An Adaptive Concurrent Priority Queue for NUMA Architectures

Foteini Strati*
fstrati@cslab.ece.ntua.gr
School of Electrical and Computer Engineering
National Technical University of Athens

Christina Giannoula*
cgiannoula@cslab.ece.ntua.gr
School of Electrical and Computer Engineering
National Technical University of Athens

Dimitrios Siakavaras
jimsiak@cslab.ece.ntua.gr
School of Electrical and Computer Engineering
National Technical University of Athens

Georgios Goumas
goumas@cslab.ece.ntua.gr
School of Electrical and Computer Engineering
National Technical University of Athens

Nectarios Koziris
nkoziris@cslab.ece.ntua.gr
School of Electrical and Computer Engineering
National Technical University of Athens

ABSTRACT

Designing scalable concurrent priority queues for contemporary NUMA servers is challenging. Several NUMA-unaware implementations can scale up to a high number of threads exploiting the potential parallelism of the *insert* operations. In contrast, in *deleteMin*-dominated workloads, threads compete for accessing the same memory locations, i.e. the first item in the priority queue. In such cases, NUMA-aware implementations are typically used, since they reduce the coherence traffic between the nodes of a NUMA system.

In this work, we propose an adaptive priority queue, called SmartPQ, that tunes itself by automatically switching between NUMA-unaware and NUMA-aware algorithmic modes to provide the highest available performance under all workloads. SmartPQ is built on top of NUMA Node Delegation (Nuddle), a low overhead technique to construct NUMA-aware data structures using any arbitrary NUMA-unaware implementation as its backbone. Moreover, SmartPQ employs machine learning to decide when to switch between its two algorithmic modes. As our evaluation reveals, it achieves the highest available performance with 88% success rate and dynamically adapts between a NUMA-aware and a NUMA-unaware mode, without overheads, while performing up to 1.83 times better performance than Spraylist, the state-of-the-art NUMA-unaware priority queue.

1 INTRODUCTION

Designing efficient and scalable concurrent data structures for contemporary NUMA servers is quite challenging, especially when they incur high contention. Pointer chasing data structures, such as linked-lists, skip-lists and search trees have inherently low contention, as their operations need to de-reference a non-constant number of pointers before completing. Recent works [6, 10] have shown that lock-free algorithms [14, 16, 18, 24, 33, 34] of such data structures can scale to hundreds of cores. On the other hand, data structures such as queues and stacks are highly contended when accessed concurrently by many threads. In these data structures, concurrent threads compete for the same memory locations, leading to excessive traffic between the nodes of a NUMA system.

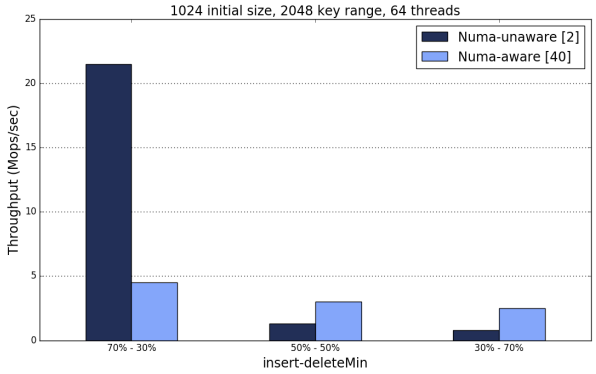

Figure 1: Throughput obtained by one NUMA-unaware [2] and one NUMA-aware [40] priority queue, both initialized with 1024 keys. The key range is set to 2048 keys and 64 threads perform a mix of *insert* and *deleteMin* operations in parallel. We executed the experiment on a 4-socket NUMA server the characteristics of which are presented in section 5.

In this work, we focus on priority queues, which exhibit medium contention and are widely used in a large number of applications, including task scheduling in real-time and computing systems [46], discrete event simulations [32, 43] and graph algorithms [25, 28, 44], e.g., Single Source Shortest Path (SSSP) problem [7] and Minimum Spanning Tree (MST) [37]. In contrast to queues and stacks, more parallelism can be extracted from priority queues, due to the potential parallelism of the *insert* operations. Therefore, concurrent NUMA-unaware implementations that scale up to a high number of threads have been proposed [2, 31]. However, in *deleteMin* operation, threads compete for removing the first element of the

* **Both authors contributed equally to this work.**

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CF '19, April 30-May 2, 2019, Alghero, Italy
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6685-4/19/05…$15.00
https://doi.org/10.1145/3310273.3323164

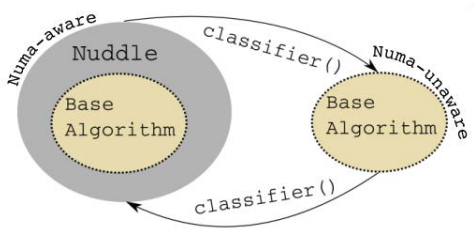

Figure 2: High-level design of SmartPQ, which dynamically adapts its algorithm to the needs of the workload according to the prediction of the classifier.

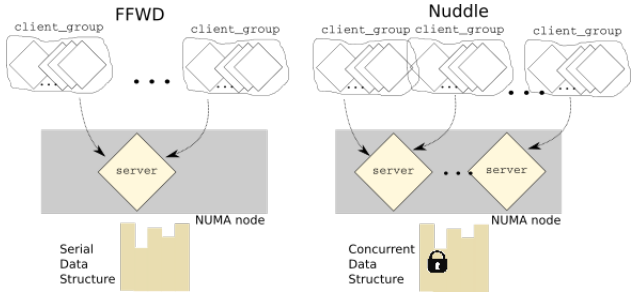

Figure 3: High-level design of FFWD [40] and Nuddle. Nuddle uses the same communication protocol as FFWD. Servers are located at the same NUMA node to provide a NUMA-aware approach, and each of them serves multiple client groups.

priority queue, creating a contention spot, which can become a performance bottleneck in NUMA machines. To alleviate this issue, NUMA-aware implementations have also been proposed [6, 40].

Even though we expect NUMA-aware implementations of priority queues to consistently outperform NUMA-unaware implementations in multi-node executions, this is not always the case. This is illustrated in Figure 1, which presents the throughput of one NUMA-unaware and one NUMA-aware priority queue on a 4-node NUMA server. Despite the fact that under low contention, i.e., 30% *deleteMin* operations, the NUMA-unaware implementation retains its high performance, when contention increases the NUMA-aware implementation performs better.

Accordingly, our goal in this work is to design a concurrent priority queue that combines both approaches and delivers the highest available performance under all workloads. Moreover, we augment our priority queue with a decision making mechanism such that to deliver the highest available performance even when the workload changes dynamically.

To this end, our contribution is threefold. First, we introduce *NUMA Node Delegation (Nuddle)*, a generic technique that can be used to wrap any NUMA-unaware data structure and transform it into an efficient NUMA-aware one. Nuddle extends FFWD [40] by allowing multiple server threads, instead of only one, to execute operations on behalf of client threads. In contrast to FFWD, Nuddle targets data structures with medium contention, such as priority queues, that are able to scale up to a number of threads.

Second, we use machine learning to predict the best-performing implementation among NUMA-unaware and NUMA-aware ones given the expected workload and contention level on the priority queue. We formulate the algorithmic mode (NUMA-unaware or NUMA-aware) as a classification problem and build a Decision Tree classifier to choose among these two.

Third, we propose *SmartPQ*, an adaptive priority queue that combines Nuddle and the classifier to deliver the best performance at any point in time. SmartPQ exploits the fact that client threads can execute operations either using Nuddle (NUMA-aware mode) or its underlying NUMA-unaware implementation (NUMA-unaware mode), with no actual change in the way data are accessed. More specifically, they can either delegate their operations to the appropriate server or directly access the data structure. Moreover, SmartPQ incorporates a decision making mechanism based on the proposed classifier to decide upon transitions between the two modes. A high level overview of SmartPQ is presented in Figure 2, where we use the term *base algorithm* to denote any arbitrary NUMA-unaware concurrent data structure. Since both algorithmic modes access the data structure in the same way, SmartPQ can switch from one mode to another without needing a synchronization point between transitions. As shown in our experimental evaluation, SmartPQ dynamically adapts between its two algorithmic modes, without overheads, and delivers the highest possible performance with 88% success rate.

2 NUMA NODE DELEGATION (NUDDLE)

2.1 Overview

NUMA Node Delegation (Nuddle) is a generic technique that can be used as a wrapper over any NUMA-unaware data structure and transform it into an efficient NUMA-aware implementation. It extends FFWD [40], a client-server software mechanism , where all operations performed by multiple client threads are delegated to a single dedicated thread, called the *server*. FFWD eliminates the need for synchronization, since the shared data structure is no longer accessed by multiple threads. Only a single server thread directly accesses the data structure and, therefore, FFWD can use a serial asynchronized implementation of the underlying data structure (Figure 3). FFWD originally targeted inherently serial data structures, whose concurrent performance cannot be better than that of single threaded execution.

On the contrary, Nuddle targets data structures that can scale up to a number of concurrent threads. Priority queues are a typical example of such a data structure. Multiple threads can concurrently execute *insert* operations, although *deleteMin* is inherently serial since all threads write to a single memory location, i.e., the element with the minimum priority. In relaxed priority queues, such as Spraylist [2], even *deleteMin* can be parallelized to some extent.

A high level design and the difference of Nuddle compared to FFWD is shown in Figure 3. We use the same protocol for client-server communication. The protocol carefully manages memory accesses to minimize cache coherence traffic and latency. Multiple clients are grouped together to minimize the response messages (one response cache line per client group). For more details the reader can refer to [40]. In contrast to FFWD, Nuddle deploys multiple servers each of which serves multiple client groups. Since multiple servers concurrently access the underlying data structure, a concurrent NUMA-unaware version is used to ensure correctness. Finally, servers are pinned to the same NUMA node to keep the data structure locally, while clients can be located at any node.

```
1
2  #define cache_line_size 128
3  typedef char cache_line[cache_line_size];
4  struct nuddle_pq {
5    nm_oblv_set *base_pq;
6    int servers, groups, clnt_per_group;
7    int server_cnt, clients_cnt, group_cnt;
8    cache_line *requests[groups][clnt_per_group];
9    cache_line *responses[groups];
10   lock *global_lock;
11 };
12
13 struct server {
14   nm_oblv_set *base_pq;
15   cache_line *my_clients[], *my_responses[];
16   int my_groups, clnt_per_group;
17 };
18
19 struct client {
20   cache_line *request, *response;
21   int clnt_pos;
22 };
23
24 struct nuddle_pq *initPQ(int servers, int max_clients) {
25   struct nuddle_pq *pq = allocate_nuddle_pq();
26   __base_init(pq->base_pq);
27   pq->servers = servers;
28   pq->clnt_per_group = client_group(cache_line_size);
29   pq->groups = (max_clients +
30     pq->clnt_per_group-1) / pq->clnt_per_group;
31   pq->server_cnt = 0;
32   pq->client_cnt = 0;
33   pq->group_cnt = 0;
34   pq->requests = malloc(groups * clnt_per_group);
35   pq->responses = malloc(groups);
36   init_lock(pq->global_lock);
37   return pq;
38 }
```

Figure 4: The data structures and initialization function of Nuddle.

2.2 Implementation Details

Even though in this work we focus on priority queues, Nuddle is a framework that wraps any existing concurrent data structure to transform it to a NUMA-aware approach. Figures 4 and 5 present the code of a priority queue based on Nuddle. We denote with red color the default operations of the base underlying data structure.

The structures. Each client has its own *struct client* structure with a dedicated request and a dedicated response cache line. The request cache line is exclusively written by the client and read by the server, while the response cache line is exclusively written by the server and read by all clients that belong in the same client group. Similarly, each server has its own *struct server* structure that includes an array of requests (*my_clients*), each of them shared with a client, and an array of responses (*my_responses*), each of them shared with a client group. Finally, the structure of Nuddle, named *struct nuddle_pq*, wraps the base algorithm (*nm_oblv_set*) and includes some additional fields needed to associate clients with servers in the initialization step.

Initialization. The proposed *initPQ()* initializes the main data structure, using the corresponding function of the base algorithm (line 25), and the additional fields needed for Nuddle. In this function, programmers specify the number of servers and the maximum number of clients to properly allocate the cache lines needed for communication among them. They also need to specify the size

```
39 struct server *initServer(struct nuddle_pq *pq, int core)
40 {
41   set_affinity(core);
42   struct server *srv = allocate_server();
43   srv->base_pq = pq->base_pq;
44   srv->my_groups = 0;
45   srv->clnt_per_group = pq->clnt_per_group;
46   acquire_lock(pq->global_lock);
47   int j = 0;
48   for(i = 0; i < pq->groups; i++)
49     if(i % pq->servers == pq->server_cnt) {
50       srv->my_clients[j] = pq->requests[i][0..gr_clnt];
51       srv->my_responses[j++] = pq->responses[i];
52       srv->my_groups++;
53      }
54   pq->server_cnt++;
55   release_lock(pq->global_lock);
56   return srv;
57 }
58
59 int insert_server(struct server *srv,
                     int key, 8-byte value)
60 {
61   return __base_insrt(srv->base_pq, key, value);
62 }
63
64 void serve_requests(struct server *srv) {
65   for(i = 0; i < srv->mygroups; i++) {
66     cache_line resp;
67     for(j = 0; j < srv->clnt_per_group; j++) {
68       key = srv->my_clients[i][j].key;
69       value = srv->my_clients[i][j].value;
70       if (srv->my_clients[i][j].op == "insert")
71         resp[j] = __base_insrt(srv->base_pq, key, value);
72       else if (srv->my_clients[i][j].op == "deleteMin")
73         resp[j] = __base_del(srv->base_pq);
74     }
75     srv->my_responses[i] = resp;
76   }
77 }
78
79 struct client *initClient(struct nuddle_pq *pq) {
80   struct client *cl = allocate_client();
81   acquire_lock(pq->global_lock);
82   cl->request = &(pq->requests[group_cnt][clients_cnt]);
83   cl->response = &(pq->responses[group_cnt]);
84   cl->pos = pq->client_cnt;
85   pq->client_cnt++;
86   if (pq->client_cnt % pq->clnt_per_group == 0) {
87     pq->clients_cnt = 0;
88     pq->group_cnt++;
89   }
90   release_lock(pq->global_lock);
91   return cl;
92 }
93
94 int insert_client(struct client *cl,
                     int key, 8-byte value)
95 {
96   cl->request = write_req("insert", key, value);
97   while (cl->response[cl->pos] == 0) ;
98   return cl->response[cl->pos];
99 }
```

Figure 5: The functions used by servers and clients in Nuddle.

of the client group that is either 7 or 15, when a cache line is 64 or 128 bytes, respectively. As explained in [40], supposing 8-byte return values, a dedicated 128-byte response cache line is shared between up to 15 clients, because it also has to include an additional toggle bit for each client. After initializing *struct nuddle_pq*, each running thread calls either *initServer()* or *initClient()* depending on its role. It initializes its own structure (*struct server* or *struct client*) with request and response cache lines of the shared corresponding arrays of *struct nuddle_pq*. Servers undertake client groups with a round robin fashion, such that the load associated with clients is balanced among them. In the function *initServer()*, it is the programmer's responsibility to properly pin servers to the hardware threads of the machine (line 40), such that they are located in the same NUMA node, and fully benefit from the Nuddle technique. Moreover, supposing that clients of the same group share the same response cache line, the programmer can pin them at the same NUMA node to reduce cache coherence traffic. Finally, since the request and response arrays of *struct nuddle_pq* are shared between all threads, a global lock is used when accessing them to ensure mutual exclusion.

Main API. The core operations of *insert* and *deleteMin* have similar API with the classic API of state-of-the-art priority queues. However, we separate the corresponding operations for clients and servers. Clients write their request to the dedicated request cache line (line 96) and then wait for the server's response, while servers directly execute operations in the data structure using the core operations of the base algorithm (line 61). In Figure 5, we omit the respective operations of *deleteMin*, since they are very similar to *insert* operations. Servers can serve clients using the *serve_requests()* function. A server iterates through each client group and executes the requested operations in the data structure. It buffers individual return values to a local cache line (*resp* in lines 71 and 73) until processing for the current client group has finished, then writes all responses to the response cache line of that client group, and proceeds to the next client group.

3 SELECTING THE ALGORITHMIC MODE

3.1 The Need for Machine Learning

To design an adaptive priority queue, we need to resolve two issues: a) how to switch from NUMA-aware mode to NUMA-unaware, or vice versa, with low overhead, and b) decide when this transition is needed. We can resolve the first issue using Nuddle, since it uses any arbitrary NUMA-unaware concurrent data structure to ensure correctness when multiple servers access the data structure. Client threads can either directly access the data structure (NUMA-unaware mode) or delegate their operation to the appropriate server (NUMA-aware mode) without violating correctness. In this section, we aim to resolve the second issue, and propose a decision making mechanism that selects the best performing algorithmic mode for each workload.

Our experimentation shows that it is not trivial to construct a statistical model that selects the correct algorithmic mode for two reasons; first the influence of the workload on the performance behavior of each data structure is not straightforward, and second, different concurrent data structures have varying behavior in servers with different architectures. Figure 6 summarizes these observations. When the number of threads increases, the contention also increases, and thus we may expect that the performance of alistarh_herlihy degrades. Instead, in 6a, when using 29 threads with 80% insert, alistarh_herlihy outperforms Nuddle. In this case, the size of the priority queue and the key range are relatively large, and the ratio of *deleteMin* operations is not high. Therefore, threads may not update the same elements, and thus, alistarh_herlihy achieves high throughput.

Similarly, in *insert*-dominated workloads, as the key range increases, the contention decreases, since threads compete less for the same elements. We might, thus, expect that after a certain point, alistarh_herlihy will always outperform Nuddle, as the contention decreases. However, as shown in Figure 6b, when using the key range of 10K, Nuddle achieves 1.58 times higher performance than alistarh_herlihy. In this figure, the performance of Nuddle remains constant, as expected, while the performance of alistarh_herlihy varies as the key range changes, due to the hyperthreading effect. We evaluate the implementations shown in this figure in the same way as described in section 5.1. When using more than 32 threads, hyperthreading is enabled in our NUMA server resulting to a varying behavior in case of alistarh_herlihy. The hyperthreading pair of threads shares the L1 and L2 caches. Thus, those threads may either thrash or benefit from each other depending on the elements accessed in each operation. Finally, Figure 6c shows that for the same workload, data structures present varying behavior in servers with different architectures. Based on these observations, we resort to machine learning as the basis of a prediction mechanism for this non-straightforward behavior.

3.2 Decision Tree Classifier

We formulate the selection of the algorithmic mode as a classification problem and leverage supervised learning techniques to train a classifier that predicts the correct algorithmic mode for each workload. For our classifier, we choose decision trees, since they are commonly used classification models, with low training and inference cost. Moreover, they are easy to interpret and thus, be incorporated to the proposed priority queue. We generate the Decision Tree classifier using the scikit-learn machine learning toolkit [35].

1) Class Definition: We define a threefold classification model for the following reason: When using one socket of the machine, Nuddle delivers the same performance as the base algorithm. In that case, threads do not need to delegate their operations, instead they directly execute them in the data structure using the underlying NUMA-unaware algorithm. In such a configuration, the classifier can predict either Nuddle or the NUMA-unaware mode. Moreover, we want to configure a transition from one to another mode to occur when the difference in their throughput is greater than a certain threshold. Otherwise, SmartPQ might continuously oscillate between the two modes, without delivering significant performance improvement or even causing performance degradation. Therefore, our model consists of the following three classes: a) the ***NUMA-unaware*** class that stands for the NUMA-unaware mode, i.e the base algorithm of Nuddle, b) the ***NUMA-aware*** class that stands for the NUMA-aware mode, i.e. Nuddle, and c) the ***neutral*** class

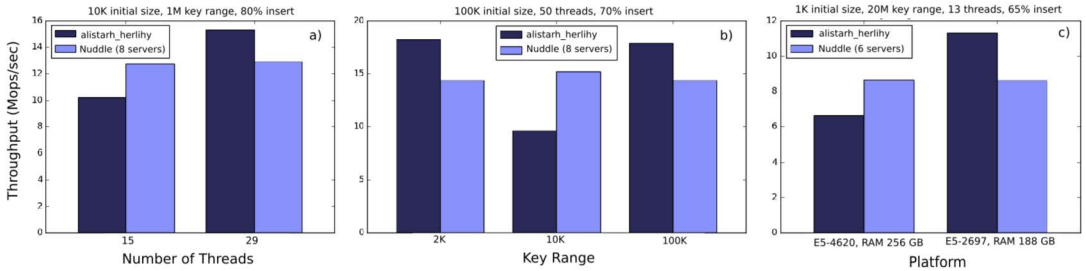

Figure 6: Experimental comparison of Nuddle and its underlying NUMA-unaware base algorithm, varying (a) the number of threads (b) the key range and (c) the experimental platform.

Feature	Definition
#threads	The number of running threads
size	The initial size of the priority queue
key_range	The range of keys used in operations
% *insert*/*deleteMin*	The ratio of *insert*/*deleteMin* operations

Table 1: The features used for classification.

that stands for "tie", meaning that any of the two modes is correct. As a result, when the classifier predicts the neutral class, SmartPQ remains at the currently selected algorithmic mode avoiding unnecessary transitions.

3) Generation of Training Data: To train our classifier, we develop microbenchmarks in which threads execute random operations on the data structure for 5 seconds. We evaluate Nuddle as the NUMA-aware mode and *alistarh_herlihy* as the NUMA-unaware mode, and select a variety of values for the features of the classifier. Our training data set consists of 5525 different workloads. Finally, we pin software threads to the hardware threads of the machine in a round-robin fashion, and thus, the classifier is only trained in this thread placement. We aim to experiment more on the thread placement of our training data set in future work.

4) Labeling of Training Data: In our data set, we set the threshold for the "tie" between the two algorithmic modes to 1.5 Million operations per second. When the difference in their throughput is less than this threshold, the *neutral* class is selected as label. Otherwise, we select the class that corresponds to the algorithmic mode that delivers the higher throughput.

The final Decision Tree classifier is quite small, as our classification problem has only 4 features. The classifier has 180 nodes, half of which are leaves. It also has a quite low depth of 8, that is the length of the longest path in the tree, and consequently, a quite low traversal cost (on average 2-4 ms in the NUMA server used in our evaluation).

4 SMARTPQ

In this section, we extend Nuddle and incorporate the decision making mechanism based on the aforementioned classifier to design a dynamically adaptive concurrent priority queue, named SmartPQ. SmartPQ delivers the highest available performance under all execution scenarios, since it can switch between the NUMA-aware Nuddle and its underlying NUMA-unaware implementation without needing a synchronization point. Figure 7 presents the modified code of Nuddle to implement SmartPQ.

We extend the main structure of Nuddle, renamed to `struct smartpq`, by adding an additional flag field, named `algo`, that keeps track of the current algorithmic mode (either Nuddle or the base algorithm). Similarly, the structures `struct client` and `struct server` are extended with an additional `algo` field, that is a pointer to the `algo` variable of the main structure, and is initialized by all threads in `initClient()` and `initServer()` (line 111). In this way, all threads share the same algorithmic mode at any point in time. We also modify the core operations of clients, `insert()` and `deleteMin()`, such that they either directly execute their requests in the data structure (line 118), or delegate them to servers (line 120), with respect to the current algorithmic mode. On the other hand, the core operations of servers do not need any modification. Finally, we wrap the lines 65-76 of the `serve_requests` function of Figure 5 with an if/else statement depending on the `algo` field, such that servers poll at clients' requests only in Nuddle mode. If the NUMA-unaware mode is selected, the function returns without doing anything. In this way, programmers do not need to take care of the calls on this function in their code, when the NUMA-unaware mode is selected.

We also incorporate the Decision Tree classifier. The function `decisionTree()` describes the API of the classifier, where the arguments are associated with its features. When the workload changes (or in frequent time lapses), one or more threads can call this function to check if a transition to another mode is needed. If this is the case, the `algo` field of `struct smartpq` is updated, SmartPQ switches mode, and all threads start executing their operations using the new algorithmic mode. If the classifier predicts the neutral class, the `algo` field is not updated, and SmartPQ remains at the current selected mode (line 126).

5 EXPERIMENTAL EVALUATION

Our experimental evaluation focuses on three aspects: the performance comparison of Nuddle with state-of-the-art concurrent priority queues, the accuracy of our classifier and the performance benefit of SmartPQ over naive approaches in synthetic benchmarks that frequently change their workloads. We use a 4-socket Intel Sandy Bridge-EP server equipped with 8-core Intel Xeon CPU E5-4620 processors providing a total of 32 physical cores and 64 hardware threads. The processor runs at 2.2GHz and each physical core has its own L1 and L2 cache of sizes 64KB and 256KB respectively. A 16MB L3 cache is shared by all cores in a socket and the RAM is 256GB. We use GCC 4.9.2 with -O3 optimization flag enabled to compile our implementations.

```
100 struct client {
101   nm_oblv_set *base_pq;
102   cache_line *request, *response;
103   int clnt_pos;
104   // 0: tie, 1: NUMA-oblivious (default), 2: NUMA-aware
105   int *algo;
106 };
107
108 struct client *initClient(struct smartpq *pq) {
109   ... lines 80-89 in Alg. 1 ...
110   cl->base_pq = pq->base_pq;
111   cl->algo = &(pq->algo);
112   release_lock(pq->global_lock);
113   return cl;
114 }
115
116 int insert_client(struct client *cl,
                      int key, float value) {
117   if( *(cl->algo) == 1) {
118     return __base_insert(cl->base_pq,key,value);
119   } else { // *(cl->algo) == 2
120     ... lines 96-98 in Alg.1 ...
121   }
122 }
123
124 void decisionTree(struct server/struct client *str,
                      int nthreads, int size,
                      int key_range, double insert) {
125   ...
126   if (algo!=0)    *(str->algo) = algo;
127 }
```

Figure 7: The modified code of Nuddle with the necessary changes to implement SmartPQ.

5.1 Performance Comparison with State-Of-The-Art Priority Queues

Our evaluation includes the following concurrent priority queues:

- alistarh_fraser: A NUMA-unaware, relaxed, lock-free priority queue [2] based on Fraser's skip-list [16], available at [10].
- alistarh_herlihy: A NUMA-unaware, relaxed, lock-free priority queue [2] based on Herlihy's skip-list [22], available at [10].
- lotan_shavit: A NUMA-unaware, lock-free priority queue designed by Lotan and Shavit [31], available at [10].
- FFWD: A NUMA-aware approach based on delegation with one dedicated server [40].
- Nuddle: The proposed NUMA-aware priority queue which uses *alistarh_herlihy* as the underlying base algorithm.

We evaluate the concurrent priority queues in the following way:

- Each execution lasts 5 seconds during which each thread performs randomly chosen operations. We also tried longer durations and had similar results.
- Each software thread is pinned to a hardware thread. Hyperthreading is enabled when using more than 32 processing threads. When exceeding the number of available hardware threads, we oversubscribe more than one thread in each hardware thread.
- We use a delay loop of 25 pause instructions between consecutive requested operations on the priority queue. This delay is intentionally added to our benchmarks in order to better simulate a real-life application, where the operations in the priority queue are intermingled with the rest instructions of the application.
- At the beginning of each run, the priority queue is initialized with elements the number of which is described at each figure.
- The first 8 threads are pinned to the first socket, and then we pin 7 processing threads to consecutive sockets (adding one client group each time in case of Nuddle.).
- In NUMA-unaware implementations, any allocation needed in the operation, is executed on demand and memory affinity is determined by the first touch policy.
- In Nuddle, the first 8 threads that represent servers, execute `serve_requests` and then a randomly chosen *insert* or *deleteMin* and repeat this process until time is up.
- In Nuddle and FFWD [40], since our server has 64-byte cache lines, the response cache line is shared among up to 7 clients, supposing 8 byte return values.
- We have disabled the automatic Linux Balancing [27] to get consistent and more stable results.
- All reported results are the average of 10 independent executions with no significant variance.

Figure 8 presents the throughput achieved by concurrent priority queues for various sizes and operation workloads. The NUMA-aware priority queues (FFWD and Nuddle) achieve high throughput in *deleteMin*-dominated workloads: FFWD outperforms NUMA-unaware implementations only at small priority queues (100K elements), while Nuddle performs best in all *deleteMin*-dominated workloads. However, their performance saturates, since they are limited by the number of servers, because only servers access the data structure. FFWD is limited to one server, while Nuddle to 8 servers. Therefore, when using more than 8 threads, the performance of Nuddle remains constant. Second, the message passing protocol between servers and clients has negligible overhead; when more and more clients are used, even though the communication traffic increases, there is not any performance drop. We conclude that, although the performance of Nuddle saturates, it achieves the highest throughput in *deleteMin*-dominated workloads and remains a high efficient NUMA-aware approach, as it avoids performance drops caused by the NUMA effect and cache coherence traffic.

On the other hand, the NUMA-unaware priority queues are highly influenced by the NUMA effect and memory invalidation traffic resulting to significant performance drops. In *insert*-dominated workloads, the performance of the lotan_shavit priority queue degrades when using more than one socket of the machine, while the other two NUMA-unaware priority queues achieve high scalability. Those are relaxed priority queues, in which *deleteMin* returns an element among the first $O(p \log 3p)$ elements, with high probability, where p is the number of threads. As a result, in *insert*-dominated workloads, the contention levels decrease, since threads do not compete for the same elements, and so does the memory invalidation traffic. In these workloads, the relaxed priority queues outperform the NUMA-aware ones.

To conclude, it turns out that there is no one-size-fits-all solution, since none of the priority queues delivers the best performance in all workloads. Nuddle is an efficient NUMA-aware approach that has high throughput in medium and highly-contented workloads. On the other hand, in case of lower contention, even in a NUMA configuration -more than one NUMA nodes used-, the relaxed NUMA-unaware approaches outperform the NUMA-aware ones delivering the best performance. It is thus desirable to design an adaptive priority queue that combines the best of both worlds, and delivers the highest performance in all cases.

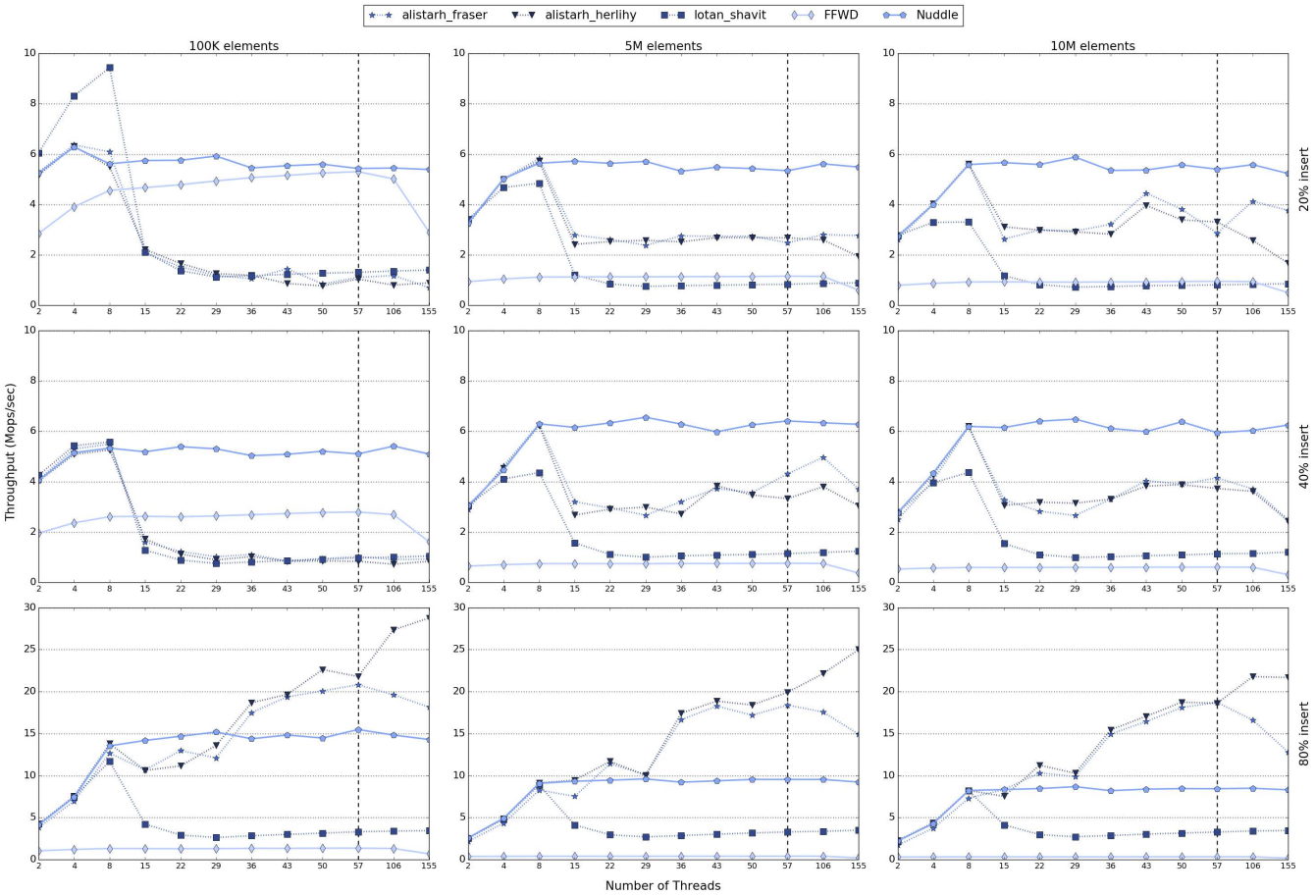

Figure 8: Throughput of concurrent priority queues implementations. The columns represent different sizes for double size key range and the rows different operation workloads. The vertical line in each plot shows the point after which we oversubscribe software threads to hardware threads.

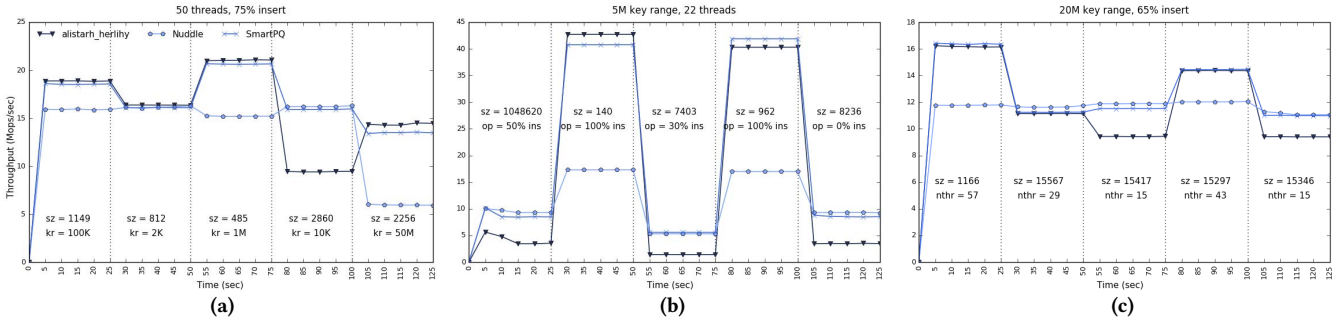

Figure 9: Throughput achieved by SmartPQ, Nuddle and the base algorithm (*alistarh_herlihy*), in dynamic benchmarks that change their workload varying a) the key range, b) the operation workload and c) the number of threads.

5.2 Decision Tree Classifier Accuracy

We evaluate the accuracy of our classifier, i.e. the percentage of correct predictions. We consider a correct prediction if the classifier predicts the algorithmic mode (NUMA-aware or NUMA-unaware mode) that has the highest throughput (Million operations per second). We use a test set of 10780 different workloads, and for each workload, we randomly select the values of its features. Then, we measure the throughput achieved by Nuddle and its underlying base algorithm, i.e. *alistarh_herlihy* priority queue, and use our classifier to predict between these two algorithmic modes. In the above test set, our classifier has 88% accuracy, meaning that it mispredicts 1300 times in 10780 different workloads. Furthermore, we measure the cost of misprediction, that is the difference in the throughput of the wrong predicted algorithmic mode from the correct mode normalized by the throughput of the predicted mode. For example, given that the throughputs of the predicted and correct mode are Y

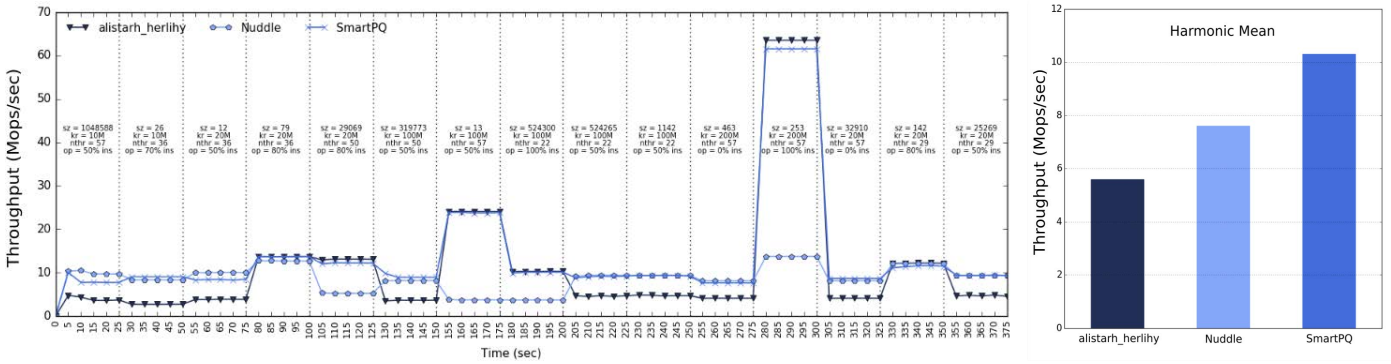

Figure 10: Throughput of SmartPQ, Nuddle and the base algorithm (*alistarh_herlihy*), in dynamic benchmarks that change their workload.

and X respectively, the misprediction cost is $((X - Y)/Y) * 100\%$. In 1300 mispredicted workloads, the geometric mean of misprediction cost is 30.2%. We conclude that the classifier provides high benefit to SmartPQ, since it has high accuracy, and even in the case of a misprediction, it results to a quite low performance drop.

5.3 Performance Benefit of SmartPQ in Dynamic Benchmarks

We present the performance benefit of SmartPQ in dynamic benchmarks, which frequently change their workload, and compare it with Nuddle and *alistarh_herlihy* priority queue. In each benchmark, we change the workload every 25 seconds and evaluate the concurrent priority queues in the same way as described in section 5.1. In SmartPQ, one dedicated thread calls the classifier in a fixed time interval of one second, and if a transition to another mode is needed, the algorithmic mode is updated, and all threads start executing their operations using the new mode. In Figures 9a, 9b and 9c, we only vary one feature in the workload, that is the key range, the ratio of *insert*/*deleteMin* and the number of threads, respectively, while in Figure 10, we vary multiple features every 25 seconds. It should be noted that the size of the priority queue changes during the execution, due to the successful *insert* and *deleteMin* operations, and none of the workloads evaluated in this section belong to the training data set used in the training process. Finally, in these benchmarks, we suppose that the workload is known a priori. In section §6, we will present some directions on how to on-the-fly extract the workload.

As already shown, there is no one-size-fits-all solution. Neither Nuddle nor *alistarh_herlihy* achieves the highest throughput in all workloads. In Figure 9c, even though the performance of Nuddle is constant, it outperforms *alistarh_herlihy*, when 15 threads (2 nodes of our machine) are used. On the other hand, SmartPQ successfully incorporates the best of both worlds. It is always able to successfully adapt to the needs of the workload, and achieve the best performance under all circumstances. Even when changing more than one features in the workload (Figure 10), the proposed classifier predicts the correct algorithmic mode, and SmartPQ is successfully adjusted to the best-performing algorithm delivering on average 1.83 and 1.47 times higher throughput compared to the naive approaches of *alistarh_herlihy* and Nuddle, respectively.

The decision making mechanism added to SmartPQ has negligible overhead. As described in section §3.2, the proposed decision tree is small and has a quite low depth of 8. Given that the traversal cost is a quite low, we call the classifier at short time intervals (every second), such that SmartPQ adapts without any delay to the best performing algorithmic mode, even in cases that the workload changes frequently. The worst case overhead of SmartPQ shown in these figures appears in the last workload of Figure 9a, in the key range of 50M. In this case, SmartPQ delivers only 0.053% lower performance compared to the baseline *alistarh_herlihy* priority queue. To conclude, SmartPQ delivers the highest possible throughput at any point in time adding negligible overhead compared to directly using the corresponding concurrent priority queue.

6 DISCUSSION & FUTURE WORK

In this work, we assume that the workload is known a priori and thus, we can easily extract the features needed for the classifier. As future work, we aim to design a lightweight technique to on-the-fly extract the features needed for the classification and dynamically detect when the workload changes. Specifically, the structure of SmartPQ will be enriched with additional fields that keep track of workload statistics. We will either spawn a background thread to keep track the requests performed on the data structure and update these fields, or all active threads can atomically update them. In frequent time lapses, either the background thread or an active thread will extract the features needed for classification based on the workload statistics, and call the classifier to predict if a transition is needed. Furthermore, SmartPQ can also contain a configuration parameter on how often to collect workload statistics.

In our experimental section, we evaluate SmartPQ using microbenchmarks in order to test SmartPQ under various workloads and contention levels. This microbenchmark-based approach is very common in related research around concurrent data structures [2, 10, 40]. It is however valuable and one of our future research directions to evaluate it in real-life applications. SmartPQ can be used as the underlying concurrent priority queue implementation in real-life applications whose workload changes over time. Examples of such applications are web servers [17, 38], shortest path algorithms [41] and scheduling in operating systems [1].

In these applications, *client threads* do not need to be pinned in hardware threads. However, for our approach to be meaningful *server threads* need to be limited in a single NUMA node. Therefore, at a real-life scenario, the server threads could be created when SmartPQ is initialized and pinned to specific hardware threads.

In this case, the servers are background threads that only accept and serve requests from the various client threads that are part of the high-level application. Moreover, in real-life applications there is a high probability that some clients do not have requests for a long time, while others might continuously request operations. In that case, if we statically associate the client groups with the servers in the initialization step, some servers might remain idle for a long time, while at the same time there are many pending client requests. Consequently, the total performance of the system will be degraded, since some servers will be underutilized. To address this issue, we aim to extend the proposed API with new functions that rebalance the load associated with clients' requests among servers, and *dynamically* associate the client groups with servers.

7 RELATED WORK

Concurrent Priority Queues. Priority queues are widely used in real world applications, which has spawned a range of concurrent data structures, either priority queues [2, 4, 5, 29, 31, 39, 41, 42, 45, 47], or generally skip lists [8, 11, 15, 16, 22, 23, 30]. Recent works [29, 31] designed lock-free priority queues that separate the logical and the physical deletion of an item to improve performance. Alistarh et al. [2] proposed a relaxed lock-based priority queue, called *Spraylist*, in which *deleteMin* returns with high probability, an item among the first $O(p \log 3p)$ items, where p is the number of threads. Apart from these NUMA-unaware approaches, NUMA-aware implementations have also been proposed. Calciu et al. [5] paired combining with elimination. Elimination allows complementary operations (*insert* and *deleteMin*) to complete without accessing the data structure, while combining is a form of delegation. Every thread can become a combiner and combine its own operation with those of other threads. Thus, threads can execute operations without directly accessing the data structure and the memory traffic is reduced resulting to a NUMA-aware design. Finally, Daly et al. [9] proposed an efficient technique to design NUMA-aware skip lists. However, it can only be applied to skip list-based data structures, whereas Nuddle is a general framework to design NUMA-aware data structures.

Black-Box Approaches. Researchers have also proposed black-box approaches that wrap existing data structures to improve their performance. Herlihy [21] constructed a universal method to design wait-free implementations of any sequential object. However, this method remains impractical due to high overheads. Hendler et al. [19] proposed flat combining; a technique that reduces synchronization by executing multiple client requests at once. Despite improvements [20], this technique provides high performance only upon certain data structures (synchronous queues). *FFWD* [40] is another universal approach that uses delegation to avoid memory invalidation traffic in the data structure. However, it is limited to single threaded performance. Calciu et al. [6] proposed a black-box technique, named *Node Replication (NR)*, to obtain concurrent data structures for NUMA machines. In NR, every NUMA node has replicas of the shared data structure which are synchronized via a shared log. Even though both FFWD and NR are general techniques, they could not substitute Nuddle in SmartPQ. Given that they access the data structure with an asynchronized manner, a synchronization point would be necessary to ensure correctness when switching to the NUMA-unaware mode.

Machine learning in Data Structures. Even though machine learning techniques are widely used to optimize modern applications [3, 13, 36, 48], there is a handful of works that employ machine learning to design efficient data structures. Recently, Eastep et al. [12] use Reinforcement Learning to optimize a parameter, called *scancount*, in flat combining. In flat combining [19], every thread publishes its request in a publication list. The combiner thread scans the list to combine these requests. The scancount parameter defines the number of scans the combiner makes over the publication list before returning control to the application. Increasing scancount reduces synchronization costs and improves temporal locality, while on the other hand, the combiner thread remains as a combiner for a long time, which in turn increases the latency of each operation. Therefore, the authors employ machine learning to dynamically tune this parameter. Another recent work [26] indicates that machine learning models can be trained to predict the position or the existence of records in a set of lookup keys, and discusses under which conditions they can outperform the traditional indexed data structures, e.g. B-trees.

8 CONCLUSION

In this paper, we propose an adaptive concurrent priority queue for modern NUMA machines, which provides the highest possible performance at any point in time. We first introduce Nuddle; a low-overhead generic approach that constructs efficient NUMA-aware concurrent data structures, using as underlying representation any arbitrary NUMA-unaware concurrent data structure, and targets data structures with medium contention that can scale up to a number of concurrent threads. Then, we train a classifier to predict the best-performing mode between Nuddle and its underlying NUMA-unaware implementation. Finally, we extend Nuddle and incorporate this decision making mechanism to propose an adaptive priority queue, named SmartPQ, which switches between a NUMA-aware and a NUMA-unaware mode, with no synchronization. Our evaluation shows that SmartPQ is always able to capture the needs of the workload, adapting to the best-performing mode for each workload phase. To the best of our knowledge, this is the first work that proposes a dynamic concurrent data structure, and we believe our work constitutes a foundation for researchers to further investigate how to design dynamic concurrent data structures that perform best under all circumstances.

9 ACKNOWLEDGMENTS

Christina Giannoula is funded for her postgraduate studies from General Secretariat for Research and Technology (GSRT) and Hellenic Foundation for Research and Innovation (HFRI). We would like to thank the anonymous reviewers, and our shepherd, Alexander Heinecke, for their insightful comments and suggestions. We also thank the graduate student, Marina Vemmou, for her suggestions, and our colleague, Athena Elafrou, for valuable discussions and feedback on earlier drafts of this work.

REFERENCES

[1] Lisa A. Torrey, Joyce Coleman, and Barton Miller. 2007. A comparison of interactivity in the Linux 2.6 scheduler and an MLFQ scheduler. *Software Practice and Experience.* 37 (04 2007), 347–364.

[2] Dan Alistarh, Justin Kopinsky, Jerry Li, and Nir Shavit. 2015. The SprayList: a scalable relaxed priority queue. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, San Francisco, CA, USA, February 7-11, 2015.* 11–20.

[3] Akrem Benatia, Weixing Ji, Yizhuo Wang, and Feng Shi. 2016. Sparse Matrix Format Selection with Multiclass SVM for SpMV on GPU. 496–505.

[4] Gerth Stølting Brodal, Jesper Larsson Träff, and Christos D. Zaroliagis. 1998. A Parallel Priority Queue with Constant Time Operations. *J. Parallel Distrib. Comput.* 49, 1 (Feb. 1998), 4–21.

[5] Irina Calciu, Hammurabi Mendes, and Maurice Herlihy. 2014. The Adaptive Priority Queue with Elimination and Combining. In *Distributed Computing*, Fabian Kuhn (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 406–420.

[6] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K. Aguilera. 2017. Black-box Concurrent Data Structures for NUMA Architectures. *SIGPLAN Not.* 52, 4 (April 2017).

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition.*

[8] Tyler Crain, Vincent Gramoli, and Michel Raynal. 2013. No Hot Spot Nonblocking Skip List. In *ICDCS*. IEEE Computer Society, 196–205.

[9] Henry Daly, Ahmed Hassan, Michael F. Spear, and Roberto Palmieri. 2018. NUMASK: High Performance Scalable Skip List for NUMA. In *32nd International Symposium on Distributed Computing, DISC 2018, New Orleans, LA, USA, October 15-19, 2018.* 18:1–18:19.

[10] Tudor Alexandru David, Rachid Guerraoui, and Vasileios Trigonakis. 2015. Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures. *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (2015).

[11] Ian Dick, Alan Fekete, and Vincent Gramoli. 2017. A skip list for multicore. *Concurrency and Computation: Practice and Experience* 29, 4 (2017).

[12] Jonathan Eastep, David Wingate, and Anant Agarwal. 2011. Smart Data Structures: An Online Machine Learning Approach to Multicore Data Structures. In *Proceedings of the 8th ACM International Conference on Autonomic Computing (ICAC '11).*

[13] Athena Elafrou, Georgios I. Goumas, and Nectarios Koziris. 2017. Performance Analysis and Optimization of Sparse Matrix-Vector Multiplication on Modern Multi- and Many-Core Processors. In *46th International Conference on Parallel Processing, ICPP 2017, Bristol, United Kingdom, August 14-17, 2017.* 292–301.

[14] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. 2010. Non-blocking Binary Search Trees. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC '10).* ACM, New York, NY, USA, 131–140.

[15] Mikhail Fomitchev and Eric Ruppert. 2004. Lock-free Linked Lists and Skip Lists. In *Proceedings of the Twenty-third Annual ACM Symposium on Principles of Distributed Computing (PODC '04).*

[16] K. Fraser. 2004. Practical Lock-Freedom. *PhD thesis, University of Cambridge, 2004.* (2004).

[17] Mor Harchol-Balter, Bianca Schroeder, Nikhil Bansal, and Mukesh Agrawal. 2003. Size-based Scheduling to Improve Web Performance. *ACM Trans. Comput. Syst.* 21, 2 (May 2003), 207–233. https://doi.org/10.1145/762483.762486

[18] Tim Harris. 2001. A Pragmatic Implementation of Non-Blocking Linked Lists. In *Proceedings of the 15th International Symposium on Distributed Computing* (proceedings of the 15th international symposium on distributed computing ed.) *(Lecture Notes in Computer Science)*, Vol. 2180. 300–314.

[19] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. 2010. Flat Combining and the Synchronization-parallelism Tradeoff. In *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '10).* ACM, 355–364.

[20] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. 2010. Scalable Flat-Combining Based Synchronous Queues. In *DISC (Lecture Notes in Computer Science)*, Vol. 6343. Springer, 79–93.

[21] Maurice Herlihy. 1991. Wait-free Synchronization. *ACM Trans. Program. Lang. Syst.* (1991).

[22] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. 2007. A Simple Optimistic Skiplist Algorithm. In *Proceedings of the 14th International Conference on Structural Information and Communication Complexity (SIROCCO'07).*

[23] Maurice Herlihy and Nir Shavit. 2008. *The Art of Multiprocessor Programming.*

[24] Shane V. Howley and Jeremy Jones. 2012. A Non-blocking Internal Binary Search Tree. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '12).* ACM, New York, NY, USA, 161–171.

[25] Vladimir Kolmogorov. 2009. Blossom V: A new implementation of a minimum cost perfect matching algorithm.

[26] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18).*

[27] L. T. Schermerhorn. 2007. Automatic Page Migration for Linux [A Matter of Hygiene].

[28] Dominique Lasalle and George Karypis. 2013. Multi-threaded Graph Partitioning. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing (IPDPS '13).* IEEE Computer Society, 225–236.

[29] Jonatan Lindén and Bengt Jonsson. 2013. A Skiplist-Based Concurrent Priority Queue with Minimal Memory Contention. In *Proceedings of the 17th International Conference on Principles of Distributed Systems - Volume 8304 (OPODIS 2013).*

[30] Zhiyu Liu, Irina Calciu, Maurice Herlihy, and Onur Mutlu. 2017. Concurrent Data Structures for Near-Memory Computing. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '17).* ACM, New York, NY, USA, 235–245.

[31] I. Lotan and N. Shavit. 2000. Skiplist-Based Concurrent Priority Queues. In *Proceedings of the 14th International Symposium on Parallel and Distributed Processing (IPDPS '00).*

[32] Romolo Marotta, Mauro Ianni, Alessandro Pellegrini, and Francesco Quaglia. 2016. A Non-Blocking Priority Queue for the Pending Event Set. In *Proceedings of the 9th EAI International Conference on Simulation Tools and Techniques (SIMUTOOLS'16).* ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), ICST, Brussels, Belgium, Belgium, 46–55.

[33] Maged M. Michael. 2002. High Performance Dynamic Lock-free Hash Tables and List-based Sets. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '02).* ACM, New York, NY, USA, 73–82.

[34] Aravind Natarajan and Neeraj Mittal. 2014. Fast Concurrent Lock-free Binary Search Trees. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '14).* ACM, New York, NY, USA, 317–328.

[35] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *J. Mach. Learn. Res.* (2011).

[36] Juan Carlos Pichel and Beatriz Pateiro-López. 2018. A New Approach for Sparse Matrix Classification Based on Deep Learning Techniques. In *CLUSTER*. IEEE Computer Society, 46–54.

[37] R. C. Prim. 1957. Shortest Connection Networks and some Generalizations. *The Bell Systems Technical Journal* 36, 6 (1957), 1389–1401.

[38] Mayank Rawat and Ajay D. Kshemkalyani. 2003. SWIFT: Scheduling in Web Servers for Fast Response Time. In *2nd IEEE International Symposium on Network Computing and Applications (NCA 2003), Cambridge, MA, USA.* 51–58.

[39] Hamza Rihani, Peter Sanders, and Roman Dementiev. 2014. MultiQueues: Simpler, Faster, and Better Relaxed Concurrent Priority Queues.

[40] Sepideh Roghanchi, Jakob Eriksson, and Nilanjana Basu. 2017. Ffwd: Delegation is (Much) Faster Than You Think. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17).* ACM, New York, NY, USA, 342–358.

[41] Konstantinos Sagonas and Kjell Winblad. 2017. The Contention Avoiding Concurrent Priority Queue. In *Languages and Compilers for Parallel Computing.*

[42] Håkan Sundell and Philippas Tsigas. 2005. Fast and Lock-free Concurrent Priority Queues for Multi-thread Systems. *J. Parallel and Distrib. Comput.* (2005).

[43] Wai Teng Tang, Rick Siow Mong Goh, and Ian Li-Jin Thng. 2005. Ladder Queue: An O(1) Priority Queue Structure for Large-scale Discrete Event Simulation. *ACM Trans. Model. Comput. Simul.* 15, 3 (July 2005), 175–204.

[44] Mikkel Thorup. 2003. Integer Priority Queues with Decrease Key in Constant Time and the Single Source Shortest Paths Problem. In *Proceedings of the Thirty-fifth Annual ACM Symposium on Theory of Computing (STOC '03).* ACM, New York, NY, USA, 149–158.

[45] Martin Wimmer, Jakob Gruber, Jesper Larsson Träff, and Philippas Tsigas. 2015. The Lock-free k-LSM Relaxed Priority Queue. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2015).* ACM, New York, NY, USA, 277–278.

[46] Yuming Xu, Keqin Li, and Jingtong Hu. 2014. A genetic algorithm for task scheduling on heterogeneous computing systems using multiple priority queues. *Inf. Sci.* 270 (2014), 255–287.

[47] Deli Zhang and Damian Dechev. 2016. A Lock-Free Priority Queue Design Based on Multi-Dimensional Linked Lists. *IEEE Trans. Parallel Distrib. Syst.* 27, 3 (March 2016), 613–626.

[48] Yue Zhao, Jiajia Li, Chunhua Liao, and Xipeng Shen. 2018. Bridging the gap between deep learning and sparse matrix format selection. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2018, Vienna, Austria, February 24-28, 2018.* 94–108.