

# An Adaptive Cache-Friendly Priority Queue: Fine-Tuning Heap Efficiency

Kiarash Parvizi

Department of Computer Engineering, Bu-Ali Sina University, Hamedan, Iran

## Abstract:

In this paper, we present a new approach to priority queues, specifically designed to enhance cache-friendliness. Our data structure incorporates three adjustable parameters, allowing for a customized heap structure that can adapt to different system conditions. A search method is used, determining the optimal parameter values, eliminating the need for manual configuration and delivering notable performance improvements, as demonstrated through rigorous testing on the heap sort algorithm. Importantly, this designed data structure can dynamically grow without the need for reconstructing the heap tree. The adaptability of our cache-friendly priority queue makes it particularly interesting in the context of modern computing, where system architectures can vary widely.

## Keywords

cache conscious – cache friendly - clustering – data structure – priority queue - cache - external memory - heap - cache efficiency – sorting – algorithms

## Introduction

In the realm of evolving computer memory systems, it is crucial to develop algorithms that align with the intricate structure of memory. Modern memory systems are characterized by a hierarchical arrangement, encompassing cache levels, main memory, and secondary storage. This stands in sharp contrast to conventional computational models, which assumed a uniform 'flat' memory access latency. Nowadays, computing platforms exhibit significant variations in access times across different memory levels, differing by several orders of magnitude. This phenomenon underscores the necessity for algorithms that can navigate this complexity effectively.

In this research paper, we focus on creating an adaptive priority queue that is optimized for cache usage. Priority queues are fundamental data structures widely used in applications like graph algorithms. Therefore, developing a priority queue that efficiently utilizes cache memory is a crucial and relevant area of study explored in this paper.

Traditionally, the majority of algorithmic research has been conducted within the framework of

the Random Access Machine (RAM) model of computation, which assumes a "flat" memory structure with uniform access time. In this standard RAM model, algorithm efficiency is gauged by the number of instructions it entails, with memory access considered to have a unit cost, regardless of the data's location within the memory. However, this model proves inadequate for accurately understanding modern computers equipped with intricate memory hierarchies. The advent of a memory hierarchy implies varying access costs based on the data's placement within the hierarchy. Algorithms that disregard this memory hierarchy face substantial performance penalties.

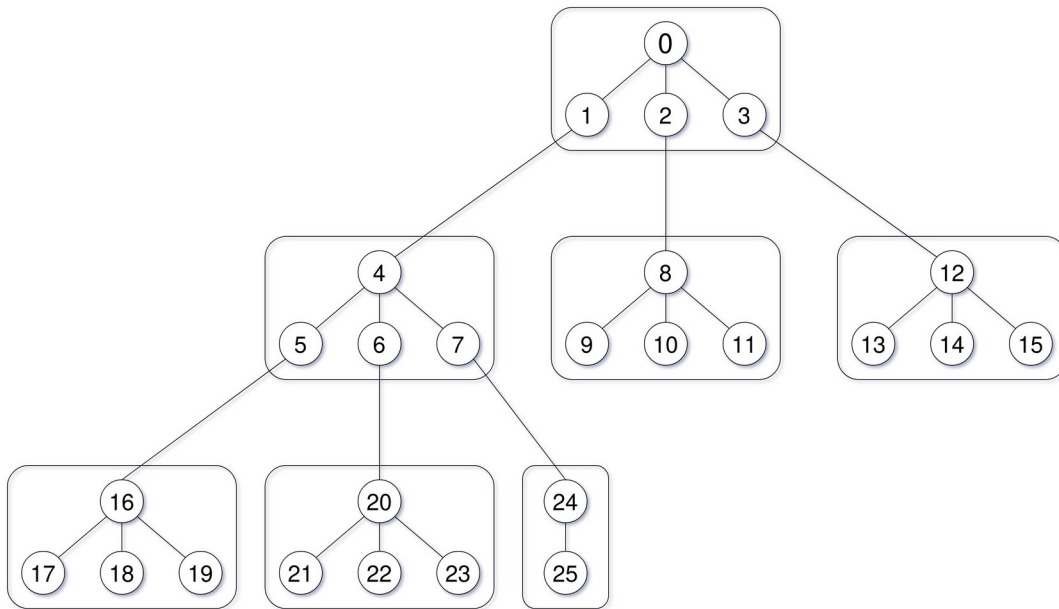
An alternative approach is the standard two-level memory hierarchy model incorporating block transfers, recognized under different names such as the external memory model, the I/O model, or the disk access model. This model characterizes a computer system with two levels: the cache, located proximally to the CPU, offers swift but limited access, while the disk, situated distantly from the CPU, provides expensive access but offers virtually limitless storage. The difference in access times between these two memory systems and the simplicity of this model render it a compelling choice for analyzing the operational costs of algorithms.

The development of memory hierarchies often incorporates the principle of locality of reference for efficient data access. This concept encompasses two key aspects. First, when a program accesses a specific memory location, it is likely to revisit that location multiple times within a short timeframe, known as temporal locality. Second, once a program accesses a particular memory location, it is probable that it will also access nearby memory locations shortly after, termed spatial locality.

Fundamental structure of a cache involves multiple cache blocks, each capable of storing multiple words. This design choice is made to leverage spatial locality, as accessing nearby memory locations often occurs in programs. The effectiveness of a cache system lies in its ability to store data that a program is likely to access again. When a program requests data that is not present in the nearest memory level, it results in a cache miss. During a cache miss, the requested word must be retrieved from the next, slower memory level, leading to performance delays. To address this challenge, researchers have explored three fundamental data placement design principles: clustering, coloring, and compression. These principles, when effectively combined, give rise to cache-conscious data structures, which play a pivotal role in optimizing system performance and minimizing access latency. The process of clustering involves organizing data structure elements within a cache block based on the likelihood of them being accessed together, thereby improving spatial and temporal locality. In the context of the new heap data structure discussed in this research paper, clustering forms the core design principle. Furthermore, our implementation of the heap structure incorporates the pointer elimination technique by organizing the entire data structure into a single compact array.

## Dynamic Cache-Friendly layout:

In this section, the details of a dynamic structure for heap implementation are expounded upon. The designed data structure, named Par-Heap, necessitates the determination of three parameters to specify its schema or layout. By adjusting the values of these parameters, various structures and clusters can be generated, endowing the data structure with remarkable flexibility and dynamism. Semi-optimal parameter values for specific machines can be ascertained, emphasizing the adaptability of the design. The implementation of the heap is array-based; wherein a heap of size  $N$  is accommodated within an array of the same size, enabling the performance of the heap-sort algorithm in-place with only constant extra memory needed.



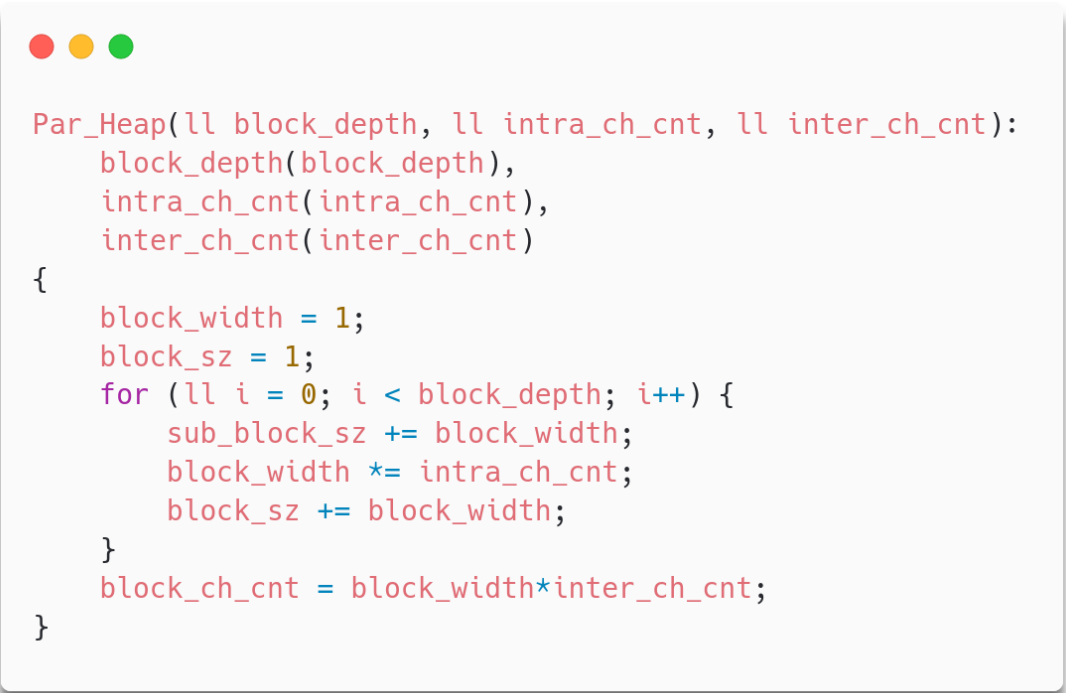
*Figure 1: Par-Heap data structure with parameters:  $intra\_child\_count = 3$ ,  $inter\_child\_count = 1$ ,  $block\_depth = 1$*

In Figure [1], it is evident that every block in the structure encompasses a full tree characterized by a height of 'block\_height' and a fanout of 'intra\_child\_count.' It is noteworthy that these specific parameters remain consistent across all blocks within the data structure. There is, however, an exception for the final block, serving as the terminal segment of the tree, which necessitates accommodating at least one node. The maximum number of children per leaf node is determined by the 'inter\_child\_count' parameter, which maintains a constant value for all leaf nodes.

By placing these structures next to each other, a comprehensive tree layout is created. When observing the layout broadly, each block acts as a super node, together forming a full tree structure. This arrangement is similar to a d-ary heap, with super nodes serving as its parts. The value of 'd' in this d-heap structure corresponds to 'block\_width \* inter\_child\_count', where 'block\_width' represents the number of leaf nodes in a block (nodes in the last layer of the block).

## Implementation

The data structure is represented by a single class. During initialization, the constructor takes three main parameters that define the structure's layout. As shown in Figure [2], it calculates additional values such as 'block\_depth' and 'block\_width' based on these parameters, which are essential for later operations.



```
Par_Heap(ll block_depth, ll intra_ch_cnt, ll inter_ch_cnt):
    block_depth(block_depth),
    intra_ch_cnt(intra_ch_cnt),
    inter_ch_cnt(inter_ch_cnt)
{
    block_width = 1;
    block_sz = 1;
    for (ll i = 0; i < block_depth; i++) {
        sub_block_sz += block_width;
        block_width *= intra_ch_cnt;
        block_sz += block_width;
    }
    block_ch_cnt = block_width*inter_ch_cnt;
}
```

*Figure 2: Par\_Heap constructor, calculating the needed values based on the given parameters*

In this paper, we focused on implementing two essential methods required for the test: 'heapify' and 'heap\_sort'. An important modification we made is in the 'heapify' method, which now accepts two arguments. One argument keeps track of the current super\_node index, and the other, called 'localI', keeps track of the position within the current super\_node.

Within the 'heapify' method, we initially determine if the current node is a block\_leaf node. Based on this and the chosen layout during class construction, we calculate the positions of the node's children. The subsequent steps follow the typical heap behavior, selecting the node with the most of a property (such as smallness) and continuing the heapify process or terminating, similar to the traditional heap method. The complete 'heapify' implementation is detailed in Figure [3].

In cases where the 'inter\_child\_count' parameter is set to one, we utilized a simplified version of the code. Here, we statically set the parameter to one and removed the 'for' loop used for finding the children of block\_leaf nodes. This optimization reduced the number of CPU instructions required. However, a similar optimization for the 'inter\_child\_count' of 2 did not significantly improve our benchmarks. As a result, we discontinued this optimization for values greater than one.

```

void heapify(ll I, ll localI) {
    // calculate the actual index of the current node in the data array
    ll index = I * block_sz + localI;
    // check if it's a block leaf node
    if (localI >= sub_block_sz) {
        // the current node is a leaf node
        // calculate the range of its childs
        ll itI = I*block_ch_cnt + 1 + (localI-sub_block_sz)*inter_ch_cnt;
        ll start = (itI) * block_sz;
        ll end = min(start + inter_ch_cnt * block_sz, n);
        // iterate over the calculated range
        int mx = data[index];
        ll mx_I = I;
        for (; start < end; start += block_sz, ++itI) {
            // compare (set max)
            if (data[start] > mx) {
                mx = data[start];
                mx_I = itI;
            }
        }
        // terminate heapify if no child of the parent has a greater value
        if (mx_I == I) return;
        // swap & recursively call heapify
        swap(data[mx_I * block_sz], data[index]);
        heapify(mx_I, 0);
        return;
    }
    // if it's not a block leaf node, then the mapping to children is done
    // the same way as the regular d-ary heap:
    // calculate the range of its childs
    ll start = I*block_sz + localI * intra_ch_cnt + 1;
    ll end = min(start + intra_ch_cnt, n);
    // iterate over the calculated range
    int mx = data[index];
    ll mx_index = index;
    for (; start < end; ++start) {
        // compare (set max)
        if (data[start] > mx) {
            mx = data[start];
            mx_index = start;
        }
    }
    // terminate heapify if no child of the parent has a greater value
    if (mx_index == index) return;
    // swap & recursively call heapify
    swap(data[mx_index], data[index]);
    heapify(I, mx_index-I*block_sz);
}

```

Figure 3: The complete implementation of the 'heapify' method used in Par\_Heap data structure. Note that 'll' is the cpp 'long long' data type.

## The search method:

In the abstract, we highlighted the need to determine suitable parameters for data structures in any system, achieved through an automated search method. In this study, we employed an exhaustive search approach due to the manageable search space. Notably, values of 'block\_depth' greater than  $1+\log(N)$  (in base 'intra\_child\_count') yield an identical structure resembling a regular d'ary heap. This occurs because all nodes are confined within the first and only super\_block. Additionally, our experiment involved setting manual limits on 'inter\_child\_count' and 'intra\_child\_count'.

The search method's effectiveness is gauged by the CPU time required to perform heap\_sort on an 80 million random integer array. The table [1] displays CPU times for various tree layouts. Among these, layout (2, 9, 1) yielded the most favorable results. Consequently, this layout will be employed in the comparison tests outlined in the upcoming Experiment section.

Inter_child_count		Block_height									
1		1	2	3	4	5	6	7	8	9	10
1	2	5.58	3.34	2.67	2.39	2.3	2.38	2.56	2.87	3.29	3.7
	3	3.55	2.23	1.98	1.92	2.26	2.61	2.92	2.96	3.13	3.35
	4	2.92	1.91	1.81	2.06	2.39	2.6	2.68	2.87	3.08	3.2
	5	2.58	1.84	1.79	2.11	2.39	2.38	2.58	2.74	2.82	2.77
	6	2.38	1.73	1.84	2.16	2.19	2.36	2.54	2.62	2.57	2.26
	7	2.28	1.69	1.81	2.18	2.15	2.35	2.46	2.5	2.21	2.13
	8	2.18	1.66	1.85	2.13	2.16	2.36	2.47	2.37	2.09	2.08
	9	2.1	1.63	1.92	2.01	2.15	2.31	2.35	2.05	1.99	1.98
	10	2.08	1.64	1.95	1.97	2.16	2.3	2.27	1.96	1.98	2.01
2	2	3.43	2.74	2.45	2.3	2.41	2.52	2.86	3.12	3.47	3.83
	3	2.61	2.16	2.02	2.11	2.37	2.72	2.97	2.97	3.16	3.41
	4	2.43	1.97	1.9	2.2	2.54	2.59	2.73	2.95	3.17	3.3
	5	2.3	1.83	1.99	2.28	2.41	2.43	2.65	2.8	2.89	2.81
	6	2.2	1.8	2.02	2.33	2.21	2.42	2.6	2.68	2.63	2.29
	7	2.12	1.76	2.07	2.28	2.19	2.42	2.54	2.54	2.26	2.17
	8	2.08	1.79	2.11	2.16	2.23	2.44	2.54	2.41	2.12	2.09
	9	2.07	1.85	2.12	2.04	2.22	2.37	2.39	2.07	2	1.98
	10	2.04	1.89	2.16	2.02	2.23	2.36	2.3	1.96	1.97	2

Table 1: cpu time (scale =  $1e7$  microseconds) needed for each layout to complete the heap sort operation on an array of size 80 million, filled with random 32bit integers

## Experiment:

The experiment involved creating arrays (containing random 32bit integer values) of various sizes ranging from 10 to 1e8. We then applied the 'make\_heap' operation followed by 'heap\_sort' on this array. We compared two main heap data structures: our Par-Heap with semi-optimal parameters and the regular binary heap. The comparison was based on the number of L2 and L3 total cache misses and the CPU execution time required to complete the task.

In our tests, the regular binary heap was implemented in two ways: one utilizing standard C++ library functions (referred to as stl-Heap) and the other method called standard-Heap. Each test was repeated 10 times, and the average results are presented in Figures [4,5,6]. It is important to note that there might be some unavoidable noise in the number of cache misses reported by the PAPI library.

The findings indicate that our Par-Heap structure, with a semi-optimal layout, demonstrated significant improvements in both the number of cache misses and CPU execution time compared to the other two methods. All programs and tests, accessible at [<https://github.com/Kiarash-Parvizi/Par-Heap>], were written in the C++ programming language and compiled using the clang compiler at the O2 optimization level.

## Hardware Specifications:

The tests were conducted on a computer with the following specifications:

```
Brand Raw: Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz
Hz Advertised Friendly: 2.8000 GHz
Hz Actual Friendly: 3.4136 GHz
Arch: X86_64
Bits: 64
Count: 8
Arch String Raw: x86_64
L1 Data Cache Size: 131072
L1 Instruction Cache Size: 131072
L2 Cache Size: 1048576
L3 Cache Size: 6291456
```

## Benchmark Results:

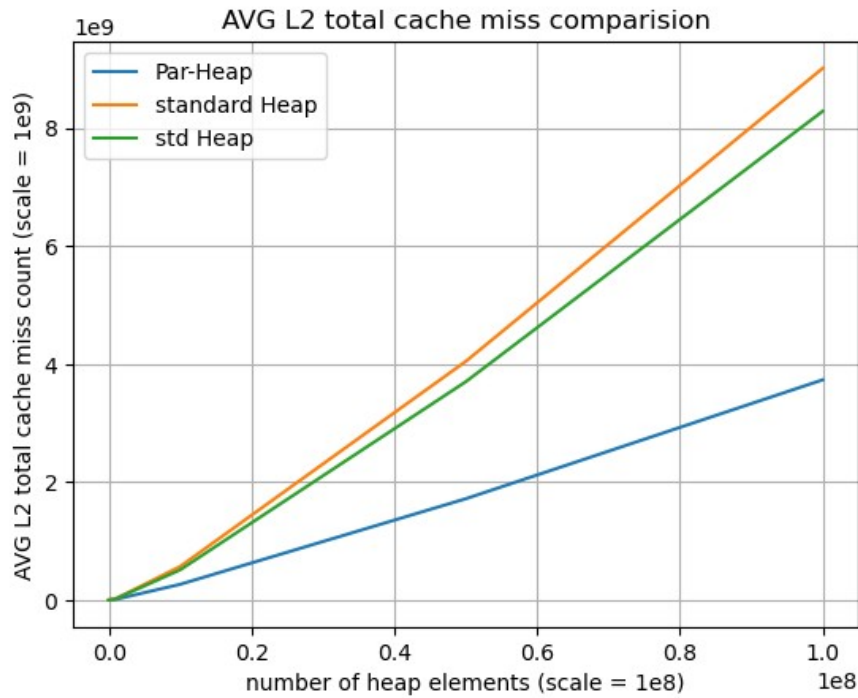


Figure 4: total L2 cache miss count for running the heap-sort algorithm with different number of heap elements

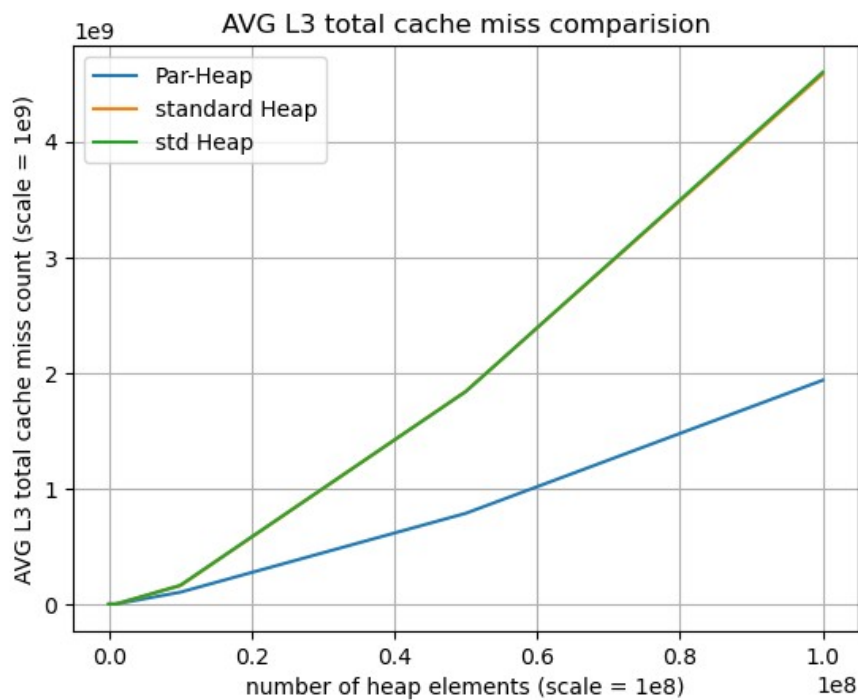


Figure 5: total L3 cache miss count for running the heap-sort algorithm with different number of heap elements. It can be seen that the 'standard' and 'std' methods have almost the same results.



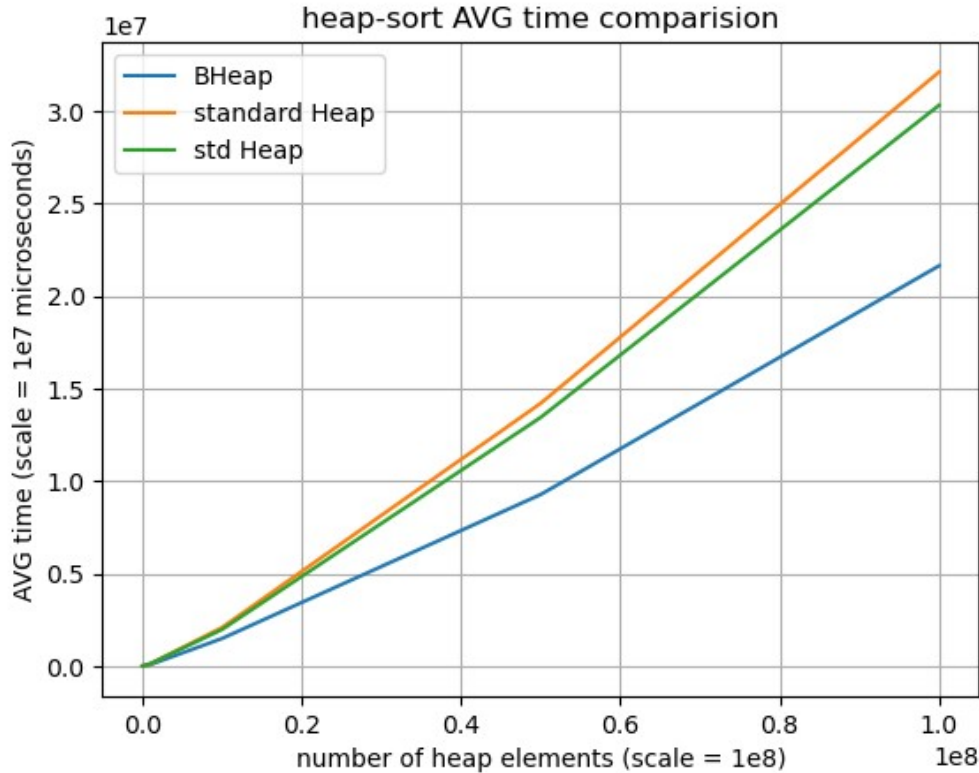


Figure 6: heap-sort runtime comparison benchmark

## Conclusion

In this research we introduced a new approach to priority queues, focusing on optimizing cache-friendliness. By incorporating adjustable parameters, we have developed a flexible heap structure that can be tailored to diverse system conditions. Extensive testing has demonstrated that implementing the techniques outlined in this research can result in substantial performance improvements.

Looking forward, future research endeavors could focus on refining the search method for parameters, aiming for a faster search algorithm.

## References

- [1] Knuth, D.E., 1997. The art of computer programming: sorting and searching (Vol. 3). Addison-Wesley Professional.
- [2] Anthony LaMarca and Richard Ladner. 1996. The influence of caches on the performance of heaps. ACM J. Exp. Algorithmics 1 (1996), 4–es. <https://doi.org/10.1145/235141.235145>

- [3] Robert W. Floyd. 1964. Algorithm 245: Treesort. *Commun. ACM* 7, 12 (Dec. 1964), 701. <https://doi.org/10.1145/355588.365103>
- [4] Peter Sanders. 2001. Fast priority queues for cached memory. *ACM J. Exp. Algorithmics* 5 (2000), 7–es. <https://doi.org/10.1145/351827.384249>
- [5] Cormen, T.H., Leiserson, C.E., Rivest R.L., Stein C., 2009. *Introduction to Algorithms* (3rd ed.). MIT Press.
- [6] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. 1999. Cache-conscious structure layout. *SIGPLAN Not.* 34, 5 (May 1999), 1–12. <https://doi.org/10.1145/301631.301633>
- [7] A. Agarwal, J. Hennessy, and M. Horowitz. 1989. An analytical cache model. *ACM Trans. Comput. Syst.* 7, 2 (May 1989), 184–215. <https://doi.org/10.1145/63404.63407>
- [8] Chilimbi, T. M. (1999). *Cache-conscious data structures, Design and implementation* (thesis). [University of Wisconsin-Madison].
- [9] Parvizi, K. (2023). Par-Heap implementation. <https://github.com/Kiarash-Parvizi/Par-Heap>
- [10] UNIVERSITY OF TENNESSEE, Performance application programming interface library. <http://icl.cs.utk.edu/projects/papi/>.