Research Paper Computer Science

Cache Oblivious Sorting Algorithm Using Sequential Processing

* Korde P. S. ** Khanale P. B

* Department of Computer Science Shri Shivaji College,Parbhani (M.S.) India

** Department of Computer Science Dnyopasak College Parbhani (M.S.) India

ABSTRACT

Algorithms that design which effectively utilized multi-layered memory hierarchies; it must rely on detailed knowledge of the characteristics of memory systems. The execution of memory has depends on its capacity of running to process job. This paper explores a simple and general approach for developing algorithmic study of cache oblivious sorting. We present three approaches for sorting. The algorithm uses array technique that is based on sorting methods. We present here algorithm that required minimum swapping in sorting.

Keywords : Cache Oblivious, Cache miss, Sorting, Sequential Processing Swapping

Introduction

In modern computers memory are organized memory levels in form hierarchies, with each level storing as a data as processing for the next. The memory have components as registers, Cache1 (level 1), Cache2 (level 2), now days Cache 3 (level 3). Each level is accessing required time when data have larger that gives for new level. The access of memory depends on current memory level which processing elements.

The concept of Cache oblivious gives by M.Frigo in 1999. The new data structure for cache oblivious for various techniques as matrix transposition, FFT, sorting [5], and also gave search trees [5] cache-aware B-trees [7]. It gives new idea and area for researcher for results cache oblivious model. Bender et al. [8] introduces new idea for dynamic search trees that process B-trees. It gives search trees with complexities matching presented in [6, 2] and variant with bounds in present format. These algorithms used for computational geometry [1], dynamic sets [7], static trees. It also gives priority queue [3] for designing graph algorithms.

Sorting technique is depending on processing of objects as searching in dictionary order like word, pattern matching, etc. The sorting algorithms like merge sort, quick sort, insertion sort have based on number of comparison when it processed. It gives complexity, which will have no knowledge of cache and its level size. It was introduced by I/O model [A. Aggarwal *et.al. 88*]. [Demaine *02*] Its relation to multilevel memory hierarchies are given in methods.

Various techniques can be found out for depth of cache oblivious sorting. In Arge et al[L. Arge *et,al 02*] developed a cache oblivious priority queue, the basic method is reduction. The author shows priority queue can be access as sequence of graph execution.

Bordal and Fagerberg in [G. S. Brodal *et,al 02*] showed how to modify cache obvious funnel sort algorithm to solve several problems with in computational geometry.

In this paper we give new idea for probably utilize algorithm, the sorting of array. We want to produce the sorting methods on general computer machines. Cache Oblivious Sorting Algorithm using Sequential Processing demonstrate that it is possible to design an algorithm where memory address copies stepwise one, which also eliminate the process for swapping. It will therefore give basic idea of the algorithm and present some of nice result from this approach.

Bubble Sort

A Bubble Sort is simple example of the application of the cache oblivious sorting. It is found in [M. Frigo *et.al. 99*], it was proved that Bubble Sort uses θ (n^2) comparisons. The general idea of Bubble sort is algorithm 1.

```
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
        [End of inner loop]
        [End of step 1 loop]
4. Exit
```

Algorithm 1 Bubble Sort

Depending upon the Programming language used, the elements of array required to execute is proportional to n^2 where n is the input items.

In this paper, we present an approach that uses an ordering of min elements array as shown in fig 1. However our first sequential sorting which totally avoid the transposition process.

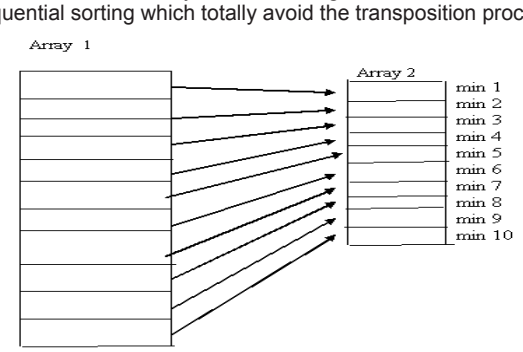

Fig 1: Cache Oblivious Sequential Processing Sorting

Cache Oblivious First Sequential Processing Sorting

First we reformulate algorithm 1 in the following form as fig 2

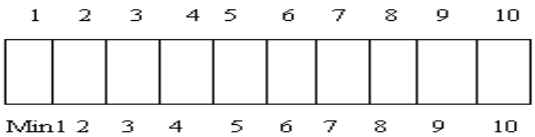

Fig 2 Array 2

In this process, we have used two arrays having same length. In this method we find minimum value at every phase store in second array list. Cache Oblivious Sequential Processing Sorting shows the locality of the element access. It shows better benefit and cache memory utilization. In Bubble sort method process execute form left to right according to permutation. It moves largest value by exchanging between right adjacent elements from lowest level. In this sorting user must complete n-1 passes. The outcome of every pass is one element is placed in correct place.

The total number of comparisons is obviously at most n^2, so it only needs to consider the lower bound. For a permutation π of elements 1, 2, -------- n. it describe the total number of exchanges by 1. For Bubble sort it shows in 1.

$$M = \sum_{i=1}^{n-1} \qquad \text{---------(1)}$$

In First sequential processing sorting we move minimum number from array and store into array 2. These processes repeat n times. In this method there are eliminating process of swapping and exchanging elements.

Cache Oblivious Second Sequential Processing Sorting
In Second process, we have used single arrays. In this method we sort only half part of array list as shown in fig 3. The remaining part of array list we processed further in next loop. It shows the better locality of the element access and can benefit from the presence of cache memory.

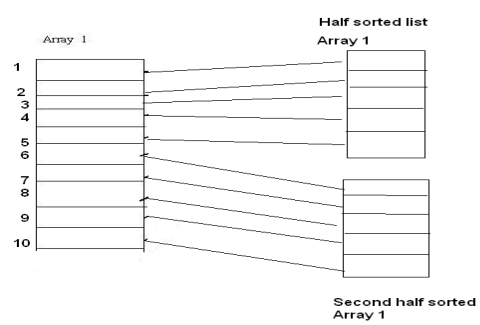

Fig 3: Cache Oblivious Second Sequential Processing Sorting

In Second process, we have used single arrays. In this method we sort only half part of array list as shown in fig 3. The remaining part of array list we processed further in next loop. It shows the better locality of the element access and can benefit from the presence of cache memory. In this sequential processing sorting we move minimum comparison from array to sort the list. These processes repeat n-i/2 times in first half, in second time it requires n/2-i time comparison. In this method there are eliminating process of swapping and exchanging elements.

Cache Oblivious Third Sequential Processing Sorting
In third event as shown in fig 4, we have used two arrays. In this process we start simultaneously sorting as ascending as half part in first array list and descending as second half part. Both process ascending and descending continuously run at same time in different array list. It shows the good locality of the element access and can benefit from the presence of cache memory.

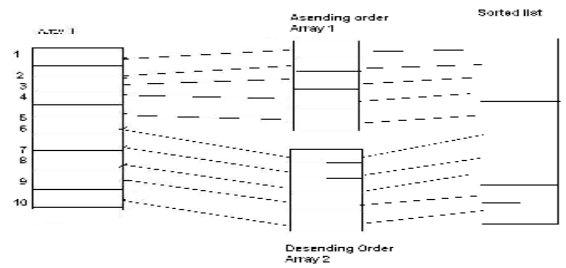

Fig 4: Cache Oblivious third Sequential Processing Sorting

In third sequential processing sorting we move same comparison from array to sort the list as bubble sort techniques. These processes repeat n-i/2 times in first half, in second time it requires n-i/2 time comparison. In this method there are dividing process of sorting and exchanging elements.

```
Algorithm 2
  1. Repeat steps 2,3,4 for i = 1 to n
  2. Repeat for j = 1 to n
     a) If( a[ j] < = min ) and ( a[ j ] < > 0 ) then
          i)    Set min =a [ j ]
          ii)   Set loc = j
                [ end of if ]
                [ end of j loop ]
  3. Set b[ i ] = min
  4. Set min = max value
     [ End of i loop ]
  5. Exit
```

In this process we formulate the Bubble sort method; here we used two arrays as array 1 and array 2. In first array we find the location of min value and place as null value and copy value in array 2. The process may continue up to last elements. Every phase we must store min value as max value. It is compulsory for every external loop.

We process n passes, since after all one element in the correct place the last remaining element must be also in its correct place. The total number comparisons are at most n^2. By using same technique we may design algorithm 3 Cache oblivious Sequential Sorting for descending.

```
Algorithm 3 ( Sequential Processing for Descending )
  1. Repeat steps 2,3,4 for i = 1 to n
  2. Repeat for j = 1 to n
     a) If( a[ j] > = min ) and ( a[ j ] < > 0 ) then
          i)    Set min =a [ j ]
          ii)   Set loc = j
                [ end of if ]
                [ end of j loop ]
  3. Set b[ i ] = min
  4. Set min = max value
     [ End of i loop ]
  5. Exit
```

In second sequential processing we process only half part sorting the value array while processing and reformulate the algorithm 1 as algorithm 4.

```
Algorithm 4 (Second Sequential Processing for Sorting )
  1. Repeat steps 2  for i = 1 to n/2-i
  2. Repeat for j = 1 to n
     a) If( a[ j] < = a[ j+1 ] ) then
          i)    t = a[ j ]
          ii)   a[ j ] = a[ j+1 ]
          iii)  a[ j+1]=t
                [ end of if ]
                [ end of j loop ]
  3. Repeat for j = 1 to n/2-i
     b) If( a[ j] < = a[ j+1 ] ) then
          iv)   t = a[ j ]
          v)    a[ j ] = a[ j+1 ]
          vi)   a[ j+1]=t
                [ end of if ]
                [ end of j loop ]
  4. Exit
```

We used single arrays as array 1. In this array we compare

the elements and place as largest or smallest element of value in same array. The second phases we continually process continue in another loop having n-2-I time comparisons up to last elements. Every phase we must store value. It is compulsory for every external loop.

We make at most n passes, since after moving all but one element in the correct place the single remaining element must be also in its correct place. The total number of exchanges is obviously at most n^2-i so we only need to consider the lower bound. By using same technique we may design algorithm Cache oblivious Sequential Sorting for descending.

In third sequential processing, simultaneously start ascending and descending process in different array list as first and second half list. Then we get perfect sorted list after combination. Only half part sorting the value array while processing and reformulate the algorithm 1 as algorithm 5.

```
Algorithm 5 (Third Sequential Processing for Sorting)
1.  Repeat steps 2,3   for i = 1 to n/2
2.  Repeat for j = 1 to  n
        i)      If( a[j] <= a[j+1 ] ) then
        ii)     t = a[ j ]
        iii)    a[ j ] = a[ j+1 ]
        iv)     a[j+1]=t
                [ end of if ]
                [ end of j loop ]
3.  Repeat for k = 1 to  n
        v)      If( a[ k] >= a[kj+1 ] ) then
        vi)     t1 = a[ k ]
        vii)    a[ k ] = a[k+1 ]
        viii)   a[[k+1]=t
                [ end of if ]
                [ end of k loop ]
                [end of i loop ]
4.  Exit
```

We used single arrays as array. In first array we compare the elements and place as ascending order manner, smallest element of value in same array. The second array at same we continually process descending order manner, largest element of value in another array. Every phase we must store value. It is compulsory for every external loop.

We make at most n passes, since after moving all but one element in the correct place the single remaining element must be also in its correct place. The total number of exchanges is obviously at most n^2-i/2 for every half loop so we only need to consider the lower bound. By using same technique we may design algorithm Cache oblivious Sequential Sorting for descending.

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

$$(n-1)+(n-2)+\ldots\ldots+2+1 = n(n-1)/2 = O(n^2)$$

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

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

$$O(n(n-1)/2) = O(n^2).$$

It was proved that [1] sorting requires $O\left(\frac{N}{B}\log_{M/B}\frac{N}{B}\right)$

block transfers and permuting an array requires

$$O\left(\min\left\{\frac{N}{B}\log_{M/B}\frac{N}{B}\right\}\right)$$

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

$$\Omega\left(\frac{N}{B}\log_{M/B}\frac{N}{B}\right)$$

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.

Implementation

Algorithm 2,3,4,5 are simple scheme we developed in previous sections. The algorithm takes n-i phases as 1, 2 ----- 5 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.

How ever we compare algorithm 1,2,3,4,5 with different array size and their execution speed shown in table 1.

Sr. no.	Array Size	Bubble Sort	First Sequential Sort	Second Sequential Sort	Third Sequential Sort
01	10	0.164835	0.164835	0.164835	0.164835
02	100	0.274725	0.274725	0.274725	0.274725
03	500	0.659341	0.659341	0.659341	0.659341

It may convert in graphical format as shown in fig 3.

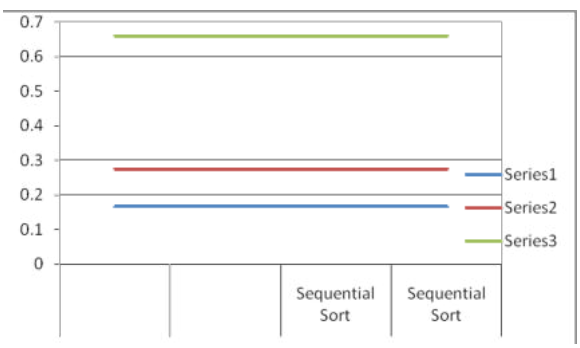

Fig 3: Time required for Bubble sort and Sequential Sort.

We want to emphasize that as in fig 3 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.

Conclusion

We have presented three sequential processing for sorting algorithm which shows better locality features. Using ideal cache model cache misses is of order of $O\left(\frac{N}{B}\log_{M/B}\frac{N}{B}\right)$ levels of data for cache. It is optimal for any algorithm that is based on sorting. Our methods give minimum swapping for address arithmetic. While this fact is not standard environment, it may be considerable advantage for implementations of sorting techniques.

REFERENCES

] A. Aggarwal and J. S. Vitter. 1988, The input/output complexity of sorting and related problems. Communications Proceedings of the ACM, 31(9):1116–1127 www.cc.gatech.edu/~bader/COURSES/UNM/ece637-Fall2003/.../AV88.pdf | 2] Demaine, E.D 2002, Cache-oblivious algorithms and data structures. Proceedings of Lecture Notes from the EEF Summer School on Massive Data Sets. , BRICS, University of Aarhus, Denmark erikdemaine.org/papers/BRICS2002/paper.pdf | 3] G. S. Brodal and R. Fagerberg,2002 Cache oblivious distribution sweeping. In Proc. 29th International Colloquium on Automata, Languages, and Programming, volume 2380 of Lecture Notes in Computer Science, Springer Verlag, Berlin. pages 426–438. www.mpi-inf.mpg.de/...of.../Brodal-Fagerberg-DistributionSweeping.pdf | 4] L. Arge, M. A. Bender, E. D. Demaine, B. Holland- Minkley, and J. I. Munro,2002 -Cache-oblivious priority queue and graph algorithm applications?, In ACM, editor, Proceedings of the 34th Annual ACM Symposium on Theory of Computing (STOC '02), ACM Press, pages 268– 276.www.cc.gatech.edu/~bader/COURSES/UNM/ece637.../ABD02b.pdf | 5] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran.1999 Cache-oblivious algorithms. In Proc. 40th Annual Symposium on Foundations of Computer Science, IEEE Computer Society Press. pages 285 297.www.cc.gatech.edu/~bader/COURSES/GATECH/CSE6140.../FLP99.pd | 6] Piyush Kumar 2009 "Cache Oblivious Algorithms" Department of Computer Science State University of New York at Stony Brook Stony Brook, NY 11790, USA com/co-chap/y MPI-Saarbr ucken. NSF (CCR-9732220, CCR-0098172) Sandia National Labs. | piyush@acm.org http://www.compgeom. www.informatik.uni-trier.de/~ley/pers/hd/k/Kumar:Piyush | 7] R. Bayer and E. McCreight. 1972 Organization and maintenance of large ordered indexes. Acta Informatica, 1:173–189 .link.springer.com/content/pdf/10.1007/978-3-642-59412-0_15.pdf | 8] M. A. Bender, E. Demaine, and M. Farach-Colton.2000 Cache-oblivious B-trees. In Proc. 41st Ann. Symp.on Foundations of Computer Science, IEEE Computer Society Press. pages 399–409. erikdemaine.org/papers/CacheObliviousBTrees_SICOMP/paper.pdf | |