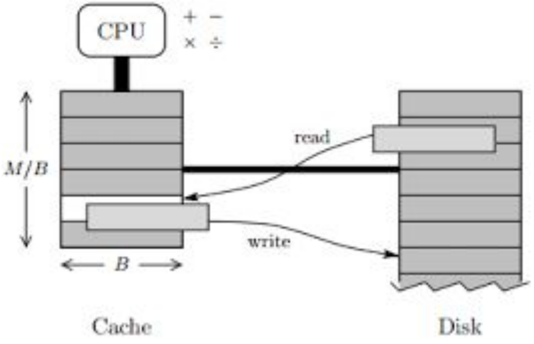
Cache Oblivious Algorithms

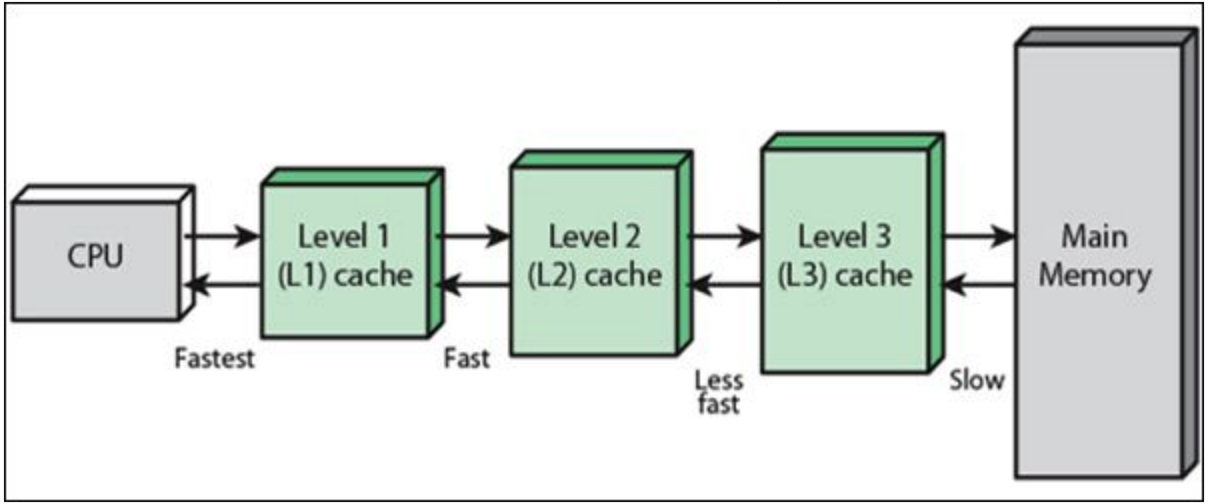
By:Tarun Gupta (180001059)
Kartik Garg (180002027)

2 level memory hierarchy model



- Basic Terminology -
  - M: Total Cache size.
  - B: Size of one block
- To compute something, data required for that computation must be transferred to cache memory from Disk.
- Transfers happen in blocks of B elements.
- The cache has total size M, stored in M/B blocks, each of size B.
- It’s not possible to divide a block in further into more pieces. Analogous to a ‘quantum’ in quantum mechanical theory.

Actual hierarchical memory model

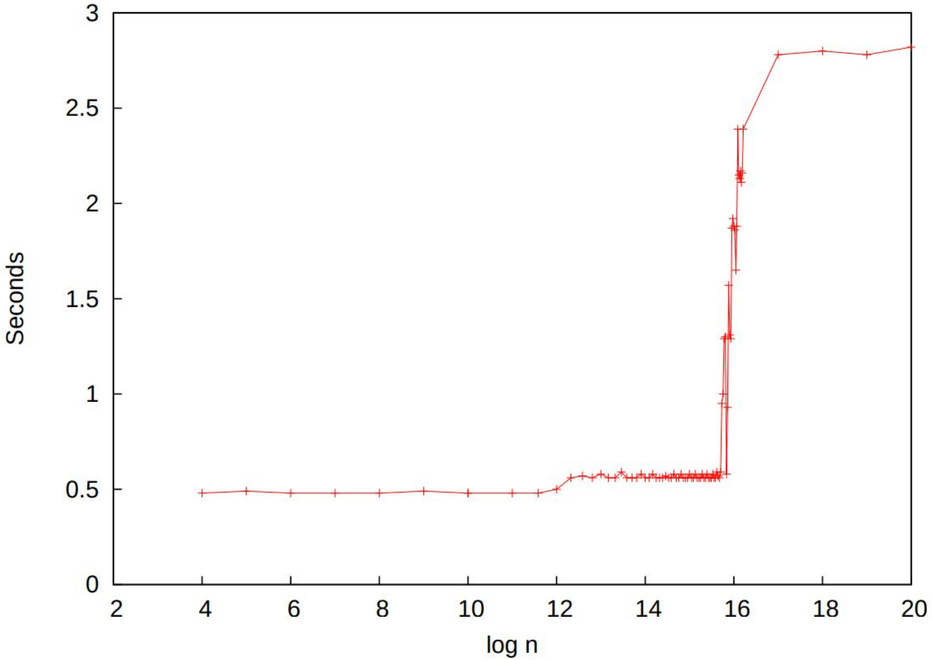


- As the distance (level) of cache increases, the speed of memory transfers increases by several orders of magnitude.
- Every level interface can have different parameters M and B.

Problem with existing algorithms and traditional time complexity analysis.

- In traditional algorithms we never take into consideration memory transfers while designing an algorithm.
- Memory transfer is very expensive!
- As the distance (level) of cache increases, the cache speed gets slower and hence computation time increases significantly.

Sudden increase in time of general algorithms:

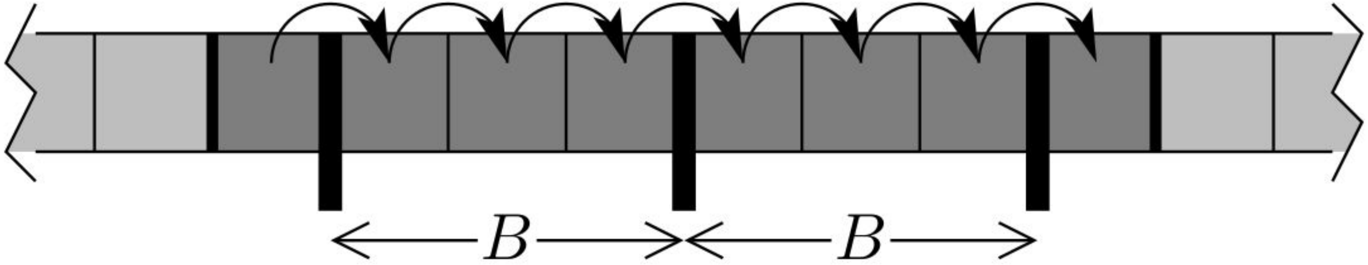


Is there a solution?

- We can explicitly write and manage our data in different levels of cache.
- These are known as **Cache aware algorithms**.
- Problems with Cache aware algorithms:
    - This requires separate software other than a normal programming language and lot of expertise of deal with.
    - Transferability of code decreases. Code needs to be customised for each system.
    - Lot of Bugs.

Better Solution: Cache Oblivious Algorithms

- A cache-oblivious algorithm is an algorithm designed to work efficiently for all cache levels without knowing M and B.
- We can gain same level of performance as cache aware algorithms!



Simple array traversal:
Memory Transfers = O(N/B)

Design of some cache oblivious algorithms

Code available at:

https://github.com/tarun360/Cache-Oblivious-Algorithms

Matrix Multiplication

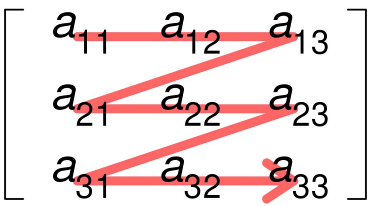
Problem: C=A·B, $c_{ij}=a_{ik} \cdot b_{kj}$

Simplest algorithm considering row major ordering is not cache-efficient.

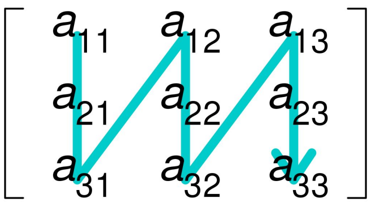
Layouts of matrices

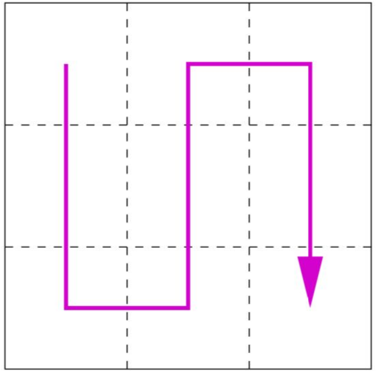
So, we will be using peano indexing to store the matrix

Row-major order

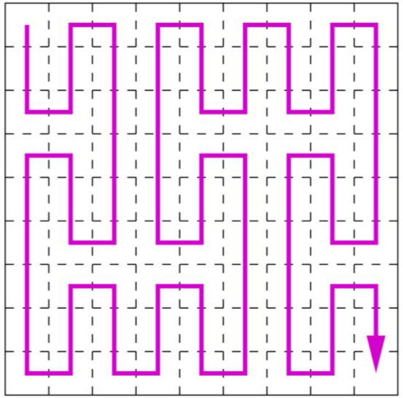


Column-major order





Peano Curve for 3X3 matrix

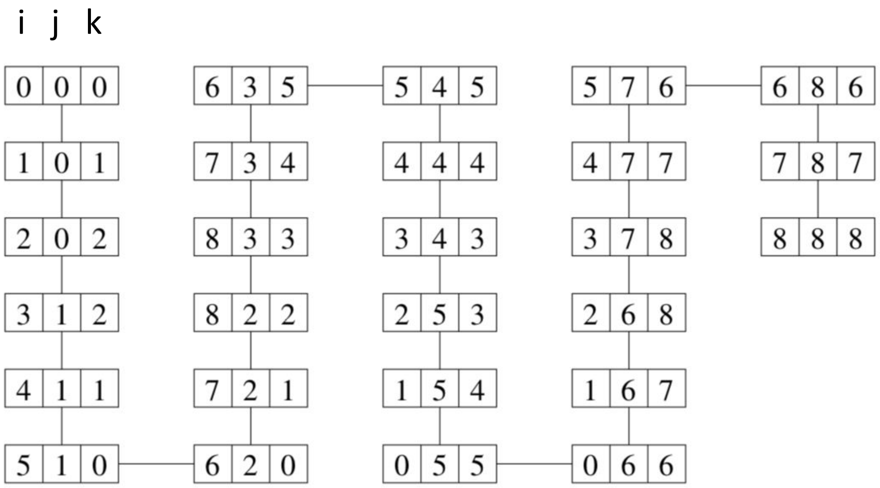


Recursive Construction of Peano Curve using that of 3X3 matrix

Matrix Multiplication on 3X3 matrix

Algorithm

- For all triples (i, j, k), difference between indexes of any components, in adjacent nodes is not greater than 1.
- Helps in finding optimal serialization.
- We can directly reuse an element or move to its neighbour after each operations, thus decreasing memory transfers.



Multiplication scheme in 3x3 matrix

Matrix Multiplication

Recursive implementation of algorithm. Here phsA, phsB, phsC takes values of 1 or -1 and helps in changing indices in (a, b, c) triples as discussed in multiplication scheme

```
/* global variables:
 * A, B, C: the matrices, C will hold the result of AB
 * a, b, c: indices of the matrix element of A, B, and C
 */
peanomult(int phsA, int phsB, int phsC, int dim)
{
   if (dim == 1) {
      C[c] += A[a] * B[b];
   }
   else
   {
      peanomult( phsA,  phsB,  phsC, dim/3); a += phsA; c += phsC;
      peanomult( phsA, -phsB,  phsC, dim/3); a += phsA; c += phsC;
      peanomult( phsA,  phsB,  phsC, dim/3); a += phsA; b += phsB;

      peanomult( phsA,  phsB, -phsC, dim/3); a += phsA; c -= phsC;
      peanomult( phsA, -phsB, -phsC, dim/3); a += phsA; c -= phsC;
      peanomult( phsA,  phsB, -phsC, dim/3); a += phsA; b += phsB;

      peanomult( phsA,  phsB,  phsC, dim/3); a += phsA; c += phsC;
      peanomult( phsA, -phsB,  phsC, dim/3); a += phsA; c += phsC;
      peanomult( phsA,  phsB,  phsC, dim/3); b += phsB; c += phsC;

      peanomult( phsA,  phsB,  phsC, dim/3); a -= phsA; c += phsC;
      peanomult( phsA, -phsB,  phsC, dim/3); a -= phsA; c += phsC;
      peanomult( phsA,  phsB,  phsC, dim/3); a -= phsA; b += phsB;

      peanomult( phsA,  phsB, -phsC, dim/3); a -= phsA; c -= phsC;
      peanomult( phsA, -phsB, -phsC, dim/3); a -= phsA; c -= phsC;
      peanomult( phsA,  phsB, -phsC, dim/3); a -= phsA; b += phsB;

      peanomult( phsA,  phsB,  phsC, dim/3); a -= phsA; c += phsC;
      peanomult( phsA, -phsB,  phsC, dim/3); a -= phsA; c += phsC;
      peanomult( phsA,  phsB,  phsC, dim/3); b += phsB; c += phsC;

      peanomult( phsA,  phsB,  phsC, dim/3); a += phsA; c += phsC;
      peanomult( phsA, -phsB,  phsC, dim/3); a += phsA; c += phsC;
      peanomult( phsA,  phsB,  phsC, dim/3); a += phsA; b += phsB;

      peanomult( phsA,  phsB, -phsC, dim/3); a += phsA; c -= phsC;
      peanomult( phsA, -phsB, -phsC, dim/3); a += phsA; c -= phsC;
      peanomult( phsA,  phsB, -phsC, dim/3); a += phsA; b += phsB;

      peanomult( phsA,  phsB,  phsC, dim/3); a += phsA; c += phsC;
      peanomult( phsA, -phsB,  phsC, dim/3); a += phsA; c += phsC;
      peanomult( phsA,  phsB,  phsC, dim/3);
   };
}
```

Matrix Multiplication

For Cache-Ignorant Matrix Multiplication,
Memory Transfers = $O(N^3)$

For Cache-Oblivious Matrix Multiplication,
Memory Transfers = $O(N^3/B\sqrt{M})$

Matrix Transposition

Problem: Transpose given matrix A of order nxm to an matrix B of order mxn such that

$$A_{ij} = B_{ji} \quad \forall i \in [1 \cdots m], j \in [1 \cdots n]$$

Cache-Ignorant Matrix Transposition

```
for (i = 1; i < n; i++){
        for (j = 0; j < i; j++){
                tmp = A[j][i];
                A[i][j]=A[j][i];
                A[j][i]=tmp;
        }
}
```

0	1	3	5	7	9	11	13
2	0	15	17	19	21	23	25
4	16	0	27	29	31	33	35
6	18	28	0	37	39	41	43
8	20	30	38	0	45	47	49
10	22	32	40	46	0	51	53
12	24	34	42	48	52	0	55
14	26	36	44	50	54	56	0

Typical access pattern for cache-ignorant algorithm

- Inner loop runs n(n-1)/2 number of times.
- No special care is made to use cache efficiently
- Memory Transfers: $\theta(n^2/B)$

Matrix Transposition

Cache-Oblivious Matrix Transposition

Basic Idea: Use a recursive approach that repeatedly divides the data set until it eventually become cache resident, and therefore cache optimal.

Matrix *A* into matrix *B* the largest dimension of the matrix is identified and split into two, creating two sub-matrices. Thus if $n \geq m$ the matrices are partitioned as:

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

This process continues recursively until individual elements of *A* and *B* are obtained at which point they are swapped.

Here, also: Memory Transfers: $\theta(n^2/B)$

0	2	4	6	14	16	22	24
1	0	8	10	18	20	26	28
3	7	0	12	30	32	38	40
5	9	11	0	34	36	42	44
13	17	29	33	0	46	48	50
15	19	31	35	45	0	52	54
21	25	37	41	47	51	0	56
23	27	39	43	49	53	55	0

Typical access pattern for cache-oblivious algorithm

Matrix Transposition

Comparison between Cache-Ignorant and Cache-Oblivious Matrix Transposition, through cache-miss rates

Dimension of matrix, N

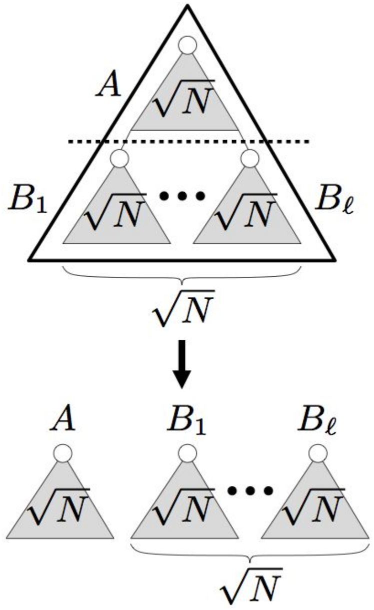
Algorithm	1024	2048	4096	8192
Cache-Ignorant	589795	2362002	9453724	37826712
Cache-Oblivious	131226	923295	7101600	56158873

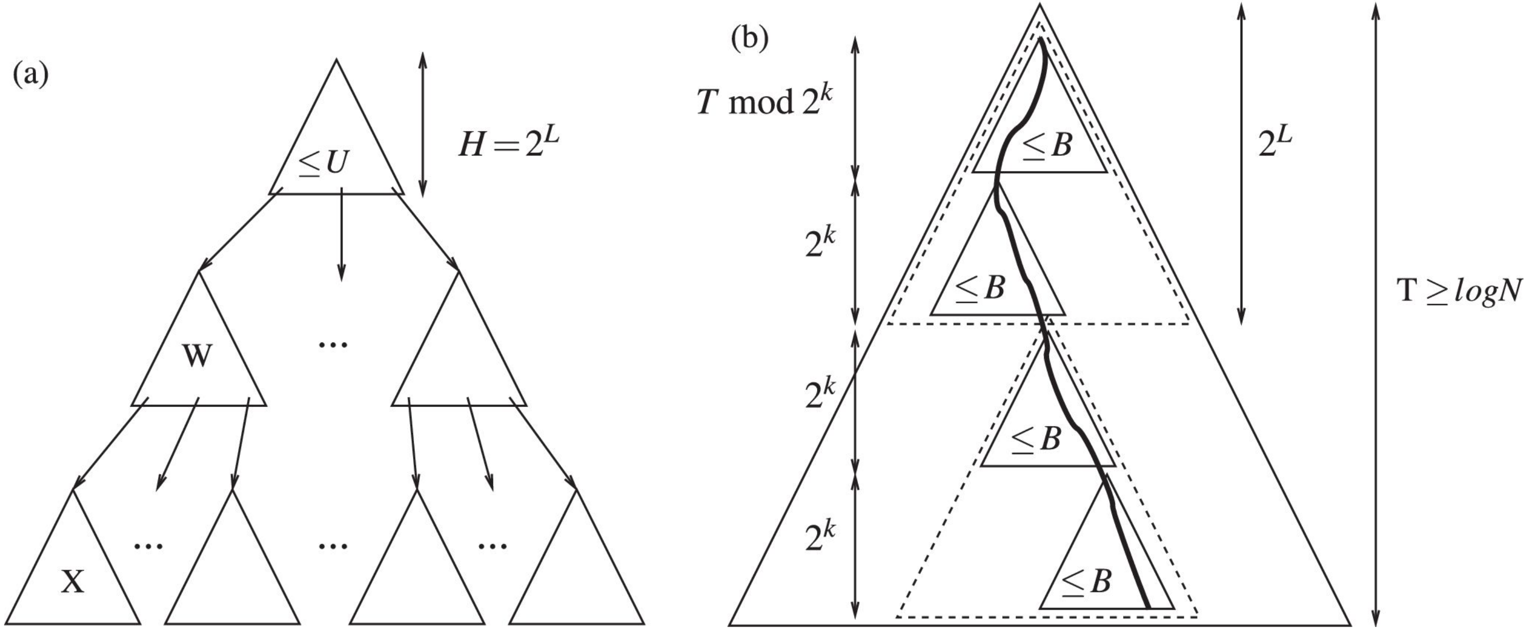
Data is calculated through Valgrind software.
Clearly, Cache-Oblivious algorithm, in this case, is performing worse than Cache-Ignorant algo.
Poor performance for N=4096 and 8194 might be due the fact that for both these dimensions, one row of the matrix is an exact multiple of the cache size.

Searching algorithm

Linear Search? - To slow.
Binary Search? - To many memory transfers.
Solution: Van Emde Boas Static Tree Layout: A strange recursive layout.

- Layout the elements in BST format.
- Conceptually split the tree at the middle level of edges, resulting in one top recursive subtree and roughly √N bottom recursive subtrees, each of size roughly √N.
- Recursively lay out the top recursive subtree, followed by each of the bottom recursive subtrees.
- The triangles are stored linearly, one after the another as shown in bottom right.





- Right picture shows a typical BST traversal done on Van Emde Boas tree.
- When the size of subtrees <= B, it fits completely in cache. Therefore, no memory transfers are required to traverse through this subtree of height log(B).
- Therefore, total number of memory transfers is O(log(N)/log(B))

Sorting algorithm: Funnelsort

- Basically a $N^{1/3}$-way merge sort.
- For now, we treat a K-funnel as a black box that merges K sorted lists of total size K^3.
- Tall-cache assumption: We assume that $M = \Omega(B^2)$.

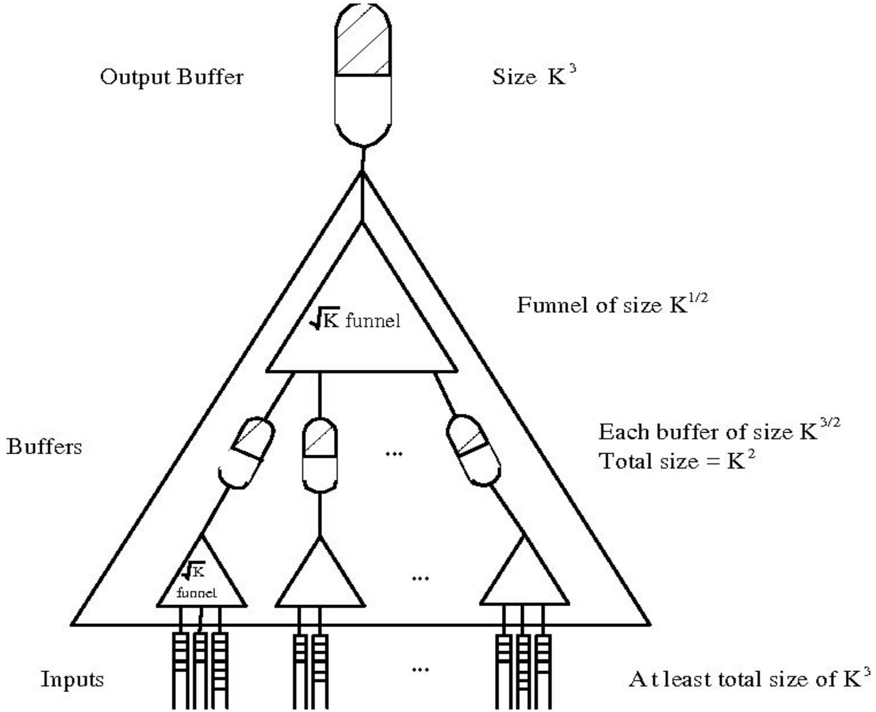
ALGORITHM:

- Split the array into $K = N^{1/3}$ contiguous segments each of size $N/K = N^{2/3}$.
- Recursively sort each segment.
- Apply the K-funnel to merge the sorted segments.

Crux of Algorithm: Efficiently merging $N^{2/3}$ sorted arrays, each of size $N^{1/3}$.

- This can be done efficiently using a K-funnel data-structure.
- Layout it same as Van Emde Boas Tree.
- A K funnel takes K sorted arrays and merges them.
- Memory Transfers =

$$O\left(\frac{K^3}{B}\log_{M/B}\frac{K^3}{B}+K\right)$$



Conclusion And Future Plans

As we have seen, cache oblivious algorithms really do work well in most of the cases.

1. Cache Oblivious algorithms significantly improve running time of algorithms.
2. Implementation and analysis of more algorithms and data structures in cache-oblivious manner.
3. Computation and analysis cache hit-miss ratio of different algorithms and comparing them with their cache-ignorant counterparts

References

- Matteo Frigo, Charles E Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. *ACM Transactions on Algorithms (TALG)*, 8(1):1–22, 2012.
- Gerth Stølting Brodal, Rolf Fagerberg, and Kristoffer Vinther. 2008. Engineering a cache-oblivious sorting algorithm. J. Exp. Algorithmics 12, Article 2.2 (2008).
- M. Blum, Robert W. Floyd, Vaughan Pratt, Ronald L. Rivest, AND Robert E. Tarjan. Time Bounds of Selection. Journal of computer and system sciences 7.
- Guy E Blelloch, Phillip B Gibbons, and Harsha Vardhan Simhadri. Low depth cache-oblivious algorithms. In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*, pp. 189–199, 2010.
- M. Bader, Ch. Zenger, A cache oblivious algorithm for matrix multiplication based on Peano’s space filling curve, in: Proceedings of the 6th International Conference on Parallel Processing and Applied Mathematics, PPAM, in press.