Chapter
Five
Chapter 5

CACHE OBLIVIOUS SORTING

1. INTRODUCTION

Sorting is a process of rearranging a sequence of objects into some kind of predefined linear order. String data is very common and most occurring data type. Sorting a string involves comparison it character by character which is more time consuming than integer sorting. Also, sorting forms the basis of many applications like data processing, databases, pattern matching and searching etc. So implementing improvements to make it fast and efficient will help in reducing the computational time and thus making our applications run faster. There are various fast and efficient string sorting algorithms. The algorithms have been divided into two categories cache-aware and cache-oblivious.

A cache oblivious sorting algorithm has not depended on the parameter M, the cache size. But many sorting algorithm however will depend on M, thereby increasing the complexity of our analysis.
In cache oblivious setting the computational model is a machine with two levels of memory: a cache limited capacity and secondary memory of infinite capacity. The core assumption of the cache oblivious model is that M and B are unknown to the algorithm whereas in the related I/O model introduced by Aggarwal and J. S. Vitter [1] the algorithm know M and B and algorithm perform block transfers explicitly. Cache oblivious model and its relation to multilevel memory hierarchies are given in [2].

Many problems can be reduced to cache obvious sorting. In particular Arge et al [3] developed a cache oblivious priority queue based on a reduction to sorting. They furthermore showed how a cache obvious priority queue can be applied to solve a sequence of graph problems.

Bordal and Fagerberg in [4] showed how to modify cache obvious funnel sort algorithm to solve several problems within computational geometry.

We investigate the probably most basic nontrivial algorithm, the sorting of array. We want to emphasize that to produce the sorting methods on computer architecture. We want to demonstrate that it is possible to construct an algorithm where memory address copies stepwise one, which also eliminate the need for transpositions. We will therefore concentrate on the basic idea of the algorithm and present some of nice properties that result from this approach.

2. Cache-Oblivious Sorting
Algorithms that use multi-layered memory hierarchies efficiently have traditionally relied on detailed knowledge of the characteristics of memory systems. The cache-oblivious approach changed this in 1999 by making it possible to design memory-efficient algorithms for hierarchical memory systems without such detailed knowledge. As a consequence, one single implementation of a cache-oblivious algorithm is efficient on any memory hierarchy. Cache-oblivious algorithms are analyzed in the ideal-cache model, which is an abstraction of real memory systems. We derive the various cache-oblivious sorting algorithms in the ideal cache model. We develop new designs that lay out a cache-oblivious memory in linear time, which is an improvement of the sorting algorithms known so far. We conclude that by combining the ideal-cache model and sorting relative performance of programs can be predicted quite precisely, provided that the analysis is carefully done.

Sorting Problem

Input : array containing x1, x2, ... , xn

Output : array with x1, x2, ... , xn in sorted order

Elements can be compared and copied
3. Cache Oblivious Sequential Sorting

Sorting is a fundamental problem in computing. Sorting very large data sets is a key routine in many external memory applications. 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. The simplest hack for sorting for N dimension in C++ could be:

1. Repeat steps 2 and 3 for k = 1 to N-1
2. Set PTR = 1
3. Repeat while PTR < N-K
   a) If DATA[PTR] > DATA[PTR+1] then
      Interchange value DATA[PTR] and DATA[PTR+1]
   b) Set PTR = PTR + 1
5. Exit

It was proved that [1] sorting requires $O \left( \frac{N}{S} \log \frac{M}{B} \frac{N}{S} \right)$ block transfers and permuting an array requires $O \left( \min \left( \frac{N}{S} \log \frac{M}{B} \frac{N}{S} \right) \right)$ block transfers where $O$ is number of elements to sort, $N$ is total number of lines, $M$ is number of words fitting in the main memory, and $B$ is number of words per disk block. Lower bounds hold for the cache-oblivious model. The lower bounds from [1] immediately give a lower bound of $\Omega \left( \frac{N}{S} \log \frac{M}{B} \frac{N}{S} \right)$ block transfers for cache-oblivious sorting. The upper bound from [1] cannot be applied to the cache-oblivious setting since these algorithm make explicit use of $B$ and $M$. 

S.R.T.M. University Nanded 131
We present an approach that uses external counter elements as shown in Algorithm 1. However, our sequential sorting, which totally avoids the transposition process, in Algorithm 1, we repeat recursion using external counter and process sorting.

```c
void sort()
{
    if(counter<len)
    {
        counter=counter+1;
        sort();
    }
    int x[100],y[100];
    int i,j,k;
    x[0]=a[cc];
    y[0]=a[cc+1];
    if(x[0]==y[0])
    {
        a[cc]=y[0];
        a[cc+1]=x[0];
    }
    if(cc==len)
    {
        cc=0;
    }
    cc=cc+1;
    newcc=newcc+1;
    if(cc==len)
    {
        cc=0;
    }
    cout<<" the value of counter is "<newcc;
}
```

*Algorithm 1: First Cache Oblivious Sequential Sorting*

The same approach applied in algorithm 2 with different execution.

The Sequential Cache Oblivious sort works in a way similar to the general sort, the difference being that it recursive execution of array into n times and therefore the process executes $n^2$ times instead of n. This similarity makes it reasonable to expect that it too exhibits good cache efficiency. Along with the fact that the algorithm works well in practice, we are convinced that it is well suited as a competitor to funnel sort. The same approach may be executed with different parameter as algorithm 2.
void sort()
{
    if(counter<newlen-1)
    {
        counter=counter+1;
        sort();
    }
    int x[100],y[100];
    int i,j,k;
    x[0]=a[cc];
    y[0]=a[cc+1];
    if(x[0]==y[0])
    {
        a[cc]=y[0];
        a[cc+1]=x[0];
    }
    if(cc==len)
    {
        cc=0;
    }
    cc=cc+1;
    newcc=newcc+1;
    if(cc==len)
    {
        cc=0;
    }
    cout<""in the value of counter is "<<newcc;
4. SEQUENTIAL SORT ANALYSIS

The number of comparison between elements and the number of exchange between elements determine the efficiency of Sequential Sort algorithm. Generally, the number of comparisons between elements in Sequential Sort can be stated as follows

\[(n-1) + (n-2) + \ldots + 2 + 1 = \frac{n(n-1)}{2} = O(n^2)\]

The \(i^{th}\) \((i \leq n - 1)\) pass performs \((n - i)\) comparisons and at most \((n - i)\) comparison. Hence, the pass takes \((n - i)\) time.

\[
f(n) = \sum_{i=1}^{n-1} O(n-i) = \sum_{i=1}^{n-1} O(i) = \sum_{i=1}^{n-1} i
\]

\[O(n(n-1)/2) = O(n^2).\]

To put this list in recursive order, and by putting successive pairs of elements in order, the smallest value in the list will be moved at the first. Thus using these techniques repetitively and reducing the length of the list to be sorted by one element (the smallest) each pass through, the complete list can be ordered. Thus to sort a list of \(n\) elements we firstly compare \((n-1)\) overlapping of elements, putting them in order, and repeat recursive process for the shortened list of \((n-1)\) elements. Thus we can repeat this process successively until the list to be sorted to only of length, and we are guaranteed that the original list has been completely sorted.
5. Recursive Cache Oblivious Sequential Sorting

The merge sort splits the list to be sorted into two equal halves, and places them in separate arrays. This sorting method is an example of the DIVIDE-AND-CONQUER paradigm i.e. it breaks the data into two halves and then sorts the two half data sets recursively, and finally merges them to obtain the complete sorted list. The merge sort is a comparison sort and has an algorithmic complexity of $O(n \log n)$. Elementary implementations of the merge sort make use of two arrays - one for each half of the data set. The following image depicts the complete procedure of merge sort as shown in fig 1.

\[ \begin{array}{c}
38 & 27 & 43 & 3 & 9 & 82 & 10 \\
38 & 27 & 43 & 3 \\
38 & 27 & 43 & 3 \\
38 & 27 & 43 & 3 \\
3 & 9 & 10 & 27 & 38 & 43 & 82 \\
9 & 82 & 10 \\
9 & 82 & 10 \\
9 & 82 & 10 \\
9 & 82 & 10 \\

\end{array} \]

Fig 1: Merge Sort Method

The algorithm for Merge sorting is presented in Algorithm 3
Algorithm 3: Merge Sort Algorithm

In the cache-oblivious setting the computational model is a machine with two levels of memory: cache of limited capacity and a secondary memory of infinite capacity. The capacity of the cache is assumed to be M elements and data is moved between the two levels of memory in blocks of B consecutive elements. Computations can only be performed on elements stored in cache, i.e. elements from secondary memory need to be moved to the cache before operations can access the elements. Programs are written as acting directly on one unbounded memory, i.e. programs are like standard RAM programs. The necessary block transfers between cache and secondary memory are handled automatically by the model, assuming an optimal offline cache replacement strategy. For the sorting problem the input is an array of N elements residing in secondary memory, and the output is required to be an array in secondary memory storing the input elements in sorted order. The sorting recursion repeat number of times using external counter elements. The recursion start and stops according to increment and decrement counter values. We design two different processes for recursive cache sequential sorting. In first process we use merge sorting using divide and conquer technique as shown in algorithm 4, 5, 6.
```csharp
void merge_sort()
{ if(counter <= (len)/2)
{ counter=counter+1;
  merge_sort(); }
  if(counter <= (len)/2)
  { counter=counter+1; }
  merge(); }
void merge()
{ if(ccc=4*len) {
  ccc=ccc+1;
  merge(); }
  clrscif);
  x[20],y[20],z[20],lowl,high1,i,tem,teml;
  lowl=cc;
  high1=cc+1;
  int t1,t2;
  t1=lowl;
  t2=high1;
  x[0]=a[t1];
  y[0]=a[t2];
  x[1]=9999;
  y[1]=9999;
  for(i=lowl;i<=high1+1;i++)
  { if(x[0]==y[0])
    { a[t1]=y[0];
      a[t2]=x[0]; }
    ccc=ccc+1;
    if(ccc=len)
    { ccc=0; }
    cout<<"in this is counter "<<ccc;
    for(i=0;i<=high1;++i)
    cout<<"in this is "<a[i].
    }
  }
void merge_sort()
{ if(counter<= (len)/2)
{ counter=counter+1;
  merge_sort(); }
  if(counter<= (len)/2)
  { counter=counter+1; }
  cout<<"the value of mun is "<<mun;
  mun=0;
  merge(); }
void merge()
{ if(mun<len)
{ mun=mun+1;
  merge(); }
  cout<<"the new length is "<<mun;
}
```

Algorithm 4: First Recursive Oblivious Sequential Sorting
Algorithm 5: Second Recursive Sequential Sorting

S.R.T.M. University Nanded
void merge_sort()
{
    if(count < (len)/2)
    {
        count = count + 1;
        merge_sort();
    } 
    else if(count > (len)/2)
    {
        count = count + 1;
    }
}

void merge()
{
    int x[20], y[20], z[20], lowl, highl, i, j, t1, t2, tem, tem1;
    lowl = cc;
    highl = cc + 1;
    t1 = lowl;
    t2 = highl;
    x[0] = a[t1];
    y[0] = a[t2];
    x[1] = 9999;
    y[1] = 9999;

    for(i = lowl; i <= highl + 1; i++)
    {
        if(x[0] >= y[0])
        {
            a[t1] = y[0];
            a[t2] = x[0];
        }
        cc = cc + 1;
        if(cc == len)
        {
            cc = 0;
        }
    }
    mc = mc + 1;
    for(i = 0; i <= highl; i++)
    {
        cout << "This is " << a[i];
    }
}

Algorithm 6: Third Recursive Cache Oblivious Sequential Sorting
In second process, we execute max and min values while processing merge sort techniques. The algorithm uses the divide and conquers technique to recognize that in a list of min and max elements, the algorithm reduces to

\[
\text{If (min _ element < first _ element) \{Swap position\}}
\]

It is shown in algorithm 7, 8, 9 respectively.
int mergesort(int arr[], int low, int high) {
    int mid;
    high = high - 1;
    if (low < high) {
        mergesort(arr, low, high);
        sort(arr, low, high);
    }
    int sort(int arr[], int L, int h) {
        int n1, n2, n3, k, loc, loca;
        for (k = 0; k < size; k++) {
            if (max >= arr[k] && arr[k] != 9999) {
                max = arr[k];
                loc = k;
            }
            if (min >= arr1[k] && arr1[k] != 0) {
                min = arr1[k];
                loca = k;
            }
        }
        ttt[cccc] = max;
        arr[loc] = 9999;
        ttt[cc] = min;
        arr1[loca] = 0;
        max = 0;
        min = 9999;
        cc = cc + 1;
        cccc = cccc - 1;
    }
    Algorithm 9: Sixth Recursive Cache Oblivious Sequential Sorting
6. Analysis of Recursive Cache Oblivious Sequential Algorithms

Merge sort works by first recursively sorting each half of the array, and then merging them.

Example \( A = \{50, 30, 40, 80, 70, 10, 90, 60\} \).

Divide \( A \) into two halves:

\[ A_1 = \{50, 30, 40, 80\}, A_2 = \{70, 10, 90, 60\} . \]

Sort each half recursively:

\[ A_1 = \{30, 40, 50, 80\}, A_2 = \{10, 60, 70, 90\} . \]

Merge them into the final sorted order:

\[ A \text{ merge } = \{10, 30, 40, 50, 60, 70, 80, 90\} . \]

Let \( f(n) \) be the (worst-case) time of sorting an array \( A \) of size \( n \). If \( n = 1 \), \( f(n) \) is obviously \( O(1) \). Next, we consider \( n > 1 \). For simplicity, let us assume that \( n \) is a power of 2.

- Dividing \( A \) into two halves \( A_1 \) and \( A_2 \) takes \( O(n) \) time.
- Sorting \( A_1 \) takes at most \( f(n/2) \) time.
- Sorting \( A_2 \) takes at most \( f(n/2) \) time.
- Merging the sorted \( A_1 \) and \( A_2 \) takes \( O(n) \) time.
Therefore:

\[ f(n) \leq 2f(n/2) + O(n) \]

Merge sort performs \( O(N \log_2 N) \) comparisons, but analyzed in the cache-oblivious model it performs \( O\left(\frac{N}{B} \log_2 \frac{N}{M}\right) \) block transfers which is a factor \( O\left(\log \frac{M}{B}\right) \) from the lower bound (assuming a recursive implementation of Merge sort, in order to get \( M \) in the denominator in the \( \log N/M \) part of the bound on the block transfers). Recursive Cache Oblivious Sequential Sorting algorithms achieve their relatively good performance for the number of block transfers from the fact that they are based on repeated scanning of arrays. As comparative Heap sort that has a very poor performance of \( O\left(\frac{N}{B} \log_2 \frac{N}{M}\right) \) block transfers.

7. IMPLEMENTATION

Cache Oblivious Sort Algorithm 1, 2, 4, 5, 6, 7, 8, 9 are simple scheme we developed in previous sections. The algorithm takes \( n-i \) phases as 1, 2 ---- \( n \) techniques to indicate the sorting process. In programming language like C or C++, Java directly used these schemes. In programming language C or C++ the counter we may design cache oblivious sequential sorting code.

However we compare algorithm 1, 2, 4, 5, 6, 7, 8, 9 with different array size and their execution speed shown in fig 1.
fig 1: Comparison of various length for sorting algorithms

We want to emphasize that as in fig 1 that is not the aim of method to produce the fastest algorithm for sorting on specific computer architecture. We want to demonstrate that it is possible to construct an algorithm where memory address copy only with stepwise one and it present nice properties that result from this approach.
8. REFERENCES


6] Piyush Kumar “Cache Oblivious Algorithms” 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/y MPI-Saarbrücken. NSF (CCR-9732220, CCR-0098172) Sandia National Labs.
9. SELF RESEARCH PAPER

1] A Research Paper "Cache Oblivious B-tree and Sorting using Sequential Accessing"
presented at Springer AIM International level conference held at Bangalore Dated 27-April 2012.