Cache-Oblivious Traversals of an Array's Pairs

Tobias Johnson

May 7, 2007

Abstract

Cache-obliviousness is a concept first introduced by Frigo et al. in [1]. We follow their model and develop a cache-oblivious algorithm for traversing all pairs of elements in a one-dimensional array and prove that it is optimally cache-efficient. Though the the traversal is recursive in nature, we demonstrate how to implement the algorithm nonrecursively, and we give experimental results.

1 Introduction

A common programming task is to iterate over all pairs of an array. For instance, an array might hold records giving information about a movie, and one might want to compute the similarity of each movie to every other movie.* Standard code for iterating through all pairs of an array of length N and executing some code on them is

for $i = 0$ to $N - 2$ **do**
  for $j = i + 1$ to $N - 1$ **do**
    f(array[i], array[j])
  end for
end for

We give a visual representation of this traversal in Figure 1.

This traversal makes poor use of the computer's cache, however. Assuming that the cache is not large enough to store the entire array, the inner loop must reload the entire array every time through. We wish instead to traverse the pairs so that accesses near to each other in time are as near as possible to each other in the array. Instead of allowing the program explicit knowledge and control of the cache, we design it to be *cache-oblivious*. The notion of cache-obliviousness was introduced by Frigo, Leiserson, Prokop, and Ramachandran in [1]. They defined a formal cache-oblivious memory model, which we use.

To achieve a cache-oblivious traversal, we give a traversal order with self-similar properties; see Figure 2 for a picture of it. We will also consider traversals of all ordered pairs of an array, in which we consider pairs like $(1, 2)$ and $(2, 1)$

*In fact, the motivation for this paper was an effort to optimize code for the Netflix Prize, a machine-learning contest (see `http://netflixprize.com/`).

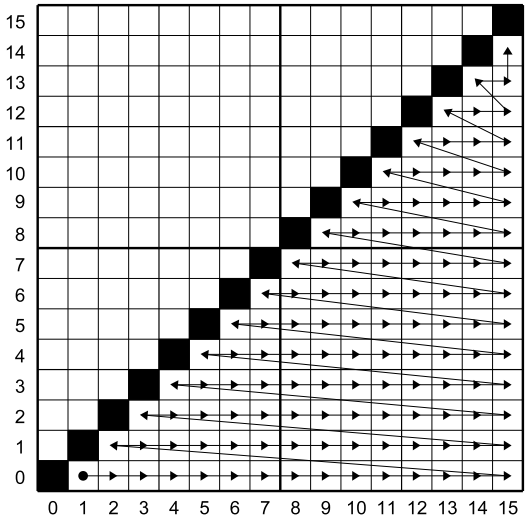

Figure 1: A standard order for traversing all unordered pairs of an array of length 16.

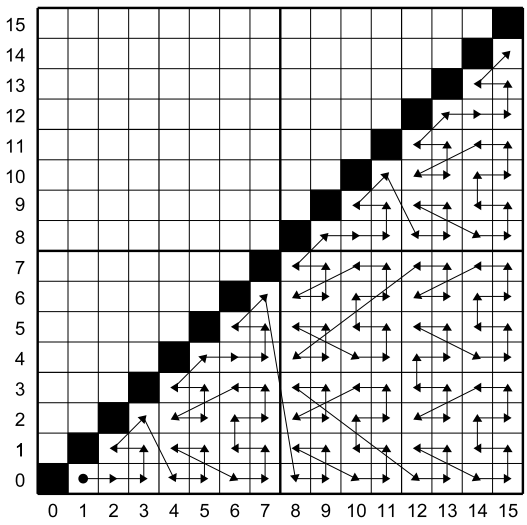

Figure 2: A cache-efficient traversal of the unordered pairs of an array of length 16.

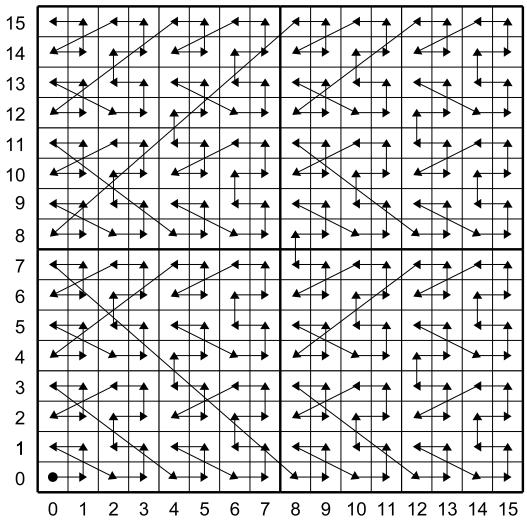

Figure 3: An cache-efficient traversal of the ordered pairs of an array of length 16

to be distinct (see Figure 3). These traversals have no practical use—we could just as well call both $f(i,j)$ and $f(j,i)$ for each unordered pair (i,j)—but we make use of them to prove facts about unordered-pair traversals.

These cache-efficient traversals all have the following property, illustrated in Figure 4:

The Locality Property. For all integers $k, u, v \geq 0$, the pairs (i,j) such that $2^k u \leq i < 2^k(u+1)$, $2^k v \leq j < 2^k(v+1)$ are accessed consecutively.

Put simply, if we tile any of Figures 1–3 with squares whose lengths are powers of two, the traversal has the property that it visits these squares one at a time, covering all pairs in one square before moving on to the next. This locality explains why the traversals are cache-efficient, and it is the only fact we need to prove that they are.

In Section 2, we explain the cache-oblivious model. In Section 3, we give a nonrecursive implementation of the cache-oblivious traversal and show that it satisfies the locality property mentioned before. In Section 4, we use the locality property to show that this algorithm is asymptotically optimal in its cache-behavior. We give experimental results in Section 5 comparing the algorithm's performance against the standard traversal technique.

2 Memory Models

Following [2], we consider two memory models. The first, the standard model, is the *external-memory model*, in which the computer is assumed to have two

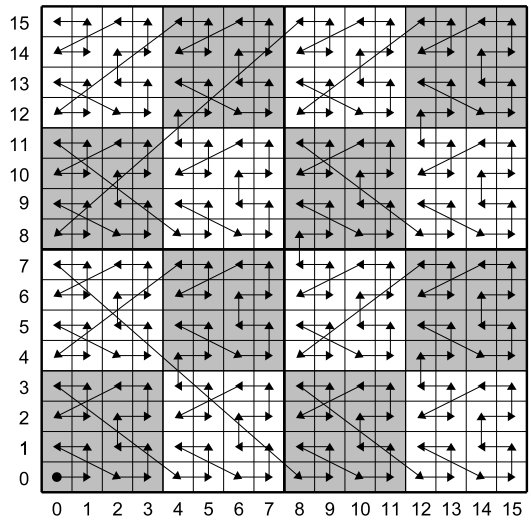

Figure 4: An ordered-pair traversal, with 4×4 squares highlighted to demonstrate the locality property.

levels of memory, the *cache* and the *disk*. (This saves the term *memory* to refer to both of them.) The cache is organized in *blocks* of size B. The program issues instructions to load blocks of data from the disk into cache, and it chooses which blocks to remove from cache when necessary. The total size of the cache is M, meaning that it contains M/B blocks.

The other model we consider is the *cache-oblivious model*. We again assume a cache of size M, organized into B blocks, but our programs have no knowledge of these values. Rather than the program explicitly loading data into the cache, we assume that the computer does this for us. We further assume that the computer is omniscient and always removes the block from cache that will be used the farthest in the future. We assume that the cache is fully associative (i.e., a block on disk can be loaded into any block in cache). In our analysis, we will seek to bound our algorithm's maximum number of *memory transfers* (often referred to as *cache misses*), instances when data must be loaded from disk to cache.

This model contains several unrealistic assumptions. A real memory hierarchy is likely to have more than two levels, an optimal cache-replacement strategy is clearly impossible, and most caches are not fully-associative. These factors turn out to be irrelevant to the asymptotic cache behavior. Frigo et al. prove in [1] that, given certain standard conditions, a cache using LRU replacement causes asymptotically the same number of memory transfers. They show that a cache-oblivious algorithm that performs optimally in a two-level model also does so in a model with more levels. Finally, they demonstrate that a fully-associative memory model can be simulated by nonfully-associated

memory with no asymptotic loss of performance.

We will assume that at each step of the traversal, the only memory accesses are to the current array elements. In particular, this means that we suppose that the loop indices are not stored in memory.

3 Traversal Algorithms

From the self-similar nature of our traversal order, exhibited in Figures 2 and 3, it is clear that we could implement our algorithm recursively. To make our algorithm fast enough to use in practice, however, we need some simple iterative implementation. We will give such a technique and prove that it produces a traversal satisfying the locality property.

As these traversals are most easily understood visually, we will consider each pair (i, j) to be the point at row i and column j. If one step of the iteration is (i, j) and the next is $(i, j + 1)$, we say that we moved to the right. Likewise, we refer to $(i, j - 1)$, $(i + 1, j)$, and $(i - 1, j)$ as being left, up, and down from (i, j), respectively.

We begin by giving an algorithm for ordered-pair traversal of an array whose length is a power of two (Figure 3).

Theorem 1. *The following set of rules, applied starting at* $(0, 0)$ *and ending when* $(2^n - 1, 0)$ *is reached, traverses (i.e., covers exactly once) all pairs* (i, j) *such that* $0 \leq i, j < 2^n$ *and satisfies the locality property.*

Algorithm 1 Given current (i, j), advance one step of an ordered-pair traversal.

Let 2^k be the greatest power of two dividing both $i + 1$ and j. We have three cases for where to move:

(1) If $\frac{i+1}{2^k}$ and $\frac{j}{2^k}$ are both odd, go up by one.

(2) If $\frac{i+1}{2^k}$ is even, go 2^k to the left and $2^k - 1$ down.

(3) If $\frac{j}{2^k}$ is even, go 2^k to the right and $2^k - 1$ down.

To prove this, we prove a stronger claim:

Lemma 2. *If we start the algorithm at any* (i_0, j_0) *such that* i_0 *and* j_0 *are both divisible by* 2^k*, it will traverse all pairs* (i, j) *such that* $i_0 \leq i < i_0 + 2^k$*,* $j_0 \leq j < j_0 + 2^k$ *before reaching* $(i_0 + 2^k - 1, j_0)$*, touching no pairs outside of this region.*

Proof. We proceed by induction. The lemma is trivially true for $k = 0$, and we use this as our base case. Now, we assume that the claim holds for k and show it for $k + 1$.

We will apply the inductive hypothesis four times, once for each of the four $2^k \times 2^k$ squares making up the $2^{k+1} \times 2^{k+1}$ square we are traversing:

Step 1: (i_0, j_0) to $(i_0, j_0 + 2^k)$

Since i_0 and j_0 are each divisible by 2^{k+1}, they are divisible by 2^k as well. We can apply the inductive hypothesis, which implies that we will traverse the pairs (i, j) satisfying $i_0 \le i < i_0 + 2^k$, $j_0 \le j < j_0 + 2^k$, ending up at $(i_0 + 2^k - 1, j_0)$. The greatest power of two dividing $i_0 + 2^k$ and j_0 is 2^k. As j_0 is divisible by 2^{k+1}, we have that $\frac{j_0}{2^k}$ is even, and Rule 3 applies, taking us to $(i_0, j_0 + 2^k)$.

Step 2: $(i_0, j_0 + 2^k)$ to $(i_0 + 2^k, j_0 + 2^k)$

We apply the inductive hypothesis again to show that from here, we traverse all pairs (i, j) satisfying $i_0 \le i < i_0 + 2^k$, $j_0 + 2^k \le j < j_0 + 2^{k+1}$, ending at $(i_0 + 2^k - 1, j_0 + 2^k)$. The greatest power of two dividing $i_0 + 2^k$ and $j_0 + 2^k$ is 2^k, and both $\frac{i_0+2^k}{2^k}$ and $\frac{j_0+2^k}{2^k}$ are odd. Rule 1 applies, taking us to $(i_0 + 2^k, j_0 + 2^k)$.

Step 3: $(i_0 + 2^k, j_0 + 2^k)$ to $(i_0 + 2^k, j_0)$

The inductive hypothesis applies again, showing that from here we traverse all pairs (i, j) satisfying $i_0 + 2^k \le i < i_0 + 2^{k+1}$, $j_0 + 2^k \le j < j_0 + 2^{k+1}$, ending at $(i_0 + 2^{k+1} - 1, j_0 + 2^k)$. The greatest power of two dividing both $i_0 + 2^{k+1}$ and $j_0 + 2^k$ is 2^k. We have $\frac{i_0+2^{k+1}}{2^k}$ is even, so Rule 2 applies, taking us to $(i_0 + 2^k, j_0)$.

Step 4: $(i_0 + 2^k, j_0)$ to $(i_0 + 2^{k+1} - 1, j_0)$

We applying the inductive hypothesis one last time, bringing us to $(i_0 + 2^{k+1} - 1, j_0)$.

Note that all the regions covered were disjoint, and together they make up all pairs (i, j) such that $i_0 \le i < i + 2^{k+1}$, $j_0 \le j < j + 2^{k+1}$. Also note that we ended at $(i_0 + 2^{k+1} - 1, j_0)$. This completes the induction step, which shows that the algorithm does traverse the pairs we claimed it did. □

The locality property follows directly from this lemma.

We need only one additional rule to handle the unordered-pair traversal (Figure 2) of an array whose length is a power of two.

Theorem 3. *The following set of rules, applied starting at* $(0, 0)$ *and ending when* $(2^n - 2, 2^n - 1)$ *is reached, traverses all pairs* (i, j) *such that* $0 \le i < j < 2^n$*, and the traversal satisfies the locality property.*

Algorithm 2 Given (i, j), advance one step of an unordered-pair traversal of an array whose length is a power of two.

- If (i, j) is "on the diagonal" (that is, $i + 1 = j$), then go one to the right and $2^k - 2$ down, where 2^k is the greatest power of two dividing $i + 2$.
- Otherwise, determine the next move using Algorithm 1.

The structure of this proof is nearly identical to the previous one.

Lemma 4. *If we start the algorithm at (i_0, i_0+1) such that i_0 is divisible by 2^k with $k \geq 1$, it will traverse each pair (i, j) such that $i_0 \leq i < j < i_0 + 2^k$ before reaching $(i_0 + 2^k - 2, i_0 + 2^k - 1)$, touching no pairs outside of this region.*

Proof. Again, we use induction. For the base case, note that the claim is trivially true for $k = 1$. Now, we assume the claim for k and show it for $k + 1$.

We will apply the inductive hypothesis twice and Lemma 2 once, reflecting that our triangle pattern is made up of two smaller triangles and one square.

Step 1: $(i_0, i_0 + 1)$ to $(i_0, i_0 + 2^k)$

Since i_0 is divisible by 2^{k+1}, it is also divisible by 2^k. Hence, the inductive hypothesis applies, showing that the algorithm covers each pair (i, j) such that $i_0 \leq i < j < i_0 + 2^k$ before reaching $(i_0 + 2^k - 2, i_0 + 2^k - 1)$. Since $(i_0 + 2^k - 2, i_0 + 2^k - 1)$ is on the diagonal, Algorithm 2 applies. The greatest power of two dividing $i_0 + 2^k$ is 2^k (note that this is true even if i_0 is divisible by a higher power of two than 2^k), so we go to $(i_0, i_0 + 2^k)$.

Step 2: $(i_0, i_0 + 2^k)$ to $(i_0 + 2^k, i_0 + 2^k + 1)$

The pair $(i_0, i_0 + 2^k)$ is not on the diagonal, since $k \geq 1$. So, Algorithm 1 applies. As i_0 and $i_0 + 2^k$ are both divisible by 2^k, by Lemma 2 repeated application of Algorithm 1 covers the region (i, j) such that $i_0 \leq i < i_0+2^k$, $i_0 + 2^k \leq j < j_0 + 2^{k+1}$, ending at $(i_0 + 2^k - 1, i_0 + 2^k)$. As this is the only pair in the region that lies on the diagonal, repeated application of Algorithm 2 has the same effect. As (i_0+2^k-1, i_0+2^k) is on the diagonal, Algorithm 2 applies. The greatest power of two dividing $i_0 + 2^k + 1$ is $2^0 = 1$. So, the algorithm takes us one to the right and -1 down, leaving us at $(i_0 + 2^k, i_0 + 2^k + 1)$.

Step 3: $(i_0 + 2^k, i_0 + 2^k + 1)$ to $(i_0 + 2^{k+1} - 2, i_0 + 2^{k+1} - 1)$

The inductive hypothesis applies again, since $i_0 + 2^k$ is divisible by 2^k. This means we cover all pairs (i, j) such that $i_0 + 2^k \leq i < j < i_0 + 2^{k+1}$ before reaching $(i_0 + 2^{k+1} - 2, i_0 + 2^{k+1} - 1)$.

We have gone from (i_0, i_0+1) to $(i_0+2^{k+1}-2, i_0+2^{k+1}-1)$, covering all squares (i, j) such that $i_0 \leq i < j < i_0 + 2^{k+1}$. This completes the induction. □

Again, the locality property follows directly from this.

We add one more rule to allow for arrays whose length is not a power of two.

Theorem 5. *The following set of rules does an unordered-pair traversal of an array of any size. The traversal is identical to the one we would get if we applied Algorithm 2 and removed all pairs that were out of bounds.*

Algorithm 3 Given (i, j), advance one step of an ordered pair traversal of an array of size N.

- If Algorithm 2 applied to (i, j) would move to a pair within bounds (that is, with both coordinates less than N), move according to it.
- Otherwise, move up by one.

Proof. Suppose we are at (i, j) and the next move according to Algorithm 2 would take us out of bounds. We will imagine that we continue to apply Algorithm 2 until we return within bounds, and then show that this would take us to $(i + 1, j)$.

We may assume that (i, j) is not on the diagonal. If it were, the next move could go at most one up and one to the right, and moving out of bounds would imply that $(i, j) = (N - 2, N - 1)$; the traversal would then be finished. So, we need only consider Algorithm 1. The only move that could possibly take us out of bounds is a rightward one, and hence Rule 3 applies. Let 2^k be the greatest power of two dividing both $i + 1$ and j. We know that $\frac{i+1}{2^k}$ is odd and $\frac{j}{2^k}$ is even, and that our next pair will be $(i - 2^k + 1, j + 2^k)$.

Since $i + 1$ is divisible by 2^k, so is $i - 2^k + 1$, and since j is divisible by 2^k, so is $j + 2^k$. Hence, Lemma 2 applies, which shows that repeated application of Algorithm 1 takes us to $(i, j + 2^k)$ without ever going left of column $j + 2^k$. Hence, as we go from $(i - 2^k + 1, j + 2^k)$ to $(i, j + 2^k)$, we never return within bounds.

Since $\frac{j}{2^k}$ is even, $\frac{j+2^k}{2^k}$ is odd. As $\frac{i+1}{2^k}$ is odd as well, Rule 1 applies and we go to $(i+1, j+2^k)$, still out of bounds. Lemma 2 applies, and repeated application of Algorithm 1 takes us to $(i+2^k, j+2^k)$, again without ever returning in bounds. As $\frac{i+2^k+1}{2^k}$ is even and $\frac{j+2^k}{2^k}$ is odd, Rule 2 applies, taking us to $(i+1, j)$, finally within bounds again.

This is exactly what we set out to prove: By moving up whenever Algorithm 2 would take us out of bounds, we achieve a traversal identical to the one we would get by following Algorithm 2 and discarding out-of-bounds entries. □

As this traversal is a subset of the one generated by Algorithm 2, it has the locality property.

4 Bounds for Cache Efficiency

We seek to show that number of memory transfers in our cache-oblivious traversal is within a constant factor of optimal. We also consider the cache-efficiency of the standard pair-traversal algorithm. We start by giving a lower bound on the number of memory transfers for any ordered-pair traversal in the standard memory model (and hence in the cache-oblivious model too). To avoid degenerate cases, we assume that $N > (2 + \epsilon)M$.

Theorem 6. *Any ordered-pair traversal of an N-element array requires $\Omega(N^2/MB)$ memory transfers.*

Proof. Consider any $4M^2$ consecutive steps of the pair traversal. The algorithm must access at least $2M$ distinct elements of the array in these steps, since for any amount fewer than $2M$ elements, there are fewer than $4M^2$ pairs consisting of just these elements. So, in the course of these $4M^2$ steps, at least M elements must be loaded into cache, requiring $\lceil M/B \rceil$ memory transfers. As the entire pair traversal has N^2 steps, the number of memory transfers in the whole traversal is at least

$$\begin{aligned}\left\lfloor \frac{N^2}{4M^2} \right\rfloor \left\lceil \frac{M}{B} \right\rceil &\geq \left(\frac{N^2}{4M^2} - 1 \right) \frac{M}{B} \\ &= \frac{N^2}{4MB} - \frac{M}{B} = \frac{N^2 - 4M^2}{4MB}.\end{aligned}$$

All we need to do is show that $N^2 - 4M^2$ is greater than some constant factor of N^2. Since $M < N/(2+\epsilon)$ by our assumption,

$$\begin{aligned}N^2 - 4M^2 &\geq N^2 - \frac{4N^2}{(2+\epsilon)^2} \\ &= N^2 - \frac{N^2}{(1+\epsilon/2)^2} \\ &= \left(1 - \frac{1}{(1+\epsilon/2)^2}\right) N^2\end{aligned}$$

which shows that the number of memory transfers is $\Omega(N^2/MB)$. □

Corollary 7. *Any unordered-pair traversal of an N-element array has the same $\Omega(N^2/MB)$ lower bound on cache efficiency.*

Proof. The argument goes through as above, but with $N(N-1)/2$ instead of N^2, which makes no difference asymptotically. □

Now, we give an upper bound on the maximum number of memory transfers in the cache-oblivious traversal. To avoid degenerate cases, we assume that the cache holds at least 10 blocks.

Theorem 8. *Given that N and M are both powers of two, the cache-oblivious algorithm for an ordered-pair traversal requires $O(N^2/MB)$ memory transfers.*

Proof. This follows quite directly from the locality property: We consider the square regions given by the locality property, and pick regions small enough that all array elements in a region can fit in cache. This gives us a bound on memory transfers for this section of the array, and we just add up these bounds.

We tile the iterative space with $\frac{M}{4} \times \frac{M}{4}$ squares; that is, we partition the pairs (i, j) with $0 \leq i, j < M/4$ into one set, those with $0 \leq i < M/4,\ M/4 \leq j < M/2$ into another, and so forth (see Figure 4). By the locality property, our

traversal covers each of these squares in turn. Each of these squares contains at most $M/2$ distinct elements of the array, in two contiguous segments. Regardless of block boundaries, these all fit in cache together: Each segment requires at most $\lceil M/4B \rceil + 1$ blocks (see [2], Theorem 1 for a detailed proof of this easy fact), so the two segments need at most $2\lceil M/4B \rceil + 2 < M/2B + 4$ blocks. Since we assumed that $M/B \geq 10$, we have $-M/2B \leq -5$, which means

$$\begin{aligned} \frac{M}{2B} + 4 &= \frac{M}{B} - \frac{M}{2B} + 4 \\ &\leq \frac{M}{B} - 1 < \left\lfloor \frac{M}{B} \right\rfloor, \end{aligned}$$

which is the number of blocks in cache at a time.

So, we need fewer than M/B memory transfers per $\frac{M}{4} \times \frac{M}{4}$ square. As there are $(4N/M)^2$ such squares, the number of memory transfers for the traversal is less than

$$\left(\frac{4N}{M}\right)^2 \frac{M}{B} = \frac{16N^2}{MB},$$

which is $O(N^2/MB)$. □

Corollary 9. *The cache-oblivious algorithm for an unordered-pair traversal makes $O(N^2/MB)$ memory transfers.*

Proof. Let N' be the smallest power of two greater than or equal to N, and let M' be the greatest power of two less than or equal to M. We have

$$N' < 2N \tag{1}$$

$$M' > M/2. \tag{2}$$

We can tile the iterative space as in the ordered-pair traversal. The unordered-pair traversal has the locality property as well, so the above argument shows that the traversal takes $O(N'^2/M'B)$. By (1) and (2), this is $O(N^2/MB)$. □

Since we've shown that all traversal algorithms have a $\Omega(N^2/MB)$ lower bound on cache-efficiency, this means that our cache-oblivious traversal algorithms have $\Theta(N^2/MB)$ cache-efficiency and are within a constant factor of optimal.

Finally, we show that the standard traversal algorithm is asymptotically less cache-efficient.

Theorem 10. *The standard algorithm for ordered- or unordered-pair traversal has cache-efficiency $\Omega(N^2/B)$.*

Proof. Recall the standard ordered-pair traversal:

```
for i = 0 to N − 1 do
  for j = 0 to N − 1 do
    f(array[i], array[j])
  end for
end for
```

At each step of the outer loop, at most M of the N elements of the array

can already be in cache. So, each iteration of the outer loop requires at least $\lceil \frac{N-M}{B} \rceil$ memory transfers. There are N steps of the outer loop, meaning that the number of memory transfers is at least

$$\left\lceil \frac{N-M}{B} \right\rceil N \geq \frac{N^2 - NM}{B} \geq \frac{N^2}{2B}$$

with the last inequality by our prior assumption that $N \geq 2M$.

The unordered-pair variation is nearly identical (but messier), and we omit it. □

5 Experimental Results

We implemented our algorithm in C++, making use of templates and function objects to allow it to execute arbitrary code on each pair. We unrolled the loop so that we need only calculate the next move once per each 16 moves. (Further loop unrolling provided no gains.) Because the code relies on function inlining, the implementation is drastically slower when compiled with the optimizer off.

To test the cache-oblivious algorithm, we wrote a program that filled an array with random integers. We considered every k of these to form a record. For each pair of records, we summed the integers making up each respective record and multiplied them together. We asked our program to find the maximum of these values, using both a standard traversal and our cache-efficient one.

Our results are shown in Figures 5 and 6. For small record sizes, the additional overhead of our method made it slower than the standard traversal. When the record size was greater than 32 bytes, our method was faster, sometimes by a factor of as much as 40%. (By percent improvement, we mean $1 - t/t'$, where t and t' are the running times of the cache-oblivious and standard algorithms, respectively.) The record size was the only variable that affected our results. In particular, changing the length of the array made no significant difference in the relative speed of our algorithm to the standard one. We used different array sizes in Figure 5 and Figure 6 only because tests with the larger array size and a large record size took a long time to run. The benchmarking was done on a computer with an 8 kilobyte, 4-way-associative L1 cache and a 512 kilobyte, 8-way-associative L2 cache, both with 64 byte lines.

We also tested the algorithm on some real-life data, asking our program to rate the similarity of all pairs of some subset of movie-watchers given in the data for the Netflix Prize. Each movie-watcher takes up a different amount of memory depending on the number of movies they have rated; the average record size was approximately 800 bytes. We improved by about 25% over the traditional traversal. Notably, this program made frequent memory accesses besides the two array elements from the pair currently traversed, not following the assumption from Section 2.

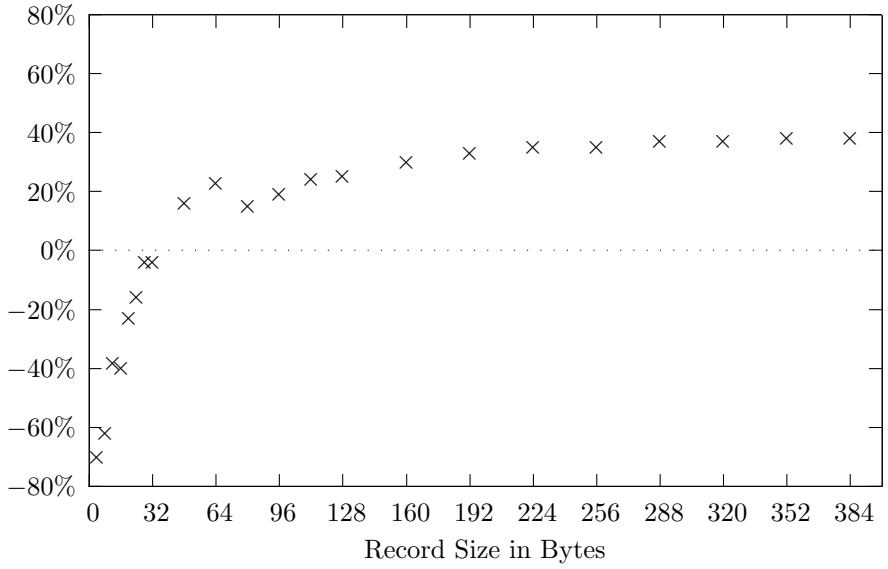

Figure 5: Percent improvement over a standard pair traversal, record sizes 4–384, array size 32768.

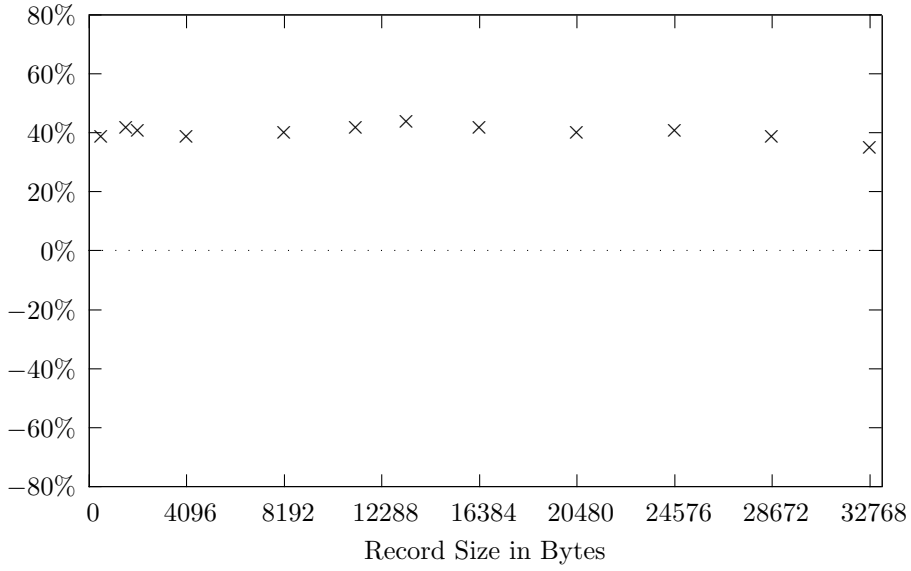

Figure 6: Percent improvement over a standard pair traversal, record sizes 512–8192, array size 8192.

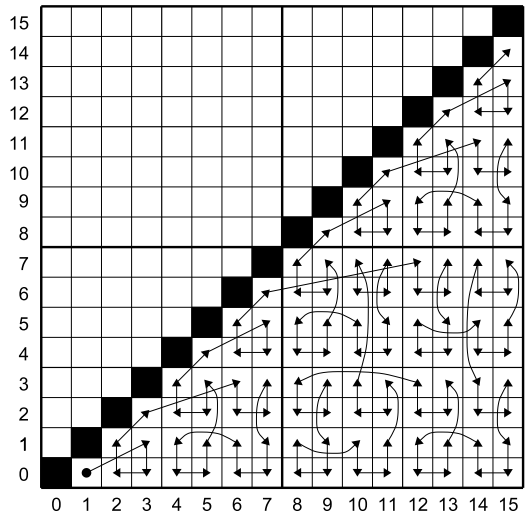

Figure 7: Traversal order with few diagonal moves.

6 Conclusion and Future Work

We presented a practical algorithm for efficiently traversing the pairs of an array. In practice, the algorithm is faster than the standard algorithm so long as the size of each array entry is large enough, and we proved that the algorithm is within a constant factor of optimal in terms of cache behavior.

Of course, in real applications, this constant term is relevant, and there may be other traversal algorithms that perform better. We examined several other traversals. Figure 7 demonstrates a traversal order that reduces the number of diagonal moves (that is, moves in which both the row and column change). Despite its apparent advantages, an implementation of this order exhibited almost identical running times as the cache-efficient algorithm given in this paper. Testing with Cachegrind, a cache profiler, showed that its cache behavior was nearly identical to our algorithm's. We were surprised by this and would consider it worthwhile to simulate a cache in more detail to determine why this is so; of course, this would leave the realm of cache-obliviousness. Figure 8 gives a traversal that reduces the number of diagonal moves and avoids large jumps, but we were not able to efficiently implement it.

7 Acknowledgments

I am grateful to my advisor, Professor Dan Spielman, for suggesting this topic and helping me with it, and to Yale's Department of Computer Science.

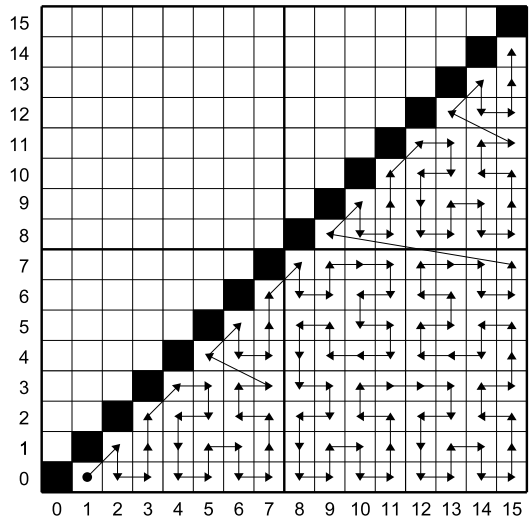

Figure 8: Traversal order with few diagonal moves and few jumps.

References

[1] M. Frigo, C. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, pages 285–297, New York, October 1999.

[2] E. Demaine. Cache-oblivious algorithms and data structures. In *Lecture Notes from the EEF Summer School on Massive Data Sets, Lecture Notes in Computer Science*, BRICS, University of Aarhus, Denmark, June 27–July 1, 2002.