-TRANSPOSE(A, B, n)
1  for $i \leftarrow 1$ to $n$
2      do for $j \leftarrow 1$ to $n$
3             do $b_{ij} \leftarrow a_{ji}$
```

Theorem 21 *The* ORD-TRANSPOSE *algorithm for matrix multiplication uses $\Theta(n^2)$ work and incurs $\Omega(n^2)$ cache misses, when transposing a $n \times n$ matrix that does not fit into cache.*

Proof. Transposing a matrix is equivalent to changing the memory layout from row- to column-major layout or *vice versa*. Here, we show that accessing in column-major order a matrix stored in row-major layout causes $\Omega(n^2)$ cache misses. After Z/L cache misses, the cache is filled and lines must be evicted. The LRU strategy replaces the lines in the same order in which they are read. Therefore, after n accesses, when a line could be reused, it has been evicted from cache by LRU replacement. Thus, all n^2 accesses are cache misses. Since $\Omega(n^3)$ is regular, it follows from Corollary 19 that ORD-TRANSPOSE incurs $\Omega(n^2)$ cache misses in the ideal-cache model. □

7.3 *Mergesort*

We have just shown that divide-and-conquer algorithms presented in Sections 2 and 3 for matrix multiplication and matrix transposition incur fewer cache misses than their iterative counterparts. In this section, we show that divide-and-conquer algorithms are not *per se* cache-optimal. Specifically, we show that Mergesort [16, p. 13] incurs $\Omega((n/L)\lg(n/Z))$ cache misses for an input of size n, which is a factor of $\Theta(\lg Z)$ more cache misses than the cache-optimal algorithms presented in Sections 4 and 5.

Mergesort is a recursive sorting algorithm that divides the input sequence in two parts, sorts them recursively and then merges the two sorted subsequences into one sorted sequence. The following pseudocode is the standard description of mergesort and can be found in a variety of textbooks [16, 34].

```
MERGESORT(A, p, r)
1  if $p < r$
2    then $q \leftarrow \lfloor (p+r)/2 \rfloor$
3         MERGESORT(A, p, q)
4         MERGESORT(A, q + 1, r)
5         MERGE(A, p, q, r)
```

We assume that the input array A of length n is stored in $\Theta(n)$ contiguous memory locations. MERGESORT uses an auxiliary procedure MERGE(A, p, q, r) that merges two sorted subarrays $A[p\,..\,q]$ and $A[q+1\,..\,r]$ into a single sorted subarray that replaces the current subarray $A[p\,..\,r]$. Merging two subarrays of length $n/2$ uses $\Theta(n)$ work and causes $O(n/L)$ cache misses assuming that $Z/L \geq 3$, since the $\Theta(n)$ data items can be accessed in linear order and each cache miss brings L elements into cache.

The work of MERGESORT is $\Theta(n \lg n)$, which is optimal in the random-access machine model [16, p. 172]. Although mergesort is a divide-and-conquer algorithm, its cache complexity is not asymptotically optimal.

Lemma 22 MERGESORT *incurs* $\Omega((n/L)\lg(n/Z))$ *cache misses for an input of size n.*

Proof. The cache complexity of MERGESORT can be described by the recurrence:

$$Q(n) = \begin{cases} \Omega(n/L) & \text{if } n \leq \alpha Z\,, \\ 2Q(n/2) + \Omega(n/L) & \text{otherwise}\,, \end{cases} \tag{7.2}$$

where α is a sufficiently small constant. The base case arises when the $\Theta(n)$ elements fit into the cache. Sorting $\Theta(n)$ elements requires $\Theta(n)$ auxiliary storage for the merging procedure. In the recursive case where $n > \alpha Z$, two subproblems of half the size are solved and then merged together. After line 3 of MERGESORT is executed, LRU replacement will have evicted most of the $n/2$ data items used in the first recursive call. Thus, $\Omega(n)$ data elements must be read from contiguous memory locations incurring $\Omega(n/L)$ cache misses. The solution of Equation (7.2) is $\Omega((n/L)\lg(n/Z))$. □

Summary

In this section, we have shown that the cache complexity of the ordinary algorithms for matrix transposition, matrix multiplication, and sorting are not asymptotically optimal. We have proven in Corollary 19 that the optimal replacement strategy can be efficiently simulated by the LRU replacement strategy. For algorithms with regular complexity bounds LRU and optimal replacement yield the same asymptotic bounds. Since it is easier to analyze the caching behavior of an algorithm when we understand what the replacement strategy does, this Corollary often helps to simplify the analyses.

SECTION 8

Other cache models

In this section we show that cache-oblivious algorithms designed in the two-level ideal-cache model can be efficiently ported to other cache models. We show that algorithms with regular complexity bounds (Equation (7.1)) (including all algorithms heretofore presented) can be ported to less-ideal caches incorporating least-recently-used (LRU) or first-in, first-out (FIFO) replacement policies [24, p. 378]. We argue that optimal cache-oblivious algorithms are also optimal for multilevel caches. Finally, we present simulation results proving that optimal cache-oblivious algorithms satisfying the regularity condition are also optimal (in expectation) in the previously studied SUMH [5, 42] and HMM [1] models. Thus, all the algorithmic results in this thesis apply to these models, matching the best bounds previously achieved.

8.1 Two-level models

Many researchers, such as [3, 25, 43], employ two-level models similar to the ideal-cache model, but without an automatic replacement strategy. In these models, data must be moved explicitly between the the primary and secondary levels "by hand."

We now show that optimal algorithms in the ideal-cache model whose cache complexity bounds are regular (Equation (7.1)) can be ported to these models to run using optimal work and incurring an optimal expected number of cache misses. Since previous two-level models do not support automatic replacement, to port a cache-oblivious algorithms to them, we implement an LRU (or FIFO) replacement strategy in software.

Lemma 23 *A (Z, L) LRU cache (or FIFO-cache) can be maintained using $O(Z)$ primary memory locations such that every access to a cache line in primary memory takes $O(1)$ expected time.*

Proof. Given the address of the memory location to be accessed, we use a 2-universal hash function [29, p. 216] to maintain a hash table of cache lines present in the primary memory. The Z/L entries in the hash table point to linked lists in a heap of memory containing Z/L records corresponding to the cache lines. The 2-universal hash function guarantees that the expected size of a chain is $O(1)$. All records in the heap are organized as a doubly linked list in the LRU order (or singly linked for FIFO). Thus, the LRU (FIFO) replacement policy can be implemented in $O(1)$ expected time using $O(Z/L)$ records of $O(L)$ words each. □

Theorem 24 *An optimal cache-oblivious algorithm with a regular cache-complexity bound can be implemented optimally in expectation in two-level models with explicit memory management.* □

Consequently, our cache-oblivious algorithms for matrix multiplication, matrix transposition, FFT, and sorting are optimal in two-level models with explicit memory management.

8.2 Multilevel ideal caches

We now show that optimal cache-oblivious algorithms also perform optimally in computers with multiple levels of ideal caches. Moreover, Theorem 24 extends to multilevel models with explicit memory management.

The $\langle (Z_1, L_1), (Z_2, L_2), \ldots, (Z_r, L_r) \rangle$ ***ideal-cache model*** consists of an arbitrarily large main memory and a hierarchy of r caches, each of which is managed by an optimal replacement strategy. The model assumes that the caches satisfy the ***inclusion property*** [24, p. 723], which says that for $i = 1, 2, \ldots, r-1$, the values stored in cache i are also stored in cache $i+1$. The performance of an algorithm running on an input of size n is measured by its work complexity $W(n)$ and its cache complexities $Q_i(n; Z_i, L_i)$ for each level $i = 1, 2, \ldots, r$.

Theorem 25 *An optimal cache-oblivious algorithm in the ideal-cache model incurs an asymptotically optimal number of cache misses on each level of a multilevel cache with optimal replacement.*

Proof. The theorem follows directly from the definition of cache obliviousness and the optimality of the algorithm in the two-level ideal-cache model. □

Theorem 26 *An optimal cache-oblivious algorithm with a regular cache-complexity bound incurs an asymptotically optimal number of cache misses on each level of a multilevel cache with LRU or optimal replacement.*

Proof. Follows from Corollary 19 and Theorem 25. □

8.3 *The SUMH model*

In 1990 Alpern *et al.* [5] presented the uniform memory hierarchy model (UMH), a parameterized model for a memory hierarchy. In the $\mathrm{UMH}_{\alpha,\rho,b(l)}$ model, for integer constants $\alpha, \rho > 1$, the size of the ith memory level is $Z_i = \alpha\rho^{2i}$ and the line length is $L_i = \rho^i$. A transfer of one ρ^l-length line between the caches on level l and $l+1$ takes $\rho^l/b(l)$ time. The bandwidth function $b(l)$ must be nonincreasing. The processor can access the cache on level 1 in constant time per access. An algorithm given for the UMH model must include a schedule that, for a particular set of input variables, tells exactly when each block is moved along which of the buses between caches. Work and cache misses are folded into one cost measure $T(n)$. Alpern *et al.* prove that an algorithm that performs the optimal number of cache misses at all levels of the hierarchy does not necessarily run in optimal time in the UMH model, since scheduling bottlenecks can occur when all buses are active. In the more restrictive SUMH model [42], however, only one bus is active at a time. Consequently, we can prove that optimal cache-oblivious algorithms run in optimal expected time in the SUMH model.

Lemma 27 *A cache-oblivious algorithm with $W(n)$ work and $Q(n; Z, L)$ cache misses on a (Z, L)-ideal cache can be executed in the $\mathrm{SUMH}_{\alpha,\rho,b(l)}$ model in expected time*

$$T(n) = O\Big(W(n) + \sum_{i=1}^{r-1} \frac{\rho^i}{b(i)} Q(n; \Theta(Z_i), L_i)\Big) ,$$

where $Z_i = \alpha\rho^{2i}$, $L_i = \rho^i$, and Z_r is big enough to hold all elements used during the execution of the algorithm.

Proof. Use the memory at the ith level as a cache of size $Z_i = \alpha\rho^{2i}$ with line length $L_i = \rho^i$ and manage it with software LRU described in Lemma 23. The rth level is the main memory, which is direct mapped and not organized by the software LRU mechanism. An LRU cache of size $\Theta(Z_i)$ can be simulated by the ith level, since it has size Z_i. Thus, the number of cache misses at level i is $2Q(n;\Theta(Z_i),L_i)$, and each takes $\rho^i/b(i)$ time. Since only one memory movement happens at any point in time and there are $O(W(n))$ accesses to level 1, the lemma follows by summing the individual costs. ☐

Lemma 28 *Consider a cache-optimal algorithm whose work on a problem of size n is lower-bounded by $W^*(n)$ and whose cache complexity is lower-bounded by $Q^*(n;Z,L)$ on an (Z,L) ideal-cache. Then, no matter how data movement is implemented in $SUMH_{\alpha,\rho,b(l)}$, the time taken on a problem of size n is at least*

$$T(n) = \Omega\Big(W^*(n) + \sum_{i=1}^{r} \frac{\rho^i}{b(i)} Q^*(n,\Theta(Z_j),L_i)\Big) ,$$

where $Z_i = \alpha\rho^{2i}$, $L_i = \rho^i$ and Z_r is big enough to hold all elements used during the execution of the algorithm.

Proof. The optimal scheduling of the data movements does not need to obey the inclusion property, and thus the number of ith-level cache misses is at least as large as for an ideal cache of size $\sum_{j=1}^{i} Z_i = O(Z_i)$. Since $Q^*(n,Z,L)$ lower-bounds the cache misses on a cache of size Z, at least $Q^*(n,\Theta(Z_i),L_i)$ data movements occur at level i, each of which takes $\rho^i/b(i)$ time. Since only one movement can occur at a time, the total cost is the maximum of the work and the sum of the costs at all the levels. ☐

Theorem 29 *A cache-oblivious algorithm that is optimal in the ideal-cache model and whose cache complexity is regular can be executed in the $SUMH_{\alpha,\rho,b(l)}$ model in optimal expected time.*

Proof. The theorem follows directly from regularity and Lemmata 27 and 28. ☐

8.4 The HMM model

Aggarwal, Alpern, Chandra, and Snir [1] proposed the hierarchical memory model (HMM) in which an access to location x takes $f(x)$ time. The authors assume that f is a monotonically nondecreasing function, usually of the form $\lceil \log x \rceil$ or $\lceil x^\alpha \rceil$.

Lemma 30 *Consider a cache-oblivious algorithm with $W(n)$ work and $Q(n;Z,L)$ cache misses on a (Z,L) ideal cache. Let $Z_1 < Z_2 < \cdots < Z_r$ be positive integers such that a* *cache of size Z_r can hold all of the data used during the execution of the algorithm. Then, the algorithm can be executed in the HMM model with cost function f in expected time*

$$T(n) = O\Big(W(n)f(s_1) + \sum_{i=2}^{r} f(s_i)Q(n;\Theta(Z_i),1)\Big) ,$$

where $s_1 = O(Z_1)$, $s_2 = s_1 + O(Z_2)$, ..., $s_r = s_{r-1} + O(Z_r)$.

Proof. Using Lemma 23 we can simulate a $\langle(Z_1,1),(Z_2,1),\ldots,(Z_r,1)\rangle$ LRU cache in the HMM model by using locations $1,2,\ldots,s_1$ to implement cache 1, locations $s_1+1, s_1+2,\ldots,s_2$ to implement cache 2, etc. The cost of each access to the ith cache is at most $f(s_i)$. Cache 1 is accessed at most $W(n)$ times, cache 2 is accessed at most $Q(n;\Theta(Z_2),1)$ times, and so forth. The lemma follows. $\square$

Lemma 31 *Consider a cache-optimal algorithm whose work on a problem of size n is lower-bounded by $W^*(n)$ and whose cache complexity is lower-bounded by $Q^*(n;Z,L)$ on an (Z,L) ideal cache. Then, no matter how data movement is implemented in an HMM model with cost function f, the time taken on a problem of size n is at least*

$$T(n) = \Omega\Big(W^*(n) + \sum_{i=1}^{r} (f(Z_{i-1}-1) - f(Z_{i-2}-1))Q^*(n;Z_i,1)\Big)$$

for any $Z_0 = 1 < Z_1 < \cdots < Z_r$ such that a cache of size Z_r can hold all of the data used during the execution of the algorithm.

Proof. The memory of the HMM model can be viewed as a cache hierarchy with arbitrary parameters $Z_0 = 1 < Z_1 < \cdots < Z_r$, where the memory elements are mapped to fixed locations in the caches. The processor works on elements in the level 0 cache with $\Theta(1)$ cost. The first $Z_1 - 1$ elements of the HMM memory are kept in the level 1 cache, the first $Z_2 - 1$ elements in the level 2 cache, etc. One element in each cache is used as a "dynamic entry" which allows access to elements on higher levels. Accessing a location at level i is then done by incorporating the memory item in the dynamic element of each of the caches closer to the processor. This "cache hierarchy" obeys the inclusion principle, but it does not do any replacement. Memory elements are exchanged—as in HMM—by moving them to the processor and writing them back to their new location.

If we charge $f(Z_{i-1}-1) - f(Z_{i-2}-1)$ to a cache miss on cache i, an access to element at position x in cache at level k costs $\sum_{i=1}^{k} f(Z_{i-1}-1) - f(Z_{i-2}-1) = f(Z_{k-1}-1) - f(0)$, which is at most $f(x)$. Thus, the access cost for accessing element x is the same in the HMM as in this "cached" HMM model. The cost $T(n)$

of an algorithm in the HMM model can be bounded by the cost of the algorithm in the multilevel model, which is at least

$$T(n) = \Omega\Big(W(n) + \sum_{i=1}^{r} (f(Z_i - 1) - f(Z_{i-1} - 1))Q(n; Z_i, 1)\Big) .$$

Since $W(n) \geq W^*(n)$ and $Q(n; Z_i, 1) \geq Q^*(n; Z_i, 1)$, the lemma follows. □

Theorem 32 *A cache-oblivious algorithm that is optimal in the ideal-cache model and whose cache complexity is regular can be executed in optimal expected time in the HMM model, if the cost function is monotonically nondecreasing and satisfies $f(2x) = \Theta(f(x))$.*

Proof. Assume that the cache at level r is big enough to hold all elements used during the execution of the algorithm. We choose $Z_1, \cdots, Z_r$ such that $2f(Z_{i-1} - 1) \leq Z_i - 1 = O(Z_{i-1} - 1)$ for all $1 < i \leq r$. Such a sequence can be computed given that f is monotonically nondecreasing and satisfies $f(2x) = \Theta(f(x))$.

We execute the algorithm as described in Lemma 30 on the HMM model with $2Z_1, 2Z_2, \ldots, 2Z_r$. The cost of warming up the caches is $\sum_{1 \leq i \leq O(Z_r)} f(i) = \Theta(Z_r f(Z_r))$ which is asymptotically no greater than the cost of the algorithm even if it accesses each input item just once. The result follows from Lemmata 18, 30 and 31. □

Summary

One strength of the ideal-cache model, compared to other models studied in the literature, is that designing and analyzing algorithms is easier. But this section shows that the assumptions of the ideal-cache model are not stronger than the assumptions of two hierarchical memory models in the literature. Specifically, we have shown that optimal cache-oblivious algorithms in the ideal-cache model are also optimal in the hierarchical memory model (HMM) [1] and in the serial uniform memory hierarchy (SUMH) model [5, 42].

Due to its simplifications, the ideal-cache model falls short of modeling some of the idiosyncrasies of a real-world memory hierarchy. It ignores issues such as conflict misses, and has only one level of caching. In developing recursive algorithms, however, we have found that these additional complications are comparatively easy to deal with once an algorithm has been designed in the ideal model.

SECTION 9

Related work

In this section, we discuss the origin of the notion of cache obliviousness. We also give an overview of other hierarchical memory models.

Our research group at MIT noticed as far back as 1994 that divide-and-conquer matrix multiplication was a cache-optimal algorithm that required no tuning, but we did not adopt the term "cache-oblivious" until 1997. This matrix-multiplication algorithm, as well as a cache-oblivious algorithm for LU-decomposition without pivoting, eventually appeared in [9]. Shortly after leaving our research group, Toledo [40] independently proposed a cache-oblivious algorithm for LU-decomposition, but with pivoting. For $n \times n$ matrices, Toledo's algorithm uses $\Theta(n^3)$ work and incurs $\Theta(1 + n^2/L + n^3/L\sqrt{Z})$ cache misses. More recently, our group has produced an FFT library called FFTW [20], which in its most recent incarnation [19], employs a register-allocation and scheduling algorithm inspired by our cache-oblivious FFT algorithm. The general idea that divide-and-conquer enhances memory locality has been known for a long time [36].

Previous theoretical work on understanding hierarchical memories and the I/O-complexity of algorithms has been studied in cache-aware models lacking an automatic replacement strategy. Hong and Kung [25] use the red-blue pebble game to prove lower bounds on the I/O-complexity of matrix multiplication, FFT, and other problems. The red-blue pebble game models temporal locality using two levels of memory. The model was extended by Savage [33] for deeper memory hierarchies. Aggarwal and Vitter [3] introduced spatial locality and investigated a two-level memory in which a block of P contiguous items can be transferred in one step. They obtained tight bounds for matrix multiplication, FFT, sorting, and other problems. The hierarchical memory model (HMM) by Aggarwal et al. [1] treats memory as a linear array, where the cost of an access to element at location x is given by a cost function $f(x)$. The BT model [2] extends HMM to support block transfers. The UMH model by Alpern et al. [5] is a multilevel model that allows I/O at different levels to proceed in parallel. Vitter and Shriver introduce parallelism, and they give algorithms for matrix multiplication, FFT, sorting, and other problems in both a two-level model [43] and several parallel hierarchical memory models [44]. Vitter [41] provides a comprehensive survey of external-memory algorithms.

Wir stehen selbst enttäuscht und sehn betroffen
Den Vorhang zu und alle Fragen offen. [...]
Verehrtes Publikum, los, such dir selbst den Schluß!
Es muß ein guter da sein, muß, muß, muß!

We feel deflated too. We too are nettled
To see the curtain down and nothing settled. [...]
You write the happy ending to the play!
There must, there must, there's got to be a way!

BERTOLD BRECHT, *Der gute Mensch von Sezuan*, 1940

SECTION 10

Conclusion

This thesis has introduced the notion of cache obliviousness and has presented asymptotically optimal cache-oblivious algorithms for fundamental problems. Figure 10 gives an overview of the known efficient cache-oblivious algorithms, most of which are described in this thesis. Two that we have not discussed are matrix addition and LUP-decomposition. For matrix addition, a simple iterative algorithm turns out to be cache-optimal if the matrix elements are read in the same order in which they are stored in memory. The algorithm for LUP-decomposition is due to Toledo [40], but it uses cache-aware algorithms as subprocedures. By applying the cache-oblivious algorithms presented here, however, his algorithm can be converted into a cache-oblivious one.

The remainder of this section outlines research questions related to cache obliviousness. Section 10.1 discusses the engineering task of implementing cache-oblivious algorithms. Section 10.2 discusses cache-oblivious data structures and briefly presents a cache-oblivious data structure for static binary search trees. Section 10.3 raises two theoretical questions about the general power of cache-oblivious algorithms. Section 10.4 discusses divide-and-conquer as a programming strategy and the tools needed to help programmers to write recursive programs. In Section 10.5, I argue that because divide-and-conquer works well with cache hierarchies and also with parallel computers, the coming revolution of shared-memory multiprocessors will make this design paradigm of paramount importance.

Algorithm	Cache complexity	Optimal?
Matrix Multiplication	$\Theta(m+n+p+(mn+np+mp)/L + mnp/L\sqrt{Z})$	tight lower bound unknown
Strassen's Algorithm	$\Theta(n+n^2/L+n^{\log_2 7}/L\sqrt{Z})$	tight lower bound unknown
Matrix Transpose	$\Theta(1+n^2/L)$	yes
Matrix Addition[†]	$\Theta(1+n^2/L)$	yes
LUP-decomposition[‡] [40]	$\Theta(1+n^2/L+n^3/L\sqrt{Z})$	tight lower bound unknown
Discrete Fourier Transform	$\Theta(1+(n/L)(1+\log_Z n))$	yes
Distribution sort	$\Theta(1+(n/L)(1+\log_Z n))$	yes
Funnelsort	$\Theta(1+(n/L)(1+\log_Z n))$	yes
Jacobi multipass filter	$\Theta(1+n/L+n^2/ZL)$	yes

Figure 10-1: Overview of the known cache-oblivious algorithms. Except for matrix addition (†) and LUP-decomposition (‡), all these algorithms are presented in this thesis.

10.1 *Engineering cache-oblivious algorithms*

The job is not done after an efficient algorithm has been designed in the ideal-cache model. The software-engineering task of programming the algorithm on a real machine remains to be done. This task often involves coping with the less-than-ideal behavior of real caches. Nevertheless, if the original algorithm in the ideal-cache model exploits locality effectively, a program based on the algorithm can usually be made to run efficiently in practice. If the algorithm fails to exploit locality in the ideal-cache model, the algorithm will be slow no matter what the real-world computer environment looks like.

The trend in architecture is towards bigger caches with steeper hierarchies and towards new cache organizations which employ more "intelligent" algorithms to use the cache memory more effectively. But even when caches get more intelligent, the algorithm designer retains responsibility to ensure that frequently accessed data has the opportunity to reside in cache.

Both cache-aware and cache-oblivious strategies can be used to achieve good caching behavior of an algorithm. This thesis has shown that optimal cache-oblivious algorithms exist which have the same cache complexity as optimal cache-aware algorithms. But how do these two strategies compare in practice? How much faster is a cache-aware algorithm optimized for a given architecture than a cache-oblivious algorithm that solved the same problem?

Initial measurements I have taken indicate that cache-oblivious algorithms can rival the performance of hand-tuned cache-aware code, but in general cache-aware programs are faster. I hope to quantify this difference, as well as resolve other empirical questions. How do more levels of caching affect the difference between cache-aware and cache-oblivious algorithms? By how much do cache-aware algorithms slow down when executed on hardware they are not optimized for? Answers to these questions will become increasingly important as cache hierarchies become more pronounced.

10.2 Cache-oblivious data structures

This section discusses cache-oblivious data structures and briefly presents a cache-oblivious data structure for static binary search trees.

As there are cache-oblivious algorithms, there are cache-oblivious data structures. The blocked layout (Figure 2-1(c)), for example, is cache aware. To optimize it for a certain cache, the line length must be known. The bit-interleaved layout (Figure 2-1(d)), however, is cache oblivious and has the same asymptotic behavior as the blocked layout for matrix multiplication. Other cache oblivious layouts for matrices exist like the Morton or Hilbert layouts discussed in [12, 13, 18].

Different data layouts can greatly affect the asymptotic behavior of an algorithm. For cache-optimal matrix multiplication, as discussed in Section 2, the tall cache requirement can be relaxed if matrices are stored in blocked (Figure 2-1(c)) or bit-interleaved order (Figure 2-1(d)). In Section 7.1 we have shown that the number of cache misses for the ordinary matrix multiplication algorithm can be reduced by a factor of $\Theta(L)$ by choosing a different data layout.

Can the idea of cache obliviousness be extended to data structures? Do efficient cache-oblivious data structures exist for dynamic data structures, such as linked lists, heaps, or trees? Although I do not yet know the answer to this question, I have been able to devise a cache-oblivious layout for static binary search trees that is $O(1)$-competitive with the performance of B-Trees [28], which are used in file systems and other out-of-core applications because of their low cache complexity.

Figure 10-2 shows the cache-oblivious layout for a complete binary search tree of height 4. Let T be a complete binary tree of height $h = \Theta(\lg n)$, where n is the number of elements in the tree. To find the layout, divide T at level $\lfloor h/2 \rfloor$, which separates T into subtree T_0 (top $\lfloor h/2 \rfloor$ levels) and $k \leq 2^{\lfloor h/2 \rfloor}$ subtrees having height at most $\lceil h/2 \rceil$. The cache-oblivious data layout $\mathcal{L}(T)$ of T is defined recursively as follows.

$$\mathcal{L}(T) = \mathcal{L}(T_0) \mathbin{\|} \mathcal{L}(T_1) \mathbin{\|} \cdots \mathbin{\|} \mathcal{L}(T_k) \,,$$

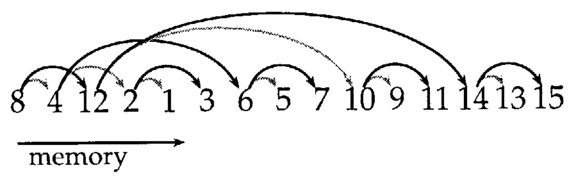


Figure 10-2: Cache-oblivious layout of a binary tree of height 4 with 15 elements 1, 2, . . . , 15. The values stored in the nodes of the tree are shown in the order in which they are stored. Pointers to left children are shown in grey and pointers to right children in black.

where $||$ is the concatenation operator. The base case, when the tree has only one node, is trivial. The cache complexity of finding an element in this data structure on a (Z, L) ideal-cache is $O(\lg_L n)$, which is asymptotically equivalent to the performance of a B-Tree. Making this layout strategy work for dynamic search trees is a high research priority.

10.3 Complexity of cache obliviousness

In this section, we discuss whether a separation theorem can be proven, showing that certain problems can only be solved cache-optimally by a cache-aware algorithm. We also discuss whether a simulation result can be proven that bounds the advantage of cache-aware algorithms over cache-oblivious algorithms.

We know now that many optimal cache-oblivious algorithms exist. But, how powerful are cache-oblivious algorithms compared to cache-aware algorithms in general?

Separation: Is there a separation in asymptotic complexity between cache-aware and cache-oblivious algorithms?

It appears that cache-aware algorithms should be able to use caches better than cache-oblivious algorithms since they have more knowledge about the system they are running on. But so far, I have not found a cache-aware algorithm that has better asymptotic behavior than a well-designed cache-oblivious algorithm. Nevertheless, I do believe a seperating problem exists. I conjecture that for such a seperating problem, the best cache-oblivious algorithm has a factor of $\Omega(\lg Z)$ more cache misses than the best cache-aware algorithm.

Simulation: Given a class of optimal cache-aware algorithms to solve a single problem, can we construct a good cache-oblivious algorithm that solves the same problem?

I believe that the gap between cache-aware and cache-oblivious algorithms (if it exists) is not bigger than a factor of $O(\lg Z)$ difference. Perhaps this result can be proven by using simulation techniques to convert a class of cache-aware algorithms into a cache-oblivious algorithm. I have not yet had much success in this line of research, however.

10.4 *Compiler support for divide-and-conquer*

This section discusses how new compiler techniques can help to ease the programming of divide-and-conquer algorithms. Most algorithms given in this thesis are divide-and-conquer algorithms. Conventional wisdom says that recursive procedures should be converted into iterative loops in order to improve performance [8]. While this strategy was effective ten years ago, many recursive programs now actually run faster than their iterative counterparts. So far most of the work by architects and compiler writers is concentrated on loop-based iterative programs. Their tools are often not appropriate for recursion and divide-and-conquer programs.

For a divide-and-conquer algorithm to be efficient, the base case must be efficiently coded. Coding recursion with a simple "unit" base case is usually easy for a programmer, but then the overhead of the recursive implementation can be substantial. To get full performance out of a recursive algorithm, it is necessary to ***coarsen*** the base case of recursion (a transformation called "unfolding" in [18]), which is analogous to loop unrolling. Coarsening of base cases is motivated by the observation that for many recursive algorithms, the overhead of recursion is often in the lowest few levels, near the leaves. With a branching factor of 2, for example, 97% of the recursive function calls are in the bottom 5 levels of recursion. The proportion is even higher for branching factors greater than 2.

Typically, a variety of coarsened base cases must be written, making it hard to code by hand. Can a compiler effectively generate coarsened base cases? This problem is much like loop-unrolling, which is already done by compilers.

Matteo Frigo, a member of our research group, and Steve Johnson, also at MIT, have implemented a discrete Fourier transform library FFTW [20] that incorporates a cache-oblivious algorithm with a specialized compiler to generate coarsened base cases. I believe that parts of their technique can be extended to general divide-and-conquer algorithms.

FFTW also employs an adaptive runtime execution, which chooses base cases during an initialization phase of the program. This strategy is effective when the question of which of several coarsened base cases yields the fastest results on a given architecture cannot be determined at compile time. An adaptive execution strategy allows the compiler to produce several distinct base-case implementations at different granularities and with different strategies, and then at runtime initialization, choose the fastest for the particular machine by timing the various alternatives. Benchmarks performed on a variety of platforms show that FFTW's performance is typically superior to that of other publicly available FFT software, and it rivals or is better than many hand-coded vendor libraries.

10.5 *The future of divide-and-conquer*

Shared-memory multiprocessors are now available as deskside workstations and will appear in desktop PC's in the next two years and in mobile laptops within five years. Divide-and-conquer algorithms seem to be a perfect match for these parallel machines in which the technologies of parallelism and caching are converging.

In a shared-memory multiprocessor machine, multiple processors, each having its own cache, work together to solve problems faster, communicating through a single shared memory.

Our research group discovered, while working on the parallel programming language Cilk [39, 9], that divide-and-conquer programs work well with shared-memory multiprocessors. In Cilk a function can be "spawned", making it logically parallel to the spawning procedure. Since the Cilk scheduler decides at runtime whether two logically parallel functions are actually executed in parallel, a Cilk program is processor oblivious. It can be effectively executed on many processors, as long as the problem has enough inherent parallelism. Rugina and Rinard [32] have experimented with automatic parallelization from C to Cilk and achieved good speedups on divide-and-conquer programs.

Recursive calls can often be replaced by recursive spawns, which allow the children to work in parallel. Once the division phase is complete, the subproblems are usually independent and can therefore be solved in parallel. Our experiments with Cilk show that divide-and-conquer algorithms scale well and have good memory behavior on a parallel machine. The number of cache misses of a Cilk program can be upper-bounded using the cache complexity of its C elision (the Cilk program without the parallel keywords) as shown in [9, 10].

Can we design algorithms which are optimal with respect to work, parallelism, and cache complexity but which are also cache oblivious and processor oblivious? I believe that resource-oblivious versions of the algorithms given in this thesis can be proven to satisfy all three optimality requirements.

Small shared-memory multiprocessors are readily avaiable: A 4-processor machine costs less than $20,000 [26]. Most of these machines are designed to be servers, but workstations intended to be used by a single user are starting to appear [35]. These machines will become more common over the next few years, and it is expected that we will see a shared-memory multiprocessor-on-a-chip within a few years [23, 27]. Writing efficient parallel programs is considered hard. Caching problems are more pronounced in these machines than they are in single-processor machines. Memory hierarchies will be bigger and steeper in the future, and cache misses will be more expensive. The new Alpha 21264 chip [14], for example, can deliver 2 words from L1-cache in one cycle, but it takes around 100 cycles to fetch from main memory. Divide-and-conquer seems to provide a way to write processor- and cache-oblivious algorithms, which will help to ease programming on the future machines.

Bibliography

[1] AGGARWAL, A., ALPERN, B., CHANDRA, A. K., AND SNIR, M. A model for hierarchical memory. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing* (May 1987), pp. 305–314.

[2] AGGARWAL, A., CHANDRA, A. K., AND SNIR, M. Hierarchical memory with block transfer. In *28th Annual Symposium on Foundations of Computer Science* (Los Angeles, California, 12–14 Oct. 1987), IEEE, pp. 204–216.

[3] AGGARWAL, A., AND VITTER, J. S. The input/output complexity of sorting and related problems. *Communications of the ACM 31*, 9 (Sept. 1988), 1116–1127.

[4] AHO, A. V., HOPCROFT, J. E., AND ULLMAN, J. D. *The Design and Analysis of Computer Algorithms.* Addison-Wesley Publishing Company, 1974.

[5] ALPERN, B., CARTER, L., AND FEIG, E. Uniform memory hierarchies. In *Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science* (Oct. 1990), pp. 600–608.

[6] BAILEY, D. H. FFTs in external or hierarchical memory. *Journal of Supercomputing 4*, 1 (May 1990), 23–35.

[7] BELADY, L. A. A study of replacement algorithms for virtual storage computers. *IBM Systems Journal 5*, 2 (1966), 78–101.

[8] BENTLEY, J. L. *Writing Efficient Programs.* Prentice-Hall, 1982.

[9] BLUMOFE, R. D., FRIGO, M., JOERG, C. F., LEISERSON, C. E., AND RANDALL, K. H. An analysis of dag-consistent distributed shared-memory algorithms. In *Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)* (Padua, Italy, June 1996), pp. 297–308.

[10] BLUMOFE, R. D., FRIGO, M., JOERG, C. F., LEISERSON, C. E., AND RANDALL, K. H. Dag-Consistent distributed shared memory. In *Proceedings of*

the 10th International Parallel Processing Symposium (IPPS) (Honolulu, Hawaii, Apr. 1996).

[11] BORODIN, A., AND EL-YANIV, R. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.

[12] CHATTERJEE, S., JAIN, V. V., LEBECK, A. R., AND MUNDHRA, S. Nonlinear array layouts for hierarchical memory systems. In *Proceedings of the ACM International Conference on Supercomputing* (Rhodes, Greece, June 1999).

[13] CHATTERJEE, S., LEBECK, A. R., PATNALA, P. K., AND THOTTETHODI, M. Recursive array layouts and fast parallel matrix multiplication. In *Proceedings of the Eleventh Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)* (Saint-Malo, France, June 1999).

[14] COMPAQ. `http://ftp.digital.com/pub/Digital/info/semiconductor/` ... `literature/dsc-library.html`.

[15] COOLEY, J. W., AND TUKEY, J. W. An algorithm for the machine computation of the complex Fourier series. *Mathematics of Computation 19* (Apr. 1965), 297–301.

[16] CORMEN, T. H., LEISERSON, C. E., AND RIVEST, R. L. *Introduction to Algorithms*. MIT Press and McGraw Hill, 1990.

[17] DUHAMEL, P., AND VETTERLI, M. Fast Fourier transforms: a tutorial review and a state of the art. *Signal Processing 19* (Apr. 1990), 259–299.

[18] FRENS, J. D., AND WISE, D. S. Auto-blocking matrix-multiplication or tracking blas3 performance from source code. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Las Vegas, NV, June 1997), pp. 206–216.

[19] FRIGO, M. A fast Fourier transform compiler. In *Proceedings of the ACM SIGPLAN'99 Conference on Programming Language Design and Implementation (PLDI)* (Atlanta, Georgia, May 1999).

[20] FRIGO, M., AND JOHNSON, S. G. FFTW: An adaptive software architecture for the FFT. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing* (Seattle, Washington, May 1998).

[21] FRIGO, M., LEISERSON, C. E., PROKOP, H., AND RAMACHANDRAN, S. Cache-oblivious algorithms. Extended abstract submitted for publication, May 1999.

[22] GOLUB, G. H., AND VAN LOAN, C. F. *Matrix Computations*. Johns Hopkins University Press, 1989.

[23] HENNESSY, J. L. Back to the future: Time to return to some long standing problems in computer systems? Plenary talk at FCRC'99.

[24] HENNESSY, J. L., AND PATTERSON, D. A. *Computer Architecture: A Quantitative Approach*, 2nd ed. Morgan Kaufmann Publishers, INC., 1996.

[25] HONG, J.-W., AND KUNG, H. T. I/O complexity: the red-blue pebbling game. In *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing* (Milwaukee, 1981), pp. 326–333.

[26] IBM. `http://www.pc.ibm.com/us/netfinity/index.html`.

[27] KENNEDY, K. Future investment in information technology research: Report of the president's information technology advisory committee. Plenary talk at FCRC'99.

[28] KNUTH, D. E. *Sorting and Searching*, second ed., vol. 3 of *The Art of Computer Programming*. Addison-Wesley, 1997.

[29] MOTWANI, R., AND RAGHAVAN, P. *Randomized Algorithms*. Cambridge University Press, 1995.

[30] NODINE, M. H., AND VITTER, J. S. Deterministic distribution sort in shared and distributed memory multiprocessors. In *Proceedings of the Fifth Symposium on Parallel Algorithms and Architectures* (Velen, Germany, 1993), pp. 120–129.

[31] PRESS, W. H., FLANNERY, B. P., TEUKOLSKY, S. A., AND VETTERLING, W. T. *Numerical Recipies in C*. Cambridge University Press, 1988.

[32] RUGINA, R., AND RINARD, M. Automatic parallelization of divide and conquer algorithms. In *Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)* (Atlanta, Georgia, May 1999), pp. 72–83.

[33] SAVAGE, J. E. Extending the Hong-Kung model to memory hierarchies. In *Computing and Combinatorics*, D.-Z. Du and M. Li, Eds., vol. 959 of *Lecture Notes in Computer Science*. Springer Verlag, 1995, pp. 270–281.

[34] SEDGEWICK, R. *Algorithms in C.* Addison-Welsey Publishing Company, 1990.

[35] SGI. `http://www.sgi.com/products/hw_workstations.html`.

[36] SINGLETON, R. C. An algorithm for computing the mixed radix fast Fourier transform. *IEEE Transactions on Audio and Electroacoustics AU-17*, 2 (June 1969), 93–103.

[37] SLEATOR, D. D., AND TARJAN, R. E. Amortized efficiency of list update and paging rules. *Communications of the ACM 28*, 2 (Feb. 1985), 202–208.

[38] STRASSEN, V. Gaussian elimination is not optimal. *Numerische Mathematik 13* (1969), 354–356.

[39] SUPERCOMPUTING TECHNOLOGIES GROUP, MIT LABORATORY FOR COMPUTER SCIENCE. *Cilk-5.2 (Beta 1) Reference Manual.* Cambridge, MA, 1998. Available on the Internet from `http://supertech.lcs.mit.edu/cilk`.

[40] TOLEDO, S. Locality of reference in *LU* decomposition with partial pivoting. *SIAM Journal on Matrix Analysis and Applications 18*, 4 (Oct. 1997), 1065–1081.

[41] VITTER, J. S. External memory algorithms and data structures. In *External Memory Algorithms and Visualization*, J. Abello and J. S. Vitter, Eds., DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society Press, Providence, RI, 1999.

[42] VITTER, J. S., AND NODINE, M. H. Large-scale sorting in uniform memory hierarchies. *Journal of Parallel and Distributed Computing 17*, 1–2 (January and February 1993), 107–114.

[43] VITTER, J. S., AND SHRIVER, E. A. M. Algorithms for parallel memory I: Two-level memories. *Algorithmica 12*, 2/3 (August and September 1994), 110–147.

[44] VITTER, J. S., AND SHRIVER, E. A. M. Algorithms for parallel memory II: Hierarchical multilevel memories. *Algorithmica 12*, 2/3 (August and September 1994), 148–169.

[45] WINOGRAD, S. On the algebraic complexity of functions. *Actes du Congrès International des Mathématiciens 3* (1970), 283–288.