Cache Oblivious Algorithms

Piyush Kumar*

Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11790, USA
piyush@acm.org
http://www.compgeom.com/co-chap/

Abstract. The cache oblivious model is a simple and elegant model to design algorithms that perform well in hierarchical memory models ubiquitous on current systems. This model was first formulated in [22] and has since been a topic of intense research. Analyzing and designing algorithms and data structures in this model involves not only an asymptotic analysis of the number of steps executed in terms of the input size, but also the movement of data optimally among the different levels of the memory hierarchy. This chapter is aimed as an introduction to the "ideal-cache" model of [22] and techniques used to design cache oblivious algorithms. The chapter also presents some experimental insights and results.

* Part of this work was done while the author was visiting MPI-Saarbrücken. The author is partially supported by NSF (CCR-9732220, CCR-0098172) and by a grant from Sandia National Labs.

Table of Contents

Cache Oblivious Algorithms 1
Piyush Kumar (University of New York at Stony Brook, NY)
1 Introduction 2
2 The Model 4
3 Algorithm design tools 6
4 Matrix transposition 10
5 Matrix multiplication 15
6 Searching using Van Emde Boas layout 18
7 Sorting 20
8 Is the model an oversimplification? 25
9 Other Results 27
10 Open problems 28
11 Acknowledgements 28

1 Introduction

A dream machine would be fast and would never run out of memory. Since an infinite sized memory was never built, one has to settle for various trade-offs in speed, size and cost. In both the past and the present, hardware suppliers seem to have agreed on the fact that these parameters are well optimized by building what is called a memory hierarchy (See Figure 1, Chapter **??**). Memory hierarchies optimize the three factors mentioned above by being cheap to build, trying to be as fast as the fastest memory present in the hierarchy and being almost as cheap as the slowest level of memory. The hierarchy inherently makes use of the assumption that the access pattern of the memory has *locality* in it and can be exploited to speed up the accesses.

The locality in memory access is often categorized into two different types, code reusing recently accessed locations (*temporal*) and code referencing data items that are close to recently accessed data items (*spatial*) [24]. Caches use both temporal and spatial locality to improve speed. Surprisingly many things can be categorized as caches, for example, registers, L1, L2, TLB, Memory, Disk, Tape etc (Chapter **??**). The whole memory hierarchy can be viewed as *levels* of caches, each transferring data to its adjacent levels in atomic units called *blocks*. When data that is needed by a process is in the cache, a *cache hit* occurs. A *cache miss* occurs when data can not be supplied. Cache misses can be very costly in terms of speed. Cache misses can be reduced by designing algorithms that use locality of memory access.

The *Random Access Model* (RAM) in which we do analysis of algorithms today does not take into account differences in speeds of random access of memory depending upon the locality of access [18]. Although there exist models which can deal with multi-level memory hierarchies, they are quite complicated to use [1, 3,

6, 5, 2, 25, 33, 34]. It seems there is a trade-off between the accuracy of the model and the ease of use. Most algorithmic work has been done in the RAM model which models a 'flat' memory with uniform access times. The external memory model (with which the reader is assumed to be familiar, See Chapter **??**) is a two level memory model, in the context of memory and disk. The 'ideal' cache oblivious model is a step towards simplifying the analysis of algorithms in light of the fact that local accesses in memory are cheaper than non-local ones in the whole memory hierarchy (and not just two-levels of memory). It helps take into account the whole memory hierarchy and the speed differences hidden therein.

In the external memory model, algorithm design is focussed on a particular level of memory (usually the disk) which is the bottleneck for the running time of the algorithm in practice. Recall (From Chapter **??**), that in the external memory model, processing works almost as in the RAM model, except that there are only M words of internal memory that can be accessed quickly. The remaining memory can only be accessed using I/Os that move B contiguous words between external and internal memory. The I/O complexity of an algorithm amounts to counting the number of I/Os needed.

The cache oblivious model was proposed in [22] and since then has been used in more than 30 papers already. It is becoming popular among researchers in external memory algorithms, parallel algorithms, data structures and other related fields. This model was born out of the necessity to capture the hierarchical nature of memory organization. (For instance the Intel Itanium has 7 levels in its memory hierarchy, See Chapter **??**). Although there have been other attempts to capture this hierarchical information the cache oblivious model seems to be one of the most simple and elegant ones. The cache oblivious model is a two level model (like the [4] model that has been used in the other chapters so far) but with the assumption that the parameters M,B are unknown to the algorithm (See Figure 1). It can work efficiently on most machines with multi-level cache hierarchies. Note that in the present volume, all external memory algorithms that have been dealt with yet, need the programmer/user to specify M,B.

This chapter is intended as an introduction to the design and analysis of cache oblivious algorithms, both in theory and practice.

Chapter Outline: We introduce the cache oblivious model in section 2. In section 3 we elaborate some commonly used design tools that are used to design cache oblivious algorithms. In section 4 we choose matrix transposition as an example to learn the practical issues in cache oblivious algorithm design. We study the cache oblivious analysis of Strassen's algorithm in section 5. Section 6 discusses a method to speed up searching in balanced binary search trees both in theory and practice. In section 7, a theoretically optimal, randomized cache oblivious sorting algorithm along with the running times of an implementation is presented. In section 8 we enumerate some practicalities not caught by the model. Section 9 presents some of the best known bounds of other cache oblivious algorithms. Finally we conclude the chapter by presenting some related open problems in section 10.

2 The Model

The ideal cache oblivious memory model is a two level memory model. We will assume that the faster level has M size and the slower level always transfers B words of data together to the faster level. These two levels could represent the memory and the disk, memory and the cache, or any two consecutive levels of the memory hierarchy (See Figure 1). In this chapter, M and B can be assumed to be the sizes of any two consecutive levels of the memory hierarchy subject to some assumptions about them (For instance the inclusion property which we will see soon). We will assume that the processor can access the faster level of memory which has size M. If the processor references something from the second level of memory, an I/O fault occurs and B words are fetched into the faster level of the memory. We will refer to a *block* as the minimum unit that can be present or absent from a level in the two level memory hierarchy. We will use B to denote the size of a *block* as in the external memory model. If the faster level of the memory is full (i.e. M is full), a block gets evicted to make space.

The ideal cache oblivious memory model enables us to reason about a two level memory model like the external memory model but prove results about a multilevel memory model. Compared with the external memory model it seems surprising that without any memory specific parametrization, or in other words, without specifying the parameters M, B, an algorithm can be efficient for the whole memory hierarchy, nevertheless its possible. The model is built upon some basic assumptions which we enumerate next.

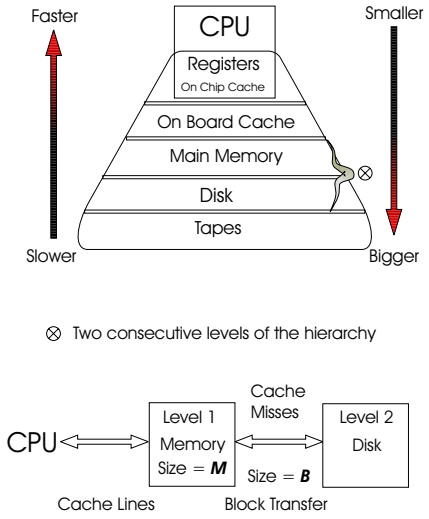

Fig. 1. The ideal cache oblivious model

Assumptions: The following four assumptions are key to the model.

Optimal replacement The *replacement policy* refers to the policy chosen to replace a block when a cache miss occurs and the cache is full. In most hardware, this is implemented as FIFO, LRU or Random. The model assumes that the cache line chosen for replacement is the one that is accessed furthest in the future. The strategy is called *optimal off-line replacement* strategy.

2 levels of memory There are certain assumptions in the model regarding the two levels of memory chosen. They should follow the *inclusion property* which says that data cannot be present at level i unless it is present at level $i+1$. In most systems, the inclusion property holds. Another assumption is that the size of level i of the memory hierarchy is strictly smaller than level $i+1$.

Full associativity When a block of data is fetched from the slower level of the memory, it can reside in any part of the faster level.

Automatic replacement When a block is to be brought in the faster level of the memory, it is automatically done by the OS/hardware and the algorithm designer does not have to care about it while designing the algorithm. Note that we could access single blocks for reading and writing in the external memory model, which is not allowed in the cache oblivious model.

We will now examine each of the assumptions individually. First we consider the optimal replacement policy. The most commonly used replacement policy is LRU (*least recently used*). In [22] the following lemma, whose proof is omitted here, was proved using a result of [35].

Lemma 1 ([22]). *An algorithm that causes $Q^*(n, M, B)$ cache misses on a problem of size n using a (M, B)-ideal cache incurs $Q(n, M, B) \leq 2Q^*(n, \frac{M}{2}, B)$ cache misses on a (M, B) cache that uses LRU, FIFO replacement. This is only true for algorithms which follow a regularity condition.*

An algorithm whose cache complexity satisfies the condition $Q(n, M, B) \leq O(Q(n, 2M, B))$ is called *regular* (All algorithms presented in this chapter are regular). Intuitively, algorithms that slow down by a constant factor when memory (M) is reduced to half, are called regular. It immediately follows from the above lemma that if an algorithm whose number of cache misses satisfies the regularity condition does $Q(n, M, B)$ cache misses with optimal replacement then this algorithm would make $\Theta(Q(n, M, B))$ cache misses on a cache with LRU or FIFO replacement.

The automatic replacement and full associativity assumption can be implemented in software by using LRU implementation based on hashing. It was shown in [22] that a fully associative LRU replacement policy can be implemented in O(1) expected time using O($\frac{M}{B}$) records of size O(B) in ordinary memory. Note that, the above description about the cache oblivious model also proves that any optimal cache oblivious algorithm can also be optimally implemented in the external memory model.

We now turn our attention to multi-level ideal caches. We assume that all the levels of this cache hierarchy follow the inclusion property and are managed by an optimal replacement strategy. Thus on each level, an optimal cache oblivious algorithm will incur an asymptotically optimal number of cache misses. From

Lemma 1, this becomes true for cache hierarchies maintained by LRU and FIFO replacement strategies.

Apart from not knowing the values of M, B explicitly, some cache oblivious algorithms (for example optimal sorting algorithms) require a *tall cache* assumption. The tall cache assumption states that $M = \Omega(B^2)$ which is usually true in practice. Its notable that regular optimal cache oblivious algorithms are also optimal in SUMH [5] and HMM [1] models. Recently, compiler support for cache oblivious type algorithms have also been looked into [39, 40].

3 Algorithm design tools

In cache oblivious algorithm design some algorithm design techniques are used ubiquitously. One of them is a *scan* of an array which is laid out in contiguous memory. Irrespective of B, a scan takes at most $1 + \lceil \frac{N}{B} \rceil$ I/Os. The argument is trivial and very similar to the external memory scan algorithm Chapter **??**. The difference is that in the cache oblivious setting the buffer of size B is not explicitly maintained in memory. In the assumptions of the model, B is the size of the data that is always fetched from level 2 memory to level 1 memory. The scan does not touch the level 2 memory until its ready to evict the last loaded buffer of size B already in level 1. Hence, the total number of times the scan algorithm will force the CPU to bring buffers from the level 2 memory to level 1 memory is upper bounded by $1 + \lceil \frac{N}{B} \rceil$.

Exercise 1 *Convince yourself that indeed a scan of an array will at most load* $O(\frac{N}{B})$ *buffers from the level 2. Refine your analysis to* $1 + \lceil \frac{N}{B} \rceil$ *buffer loads.*

3.1 Main Tool: Divide and conquer

Chances are that the reader is already an expert in divide and conquer algorithms. This paradigm of algorithm design is used heavily in both parallel and external memory algorithms. It should not come as a surprise to the reader that cache oblivious algorithms make heavy use of this paradigm and lot of seemingly simple algorithms that were based on this paradigm are already cache oblivious! For instance, Strassen's matrix multiplication, quicksort, mergesort, closest pair [18], convex hulls [7], median selection [18] are all algorithms that are cache oblivious, though not all of them are optimal in this model. This means that they might be cache oblivious but can be modified to make fewer cache misses than they do in the current form. In the section on matrix multiplication we will see that Strassen's matrix multiplication algorithm is already optimal in the cache oblivious sense.

Why does divide and conquer help in general for cache oblivious algorithms? Divide and conquer algorithms split the instance of the problem to be solved into several sub problems such that each of the sub problems can be solved independentally. Since the algorithm recurses on the sub problems, at some point of time, the sub problems fit inside M and subsequent recursion, fits the sub

problems into B. For instance lets analyze the average number of cache misses in a randomized quicksort algorithm. This algorithm is quite cache friendly if not cache optimal and is described in Section 8.3 of [18].

Lemma 2. *Randomized version of quicksort incurs an expected number of $O(\frac{N}{B}\log_2(\frac{N}{B}))$ cache misses.*

Proof. Choosing a random pivot makes at most one cache miss. Splitting the input set into two output sets, such that the elements of one are all less than the pivot and the other greater than or equal to the pivot makes at most $\mathrm{O}(1+\frac{N}{B})$ cache misses by Exercise 1.

As soon as the size of the recursion fits into B, there are no more cache misses for that subproblem ($Q(B) = O(1)$). Hence the average cache complexity of a randomized quicksort algorithm can be given by the following recurrence (which is very similar to the average case analysis presented in [18]).

$$Q(N) = \frac{1}{N}\left[\sum_{i=1..\frac{N-1}{B}} (Q(i) + Q(N-i)) + (1 + \lceil\frac{N}{B}\rceil)\right] \tag{1}$$

which solves to $\mathrm{O}(\frac{N}{B}\log_2\frac{N}{B})$.

Exercise 2 *Solve the recurrence 1.*

Hint 1 *First simplify the recurrence to*

$$Q(N) = \frac{2}{N}\left[\sum_{i=1..\frac{N-1}{B}} Q(i) + \Theta(1+\frac{N}{B})\right] \tag{2}$$

and then you can take help from [23].

Exercise 3 *Prove that mergesort also does $O(\frac{N}{B}\log_2\frac{N}{B})$ cache misses.*

As we said earlier, the number of cache misses randomized quicksort makes is not optimal. The sorting lower bound in the cache oblivious model is also the same as the external memory model (See Chapter **??**). We will see later in the chapter a sorting algorithm that does sorting in $\mathrm{O}(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B})$ cache misses.

We now will analyze another cache oblivious algorithm that is also optimal, the *median finding* as given in [18] (Section 10.3). The recurrence for finding the median of N entities is given by:

$$Q(N) = Q(\frac{N}{5}) + Q(\frac{7N}{10}+6) + O(1+\frac{N}{B})$$

Solving which we get, $Q(N) = O(1+\frac{N}{B})$. Note that again $Q(B) = O(1)$ so the recursion stops adding cost to cache misses when the instance of the problem becomes smaller than B.

3.2 Randomization

Two basic tools that are almost always used in randomized algorithms are selection and permutation. Permuting a set or an array randomly can be done exactly the way distribution sort (See section 7) is done, except, that when comparing two objects, the outcome is defined by a random coin. We don't know of a simple way to do randomized shuffling cache obliviously yet. The same sorting lower bounds hold for shuffling too.

Random subset selection is a more interesting problem. In this section we describe how one could pick a random subset of size n^ϵ from an array of size n without incurring too many cache misses. This could be done in work complexity $O(n^\epsilon)$ and cache complexity $O(min(1+n^\epsilon, \frac{N}{B}))$. Use algorithm S of Knuth [27], Sec 3.4.2 with $N = n, n = n^\epsilon$.

By using the HyperGeometric distribution, a method mentioned as an exercise in Knuth's book is cache oblivious for sampling without replacement. We can sample with or without replacement k objects from n objects by incurring almost optimal number of cache faults.

This seems to be faster than linearly scanning the whole input array if the array is very big and in practice, this method looks quite promising. For instance we could generate 2^{15} random numbers in sorted order for $n = 2^{30}$ in .2 seconds on a 1Ghz Laptop without any effort to optimize the code.

Another method good for practical implementations is to generate n^ϵ numbers in an array ranging from $1..n$, sort the array and then pick the unique elements using a linear scan. Another very simple method to choose random samples is the use of Bernoulli trials.

In section 7 we use sampling with replacement. This kind of sampling appears wasteful, but we use it here to keep the analysis clean. Sampling without replacement would result in a marginally sharper analysis, at the cost of complicating the analysis.

3.3 Amortization

This technique of analysis can be used to show that the average cost of an operation is small, even though a single operation might take a long time to complete. It guarantees average performance of an operation in the worst case. Amortized analysis is helpful in gaining an insight of the design of a particular data structure or algorithm [18].

In this section we take an example to illustrate how amortization could help in design and analysis of a cache oblivious data structure called the packed memory structure [9]. This structure can help one maintain a dynamic van Emde Boas layout (section 6) of a strongly weight balanced search tree [9]. This structure can also be used as a cache oblivious linked list data structure and has been used to design dynamic dictionaries [10]. Our description mostly follows [9].

In the packed memory problem, also known as ordered file maintenance problem, we want to store N elements in an array of size $|A| = O(N)$. Note that $|A|$ is not N but cN for some $c > 1$.

We are supposed to design a data structure that supports inserts and deletes without doing too many cache misses and without taking too much time. The inserts in this structure are similar to linked list inserts, they also come with a pointer to the element after which the element needs to be inserted. Another constraint of the structure is the density constraint, any set of k contiguous elements are stored in a contiguous array of $O(k)$. Its not hard to imagine that to create such a structure, one should cleverly maintain gaps between elements such that inserts and deletes don't do too much damage in running time or locality of operation. Roughly speaking, when a subarray becomes too full or too empty, one could evenly redistribute the elements in a larger subarray. The subarray sizes range from $O(\log N)$ to N and are powers of two.

Let us first define some terms that we will need in the design. The density of a subarray u, denoted by $d(u)$ is equal to the number of elements stored in the subarray divided by the size of the subarray. Associated with each subarray is a density threshold. This thresholding is very similar to [26] except that [9] also uses a lower bound threshold. So this means that $d(u)$ for each subarray is bounded between an interval depending on the height of the node u in the tree. We will describe the data structure as a conceptual tree on the array A. Let the array A be split into $\log |A|$ size subarrays. These subarrays will form the leaves of a tree on whose nodes we will maintain density thresholds. Let the root node be at height zero, then leaf nodes are at height $\log \log |A|$. Each node maintains a upper and lower bound thresholds on the density. As soon as a node is discovered whose density $d(u)$ violates the thresholds, it is "fixed" so that the density lies again between the thresholds. The bounds for node densities $d(u)$ are (τ_k, ρ_k) where k is the height of the node u. τ_k and ρ_k are arithmetic progressions, $\tau_k = \tau_0 + k\delta$ and $\rho_k = \rho_0 - k\delta'$ where

$$0 < \rho_{\log \log N} < \rho_0 < \tau_0 < \tau_{\log \log N} = 1$$

$$\delta = \frac{\tau_{\log \log N} - \tau_0}{\log \log N}$$

$$\delta' = \frac{\rho_{\log \log N} - \rho_0}{\log \log N}$$

We will only describe insertions here, deletions are very similar and in fact easier than insertions. When a element x needs to be inserted, first the leaf that needs to contain it is located. If inserting x in the leaf does not interfere with the density threshold upper and lower bounds we are done. If not, then we walk up the tress from the leaf and locate the first node whose density threshold bounds are not violated. At this point, this node is *rebalanced*. Rebalancing means redistributing all the elements in the subarray corresponding to the node, evenly in the entire space available to the node. This also makes sure that the densities of the children of the rebalanced node respect their upper and lower bounds (Because thresholds become more relaxed down the tree).

We will now see how amortized analysis will help us prove that each insertion can be done in $O(\log^2 N)$ amortized time and $O(1 + \frac{\log^2 N}{B})$ amortized memory transfers. Suppose node u has depth l and was rebalanced, i.e. $\rho_l \leq d(u) \leq \tau_l$.

Some child of u, say v has violated its bounds for the density thresholds. Note that $\rho_l \leq d(v) \leq \tau_l$. For u to overflow again, we need at least $size(u)(\tau_{l+1} - \tau l)$ inserts, where $size(u)$ denotes the total number of elements that can be stored in node u. The number of elements touched while rebalancing u is $size(u)$. Thus the average work done per insertion in the node v is

$$\frac{size(u)}{size(v)(\tau_{l+1} - \tau_l)} = \frac{2}{(\tau_{l+1} - \tau_l)} = \frac{2 \log\log N}{\tau_{\log\log N} - \tau_0} = O(|A|) = O(\log N)$$

This shows the amortized number of array positions touched per insertion of an element x into each node u of the tree. Each time we insert x into A, we insert it into $\log |A|$ different nodes, hence total number of array positions touched is $O(\log^2 N)$. Each time we perform an insertion, we only scan A to the left and to the right, to decide how big a subarray we should rebalance (Note that this means we always touch contiguous memory locations). Thus the amortized number of memory locations touched is $O(1 + \frac{\log^2 N}{B})$.

Exercise 4 *Extend the analysis of the packed memory structure to handle deletions.*

Exercise 5 *Modify the data structure to support dynamic dictionary queries, Predecessor(x), ScanForward(x), ScanBackward(x) [10]. Before attempting this exercise, read section 6.*

4 Matrix transposition

Matrix transposition is a fundamental operation in linear algebra and in fast fourier transforms and has applications in numerical snalysis, image processing and graphics. The simplest hack for transposing a $N \times N$ square matrix in C++ could be:

```
for (i = 0; i < N; i++)
   for (j = i+1; j < N; j++)
       swap(A[i][j], A[j][i])
```

In C++ matrices are stored in "row-major" storage, i.e. the rightmost dimension varies the fastest (Chapter **??**). In the above case, the number of cache misses the code could do is $O(N^2)$. The optimal cache oblivious matrix transposition makes $O(1 + \frac{N^2}{B})$ cache misses. Before we go into the divide and conquer based algorithm for matrix transposition that is cache oblivious, lets see some experimental results. (See Figure 2, 3, 4, 5 , 6 and 7).

Here is the C/C++ code for cache oblivious matrix transposition. The following code takes as input a submatrix given by $(x, y) - (x + delx, y + dely)$ in the input matrix I and transposes it to the output matrix O. `ElementType`[1] can be any element type , for instance `long`.

[1] In all our experiments, `ElementType` was set to `long`.

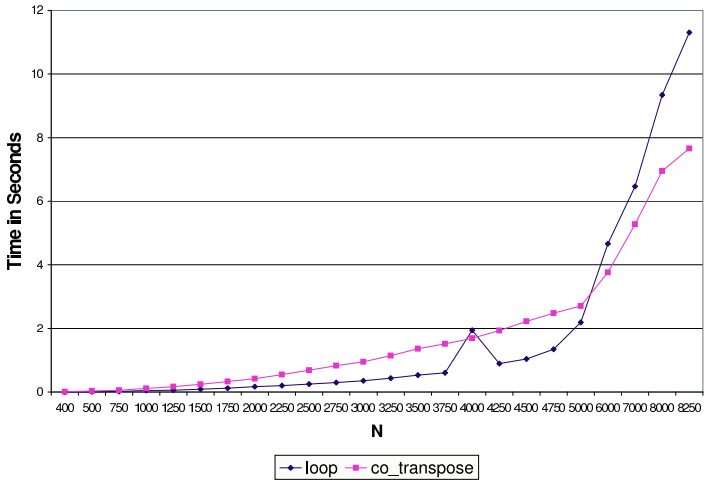

Fig. 2. Experiments with simple for loop and a cache oblivious matrix transposition subroutine on a Windows NT running on 1Ghz/512Mb RAM notebook, compiled with g++

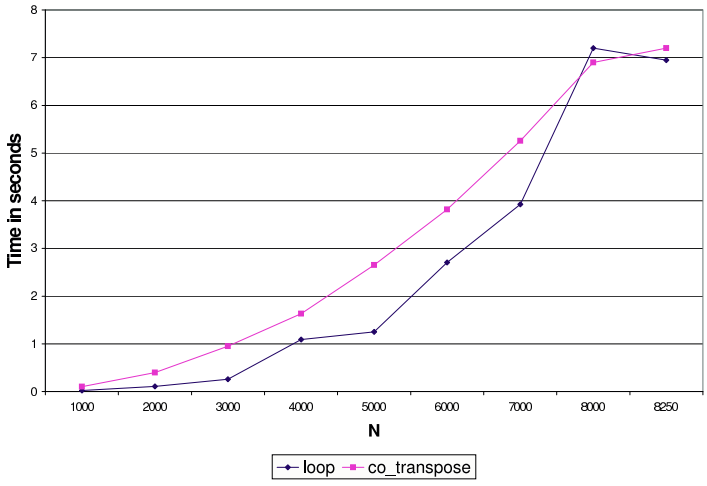

Fig. 3. Same experiment as in Figure 2 but now compiler has -O3 flag set

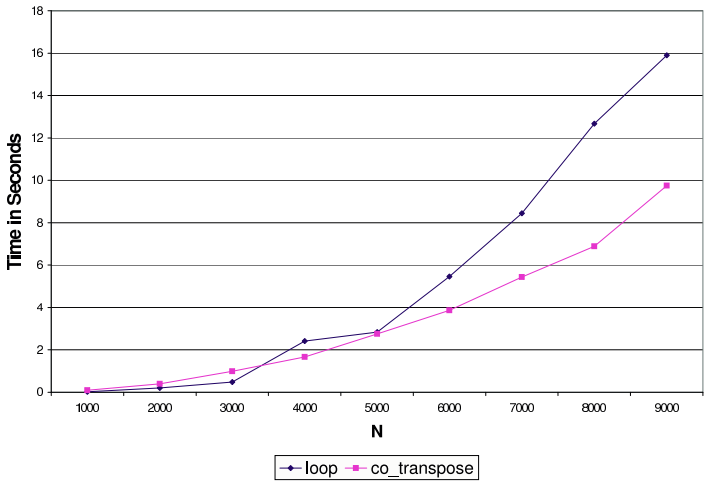

Fig. 4. Same experiment as in Figure 2 but now on a Linux machine, with Athlon 1Ghz processor and 512Mb RAM

```
void transpose(int x, int delx, int y, int dely,
                ElementType I[N][P], ElementType O[P][N]){
        // Base Case of recursion
        // Should be tailored for specific machines if one wants
        // this code to perform better.
        if((delx == 1) && (dely == 1)) {
                O[y][x] = I[x][y];
                return;
        }
        // Divide the transposition into two sub transposition
        // problems, depending upon which side of the matrix is
        // bigger.
        if(delx >= dely){
                int xmid = delx / 2;
                transpose(x,xmid,y,dely,I,O);
                transpose(x+xmid,delx-xmid,y,dely,I,O);
                return;
        }
        // Similarly cut from ymid into two subproblems
        ...
}
```

The code works by divide and conquer, dividing the bigger side of the matrix in the middle and recursing. We also compared matrix transposition code for a general $N \times P$ matrix (using two for loops) with the cache oblivious code (See Figure 5 and 6).

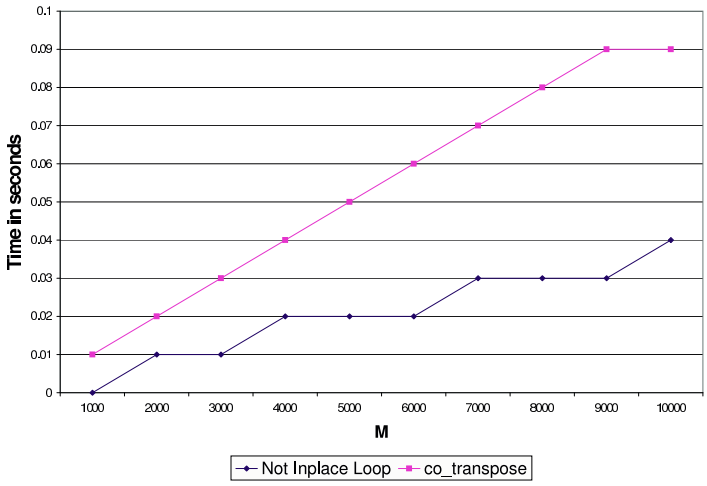

Fig. 5. A transpose for an $N \times P$ matrix where $P = 100$. Experiments on Linux machine as in Figure 4

Exercise 6 *Prove that on an $N \times N$ input matrix, the above code causes at most $O(1 + \frac{N^2}{B})$ cache misses.*

Hint 2 *Let the input be a matrix of $N \times P$ size. There are three cases:*

Case I $max\{N, P\} \leq \alpha B$ In this case,

$$Q(N, P) \leq \frac{NP}{B} + O(1)$$

Case II $N \leq \alpha B < P$ In this case,

$$Q(N, P) \leq \begin{cases} O(1 + N) & \text{if } \frac{\alpha B}{2} \leq P \leq \alpha B \\ 2Q(N, P/2) + O(1) & N \leq \alpha B < P \end{cases} \tag{3}$$

Case III $P \leq \alpha B < N$ Analogous to Case III.

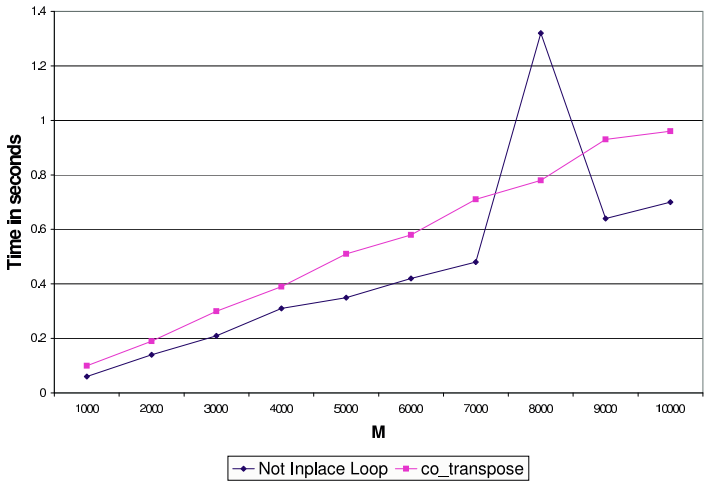

Fig. 6. Same experiment as in Figure 5 but now $P = 1000$.

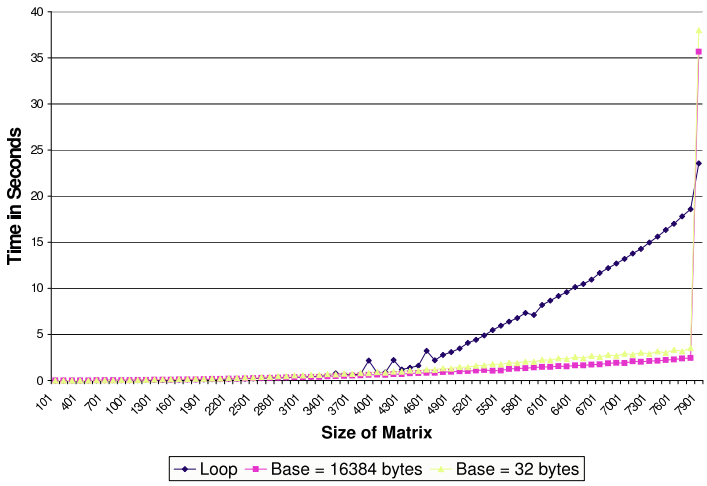

Fig. 7. The graph compares a simple for loop implementation with a blocked cache oblivious implementation. In the blocked cache oblivious implementation, we stop the recursion when the problem size becomes less than a certain block size and then use the simple for loop implementation inside the block. Note that using different block sizes has little effect on the running time. This experiment was done on the cygwin platform used earlier.

Case IV $min\{N, P\} \geq \alpha B$

$$Q(N,P) \leq \begin{cases} O(N+P+\frac{NP}{B}) & \text{if } \frac{\alpha B}{2} \leq N, P \leq \alpha B \\ 2Q(N,P/2)+O(1) & P \geq N \\ 2Q(N/2,P)+O(1) & N \geq P \end{cases} \quad (4)$$

The above recurrence solves to $Q(N,P) = O(1+\frac{NP}{B})$.

There is a simpler way to visualize the above mess. Once the recursion makes the matrix small enough such that $max(N,P) \leq \alpha B \leq \beta\sqrt{M}$ (here β is a suitable constant), or such that the submatrix (or the block) we need to transpose fits in memory, the number of I/O faults is equal to the scan of the elements in the submatrix. A packing argument of these not so small submatrices (blocks) in the large input matrix shows that we do not do too many I/O faults compared to a linear scan of all the elements.

Remark: Note that the timings of this algorithm are not so impressive except Figure 7. The algorithm given here is not the best for matrix transposition. If the reader is interested in practicality of matrix transposition, excellent references are [14, 39]. Also note that if one expands the base case (instead of moving single elements around, one moves multiple elements) the performance improves. Changing the environment (OS, Processor etc.) has a significant impact on performance of cache oblivious algorithms (implemented without any kind of blocking). The experimental results reported here do not match the results of [14]. Their implementation of cache oblivious matrix transposition always ran faster than their naive implementation, which was not the case for us. This could occur because of more than one reasons, one of them being use of loop unrolling directives to the compiler while optimization, function call overheads because we recurse down to the base case of size 1, whereas one could stop the recursion at some other constant size to speed the code up.

Figure 7 shows the effect of using blocked cache oblivious algorithm for matrix transposition. Note that in this case, the simple for loop algorithm is almost always outperformed. This comparison is not really fair. The cache oblivious algorithm gets to use blocking whereas the naive for loop moves one element at a time. A careful implementation of a blocked version of the simple for loop might beat the blocked cache oblivious transposition algorithm in practice. (See the timings of Algorithm 2 and 5 in [14]). The same remark also applies to matrix multiplication. (Figure 8)

5 Matrix multiplication

Matrix multiplication is one of the most studied computational problems: We are given two matrices of $m \times n$ and $n \times p$ and we want to compute the product matrix of $m \times p$ size. In this section we will use $n = m = N$ although the results can easily be extended to the case when they are not equal. Thus, we are given

two $N \times N$ matrices $x = (x_{i,j})$, $y = (y_{i,j})$, and we wish to compute their product z, i.e. there are N^2 outputs where the (i,j)'th output is

$$z_{i,j} = \sum_{k=1}^{N} x_{i,k} \cdot y_{k,j}$$

In 1969' Strassen surprised the world by showing an upper bound of $O(N^{\log_2 7})$[36] using a divide and conquer algorithm. This bound was later improved and the best upper bound today is $O(N^{2.376})$ [17]. We will use Strassen's algorithm as given in [18] for the cache oblivious analysis.

The algorithm breaks the three matrices x, y, z into four submatrices of size $\frac{N}{2} \times \frac{N}{2}$, rewriting the equation $z = xy$ as :

$$\begin{pmatrix} r & s \\ t & u \end{pmatrix} = \begin{pmatrix} a & c \\ b & d \end{pmatrix} \times \begin{pmatrix} e & g \\ f & h \end{pmatrix}$$

Now we have 8 subproblems (matrix multiplications) of size $\frac{N}{2} \times \frac{N}{2}$ and four additions (which will lead us to a $O(N^3)$ algorithm). This can be done more cleverly by only solving 7 subproblems and doing more additions and subtractions but still keeping the number of additions and subtraction a constant (See [18] for more details). The recurrence for the internal work now becomes :

$$T(N) = 7T(\frac{N}{2}) + \Theta(N^2)$$

which solves to $T(N) = O(N^{\lg 7}) = O(N^{2.81})$. The recurrence for the cache faults is:

$$Q(N) \leq \begin{cases} O(1 + N + \frac{N^2}{B}) & \text{if } N^2 \leq \alpha M \\ 7Q(\frac{N}{2}) + O(1 + \frac{N^2}{B}) & \text{otherwise.} \end{cases} \tag{5}$$

which solves to $Q(N) \leq O(N + \frac{N^2}{B} + \frac{N^{\lg 7}}{B\sqrt{M}})$. We also implemented a blocked cache oblivious matrix transposition routine that works very similar to the transposition routine in the previous section. It breaks the largest of the three dimensions of the matrices to be multiplied (divides it by 2) and recurses till the block size is reached. Note that this algorithm is not optimal and does $O(1+\frac{N^2}{B}+\frac{N^3}{B\sqrt{M}})$ cache misses.

We also present here experiments of a simpler divide and conquer matrix multiplication algorithm. The worst case of the implementation is $O(N + N^2 + \frac{N^3}{B\sqrt{M}})$. The experimental results are shown in Figure 8. An excellent reference for practical matrix multiplication results is [19].

Exercise 7 *Implement and analyze a matrix multiplication routine for $N \times N$ matrices.*

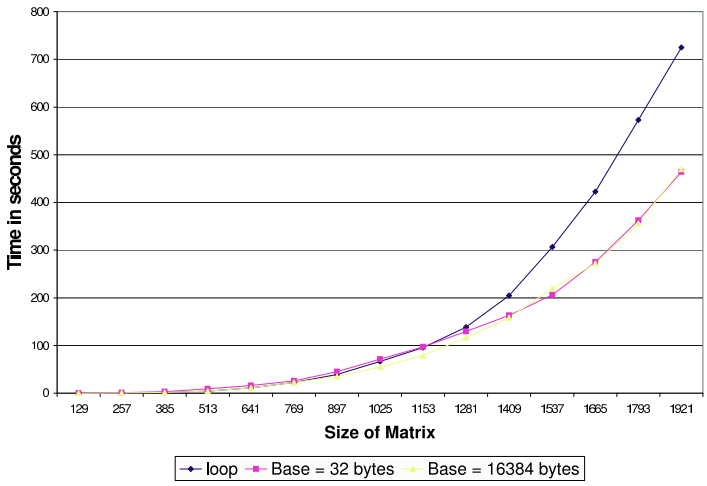

Fig. 8. Blocked cache oblivious matrix multiplication compared with simple for loop based matrix multiplication. This experiment was done on an dual processor Intel Itanium with 2Gb RAM. Only one processor was being used. Note that on this platform, the blocked implementation consistently outperforms the simple loop code. The base case has little effect for large size matrices.

6 Searching using Van Emde Boas layout

In this section we report a method to speed up simple binary searches on a balanced binary tree. This method could be used to optimize or speed up any kind of search on a tree as long as the tree is static and balanced. It is easy to code, uses the fact that the memory consists of a cache hierarchy, and could be exploited to speed up tree based search structures on most current machines. Experimental results show that this method could speed up searches by a factor of 5 or more in certain cases!

It turns out that a balanced binary tree has a very simple layout that is cache-oblivious. By layout here, we mean the mapping of the nodes of a binary tree to the indices of an array where the nodes are actually stored. For example Figure 9 shows a layout of the tree in the figure. The nodes should be stored in the bottom array in the order shown for searches to be fast and use the cache hierarchy.

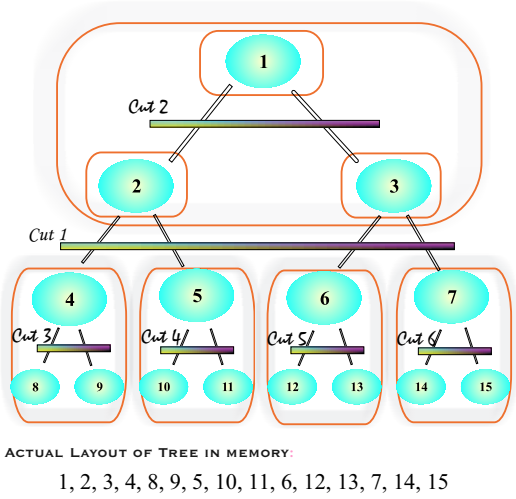

Fig. 9. An example of the Van Emde Boas layout using a balanced binary tree.

Given a complete binary tree, we describe a mapping from the nodes of the tree to positions of an array in memory. Suppose the tree has N items and has height $h = \log N + 1$. Split the tree in the middle, at height $h/2$. This breaks the tree into a top recursive subtree of height $\lfloor h/2 \rfloor$ and several bottom subtrees $B_1, B_2, ..., B_k$ of height $\lceil h/2 \rceil$. There are $\sqrt{N}$ bottom recursive subtrees, each of size $\sqrt{N}$. The top subtree occupies the top part in the array of allocated nodes, and then the B_i's are laid out. Every subtree is recursively laid out.

Another way to see the algorithm is to run a breadth first search on the top node of the tree and run it till $\sqrt{N}$ nodes are in the BFS, See Figure 10. The

figure shows the run of the algorithm for the first BFS when the tree size is $\sqrt{N}$ Then the tree consists of the part that is covered by the BFS and trees hanging out. BFS can now be recursively run on each tree, including the covered part. Note that in the second level of recursion, the tree size is $\sqrt{N}$ and the BFS will cover only $N^{\frac{1}{4}}$ nodes since the same algorithm is run on each subtree of $\sqrt{N}$. The main idea behind the algorithm is to store recursive sub-trees in contiguous blocks of memory.

Lets now try to analyze the number of cache misses when a search is performed. We can conceptually stop the recursion at the level of detail where the size of the subtrees has size $\leq B$. Since these subtrees are stored contiguously, they at most fit in two blocks. (A block can not span three blocks of memory when stored) The height of these subtrees is $\log B$. A search path from root to leaf crosses $O(\frac{\log N}{\log B}) = O(\log_B N)$ subtrees. So the total number of cache misses is bounded by $O(\log_B N)$.

Exercise 8 *Show that the Van Emde Boas layout can be at most a constant factor of* 4 *away from an optimal layout (which knows the parameter* B*). Show that the constant* 4 *can be reduced to* 2 *for average case queries. Can one get a better constant factor than* 2*?*

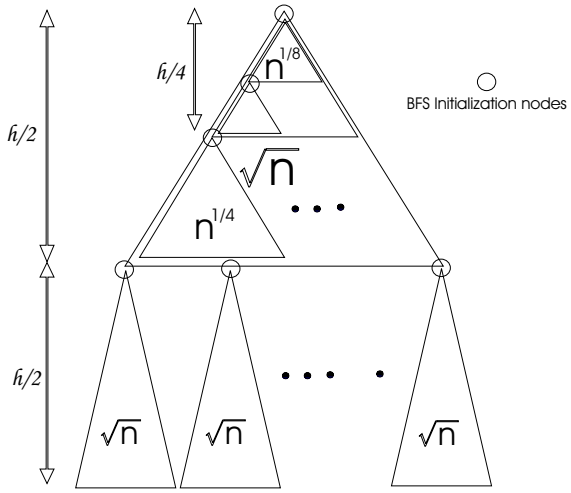

Fig. 10. Another view of the algorithm in action

6.1 Experiments

We did a very simple experiment to see how in real life, this kind of layout would help. A vector was sorted and a binary search tree was built on it. A query

vector was generated with random numbers and searched on this BST which was laid out in pre-order. Why we chose pre-order compared to random layout was because most people code a BST in either pre/post/in-order compared to randomly laying it (Which incidentally is very bad for cache health).

Once this query was done, we laid the BST using Van Emde Boas Layout and gave it the same query vector. Before timing both trees, we made sure that both had enough queries to begin with otherwise, the order of timing could also effect the search times. (Because the one that is searched last, has some help from the cache). The code written for this experimentation is below 300 lines The results reported in Figure 11 used cygwin[2] g++ version 2.95 with compiler optimization $-O3$ and was running on my laptop with 1Ghz processor, 512Mb RAM and Windows 2000 as the Operating System. The experiment reported in Figure 12 were done on a Itanium dual processor system with 2Gb RAM. (Only one processor was being used)

Currently the code copies the entire array in which the tree exists into another array when it makes the tree cache oblivious. This could also be done in the same array though that would have complicated the implementation a bit. One way to do this is to maintain pointers to and from the array of nodes to the tree structure, and swap nodes instead of copying them into a new array. Another way could be to use a permutation table. We chose to copy the whole tree into a new array just because this seemed to be the simplest way to test the speed up given by cache oblivious layouts. For more detailed experimental results on comparing searching in cache aware and cache oblivious search trees, the reader is referred to [28]. There is a big difference between the graphs reported here for searching and in [28]. One of the reasons might be that the size of the nodes were fixed to be 4 bytes in [28] whereas the experiments reported here use bigger size nodes.

7 Sorting

Sorting is a fundamental problem in computing. Sorting very large data sets is a key routine in many external memory applications. We have already seen how to do optimal sorting in the external memory model. In this section we outline some theory and experimentation related to sorting in the cache oblivious model. Some excellent references for reading more on the influence of caches on the performance of sorting are [29, 31], Chapter **??**.

7.1 Randomized distribution sorting

There are two optimal sorting algorithms known, funnel sort and distribution sort. Funnel sort is derived from merge sort and distribution sort is a generalization of quick sort. We will see the analysis and implementation of a variant

[2] Cygwin is a linux like environment which runs on windows. It's available from `http://www.cygwin.org`

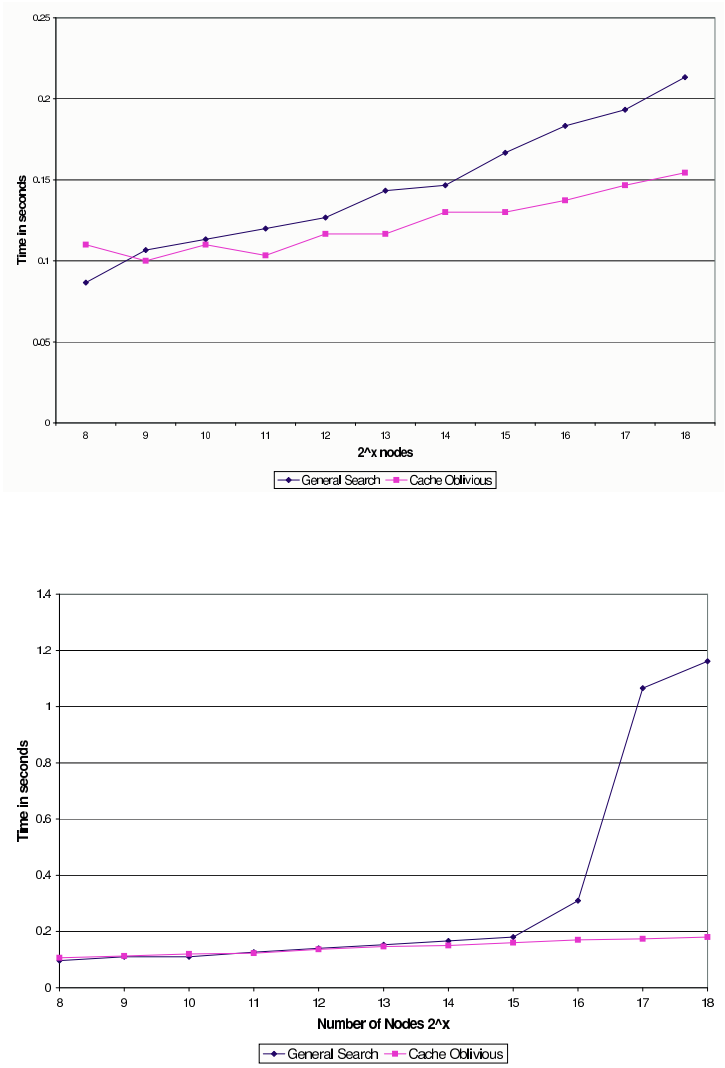

Fig. 11. Top: 32 byte nodes, Bottom: 256 byte tree nodes. All Timings are from internal memory. For bigger node sizes, and trees that do not fit into internal memory, we expect the speed up to be more than internal memory. These experiments were done on my PIII notebook

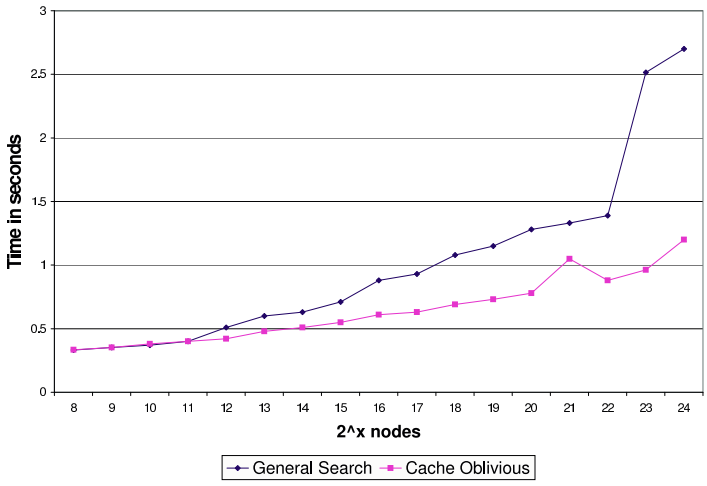

Fig. 12. Similar to the last experiment, this experiment was performed on a Itanium with 48 byte node sizes.

of distribution sort that is optimal and easier to implement than the original distribution sort of [22] (The original algorithm uses median finding and is deterministic). The algorithm for randomized distribution sort is presented in Algorithm 1.

Algorithm 1 procedure `Sort`(A)

Require: An input array A of Size N

1: Partition A into $\sqrt{N}$ sub arrays of size $\sqrt{N}$. Recursively sort each subarray.
2: $R \leftarrow$ `Sample`$(A, \sqrt{N})$
3: Sort(R) recursively. $R = \{r_0, r_1, ..., r_{\sqrt{N}}\}$.
4: Calculate counts c_i such that $c_i = |\{x \mid x \in A \text{ and } r_i \leq x < r_{i+1}\}|$
5: $c_{1+\sqrt{N}} = |A| - \Sigma_i c_i$.
6: Distribute A into buckets $B_0, B_1, ...B_{1+\sqrt{N}}$ where last element of each bucket is r_i except the last bucket. For the last bucket, the last element is maximum element of A. Note that $|B_i| = c_i$.
7: Recursively sort each B_i
8: Copy sorted buckets back to A.

The sorting procedure assumes the presence of three extra functions which are cache oblivious, choosing a random sample in Step 2, the counting in Step 4 and the distribution in Step 6. The random sampling step in the algorithm is used to determine splitters for the buckets, so we are in fact looking for splitters.

The function `Sample`(X, k) in step 2 takes as input an array X and the size of the sample that it needs to return as output. A very similar sampling is needed in Sample Sort [11] to select splitters. We will use the same algorithm here. What we need is a random sample such that the c_i's do not vary too much from $\sqrt{N}$ with high probability. To do this, draw a random sample of $\beta\sqrt{N}$ from A, sort the sample and then pick each βth element from the sorted sample. In [11], β is referred to as the *oversampling ratio*, the more the β the greater the probability that c_i's concentrate around $\sqrt{N}$.

The distribution and counting phases are quite analogous. Here is how we do distribution:

Algorithm 2 procedure `Distribute`(`int` i, `int` j, `int` m)

```
1: if m = 1 then
2:    CopyElements(i, j)
3: else
4:    Distribute(i, j, m/2)
5:    Distribute(i + m/2, j, m/2)
6:    Distribute(i, j + m/2, m/2)
7:    Distribute(i + m/2, j + m/2, m/2)
8: end if
```

In coding one would have to also take care if m is odd or even and appropriately apply floor ceiling operations for the above code to work. In our code, we treated it by cases, when m was odd and when m was even, the rounding was different. Anyway, this is a minor detail and can be easily figured out. In the base case `CopyElements`(i, j) copies all elements from subarray i that belong to bucket j. If the function `CopyElements`(i, j) is replaced by `CountElements`(i, j) in `Distibute`(), the counting that is needed in Step 4 of Algorithm 1 can be performed. `CountElements`() increments c_j's depending upon how many elements of subarray i lie between r_j and r_{j+1}. Since these subarrays are sorted, this only requires a linear scan.

Before we goto the analysis, lets see how the implementation works in practice. We report here two experiments where we compare C++ STL sort with our implementation. In our implementation we found out that as we recurse more, the performance of our sorting algorithm deteriorates. In the first graph in Figure 13, Series 1 is the STL Sort and Series 2 is our sorting implementation. The recursion is only 1 level, i.e. after the problem size becomes $\sqrt{N}$ STL Sort is used. In Figure 14 two levels of recursion are used. Note that doing more levels of recursion degrades performance. At least for 1 level of recursion, the randomized distribution sort is quite competitive and since these experiments are all in-memory tests, it might be possible to beat STL Sort when the data size does not fit in memory but we could not experiment with such large data sizes.

Lets now peek at the analysis. For a good random sample, $Pr(c_i \geq \alpha\sqrt{N}) \leq Ne^{-(1-\frac{1}{\alpha})^2\frac{\beta\alpha}{2}}$ (for some $\alpha > 1, \beta > \log N$) [11]. This follows from Chernoff

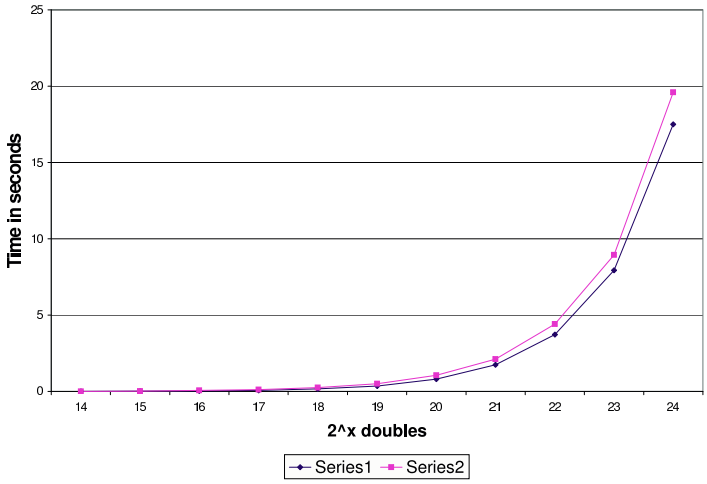

Fig. 13. Two Pass cache oblivious distribution sort, one level of recursion

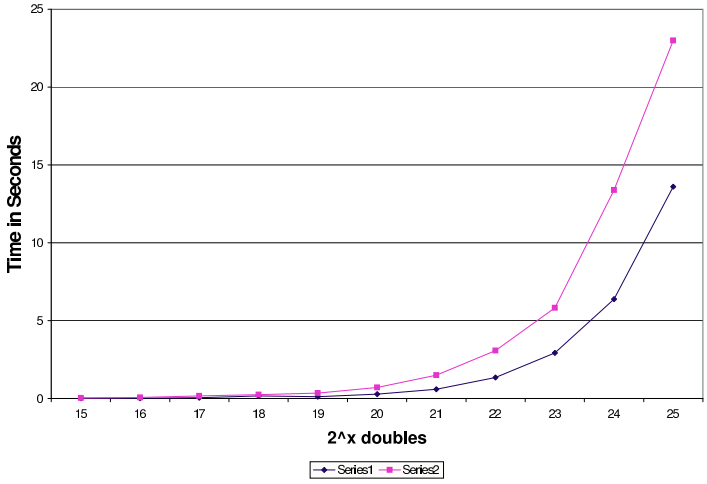

Fig. 14. Two Pass cache oblivious distribution sort, two levels of recursion

bound type arguments. Once we know that the subproblems we are going to work on are bounded in size with high probability, the expected cache complexity follows the recurrence:

$$Q(N) \leq \begin{cases} O(1+\frac{N}{B}) & \text{if } N \leq \alpha M \\ 2\sqrt{N}Q(\sqrt{N}) + Q(\sqrt{N}\log N) + O(1+\frac{N}{B}) & \text{otherwise.} \end{cases} \quad (6)$$

which solves to $Q(n) \leq O(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B})$.

Exercise 9 *A sorting algorithm called columnsort [30] was shown to be practical for large inputs [15]. The cache complexity of colum sort can be defined by the following recurrance.*

$$Q(n) = \begin{cases} O(1+\frac{N}{B}) & \textit{if } N \leq \alpha M \\ 8N^{\frac{1}{4}}Q(N^{\frac{3}{4}}) + O(1+\frac{N}{B}) & \textit{otherwise.} \end{cases} \quad (7)$$

where α is a sufficiently small constant. Note that the recurrance above is very similar to the recurrance of cache oblivious distribution sort [22] except for the fact that the constant is 8 instead of 2. Prove that column sort is not optimal in the cache oblivious model.

8 Is the model an oversimplification?

In theory, both the cache oblivious and the external memory models are nice to work with, because of their simplicity. Lot of the work done in the external memory model has been turned into practical results as well. Before one makes his hand "dirty" with implementing an algorithm in the cache oblivious or the external memory model, one should be aware of practical things that might become detrimental to the speed of the code but are not caught in the theoretical setup.

Here we list a few practical glitches that are shared by both the cache oblivious and the external memory model. The ones that are not shared are marked[3] accordingly. A reader that wants to use these models to design practical algorithms and especially one who wants to write code, should keep these issues in mind. Code written and algorithms designed keeping the following things in mind, could be a lot faster than just directly coding an algorithm that is optimal in either the cache oblivious or the external memory model.

TLB^o TLBs are caches on page tables, are usually small with 128-256 entries and are like just any other cache. They can be implemented as fully associative. The model does not take into account TLB misses. For the importance of TLB on performance of programs refer to the section on cache oblivious models in Chapter **??**.

[3] A superscript 'o' means this issue only applies to the cache oblivious model.

Concurrency The model does not talk about I/O and CPU concurrency, which automatically looses it a 2x factor in terms of constants. The need for speed might drive future *uniprocessor* systems to diversify and look for alternative solutions in terms of concurrency on a single chip, for instance the hyper-threading[4] introduced by Intel in its latest Xeons is a glaring example. On these kind of systems and other multiprocessor systems, *coherence misses* might become an issue. This is hard to capture in the cache oblivious model and for most algorithms that have been devised in this model already, concurrency is still an open problem. A parallel cache oblivious model would be really welcome for practitioners who would like to apply cache oblivious algorithms to multiprocessor systems.

Associativity^o The assumption of the fully associative cache is not so nice. In reality caches are either direct mapped or k-way associative (typically $k = 2, 4, 8$). If two objects map to the same location in the cache and are referenced in temporal proximity, the accesses will become costlier than they are assumed in the model (also known as cache interference problem [37]). Also, $k-$way set associative caches are implemented by using more comparators. (See Chapter **??**)

Instruction/Unified Caches Does not deal with the issue of instruction caches. Rarely executed, special case code disrupts locality. Loops with few iterations that call other routines make loop locality hard to exploit and plenty of loopless code hampers temporal locality. Issues related to instruction caches are not modeled in the cache oblivious model. *Unified caches* (e.g. the latest Intel Itanium chips L2 and L3 caches) are used in some machines where instruction and data caches are merged(e.g. Intel PIII, Itaniums). These are another challenge to handle in the model.

Replacement Policy^o Current operating systems do not page more than 4Gb of memory because of address space limitations. That means one would have to use legacy code on these systems for paging. This problem makes portability of cache oblivious code for big problems a myth! In the experiments reported in this chapter, we could not do external memory experimentation because the OS did not allow us to allocate array sizes of more than a GB or so. One can overcome this problem by writing one's own paging system over the OS to do experimentation of cache oblivious algorithms on huge data sizes. But then its not so clear if writing a paging system is easier or handling disks explicitly in an application. This problem does not exist on 64-bit operating systems and should go away with time.

Multiple Disks^o For "most" applications where data is huge and external memory algorithms are required, using Multiple disks is an option to increase I/O efficiency. As of now, the cache oblivious model does not handle this case, though it might not be tough to introduce multiple disks in the model.

[4] One physical processor Intel Xeon MP forms two logical processors which share CPU computational resources The software sees two CPUs and can distribute work load between them as a normal dual processor system.

Write-through caches^o L1 caches in many new CPUs is write through, i.e. it transmits a written value to L2 cache immediately [21, 24]. Write through caches are simpler to manage and can always discard cache data without any bookkeeping (Read misses can not result in writes). With write through caches (e.g. DECStation 3100, Intel Itanium), one can no longer argue that there are no misses once the problem size fits into cache! *Victim Caches* implemented in HP and Alpha machines are caches that are implemented as small buffers to reduce the effect of conflicts in set-associative caches. There also should be kept in mind when designing code for these machines.

Complicated Algorithms^o **and Asymptotics** For non-trivial problems the algorithms can become quite complicated and impractical, a glaring instance of which is sorting. The speed by which different levels of memory differ in data transfer are constants! For instance the speed difference between L1 and L2 caches on a typical Intel pentium can be around 10. Using an $O()$ notation for an algorithm that is trying to beat a constant of 10, and sometimes not even talking about those constants while designing algorithms can show up in practice (Also see Chapter **??**). For instance there are "constants" involved in simulating a fully associative cache on a k-way associative cache. Not using I/O concurrently with CPU can make an algorithm loose another constant. Can one really afford to hide these constants in the design of a cache oblivious algorithm in real code?

Despite these limitations the model does perform very well for some applications [14, 28, 20], but might be outperformed by more coding effort combined with cache aware algorithms [32, 14, 28, 20]. Here's an intercept from an experimental paper by Chatterjee and Sen [14].

> Our major conclusion are as follows: Limited associativity in the mapping from main memory addresses to cache sets can significantly degrade running time; the limited number of TLB entries can easily lead to thrashing; the fanciest optimal algorithms are not competitive on real machines even at fairly large problem sizes unless cache miss penalties are quite high; low level performance tuning "hacks", such as register tiling and array alignment, can significantly distort the effect of improved algorithms, ...

9 Other Results

We present here problems, related bounds and references for more interested readers. Note that in the table, *sort*() and *scan*() denote the number of cache misses of scan and sorting functions done by an optimal cache oblivious implementation.

Data Structure/Algorithm	Cache Complexity	Operations
Array Reversal	scan(N)	
List Ranking [16]	sort(N)	
LU Decomposition [38]	$\Theta(1 + \frac{N^2}{B} + \frac{N^3}{B\sqrt{M}})$	On $N \times N$ matrices
FFT [22]	sort(N)	
B-Trees [10, 13]	Amortized $O(\log_B N)$	Insertions/Deletions
Priority Queues [8, 12]	$O(\frac{1}{B} \log_{\frac{M}{B}} \frac{N}{B})$	Insertions/Deletions/deletemin

10 Open problems

Sorting strings in the cache oblivious model is still open. Optimal shortest paths and minimum spanning forests still need to be explored in the model. Optimal simple convex hull algorithms for $d-$dimensions is open. There are a lot of problems that still can be explored in this model both theoretically and practically.

One of the major open problems is to extensively evaluate how practical cache oblivious algorithms really are. For instance, our sorting experiment results don't look very promising. Is there a way to sort in the cache oblivious model that is at par with cache aware sorting codes already available? (Toy experiments comparing quicksort with a modified funnelsort or distribution sort don't count!) Currently the only impressive code that might back up "practicality" claims of cache oblivious algorithms is FFTW [20]. It looks like that the practicality of FFTW is more about coding tricks (special purpose compiler) that give it its speed and practicality than the theory of cache obliviousness. Moreover, FFTW only uses cache oblivious FFT for its base cases (Called codelets).

Matrix multiplication and transposition using blocked cache oblivious algorithms do fairly well in comparison with cache aware/external memory algorithms. B-Trees also seem to do well in this model [10, 13]. Are there any other non-trivial optimal cache oblivious algorithms and data structures that are really practical and come close to cache aware algorithms, only time will tell!

For some applications, it might happen that there are more than one cache oblivious algorithms available. And out of the few cache oblivious algorithms available, only one is good in practice and the rest are not. For instance, for matrix transposition, there are at least two cache oblivious algorithms coded in [14]. There is one coded in this chapter. Out of these three algorithms only one really performs well in practice. In this way, cache obliviousness might be a necessary but not a sufficient condition for practical implementations. Another example is FFTW, where a optimal cache oblivious algorithm is outperformed by a slightly suboptimal cache oblivious algorithm. Is there a simple model that captures both necessary and sufficient condition for practical results?

11 Acknowledgements

The author would like to thank Michael Bender, Matteo Frigo, Joe Mitchell, Edgar Ramos and Peter Sanders for discussions on cache obliviousness and to

MPI Informatik, Saarbrücken, Germany, for hosting him. The section on randomization came up in discussions with Peter Sanders.

References

1. A. Aggarwal, B. Alpern, A. K. Chandra, and M. Snir. A model for hierarchical memory. In *Proc. 19th Annu. ACM Sympos. Theory Comput.*, pages 305–313, 1987.
2. A. Aggarwal and A. K. Chandra. Virtual memory algorithms. In *Proc. 20th Annu. ACM Sympos. Theory Comput.*, pages 173–185, 1988.
3. A. Aggarwal, A. K. Chandra, and M. Snir. Hierarchical memory with block transfer. In *Proc. 28rd Annu. IEEE Sympos. Found. Comput. Sci.*, pages 204–216, 1987.
4. A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Commun. ACM*, 31:1116–1127, 1988.
5. B. Alpern, L. Carter, and E. Feig. Uniform memory hierarchies. In *focs*, pages 600–608, 1990.
6. B. Alpern, L. Carter, E. Feig, and T. Selker. The uniform memory hierarchy model of computation. *Algorithmica*, 12(2-3), 1994.
7. N. M. Amato and Edgar A. Ramos. On computing Voronoi diagrams by divide-prune-and-conquer. In *Proc. 12th Annu. ACM Sympos. Comput. Geom.*, pages 166–175, 1996.
8. L. Arge, M. A. Bender, E. D. Demaine, B. Holland-Minkley, and J. I. Munro. Cache-oblivious priority queue and graph algorithm applications. In *Proceedings of the 34th Annual ACM Symposium on Theory of Computing (STOC)*, pages 268–276, 2002.
9. M. A. Bender, E. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 399–409, 2000.
10. M. A. Bender, Z. Duan, J. Iacono, and J. Wu. A locality-preserving cache-oblivious dynamic dictionary. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 29–38, 2002.
11. G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. Greg Plaxton, S. J. Smith, and M. Zagha. A comparison of sorting algorithms for the connection machine CM-2. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 3–16, 1991.
12. G. S. Brodal and R. Fagerberg. Funnel heap - a cache oblivious priority queue. In *Proc. 13th Annual International Symposium on Algorithms and Computation*, Lecture Notes in Computer Science. 2002.
13. G. S. Brodal, R. Fagerberg, and R. Jacob. Cache oblivious search trees via binary trees of small height. Technical Report BRICS-RS-01-36, BRICS, Department of Computer Science, University of Aarhus, October 2001.
14. S. Chatterjee and S. Sen. Cache-efficient matrix transposition. In *HPCA*, pages 195–205, 2000.
15. G. Chaudhry, T. H. Cormen, and L. F. Wisniewski. Columnsort lives! an efficient out-of-core sorting program. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 169–178, 2001.
16. Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proc. 6th ACM-SIAM Sympos. Discrete Algorithms*, pages 139–149, 1995.

17. D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progression. *Journal of Symbolic Computation*, 9:251–280, 1990.
18. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.
19. N. Eiron, M. Rodeh, and I. Steinwarts. Matrix multiplication: A case study of algorithm engineering. In *2nd Workshop on Algorithm Engineering*, volume 16, pages 98–109, 1998.
20. M. Frigo. A fast fourier transform compiler. In *PLDI'99 — Conference on Programming Language Design and Implementation*, Atlanta, GA, 1999.
21. M. Frigo. Portable high-performance programs. Technical Report MIT/LCS/TR-785, 1999.
22. M. Frigo, Charles E. Leiserson, H. Prokop, and S. Ramachandran. Cache oblivious algorithms. In *Proc. 40th Annual Symposium on Foundations of Computer Science*, October 1999.
23. R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics*. Addison-Wesley, Reading, MA, 1989.
24. J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 1990.
25. J. W. Hong and H. T. Kung. I/o complexity: The red-blue pebble game. In *stoc*, pages 326–333, 1981.
26. A. Itai, A.G. Konheim, and M. Rodeh. A sparse table implementation of priority queues. In *Automata, Languages and Programming, 8th Colloquium*, volume 115, pages 417–431, 13–17 July 1981.
27. D. E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, 2nd edition, 1981.
28. R. E. Ladner, R. Fortna, and B. H. Nguyen. A comparison of cache aware and cache oblivious static search trees using program instrumentation. In *To appear in LNCS volume devoted to Experimental Algorithmics*, April 2002.
29. A. LaMarca and R.E. Ladner. The influence of caches on the performance of sorting. In *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 370–379, 5–7 January 1997.
30. T. Leighton. Tight bounds on the complexity of parallel sorting. In *STOC 84*, pages 71–80, 1984.
31. C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. B. Lomet. Alphasort: A risc machine sort. In R. T. Snodgrass and M. Winslett, editors, *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, May 24-27, 1994*, pages 233–242. ACM Press, 1994.
32. N. Rahman, R. Cole, and R. Raman. Optimized predecessor data structures for internal memory. In *5th Workshop on Algorithms Engineering (WAE)*, 2001.
33. J. E. Savage. Extending the Hong-Kung model to memory hierachies. In *Proceedings of the 1st Annual International Conference on Computing and Combinatorics*, volume 959 of *LNCS*, pages 270–281, August 1995.
34. S. Sen and S. Chatterjee. Towards a theory of cache-efficient algorithms. In *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 829–838, January 2000.
35. D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28:202–208, 1985.
36. V. Strassen. Gaussian elimination is not optimal. *Numer Math*, 13:354–356, 1969.
37. O. Temam, C. Fricker, and William Jalby. Cache interference phenomena. In *Measurement and Modeling of Computer Systems*, pages 261–271, 1994.

38. S. Toledo. Locality of reference in *LU* decomposition with partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 18(4):1065–1081, October 1997.
39. D. S. Wise. Ahnentafel indexing into morton-ordered arrays, or matrix locality for free. In *Euro-Par 2000 – Parallel Processing*, volume 1900 of *LNCS*, pages 774–784, August 2000.
40. Q. Yi, V. Advi, and K. Kennedy. Transforming loops to recursion for multi-level memory hierarchies. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 169–181, Vancouver, Canada, June 2000. ACM.