Report Seminar
Algorithm Engineering

G. S. Brodal, R. Fagerberg, K. Vinther: "Engineering a Cache-Oblivious Sorting Algorithm"

Iftikhar Ahmad

Chair of Algorithm and Complexity
Department of Computer Science
Albert Lüdwig University Freiburg
razmian_1167@yahoo.com

1 Introduction

In this chapter the need for the cache oblivious algorithm is described which is followed by the working of two of the cache oblivious sorting algorithms- Funnel Sort and Lazy Funnel Sort. The drawbacks of Funnel Sort are discussed and how these are overcome by Lazy Funnel Sort are described. At the end the problem statement of the research paper is highlighted.

1.1 Algorithm Analysis:

One of the most important considerations in the design of the algorithm is the efficiency. There are three general ways to measure the efficiency of algorithms. These include experimental analysis, average case and worst case analysis. The last two approaches are purely theoretical and do not consider the real world scenarios and does not consider factors that may affect the running time of algorithm. But the advantage of theoretical analysis is that it analyzes the algorithm independent of the hardware architecture on which the algorithm will be implemented, thus providing an insight into running time of algorithm independent of the hardware architecture. Hence it becomes really difficult to rely on single model for algorithm complexity. So we have to rely on experimental results as well to gauge the effectiveness of the algorithm in real world scenario.

When we consider the actual running time of an algorithm, then the memory system of computer has significant effect on the running time. It means the memory access time plays an important role in overall running time of algorithm, the latency associated with transfer of data from higher level to lower level, for example from Hard Disk to RAM or from RAM to Cache. In order for the algorithm to perform better in real world scenario it has to minimize the latency involved. Though the CPU speed is increasing following the $Moore's Law$ (The number of transistor on single chip doubles every eighteen months) , the same performance increase in the RAM is not visible. As per a study [1] the increase in speed of CPU and RAM is 55% and 7% respectively per year and this results in wide CPU-RAM performance gap. The figure below illustrates the gap more illustratively:

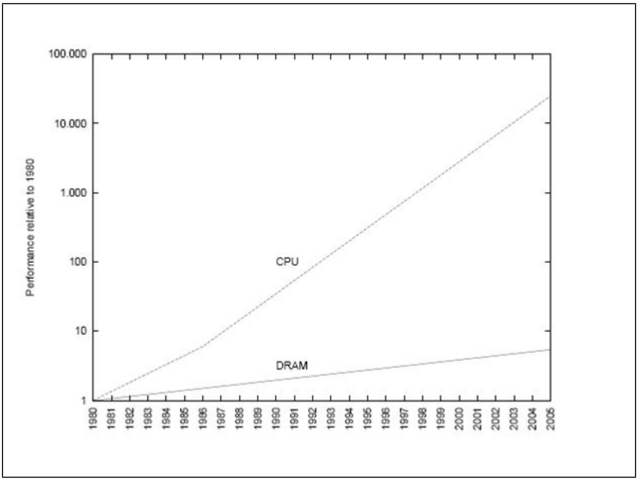


Figure 1: CPU-RAM Performance Gap

For more precise analysis of algorithms, the computational models must consider the memory architecture and factors such as hierarchy of memory, latency etc. We will discuss the model later but before that we need to understand the memory hierarchy of simple computer.

1.2 Ideal Cache Model

Let us consider a simple computer with two level of memory hierarchy as shown in the figure 2 below:

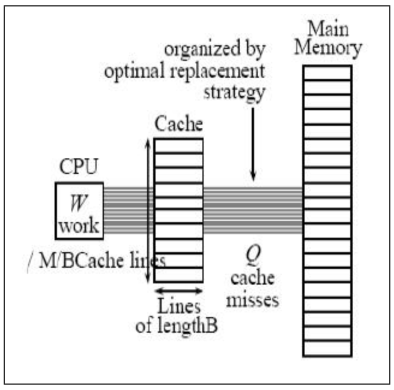


Figure 2: A Simple Computer with Two Level of Memory Hierarchy

We assume that Main Memory (RAM) is of arbitrarily large size, cache is partitioned into small blocks called *CacheLine* and each cache line consists of B

consecutive locations. Data from main memory to cache is transferred in form of blocks such that size of each block is B. The total size of cache is M so there can be a total of M/B cache lines. We work on assumption of tall cache, which means that $M \geq B^2$. A cache hit occurs whenever data referenced by CPU is present in the cache and if the data is not present in cache and it has to be fetched from main memory then it is referenced as cache miss. We also assume that there is an optimal offline strategy for cache line replacement.

As for as the complexity in ideal cache model is concerned, there are two approaches:

– Work Complexity, W(n), which is conventional running time in RAM
– Cache Complexity, Q(n, M, B), the number of cache misses as function of M and B.

where

n = input size

1.3 Cache Aware Vs Cache Oblivious Algorithms

Cache Aware and Cache Oblivious Algorithms are two different attempts that have been made to reduce the gap between actual running and theoretical running time of algorithms. The cache aware algorithms are based on parameters that can be tuned to optimize the cache complexity, these algorithms have prior knowledge about the size of cache and block etc and in practice are designed to for two known level of memory hierarchy. This makes them sub optimal for other memory levels and they often do not perform well.

In contrast to Cache Aware algorithms, Cache Oblivious algorithms have no prior information about the hardware parameters like cache size etc.

Advantages of Cache Oblivious Algorithms:

– The algorithm performs significantly better even without having any prior information about characteristics of memory hierarchy.
– There is no need to hardwire the memory characteristics in algorithm.
– Optimizing algorithm for one unknown level optimizes it for all levels.
– Cache Oblivious Algorithms are more robust and portable and thus are useful for development of software libraries.

Design Technique for Cache Oblivious Algorithm: One of the main techniques used to design cache oblivious algorithm is Divide and Conquer [2]. As like in traditional divide and conquer approach, the problem is divided repeatedly into small problems till a base case is achieved. In cache oblivious algorithms base case is reached when the sub problem is easily solvable, that is the sub problem should not cause cache miss and should fit in cache.

1.4 Cache Oblivious Sorting Algorithms:

Sorting algorithm is one of the most comprehensively researched topic in classical computer science. Sorting is being used in many of the graph, geometric and scientific applications as sub-routine. This means that efficient sorting algorithms will obviously increase the efficiency of the whole application. There are several Cache Oblivious Sorting algorithms presented; most notable of them are Funnel Sort[3], Lazy Funnel Sort[4] and Distribution based sorting algorithm[3].

Funnel Sort: Funnel Sort was the first cache oblivious sorting algorithm presented by Frigo et al[3]. The asymptotic working complexity of funnel sort is $O(nlogn)$ where as if tall cache assumption holds then cache complexity is O(N/B $\log_{M/B}$N/B). The K-Funnel form the basis for the Funnel Sort.

A K-Funnel merges K^3 elements at single invocation and consists of $\sqrt{K}$ funnels at bottom; each bottom funnel is connected to top funnel via $\sqrt{K}$ buffers. Figure 3 shows a 16-funnel with 16 input streams.

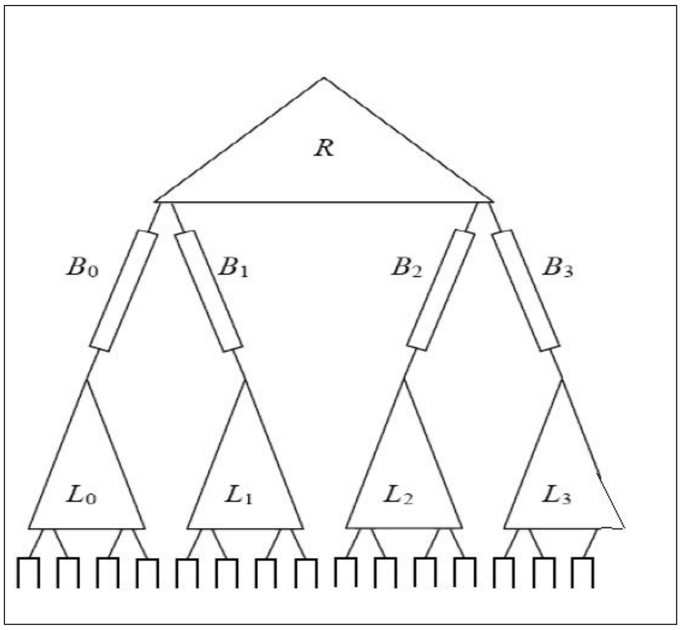


Figure 3: A 16-funnel with 16 input streams

Funnel Sort is laid in memory via van Emde Boas (vEB) layout; it means that first top tree R is laid out in memory followed recursively by B_0, L_0, B_1 and so on. The buffer size (B_0, B_1, B_2, B_3) is double than that of the input streams to accommodate the output arising from there. For initial invocation there must be K^2 elements in input buffer but as the merging progresses there can be lesser and lesser elements. If at some instance there are not sufficient input elements to satisfy the K^3 condition then it simply outputs what is there.

Complexity of Funnel Sort:
The cache complexity of funnel sort is O(N/B $\log_{M/B}$N/B)
Where
N = Input size
M = Size of Cache
B = Size of Block (line)
N/B = Total number of blocks transferred from higher level memory to lower level, so we have N/B blocks in cache to sort, similarly N/B can also be the maximum number of cache misses.

Drawbacks of Funnel Sort:
Some of the drawbacks of Funnel Sort include:

– In practice it is not always possible to split K-Funnel into $\sqrt{K}$ bottom funnels, it may lead to rounding errors.
– The flow of data is not efficient as the both the buffers must be filled before a merge occurs (except for the last when there is no more input stream available).
– vEB layout performs well for binary trees but does not perform well for complex data structures.

Lazy Funnel Sort: In an attempt to eliminate the drawbacks of Funnel Sort, a variant of Funnel Sort called Lazy Funnel sort was introduced[4]. Lazy Funnel sort is based on binary mergers which takes two input streams as input and deliver the output as merged sorted stream. Invocation of merger occurs through a recursive procedure which ensures that either the output buffer is full or both input streams are exhausted. The algorithm is given below:

```
ProcedureFill(v)
   while out-buffer not full
      if left in-buffer empty
           Fill(left child)
      if right in-buffer empty
           Fill(right child)
   perform one merge step
```

The procedure Fill() checks the emptiness of buffers rather than the fullness as in the case of Funnel Sort, and perform one merge step afterword.

In Lazy Funnel sort, K-merger is the basic data structure used. A K-merger is perfect binary tree with $K - 1$ binary mergers and output buffer at the root of size K^d, where d≥1. The following figure shows a 16-merger with 15 binary mergers.

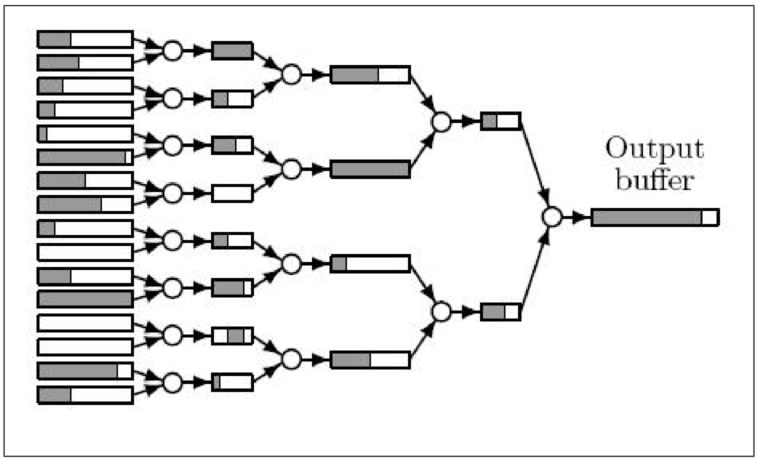


Figure 4: 16-Merger with 15 binary mergers.

For sorting N elements, lazy funnel sort recursively sort N$^{1/d}$ segments each of size N$^{1-1/d}$. N$^{1/d}$ mergers are used for merging process.

The lazy Funnel Sort removes the drawbacks of Funnel Sort, for example the need to have $\sqrt{K}$ bottom funnels is replaced by binary mergers, this also provide an easy and efficient implementation using vEB layout which is best in representing binary trees.

1.5 Problem Statement

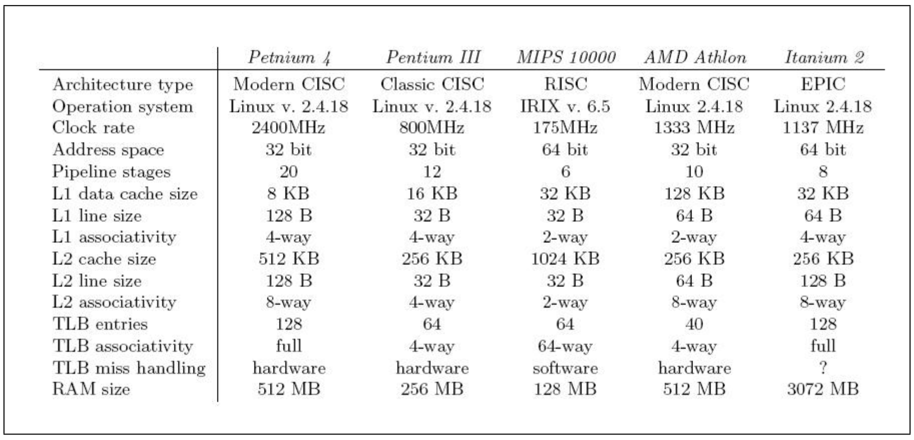
To explore a number of implementation issues and parameter choices for cache oblivious sorting algorithm and settle the best choices through experiments.

2 Engineering the Algorithm

In this chapter, the choice for the algorithm is made, then it is discussed how the said algorithm can be optimized by tuning different parameter choices. First it is discussed which algorithm is best choice for tuning and then steps for the tuning are discussed.

2.1 Methodology:

First the concentration will be on selecting an optimal algorithm for cache oblivious sorting. After the selection of algorithm, the fine tuning of the algorithm will be conducted by varying different parameter values and examining its running time. The best parameter choices will then be used to construct the most optimal sorting algorithm. After the construction of most optimal cache oblivious sorting algorithm, it will be tested against a number of present day algorithms and the running time will be observed on different hardware architectures. C++ was used as the delvelopment language. Figure 5 provides details of different hardware architectures used.



	Petnium 4	*Pentium III*	*MIPS 10000*	*AMD Athlon*	*Itanium 2*
Architecture type	Modern CISC	Classic CISC	RISC	Modern CISC	EPIC
Operation system	Linux v. 2.4.18	Linux v. 2.4.18	IRIX v. 6.5	Linux 2.4.18	Linux 2.4.18
Clock rate	2400MHz	800MHz	175MHz	1333 MHz	1137 MHz
Address space	32 bit	32 bit	64 bit	32 bit	64 bit
Pipeline stages	20	12	6	10	8
L1 data cache size	8 KB	16 KB	32 KB	128 KB	32 KB
L1 line size	128 B	32 B	32 B	64 B	64 B
L1 associativity	4-way	4-way	2-way	2-way	4-way
L2 cache size	512 KB	256 KB	1024 KB	256 KB	256 KB
L2 line size	128 B	32 B	32 B	64 B	128 B
L2 associativity	8-way	4-way	2-way	8-way	8-way
TLB entries	128	64	64	40	128
TLB associativity	full	4-way	64-way	4-way	full
TLB miss handling	hardware	hardware	software	hardware	?
RAM size	512 MB	256 MB	128 MB	512 MB	3072 MB

Figure 5: Specification of Machines.

2.2 Selecting Cache Oblivious Sorting Algorithm:

There are three approaches for cache oblivious sorting algorithm, namely Funnel Sort, Lazy Funnel Sort and Distribution based Algorithm. All the three have same optimal bound O(N/B $\log_{M/B}$N/B) but the fact is that Funnel Sort and Distribution based Algorithms are structurally more complex than Lazy Funnel Sort, so Lazy Funnel Sort is the algorithm selected based on the structural simplicity.

2.3 Optimizing Lazy Funnel Sort:

To optimize Lazy Funnel Sort some of its parameters and design issues can be tuned to get optimum results, some of these choices include:

- K-Merger Structure, How should the funnel be laid out in memory?
- Degree of Basic Merger, Is binary merger the optimum choice?
- Parameter α and d?
  - α and d are variables that effect the output buffer size α K^d

K-Merger Structure: The optimal choice to represent funnel in memory happened to be vEB layout, similarly best results were obtained when recursive invocation was used instead of iterative method. Further the pointer based invocation produced better results than implicit navigation.

Degree of Basic Merger: The experiments showed that instead of using two way basic mergers, four way merger produced better results. The obvious reason for this is that four way mergers eliminate every other level in tree and thus reducing the overall flow of data.

Parameter α and d: Experiments were performed to find the optimum choice for α and d. Different combinations were checked for d = [1.5;3] and α= [1;40]. The optimum results reflected from the experiments were $\alpha \approx 16$, d $\approx$ 2.5

2.4 Implementation of Lazy Funnel Sort:

After performing the experiments to find the optimal choices for different design and parameter choices, the optimum Lazy Funnel Sort was implemented. As the original version of Lazy Funnel Sort uses binary merger so it was decided to implement two version FunnelSort2 and FunnelSort4 for binary and 4 way basic merger respectively with the following values:

- Recursive implementation of pointer based vEB layout.
- (α,d) = (16, 2)

2.5 Selecting Competitors for Lazy Funnel Sort:

In order to measure the efficiency of the optimized lazy funnel sort, it is desired to find its competitiveness with some of the currently used sorting algorithms. The competitors were selected form wide range of sorting algorithms that covers the standard QuickSort algorithm to cache aware sorting algorithms.

The competitors selected are:

- Quick Sort implementation.
  - std::sort, for STL library of GCC v 3.2 (GCC)
  - std::sort, for STL library of Intel C++ v 7.0 (Dink)
  - Authors own implementation based on Bentley and McIlroy (Mix)
- Cache Aware Sorting Algorithm
  - TPIE sorting routine AMI_SORT
  - R-merge
  - msort-c
  - msort-m

TPIE is library for external memory computation and are the algorithms are highly optimized. R-merge is also a cache aware algorithm. The other cache aware algorithms selected are msort-c and msort-m [5].

3 Experiments and Results:

3.1 Experimental setup:

The experiments were performed on entire data being in RAM as well as on inputs residing on disk. The experiments were performed 21 times and median was reported. In external memory the experiments were performed only once.

3.2 Results:

Lazy Funnel Sort Vs QuickSort: Lazy Funnel Sort performs better on small input sizes in RAM but looses to GCC QuickSort by 10-40% but on larger input sizes the gain ratio of lazy funnel sort is almost the same against the competitors on three architectures. The two architectures where Lazy Funnel Sort was outperformed by GCC were MIPS 10000 and Pentium 4. The justification for the better performance of GCC on MIPS is that MIPS has slow processor speed and Pentium 4 is using PC800 bus which causes a delay in access time to RAM.

Lazy Funnel Sort Vs Cache Aware Sorting Algorithms: Lazy Funnel Sort performs better against two of the cache aware sorting algorithms msort-c and msort-m. Against R-merge Lazy Funnel Sort performs better on all architectures except MIPS. The obvious reason for this is the fact that MIPS is RISC based architecture having large number of registers, something which R-merge is designed to exploit.

Similarly for the experiments performed on disk, TPIE leads the way. Funnel-Sort is second and performs better than GCC. The performance gain over GCC increases as the input size grows.

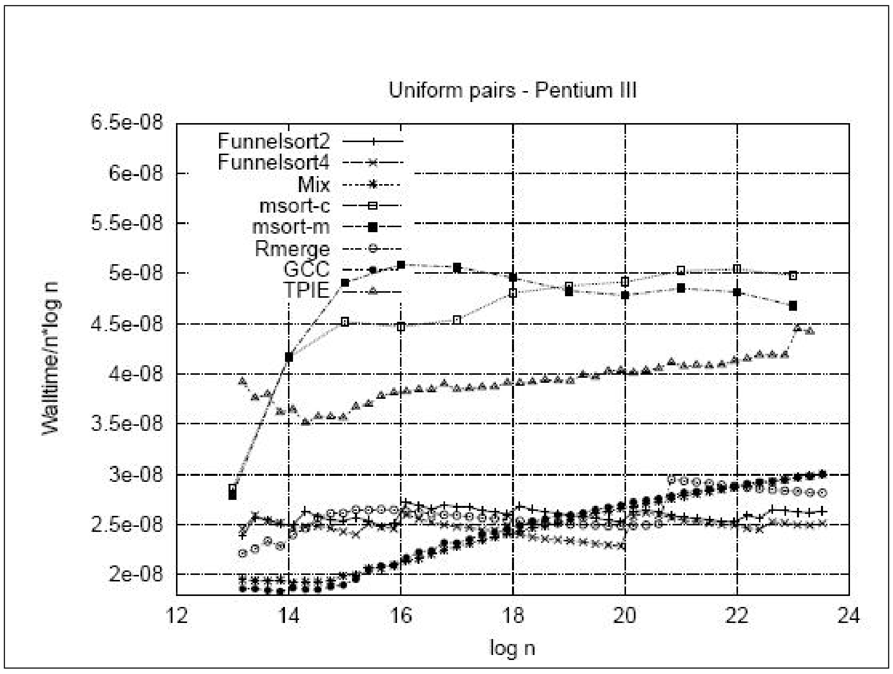


Figure 6: Results for experiments performed on Pentium III

The graphs can be viewed in the *AnnexureA*

3.3 Conclusion:

A careful implementation of Lazy Funnel Sort performs better in most of the experiments. It was also observed that in situation when Lazy Funnel Sort was outperformed by the competitors it was not by high margin and Lazy Funnel Sort was second best. Similarly Lazy Funnel Sort was also competitive on both RAM and disk and was the best algorithm to adapt to changes in memory hierarchy.

4 References

[1] Frederik Ronn, Cache-Oblivious Searching and Sorting, Master's Thesis, Department of Computer Science, University of Copenhagen, 2003

[2] E. Demaine, Cache-Oblivious Algorithms and Data Structures, Preliminary lecture notes - handed out at the EFF Summer School on Massive Data Sets. June 27-July 1, BRICS, University of Aarhus.

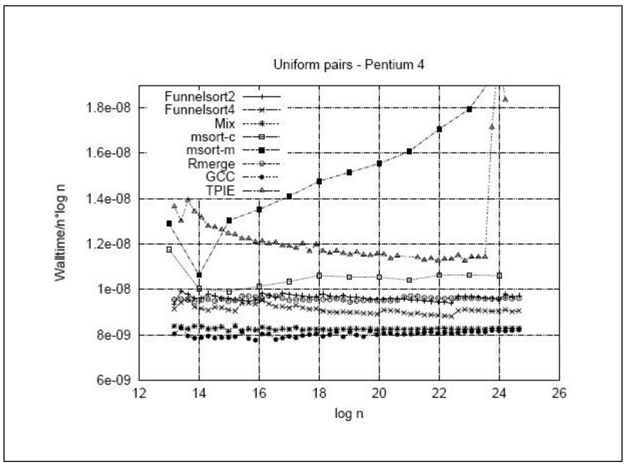
[3] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In 40th Annual Symposium on Foundations of Computer Science,pages 285-297. IEEE Computer Society Press, 1999.

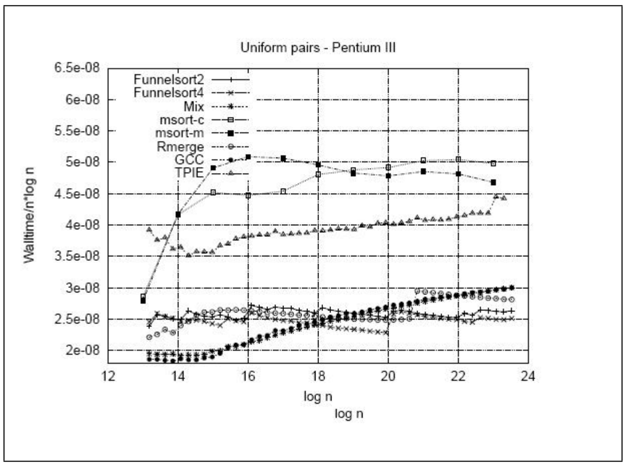
[4] G. S. Brodal and R. Fagerberg, Cache-Oblivious Distribution Sweeping, Proceedings of the 29th International Colloquium on Automata, Languages, and Programming, Lecture Note in Computer Science 2380, Springer-Verlag (2002), 426-438.

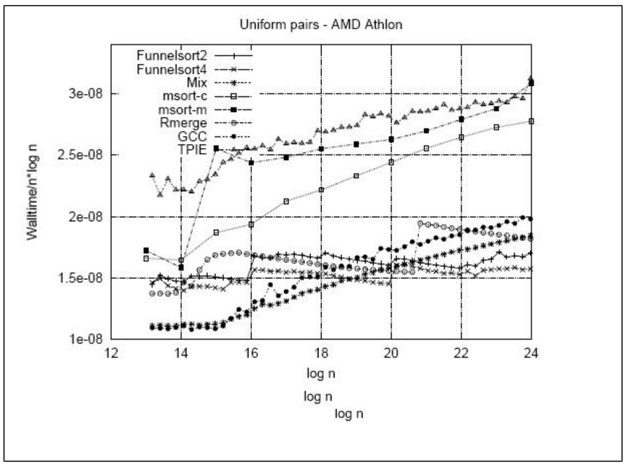
[5] L. Xiao, X. Zhang, and S. A. Kubricht. Improving memory performance of sorting algorithms. ACM Journal of Experimental Algorithmics, 5(3), 2000.

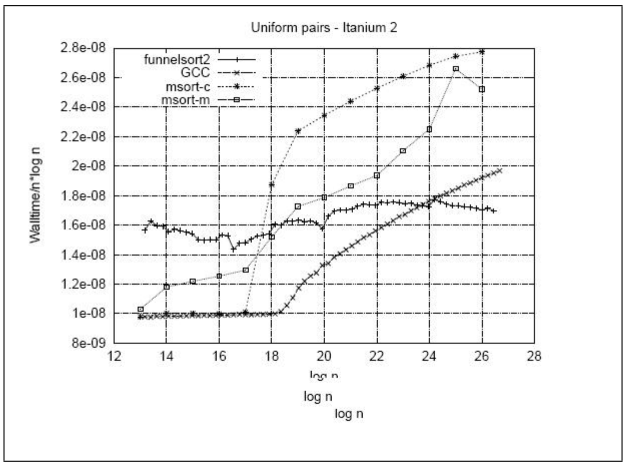
5 Annexure A

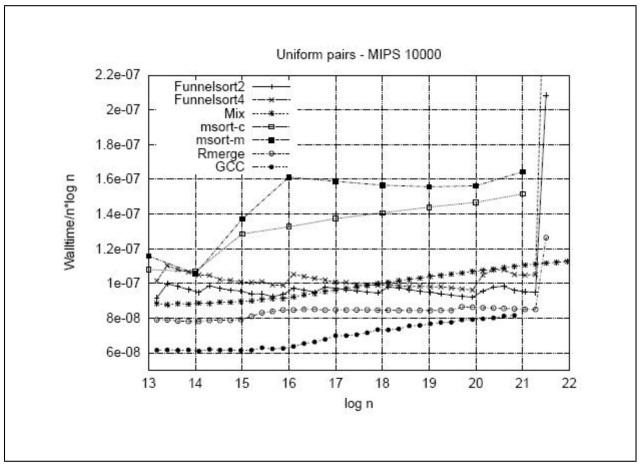
5.1 Results for Input in RAM



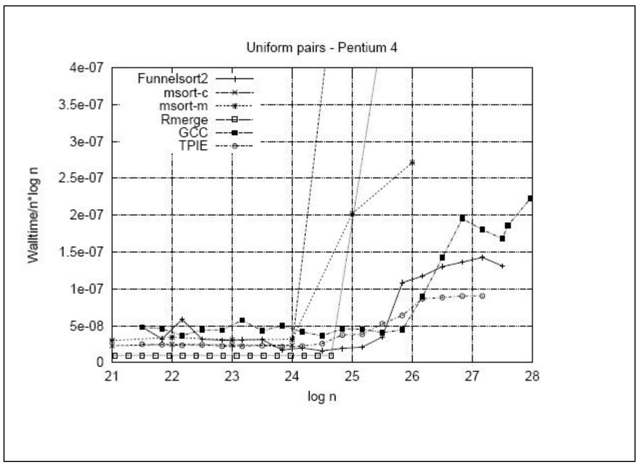


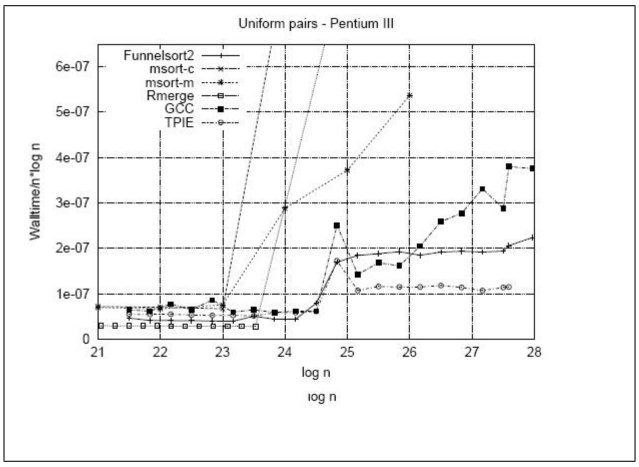

Uniform pairs - AMD Athlon
Funnelsort2
Funnelsort4
Mix
msort-c
msort-m
Rmerge
GCC
TPIE
Walltime/n*log n
log n
log n
log n

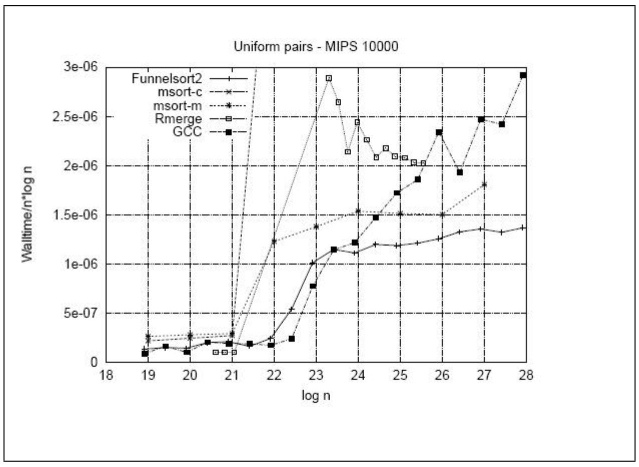

Uniform pairs - Itanium 2
funnelsort2
GCC
msort-c
msort-m
Walltime/n*log n
log n
log n
log n


Uniform pairs - MIPS 10000
Funnelsort2
Funnelsort4
Mix
msort-c
msort-m
Rmerge
GCC
Walltime/n*log n
log n

5.2 Results for Input in RAM






Uniform pairs - MIPS 10000
Funnelsort2
msort-c
msort-m
Rmerge
GCC
Walltime/n*log n
log n
3e-06
2.5e-06
2e-06
1.5e-06
1e-06
5e-07
0
18
19
20
21
22
23
24
25
26
27
28