# SNUG: Architectural Support for Relaxed Concurrent Priority Queueing in Chip Multiprocessors

Azin Heidarshenas *
heidars2@illinois.edu
University of Illinois at Urbana-Champaign

Tanmay Gangwani *
gangwan2@illinois.edu
University of Illinois at Urbana-Champaign

Serif Yesil
syesil2@illinois.edu
University of Illinois at Urbana-Champaign

Adam Morrison
mad@cs.tau.ac.il
Tel Aviv University

Josep Torrellas
torrella@illinois.edu
University of Illinois at Urbana-Champaign

## ABSTRACT

Many parallel algorithms in domains such as graph analytics and simulations rely on priority-based task scheduling. In such environments, the data structure of choice is a concurrent priority queue (PQ). Unfortunately, PQ algorithms exhibit an undesirable tradeoff. On one hand, strict PQs always dequeue the highest-priority task, and thus fail to scale because of contention at the head of the queue. On the other hand, relaxed PQs avoid contention by dequeuing tasks that are sometimes so far from the head that the resulting schedule misses the benefit of priority-based scheduling.

We propose a novel architecture for relaxing PQs without straying far from the priority-based schedule. Our chip-level architecture, called SNUG, distributes the PQ into subqueues, and maintains a set of *Work registers* that point to the highest-priority task in each subqueue. SNUG provides an instruction that picks a high-quality task to execute. The instruction periodically switches between obtaining an accurate global snapshot, and visiting only local subqueues to reduce traffic. Overall, SNUG dequeues high-quality tasks while avoiding both hotspots and excessive network traffic. We evaluate SNUG on graph analytics and event simulation programs. On a simulated 64-core chip, SNUG reduces the average execution time of the programs by 1.4×, 2.4× and 3.6× compared to state-of-the-art concurrent skip list, SprayList, and software-distributed PQs, respectively.

## CCS CONCEPTS

• **Computer systems organization → Multicore architectures**.

## KEYWORDS

Concurrent Priority Queues; Graph Algorithms; Hardware Acceleration; Multicore Architectures

*Both authors contributed equally to this research.

**ACM Reference Format:**
Azin Heidarshenas, Tanmay Gangwani, Serif Yesil, Adam Morrison, and Josep Torrellas. 2020. SNUG: Architectural Support for Relaxed Concurrent Priority Queueing in Chip Multiprocessors. In *2020 International Conference on Supercomputing (ICS '20), June 29-July 2, 2020, Barcelona, Spain.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3392717.3392740

## 1 INTRODUCTION

Many parallel algorithms in domains such as graph analytics [11, 35, 37], discrete event simulation [15], and machine learning [3] rely on *priority-based* task scheduling. When the algorithm creates a task $T$, it assigns it a priority $p$, and if $T_1.p < T_2.p$, then $T_1$ should execute before $T_2$. (That is, lower $p$ values mean higher priorities.) The data structure commonly used to implement priority-based task scheduling is the *concurrent priority queue* (PQ). A PQ maintains a collection of items, each associated with a priority. It supports two basic operations: *enqueue*, which adds an item to the correct position in the queue, and *dequeue*, which removes the item with the highest priority from the head of the queue.

Unfortunately, concurrent PQs are not scalable. Since every thread executing *dequeue* tries to remove the same highest-priority item, the head becomes a synchronization hotspot [8, 21, 30–32, 43]. The resulting serialization and synchronization overheads can dominate execution time.

To avoid this problem, researchers have proposed to *relax* PQ semantics and allow *dequeue* to return an item that is not the highest-priority one [2, 39, 47, 48]. Relaxed PQ algorithms use various strategies to find the item to dequeue. For example, they perform a short random walk on a skip list to find an item to dequeue [2], or pick the highest-priority item between a thread-local PQ and a global shared PQ [47]. These strategies alleviate the bottleneck at the head.

Relaxed PQ algorithms face the danger of straying too much from the desired task execution order and ending-up performing *wasted work*. For example, in discrete event simulation, a thread may process an event that is far in the future, only to have to reprocess it again later, as new events that occur from now until that time change the system state. Thus, the key difficulty in designing a relaxed PQ is to consistently return high-priority items.

In practice, existing relaxed PQs often fail to achieve this goal. For example, the SprayList [2] returns an item among the first $O(t \log^3 t)$ ones in the PQ with high probability, where $t$ is the number of threads. For a 64-thread execution, this translates to a weak guarantee of returning an item within the first $13,824$ in the queue (ignoring constant factors). Similarly, with the recommended parameter $k = 256$, the $k$-LSM PQ [47] only guarantees returning an item within the first 16, 384 in the queue in a 64-thread execution. Current priority-based task scheduling algorithms thus pose a synchronization vs. work-efficiency tradeoff: alleviating the synchronization hotspot by using relaxed PQs leads to wasted work.

This paper introduces novel hardware support to address this tradeoff. Our chip-level architecture, called Snug, relaxes the priority order in a PQ algorithm slightly, to alleviate contention while inducing, on average, little wasted work. Snug distributes the PQ into subqueues, and maintains a set of *Work registers* that point to the highest-priority task in each subqueue. A new *PickHead* instruction returns a high-priority task from the combined PQ.

*PickHead* employs an adaptive technique to pick a high-priority task without congesting the network. Sometimes, *PickHead* reads all Work registers in parallel, saves the result, and returns a random task from within the *R* highest-priority ones observed. *R* is called the *Relaxation Count*, and is dynamically adapted by Snug, based on the rate of synchronization failures—which indicate the degree of contention. In later *PickHead* invocations, *PickHead* reuses the saved information to avoid rereading the Work registers. Finally, *PickHead* sometimes reads only from a set of local Work registers, to save traffic. Overall, *PickHead* performs high-quality task selection while avoiding hotspots, minimizing wasted work, and consuming acceptable network bandwidth.

**Snug Effectiveness.** We evaluate Snug on a simulated 64-core chip using graph and discrete event simulation applications with various inputs. We compare the execution time of the applications using Snug and using several other PQ algorithms. Such algorithms include software-only PQs based on a concurrent skip list, SprayList, and distributed skip list; and a hardware-based centralized PQ. Snug reduces the average execution time of the applications by 1.4×, 2.4× and 3.6× compared to state-of-the-art concurrent skip list, SprayList, and software-distributed PQs, respectively. Compared to the latter two relaxed PQ designs, Snug reduces the number of wasted tasks by 3.3× and 36.8×, respectively.

**Contributions.** We make the following contributions:
• The Snug architecture, which consists of the *PickHead* instruction, the Work registers, and other ISA and microarchitectural extensions. This design adapts its degree of relaxation dynamically.
• Simulation-based evaluation of Snug using graph and discrete event simulation applications, and a comparison to several other software- and hardware-based PQs.

## 2 MOTIVATION AND BACKGROUND

### 2.1 Need for Priority-Based Task Scheduling

We target parallel algorithms that benefit from *priority-based scheduling*. Such algorithms decompose work into tasks (which can generate new tasks as they run) and execute them in order according to some notion of priority. Processing a task out of priority order leads to *wasted work*, where a task executes inefficiently and/or the results of its computation are thrown away later.

Many parallel algorithms in a wide range of domains exhibit the above structure. To name some examples, in discrete event simulation, Time Warp [15, 22] optimistically executes simulated events (i.e., tasks) in parallel, rolling back events that are discovered to have been simulated before their dependencies are satisfied. Priority-based scheduling minimizes such rollbacks. In machine learning, task execution order significantly impacts convergence time in residual belief propagation [3, 13], an algorithm for performing inference on graphical models. Parallel algorithms for important graph problems, such as Single-Source Shortest-Paths (SSSP) [11, 35], Minimal Spanning Tree (MST) [6, 37], and Betweenness Centrality [7], also leverage priority-based scheduling.

Here, we use SSSP as a running example. SSSP is a classic graph problem that is used in many domains. SSSP is also a building block for computing Betweenness Centrality which, in turn, has wide application in network theory, social networks [49], and biology [50].

**The SSSP Example.** Given a weighted directed graph and a *source* node $s$, SSSP finds the weight of the shortest path from $s$ to every other node in the graph. Most SSSP algorithms use *relaxations* [10], in which the algorithm tests whether a shortest path found so far can be improved. Each node $v$ is associated with a $dist(v)$ label (initially 0 for $s$ and $\infty$ for all other nodes). A relaxation considers an edge $(u, v)$ of weight $w$. If $dist(v) > dist(u) + w$, the algorithm updates $dist(v)$ to be $dist(u) + w$. Updates of a node's label are synchronized (e.g., with atomic instructions), allowing relaxations to run in parallel. The algorithm thus converges to the labels containing the weights of the shortest paths from $s$.

Any relaxation that does not update a node's label to its true distance is *wasted work*, as it will be overwritten by the relaxation that updates the label to the true distance. Dijkstra's SSSP algorithm [10, 11] relaxes each edge exactly once. It partitions the graph into *explored* nodes, whose distance from $s$ is known, and unexplored nodes. In each iteration, it picks the unexplored node $u$ with the smallest label, marks it explored, and relaxes every edge $(u, v)$.

**Priority-based Task Scheduling.** By using node labels as *priorities* and defining a task as relaxing every outgoing edge of a node, we can approximate Dijkstra's algorithm in a multiprocessor and minimize wasted work. Note that due to parallelism, wasted work is not guaranteed to disappear. This is because two tasks may perform conflicting relaxations.

### 2.2 Concurrent Priority Queue Algorithms

Concurrent Priority Queues (PQs) are natural data structures to implement priority-based task scheduling. A PQ maintains a collection of items (i.e., tasks or nodes), each associated with its own priority $p$. We consider lower $p$ values to be higher priorities. A PQ supports two operations: *enqueue*, which adds an item to the collection in its correct position, and *dequeue*, which removes the item with the highest priority from the collection and returns it.

PQs suffer from scalability problems, due to synchronization hotspots resulting from the fact that every thread executing *dequeue* tries to remove the same, highest-priority item. Motivated by this scalability problem, researchers have proposed *relaxed* PQs, which allow a *dequeue* to return a task that is not the highest-priority one [2, 39, 47, 48]. Relaxed PQs alleviate the synchronization hotspot, but increase the amount of wasted work. As a result, the performance of the application may improve or degrade with a relaxed PQ.

**Contention in Standard PQs.** Modern PQs [8, 14, 30, 32, 43] are implemented based on the *skip list* data structure [38]. A skip list is conceptually a sorted linked list in which some nodes contain "hints" that enable searching in logarithmic time. PQs implement *enqueue* by inserting an item into the skip list (which is sorted by priority), and *dequeue* by removing the head item.

For simplicity, we explain the PQ synchronization issues using the simpler *linked list*-based PQ, rather than the skip list-based PQ, and cover skip lists in the Appendix. Linked-list insertions are performed by searching the sorted list for the correct place to insert the new item, and linking a new node there. Searching is done using only reads, without acquiring locks [17], and so *enqueue* operations can proceed in parallel and scale well.

Linked-list deletions are done in two phases [17, 34]: the node is first *logically* deleted by setting a "deleted" flag in the node, and then *physically* deleted by updating the *next* pointer of its predecessor. We explain the details in the Appendix. Both of these steps involve synchronization, either through locks or atomic Compare-And-Swap (CAS) instructions. Crucially, threads concurrently performing a *dequeue* all attempt to delete the same node, resulting in a significant synchronization bottleneck. PQ research has focused on reducing the impact of this bottleneck—e.g., by batching physical deletions [30]—but inevitably hits a scalability limit.

**Relaxed PQs.** Relaxed PQs eliminate the bottleneck at the PQ head by distributing *dequeue* operations, often using randomization. The SprayList [2] is based on a skip list, but a *dequeue* chooses its target item by performing a short random walk on the list. MultiQueues [39] and Sheaps [4, 29] distribute the data structure, composing the logical PQ out of per-thread PQs. Dequeues are performed from a remote queue only if the local one is empty [29], or from a queue chosen randomly [39]. Other approaches combine per-thread PQs with a global PQ [47].

Relaxed PQ designs trade off synchronization costs with increased probability of wasting work, since the tasks returned by a relaxed PQ may be far from the highest-priority one.

## 3 THE SNUG ARCHITECTURE

### 3.1 Main Idea

Our goal is to design architectural support for a PQ that minimizes both wasted work and synchronization overhead. We accomplish it with SNUG, which is a novel chip-level architecture for high-performance, distributed PQs. SNUG exists in the context of a directory-based manycore. Each core (or core cluster) owns a module of the distributed directory. Any programmer-declared logical queue is distributed into multiple physical queues. Each physical queue's head is in a different directory module.

When a thread running on a core calls the *enqueue* operation, the PQ library leverages the SNUG hardware to enqueue the node at the correct spot in the core's *local* physical queue. When the thread calls the *dequeue* operation, the PQ library leverages the SNUG support to return to the thread a node with one of the highest priorities in the distributed PQ. As we will see, the hardware uses a special algorithm, so as to minimize both wasted work and synchronization overhead, and avoid excessive traffic. Overall, with SNUG, enqueues are fast because they are local, while dequeues are both fast and return high-priority nodes thanks to special hardware.

Figure 1 shows the SNUG architecture. In a tiled manycore, each node has two hardware structures: a set of *Work Registers* in the directory module, and a *PickHead Module* in the core. Each Work register can function as the head of a work queue. Cores can access all the Work registers in the chip as memory-mapped locations in an uncacheable virtual address range. When a Work register is used, it has two values: a pointer to the first node in the corresponding queue, and that node's priority. Any core in the chip can read a Work

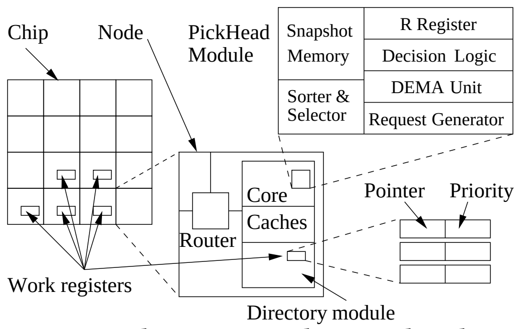


**Figure 1: SNUG architecture in a directory-based manycore. Each directory module has a set of Work registers, and each core has a *PickHead* module.**

register without causing ping-ponging of cached data across the network. In addition, there are instructions that read both pointer and priority together, and that modify both pointer and priority atomically.

When a thread calls the *dequeue* operation, the PQ library issues a new instruction called *PickHead*. The instruction may access multiple queues, only the local queue, or no queue at all. For each of the queues accessed, the network transaction returns the information in the Work register.

The *PickHead* module in Figure 1 supports the *PickHead* instruction. When the instruction visits multiple queues, it issues parallel requests that read each of the relevant Work registers in parallel. The values returned (pointer to the node at the head and its priority) are stored in a small *Snapshot Memory* in the module. Then, *PickHead* does *not* return to the software the pointer to the very highest-priority node in the Snapshot memory. Doing so would cause all the threads to attempt a CAS on the same node. Instead, the SNUG hardware sorts the pointers to the nodes in the Snapshot memory based on their priority, considers only the top $R$ of them, and picks one of them at random to return to the software. The software will then attempt to perform a CAS on the node.

$R$ is the *Relaxation* count. It is stored in the *R Register* of the *PickHead* module (Figure 1) and is dynamically adapted by SNUG based on the rate of synchronization failures.

Sometimes, *PickHead* chooses to visit no queue. It simply reuses information in the Snapshot memory, returning another of the node pointers there. This choice saves traffic, and is selected a few times right after the Snapshot memory is refreshed. However, using this choice several times in a row risks using stale data, which will result in the failure of the subsequent CAS operation.

Finally, *PickHead* sometimes chooses to visit only the local queue. This choice triggers a cheap transaction, which induces minimal traffic and contention. However, it may or may not provide a pointer to a high-priority node to dequeue. This choice is selected several times after multiple entries from the Snapshot memory have been used, to reduce traffic pressure.

After *PickHead* returns a pointer to a node, the library attempts to dequeue the node with a CAS. If it fails, it calls *PickHead* again, and the process repeats until a node is successfully dequeued.

### 3.2 PickHead Instuction and Module

The goal of the *PickHead* instruction is to return a pointer to one of the highest-priority nodes in the PQ with minimal overhead. The instruction uses the *PickHead* module shown in Figure 2. As the instruction starts executing, a Decision Logic module makes one of three choices: issue a Global access to multiple Work registers, issue a local access, or issue no network access at all.

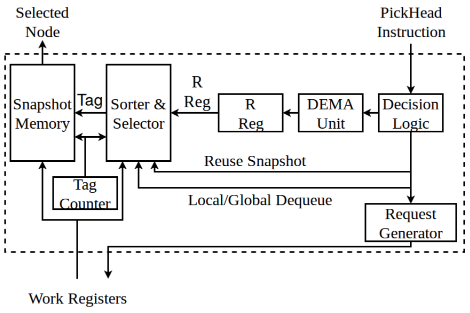


**Figure 2: *PickHead* module.**

**Global Access.** Under certain conditions, the *PickHead* instruction issues as many network requests as there are physical queues in the requested logical queue. These requests proceed in parallel, each visiting a queue and returning a 3-tuple to the Snapshot memory and Sorter & Selector module (Figure 2). A tuple has the queue ID (i.e., the virtual address (VA) of the queue), and the two contents of its Work register. Now, the *PickHead* module needs to pick one of these node pointers to return to the software, which will then perform a CAS to attempt to dequeue the node from the corresponding queue.

As indicated above, if the chosen node was always the highest-priority one, we would induce high PQ contention. Hence, the Sorter & Selector module (Figure 2) sorts the node pointers as they arrive in decreasing priority, and then considers the subset of them that have the highest priorities. This subset is called the *Inclusion Set*. Its size is equal to the Relaxation count stored in the R register (Figure 2). Finally, the hardware randomly selects one of the node pointers in the Inclusion Set and returns it to the software. Section 4.2 describes this step.

To set *R*, we would ideally use feedback from the contention for the PQ and the amount of wasted work performed by the application. Unfortunately, wasted work is very application dependent, and programmers may find it difficult to estimate. On the other hand, PQ contention is easy to measure. Hence, in Snug, the *PickHead* instruction takes an operand (*PrevFailed?*) that is set if the CAS on the node provided by *PickHead* in the previous Global access failed. This operand is set and reset inside the PQ dequeue library. In general, if the frequency of failures of CAS operations is high, we increase *R*; if it is low, we decrease *R*. The Double Exponential Moving Average (DEMA) unit (Figure 2) uses the CAS success/failure history to set *R*. Section 4.3 explains the algorithm used.

**Local Access.** In this case, *PickHead* accesses only the local queue. This operation is inexpensive, but can potentially obtain a pointer to a node with a lower priority than if we performed a Global access. The returning data bypasses the Snapshot memory and Sorter & Selector, and is returned to the software.

In networks organized in clusters, this transaction could involve *PickHead* accessing all the queues in the local cluster. If so, this case would proceed as explained for the Global access: the multiple responses would be received in the Snapshot memory and sorted, and one node pointer from the highest-priority ones would be returned to the software.

To balance the desire to reduce the global traffic and to pick high-priority nodes, we design *PickHead* to perform a fixed number of local accesses between any pair of consecutive Global accesses. Section 4.1 explains the algorithm used.

**No Network Access.** After a *PickHead* performs a Global access, several subsequent *PickHead* executions reuse the information in the Snapshot memory. The goal is to reduce the network traffic without hurting the quality of node selection much. Specifically, a prior Global access brought node pointers from all the queues, sorted them, picked one, and consumed the corresponding node. Now, the *PickHead* instruction reuses the sorted array of pointers in the Sorter & Selector module as follows. The hardware re-considers all the non-consumed entries in the Sorter & Selector module, and randomly selects one to return to the software. That entry is then marked as consumed. Note that the algorithm is otherwise not selective in what nodes to re-consider—this is to maximize the chances of attaining an available node. This process is repeated a few times before the snapshot is discarded. Section 4.1 presents the algorithm.

## 3.3 Interface to the Software

The Snug hardware is accessible with the API in Table 1. The table lists the inputs and outputs of each of the four Snug operations. While different implementations are possible, we envision *PickHead*, *UpdateHead*, and *FetchHead* to be actual instructions, with their operands placed in registers—some encoded in the instruction, and some used implicitly. The other operation, *AllocHeads*, is a subroutine that invokes the memory manager.

*AllocHeads* allocates Work registers. It takes two inputs and one output. The inputs are the number of programmer-visible logical queues to allocate, and an array with the number of physical queues that each logical queue should have. The physical queues of each logical queue are distributed into as many different directory modules as possible. *AllocHeads* returns the virtual addresses (VAs) of all the physical queues allocated—i.e., the VAs of all the Work registers allocated.

| |
|---|
| *AllocHeads*<br>input: # of logical queues<br>input: Array with the # of physical queues for each logical queue<br>output: VAs of all the physical queues allocated |
| *PickHead*<br>input: ID of the logical queue<br>input: Did the CAS after previous Global access to the queue fail?<br>output: VA of the chosen physical queue to dequeue from<br>output: Pointer to the node at the head of the chosen queue |
| *UpdateHead*<br>input: VA of the physical queue<br>input: Pointer to the node at the head of the physical queue<br>input: Pointer to the node to place at the queue's head<br>input: Priority of the node to place at the queue's head<br>output: Successful or failed outcome |
| *FetchHead*<br>input: VA of the physical queue<br>output: Pointer to the node at the head of the physical queue |

**Table 1: API of the Snug hardware. VA means virtual address.**

*PickHead* was described in Section 3.2. It takes as inputs the ID of a logical queue and a boolean that indicates if the CAS on the node provided by *PickHead* in the previous Global access to the queue failed. The outputs are the VA of the chosen physical queue to dequeue from, and a pointer to the node at the head of that queue. In the worst case, the latency of this instruction is that of multiple

```
1  void enqueue( int LogQueID, Node *new) {
2      Node **head = &queue[LogQueID][local];  // local queue
3      // head is the uncacheable address of the head of the queue
4      while (true) {
5          Node **prev = head;
6          Node *curr = FetchHead (head);  // get node at head of queue
7          while (new->priority > curr->priority) {  //high num is low prio
8              prev = &curr->next;
9              curr = curr->next;
10         }
11         // insert node between prev and curr
12         new->next = curr;
13         if (prev == head)
14             ok = UpdateHead (head, curr, new, new->priority);
15         else
16             ok = CAS(prev, curr, new);
17         if (ok)
18             return; // success
19 }   }
```

**(a) Enqueue**

```
20 PrevFailed? = no; // local variable, initially false
21 void* dequeue( int LogQueID) {
22     // " first " will get information on the node to dequeue:
23     // VA of its physical queue, and a pointer to the node
24     struct {
25         Node **physqueue;
26         Node* head;
27     } first ;
28     while (true) {
29         first = PickHead (LogQueID,PrevFailed?); // return node
30         if (first.head == NULL)
31             return NULL; // empty
32         Node *next = first.head->next;
33         success = UpdateHead (first.physqueue, first.head, next, next->priority );
34         if (Global)
35             PrevFailed? = (success ? no : yes);
36         if (success)
37             return first.head;
38 }   }
```

**(b) Dequeue**

**Figure 3: Code to enqueue (a) and dequeue (b) a node with Snug. Recall that enqueues are always local.**

uncached accesses in parallel to directory modules, plus sorting the incoming node priorities in hardware.

*UpdateHead* performs a CAS to modify a Work register and, hence, change the head of the corresponding physical queue. It changes the two fields of the Work register—i.e., the pointer to the head node and its priority—atomically. It is used in both the *enqueue* and *dequeue* PQ library operations. *UpdateHead* takes four inputs (Table 1): the VA of the physical queue, the expected value of the pointer to the node at the head of the queue, a pointer to the node that the instruction wants to place at the head of the queue, and that node's priority. If *UpdateHead* succeeds, both pointer and priority in the Work register are updated; if it fails, no change is made. *UpdateHead* returns a boolean with the outcome of the operation.

This operation is performed in a CAS hardware unit in the directory module. In this way, operands do not have to flow from the directory module to the core and back. A similar unit is provided in current GPUs to perform CASes in the cache hierarchy [27, 36]. Also note that, with this instruction, Snug can perform arbitrary writes to the queue head. Overall, the overhead of this instruction is an uncached access to a directory module that includes a CAS operation.

*FetchHead* reads the pointer field of a Work register and, thus, obtains the head of a physical queue. It is used as part of the *enqueue* library, which needs to check the current value of the queue head to be able to change it with *UpdateHead*. *FetchHead* takes as input the VA of the physical queue. Its output is a pointer to the node at the head of the queue. Its overhead is an uncached access to a directory module.

## 3.4 Enqueue and Dequeue Operations

The previous instructions are not typically used directly by programmers. Instead, they are used in the PQ library to allocate queues, and to enqueue and dequeue nodes. Figures 3(a) and (b) show pseudo-code for the routines that enqueue and dequeue a node, respectively. Programs that call the PQ library do not know about physical queues; they reference logical queues.

For simplicity, Figures 3(a) and (b) focus on the physical deletions. We omit the details of handling *logical* deletions (Section 2.2), which are orthogonal to Snug. In addition, the figures use simple linked lists. In practice, high-performance concurrent PQ libraries use the more efficient skip list [32].

The *enqueue* routine (Figure 3(a)) is called with the ID of a logical queue and a node to enqueue. The routine first determines the VA of the local physical queue (Line 2); this is the queue where the node will be enqueued. The routine then uses *FetchHead* to read the pointer to the node at the head of the queue (Line 6). It then follows the linked list of nodes, reading the priority of each node, to find the place to insert the new node (Lines 7-10). If the node needs to be placed at the head, it uses *UpdateHead* to do so and fill the Work register (Line 14). Otherwise, it uses a plain CAS (Line 16). Either *UpdateHead* or CAS can fail; if it does, the routine goes back to walking the queue to find where to enqueue.

The *dequeue* routine (Figure 3(b)) is called with the ID of a logical queue. It returns one of the highest-priority nodes from the logical queue (not necessarily the highest one), by dequeueing it from the appropriate physical queue. The routine uses *PickHead* to find the VA of the physical queue and a pointer to the node (Line 29). Then, the routine tries to dequeue the node using *UpdateHead* on the appropriate physical queue (Line 33). If this was a Global access, the routine sets *PrevFailed?* based on whether *UpdateHead* succeeded or failed (Line 35). *PrevFailed?* will be used the next time that *PickHead* performs a Global access. In any case, if *UpdateHead* succeeds, the routine returns a pointer to the dequeued node (Line 37). Otherwise, the routine repeats the use of *PickHead* and *UpdateHead* until the dequeue succeeds or there is no node to dequeue.

## 4 DETAILED ASPECTS OF SNUG DESIGN

### 4.1 Algorithm to Pick the Node to Process

Snug's algorithm for picking the next node to dequeue maintains a balance between two objectives. On one hand, Snug wants to observe globally up-to-date information about the PQ, so that it can pick a high-priority node and minimize wasted work. On the other hand, Snug also wants to avoid generating excessive network traffic, which would be the case if all *PickHead* invocations visited all the Work registers. As per Section 3.2, Snug achieves this balance by complementing Global accesses with (i) reuses of Snapshot memory information, and (ii) accesses to the Local queue.

The Decision Logic unit in the *PickHead* module (Figure 2) divides the stream of *PickHead* instruction executions into *phases*. Each phase begins with a *PickHead* instruction that performs a Global access, followed by *U PickHead* executions that reuse the snapshot, and finally *L PickHead* executions that access the local queue.

A Global access reads many pointers to nodes, each of which is at the head of a physical queue. One of these pointers is returned to the software. However, many of the pointers read by the Global access are good candidates for processing. Hence, in the next $U$ calls to *PickHead*, we want it to return some of these nodes. Eventually, the snapshot data becomes stale, as other cores have dequeued nodes from the various queues. Therefore, after $U$ *PickHead* executions, SNUG discards the snapshot.

SNUG then switches to visiting only the local queue for $L$ times. This decision trades off the quality of the executed work for short periods, in order to avoid excessive network traffic. If the network is organized in clusters, each of these $L$ *PickHead* executions could access all of the local queues in the cluster, and return one of the best nodes in them. In any case, following these $L$ local accesses, a new phase begins, and the next *PickHead* triggers a Global access.

## 4.2 Sorting the Nodes

The Sorter & Selector module sorts the 3-tuple responses as they arrive from memory. We use an inexpensive sorter that performs *pipelined linear insertion sort* (Figure 4). As each of the 3-tuples arrives, a Tag Counter (Figure 2) assigns it a tag. Then, the incoming priority and tag are fed to the Sorter & Selector module, where the earlier arrivals are already sorted in registers. The incoming priority is then compared to the existing priorities in sequence, until the hardware determines its correct position (Figure 4). At that point, the incoming priority and tag are stored in the corresponding register, and all remaining existing priorities are shifted one position to the right. Details of a similar system can be found in [33].

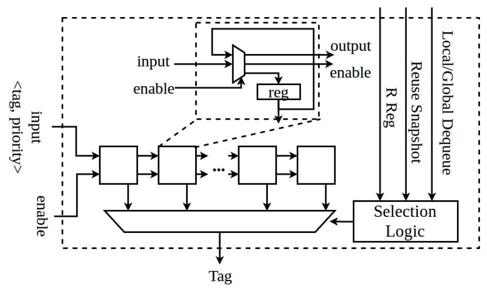


**Figure 4: Sorter & Selector module.**

The hardware complexity of this module and the sorting time scale linearly with the number of physical queues. Specifically, for $n$ queues, the sorting network has $n$ comparators. When the last tuple is received, it takes $n$ steps to complete the sorting. Our analysis with design tools estimates that each step takes 1ns, or 2 processor cycles. Hence, sorting takes $2n$ processor cycles after the last tuple arrives.

After the sorting, some selection logic reads a register that contains the current time and performs a modulo operation of the time with the current value of $R$. The result is an index that the module uses to select one of the top $R$ entries from the sorted array of priorities (Figure 4). The chosen tag is then passed to the Snapshot Memory, and used to read the corresponding physical queue ID and node pointer, which are then passed to the software. The sorted list of priorities is kept latched in the Sorter & Selector module for reuse by future *PickHead*s.

## 4.3 Setting the Relaxation Count (R)

The Relaxation count ($R$) is a parameter that determines the quality of the nodes obtained by the Global accesses and the No-network accesses. It is defined as the maximum number of sorted nodes in the Sorter & Selector module from which one will be returned to the software. If $R$ is too high, it may induce wasted work; if it is too low, it may cause *UpdateHead* failures in the *dequeue* routine (Figure 3(b)) because many cores will collide. To set $R$, we use real-time feedback from the application on how frequently these *UpdateHead* operations fail. Since only a *PickHead* that performs a Global access creates a fresh snapshot, we use the failure rate of only the *UpdateHead* operation immediately following a Global access, to decide how to change $R$. The value of $R$ serves as an indicator of the global contention in the system.

While the optimal $R$ is likely to change across the execution time of an application, it is important that its value be impervious to short-term fluctuations of *UpdateHead* failure rate, and instead reflect long-term trends. For this reason, we use the Double Exponential Moving Average (DEMA), which is based on the Exponential Moving Average (EMA) [45], a widely used indicator in statistical technical analysis. DEMA smoothing is preferred over EMA smoothing, especially when the series exhibits some trends.

As shown in Figure 2, the *PickHead* module includes a DEMA Unit, which receives information from the Decision Logic on how frequently *UpdateHead* failures occur for a Global access. The information is a single bit: 1 for failure, 0 for success. Intuitively, observing many occurrences of 1 suggests that $R$ is too small, while observing many 0 values suggests it is too high.

We define a *segment* as a series of bits of the same value (either 1 or 0) passed by the Decision Logic to the DEMA unit. On a segment termination, the DEMA unit uses the length of the segment to compute the following:

$$EMA = \alpha * sign * length + (1 - \alpha) * EMA$$
$$DEMA = \beta * EMA + (1 - \beta) * DEMA$$

where *length* is the number of entries in the segment, and $\alpha$ and $\beta$ are constants. The variable *sign* is 1 for a segment of 1s and -1 for a segment of 0s. With this setup, *UpdateHead* failures tend to push the DEMA slowly in the positive direction, while a string of *UpdateHead* successes move it in the opposite direction. Note that the DEMA changes only slowly.

When the application starts, the DEMA unit uses a default initial value $R_0$. During execution, the unit divides the time into *windows* of fixed size, and keeps calculating the DEMA. It keeps a positive DEMA threshold ($T_{pos}$) and a negative one ($T_{neg}$). If the DEMA has been above $T_{pos}$ anytime in each of the previous two windows, the DEMA unit increases $R$ one notch; if the DEMA has been below $T_{neg}$ anytime in each of the previous two windows, $R$ decreases by one notch. Figure 5 shows an example of this algorithm.

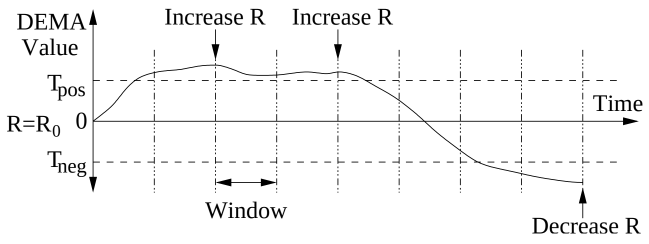


**Figure 5: Example of how SNUG sets *R*.**

Since each core computes its DEMA value independently, each core modulates its own $R$ independently. Interestingly, relaxation by one core has the serendipitous effect of reducing contention for sibling cores. As such, in most scenarios, some cores converge on smaller $R$ values than others.

$R$ changes as the application executes different sections, and as different applications execute. However, this process is invisible to the programmer. After each context switch, $R$ is reset to $R_0$.

## 4.4 Multi-Socket Configuration

SNUG does not have any centralized hardware and relies on hardware cache coherence. Consequently, it can also be implemented in a cache-coherent multi-socket system. In such a system, the *PickHead* requests may need to obtain the data from remote sockets, like regular loads. The software needs no changes.

## 5 EVALUATION ENVIRONMENT

### 5.1 Architecture Modeled

We perform our evaluation with cycle-level simulation of a 64-core chip using the gem5 simulator [5]. Table 2 shows the baseline architecture modeled. We model a hierarchical network with clustering: every 8 cores share a single L2, and the 8 cache-coherent L2s are connected to a shared, 8-banked L3 with a crossbar.

| Parameter | Value |
|---|---|
| Architecture | 64 cores on chip, 2GHz cores |
| Private L1 Caches | 32KB-I, 32KB-D WB, 8-way, 2 cycles hit latency |
| Per-cluster L2 Cache | 1MB WB, 16-way, 12 cycles hit latency |
| Shared L3 | 16MB WB, 8 banks, 16-way, 30 cycles hit latency |
| Cache line size | 64B |
| Coherence | Two-level MOESI directory protocol |
| Network | 32B wide, hierarchical, crossbar with snoop filters |
| Main memory | $\approx$ 200 cycles |
| SNUG Parameters | |
| Reuses ($U$); Local acc. ($L$) | 4; 4 |
| $R_0$; DEMA Window | 32; 100K cycles |
| DEMA thresholds/constants | $T_{pos}$ = 5, $T_{neg}$ = -2.5, $\alpha$ = 0.6, $\beta$ = 0.6 |
| Sorting network delay | 64 ns = 128 cycles |

**Table 2: Parameters of the architecture evaluated.**

The latency of the new instructions is modeled as follows. *FetchHead* is an uncached access to one directory module. *UpdateHead* is like *FetchHead* plus 4 cycles for a CAS. *PickHead*'s latency depends on the operation: in a Global access, it is $n$ parallel uncached accesses to directories, plus $2n$ cycles to sort the incoming messages (Section 4.2) plus 4 cycles to return the node pointer to the instruction; in a Local access, it is one uncached accesses to a directory plus 4 cycles to return the node pointer to the instruction; in a No-network access, it is 4 cycles to return the node pointer to the instruction. *AllocHeads* is invoked before the section of the application that is timed.

### 5.2 Concurrent PQs Compared

We compare the following concurrent PQs:

**Concurrent Skip List (*SW-SK*).** This is a skip list implementation [30] where the levels for new skipnodes are chosen based on a geometric distribution. The maximum number of levels is 24.

**Concurrent Spraylist (*SW-SP*).** This SprayList builds on top of the skip list by spraying the pops over a range of starting nodes in the list. We use the SprayList parameters as advised by the authors in [2]. The spray is started at the height of $\lfloor \log_2 t \rfloor + 1$, and the jump length at each level is $\lfloor \log_2 t \rfloor + 1$, where $t$ is the number of threads. The number of levels to descend between jumps is 1, and the maximum number of levels is 24.

**Distributed Software (*SW-D*).** This is a software distributed PQ with a per-core skip list. Threads always enqueue to their local skip list. For dequeue, the local skip list is tried first. If the queue is empty, the thread attempts to steal work from nearby skip lists.

**Centralized Hardware (*HW-C*).** This is a centralized version of SNUG. It consists of a single shared skip list with a single, centralized Work register. It uses *FetchHead* and *UpdateHead* to access the Work register. A single Work register obviates the need for *PickHead*.

**Distributed Hardware (*HW-D*). This is** SNUG. It uses per-core skip lists, each of which is supported by a Work register at the directory. Threads always enqueue to their local queue. For dequeue, a *PickHead* instruction returns the node to dequeue as described in Section 4.1. For the $L$ local accesses, *PickHead* visits the local queue. However, if the data in the Snapshot memory indicates that the local queue is empty, *PickHead* instead randomly visits one of the queues that the Snapshot memory indicates are not empty. Table 2 lists SNUG's parameters.

**Software Version of** SNUG. In this case, a global dequeue scans in software all the heads of the distributed queues. This PQ is an order of magnitude slower than *SW-D*, due to the overhead of serially scanning all the queue heads. Therefore, we omit it from the evaluation.

**Omitted PQs.** We have also evaluated the OBIM priority-based scheduler from Galois [35]. OBIM trades off synchronization time in exchange for executing tasks out of priority order and potentially performing unnecessary work. OBIM has been tuned for multi-socket platforms, which have sizable synchronization costs. For the manycore architecture considered in our paper, with a single chip and very low synchronization costs, we find that OBIM sometimes performs substantially worse than most of the other PQ implementations—even after careful parameter tuning. Hence, since we are unable to tune OBIM well for our hardware, we do not include it in the evaluation.

We also do not include a comparison to the $k$-LSM PQ [47]. $k$-LSM is a two-level software PQ where the access frequency to the top-level PQ is controlled by the $k$ parameter. As $k$ grows, fewer accesses go to the top-level queues and the quality of returned tasks decreases. We do not present the $k$-LSM PQ because we found that on our inputs, SW-SK is on average 1.45× faster than $k$-LSM (even for the best value of $k$ we found, which is $k = 256$) in experiments on a real machine with 64 threads. We remark that prior work [46] only evaluated the $k$-LSM PQ on a random graph input, whereas we use more realistic inputs, as described below.

### 5.3 Applications Evaluated

We execute four applications with a variety of inputs. The applications come from graph analytics (SSSP, BFS, and A*), and from an event-driven simulation model (SIMUL).

**Graph Analytics.** SSSP uses Dijkstra's algorithm [10] to compute the shortest distance to all graph nodes, starting from a source node (Section 2.1). We base our implementation on the push-operator [35], since we found it to outperform the pull-operator based approach. Breadth-First Search (BFS) uses breadth-first search in graphs where the weight of all the edges is 1, which drastically changes the task scheduling behavior compared to SSSP. A-star (A*) is a pathfinding algorithm used to compute the shortest distance from a source to a target node. It relaxes only a subset of the graph nodes, and uses heuristics to guide the searching process.

**Event-driven Simulation.** SIMUL is a system-modeling program with a variety of use cases [20, 25]. We model the execution of

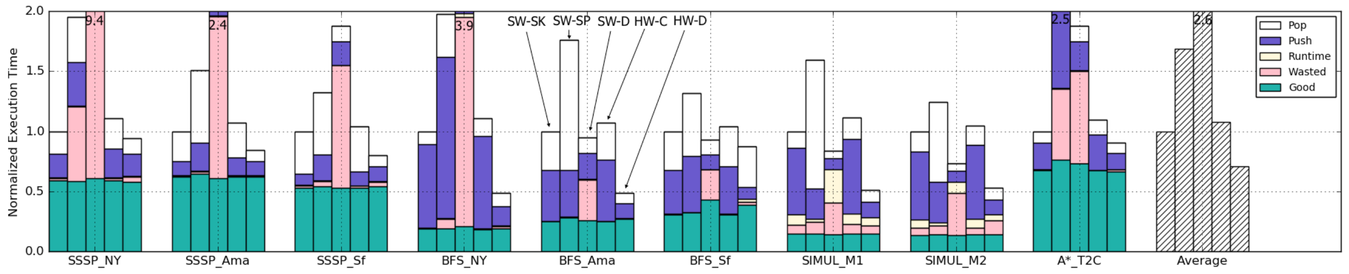


**Figure 6: 64-core execution time of the evaluated applications and inputs on different priority queue implementations.**

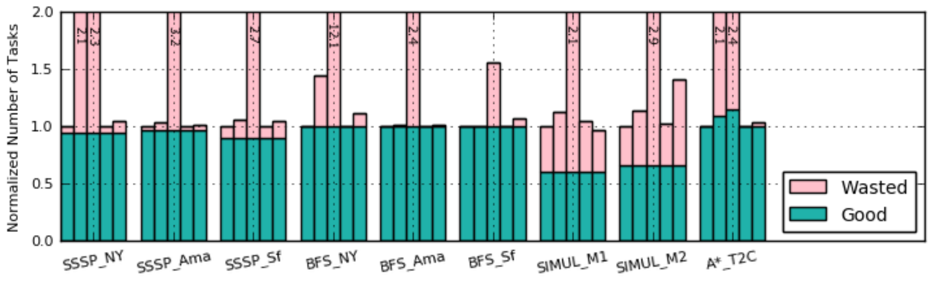


**Figure 7: Breakdown of the tasks in the applications into good and wasted tasks. The bars in each application and input are ordered as in Figure 6.**

discrete events, some of which have dependencies on other events. It uses a BFS-style graph traversal on the dependency graph. When a thread dequeues an event, it checks if all the event's dependencies have been executed. If true, the event executes and the dependents are pushed to the PQ; otherwise the event is ignored. For inputs, we generate a matrix of dependencies between events based on the approach outlined in [2]. The number of events that are dependent on a given event is parameterized with $\delta$, and the mean distance between the source and dependent node is parameterized with $\gamma$.

Table 3 shows the inputs evaluated. By default, runs are with 64 threads. The distributed PQs use 64 queues, one local to each core. *HW-D*, which is Snug, uses 64 Work registers.

| Input | Description |
|---|---|
| Graphs for SSSP and BFS: | |
| *NY Road (NY)* | Road network for New York City with edge weights as the travel time [12]. $\|V\|$ = 264,346, $\|E\|$ = 733,846 |
| *Amazon (Ama)* | Amazon product co-purchasing network from the SNAP dataset [41]. $\|V\|$ = 262,111, $\|E\|$ = 1,234,877 |
| *Scalefree (Sf)* | Graph with power law degree distribution, built with R-MAT [9, 35]. $\|V\|$ = 260,237, $\|E\|$ = 2,097,152 |
| Matrices for SIMUL: | |
| $M_1$ | Number of events = 12K, $\delta$ = 15, $\gamma$ = 100 |
| $M_2$ | Number of events = 12K, $\delta$ = 15, $\gamma$ = 200 |
| Grid for A*: | |
| *TaleofTwoCities (T2C)* | 2D Grid map from Starcraft benchmark set [42] |

**Table 3: Program inputs.**

In the applications, a thread repeatedly dequeues a task from the PQ, processes it, and (possibly) enqueues one or more tasks to the PQ. We categorize the work into *Good Work* and *Wasted Work*. For SSSP, BFS and A*, good work consists of graph edge relaxations that update the label of a node to its true distance (Section 2.1), and wasted work consists of all the remaining, redundant relaxations. Wasted work occurs either due to thread concurrency or, most notably, relaxed PQ semantics. For SSSP and BFS, the number of tasks performing good work remains constant across the different PQ designs since all the graph nodes are reduced to their shortest distance. In contrast, A* terminates when the target node is relaxed to its shortest distance. Since different PQs explore the grid in different ways, the number of tasks performing good work can change with different PQs. For SIMUL, good work consists of dequeueing an event after all of its dependencies have executed, thereby enabling its own execution. Otherwise, the task is classified as wasted. The number of tasks performing good work remains constant across the different PQ designs.

## 6 EVALUATION

### 6.1 Execution Time

Figure 6 compares the execution time of the applications and inputs under the different PQ implementations. For each application and input, we show, from left to right, *SW-SK*, *SW-SP*, *SW-D*, *HW-C*, and *HW-D*, all normalized to *SW-SK*. The time is broken down into *Pop* synchronization (the time spent in PQ *dequeue* operations), *Push* synchronization (the time spent in PQ *enqueue* operations), runtime (the execution of auxiliary code, such as thread termination), wasted work, and good work.

To help understand these bars, Figure 7 provides a breakdown of the total tasks executed in the program, classified into good and wasted tasks. The total number of tasks is normalized to the number in *SW-SK*. The bars are ordered as in Figure 6. Note that wasted tasks affect the execution in two ways. First, they induce redundant processing of graph edges or events. These cycles appear as wasted work in Figure 6. Second, they cause extraneous PQ enqueues and dequeues, hence increasing Push and Pop times in Figure 6.

In the following, we discuss all the PQ implementations in detail except for *HW-C*. *HW-C* differs from *SW-SK* mostly by having a single Work register, pointing to the highest-priority task. *HW-C*'s overall performance is similar to *SW-SK*. On average, it performs slightly worse than *SW-SK* due to the extra overheads of the new instructions. *Recall that* HW-D *is* Snug.

**SSSP.** As shown in Figure 6, *HW-D* performs better than the other PQs. On average across the three inputs, it reduces the execution

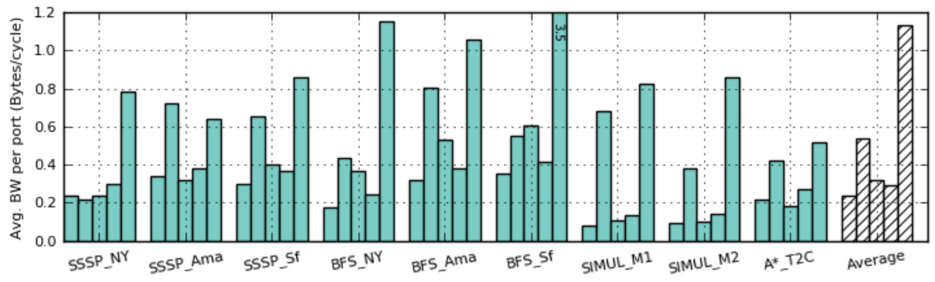


**Figure 8: Average bandwidth consumption in each of the global crossbar ports connected to the core clusters. The bars in each application and input are ordered as in Figure 6.**

time by 1.2×, 1.8×, and 5.1× compared to *SW-SK*, *SW-SP*, and *SW-D*, respectively. Compared to *SW-SK*, *HW-D* reduces the Pop time substantially, while only increasing the Push time slightly. *HW-D* reduces Pop because it eliminates the contention on the queue head with the use of distributed queues and the *PickHead* instruction, avoiding contention. *HW-D*'s Push time is slightly higher than *SW-SK* because *HW-D* is more relaxed, and ends up creating slightly more bad work (Figure 7) and enqueuing more tasks.

*HW-D* performs much better than *SW-SP* and *SW-D*, as *SW-SP* suffers from Pop time, and *SW-D* from wasted work. *SW-SP* does not work well for the relatively short PQs in these applications. *SW-SP* returns tasks from among the first $O(t \log^3 t)$ ones in the PQ (with high probability), where $t$ is the number of threads (Section 2.1). *SW-SP*'s random walks often reach the end of list; when this happens, *SW-SP* performs a linear scan of the PQ, to make sure no work was missed [2], and this increases Pop time. In addition, in sparse graphs like NY-Road, the short PQ causes *SW-SP* to return a random task, ignoring priorities, and causing wasted work.

*SW-D* has significant wasted work for all three graphs. This is because, in this relaxed PQ, each thread takes tasks from its own queue, without looking for the highest-priority task. The lack of synchronization overheads in *SW-D* is unable to compensate for this wasted work.

**BFS.** *HW-D* also performs well in BFS, reducing the average execution time by 1.7×, 3.1×, and 3.7× compared to *SW-SK*, *SW-SP*, and *SW-D*, respectively (Figure 6). In BFS, all the edges of the graph have a weight of 1. This increases the available parallelism, since there are now multiple paths of the same distance from the source node to each destination. This changes the behavior of the PQ implementations.

*HW-D* attains better performance than *SW-SK* and *SW-SP* because of reduced synchronization. The Push time is very high for these PQs, because many tasks now have the same priority and so get pushed to the same region of the skip list—leading to contention. *HW-D* reduces the Push time because enqueues do not contend with each other and, in addition, each enqueue traverses a shorter local queue.

*SW-D* has substantial wasted work. The reason is that each thread dequeues tasks from its local queue, and so is less likely to find the highest-priority tasks. *HW-D* eliminates most of the wasted work because dequeuing with *PickHead* enables a global view of available tasks, and provides higher-priority tasks. Note that on average, the wasted work with *SW-D* in BFS is lower than in SSSP. This is because in BFS, many more tasks have the same priority, and so the local queue is more likely to have one of the globally-highest priority tasks.

**SIMUL.** *HW-D* reduces the average execution time by 1.9×, 2.7×, and 1.5× compared to *SW-SK*, *SW-SP* and *SW-D*, respectively. The *SW-SK* and *SW-SP* PQ implementations have significant synchronization overheads. Specifically, *SW-SK* has Push contention similar to BFS, and *SW-SP* has high Pop time due to the previously-discussed issue of a sub-optimal spray. *SW-D* is the best software alternative. It reduces the Push and Pop times by using local enqueues and dequeues. Local dequeues, however, lead to many task dependency violations—a task is dequeued for execution before its dependencies have executed. This causes wasted work with *SW-D* (Figure 7). *HW-D* has lower contention overheads compared to the *SW-SK* and *SW-SP* PQs, and less task dependency violations compared to *SW-D*.

**A*.** *HW-D* also performs best, reducing the execution time by 1.1×, 2.8×, and 2.1× compared to *SW-SK*, *SW-SP* and *SW-D*, respectively. In A*, the priority with which a task is inserted in the PQ includes an extra heuristic cost, which is the Euclidean distance to the target. We see that *SW-SP* and *SW-D* have significant wasted work due to the relaxation of nodes away from the optimal path from source to target. *SW-SK* performs worse than *HW-D* due to higher Push time.

**Overall.** The data illustrates the main PQ tradeoff: the *SW-SK* and *SW-SP* PQs suffer from Push and Pop contention, while the *SW-D* PQ suffers from wasted work due to bad task selection. *HW-D* addresses both problems. Averaged across all applications and inputs, *HW-D* in Figure 6 reduces the execution time by 1.4×, 2.4×, and 3.6× compared to state-of-the-art skip list, SprayList, and software distributed PQs, respectively. These are substantial execution time reductions, as they correspond to 64-core executions. Compared to the latter two PQ designs, *HW-D* reduces the number of wasted tasks by 3.3× and 36.8×, respectively.

## 6.2 Network Traffic

To assess Snug's network traffic, Figure 8 shows the the average bandwidth consumption in the ports of the global crossbar that connect to the core clusters, in bytes per cycle.

We see that *HW-D* consumes more bandwidth than the other concurrent PQs. The average increase ranges from 2.2× relative to *SW-SP*, to 5.6× relative to *SW-SK*. However, *HW-D*'s bandwidth consumption is modest in absolute terms. On average across all applications, Snug consumes 1.13 bytes per cycle, or 3.5% of the crossbar's 32 bytes per cycle link capacity, which is the same link capacity assumed in related work [24] and deployed in Intel's Haswell.

Table 4 examines the behavior of the *PickHead* instruction accesses, averaged over all the cores. *GA* is the percentage of *PickHead* accesses that are Global. We can see that, typically, about 1 in 6 accesses are Global. TS is the average number of instructions per

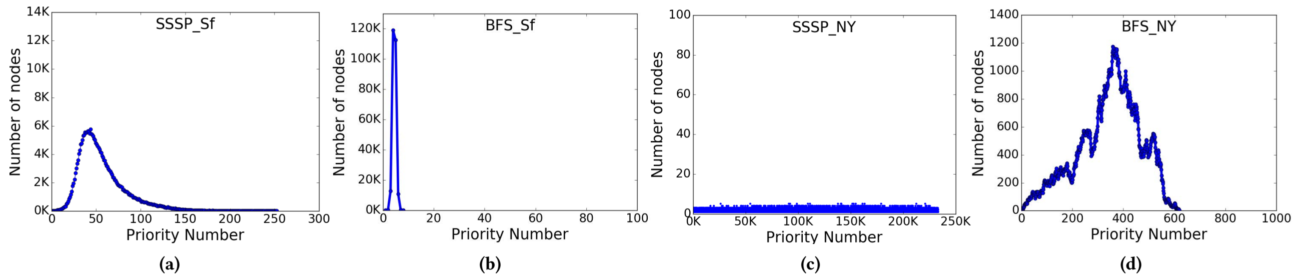


**Figure 9: Priority distribution for applications and inputs.**

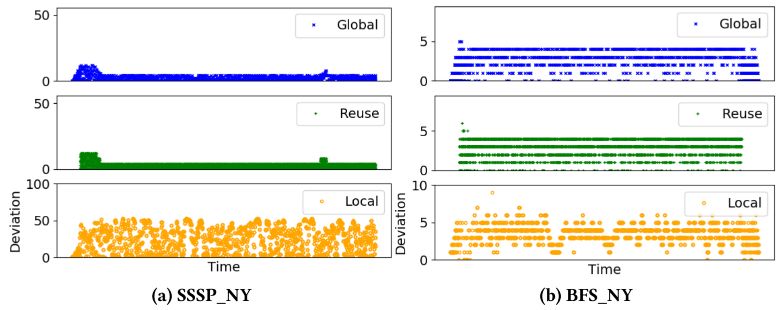


**Figure 10: Difference between the priority returned by *PickHead* and the best priority in snapshot.**

task. We see that this number is about 1300-4400. Next, we show the failure rate of *PickHead* accesses, broken down by access type. GF, RF and LF are the failure rates for Global, Reuse and Local accesses, respectively. On average, we observe that Local accesses have the least failure rate, followed by Global and Reuse. Reuse accesses fail often.

| | SSSP | | | BFS | | | SIMUL | | A* |
|---|---|---|---|---|---|---|---|---|---|
| | NY | Ama | Sf | NY | Ama | Sf | M1 | M2 | T2C |
| GA | 19% | 15% | 21% | 18% | 15% | 43% | 16% | 19% | 18% |
| TS | 2717 | 3993 | 4438 | 1297 | 1633 | 2144 | 2017 | 1895 | 3750 |
| GF | 48% | 28% | 45% | 38% | 32% | 92% | 35% | 30% | 21% |
| RF | 73% | 80% | 71% | 76% | 81% | 75% | 55% | 56% | 74% |
| LF | 11% | 8% | 13% | 12% | 14% | 5% | 7% | 8% | 5% |

**Table 4: *PickHead* accesses in *HW-D* and task size.**

## 6.3 Analysis of Wasted Work

Figure 6 showed that *SW-D* suffers from substantial wasted work in SSSP and, for the NY input, in BFS. To understand why, recall that a thread in *SW-D* dequeues tasks from its local queue. If it obtains tasks with priority far from the globally-highest priority, *SW-D* will have wasted work. However, if the application is such that there is a large number of tasks for each priority, then there is a greater chance that the locally-best task has a priority similar to the globally-best priority. In that case, *SW-D* will have much less wasted work.

Figure 9 explains the different behavior of the applications and input data sets. For SSSP and BFS, it shows the number of nodes in the graph that are at a certain distance from the source node (i.e., their priority). Figures 9a and 9b show the Sf graph (SSSP_Sf and BFS_Sf). We see that the distribution of priorities is different. In SSSP_Sf, the edge weights yield 250 priorities, most with only few nodes. In BFS_Sf, there are only a handful of priorities, each with tens of thousands of nodes. Hence, *SW-D* has less wasted work in BFS_Sf than in SSSP_Sf.

In contrast, Figures 9c and 9d show the NY graph (SSSP_NY and BFS_NY). There are few nodes for each priority—especially in SSSP. *SW-D* has a high chance of picking tasks with priority far from the globally-highest priority. As a result, *SW-D* has wasted work in SSSP_NY and BFS_NY. In *HW-D*, the *PickHead* instruction obtains a global view of the tasks in all the queues. Thanks to this, *HW-D* picks high quality tasks, and has negligible wasted work.

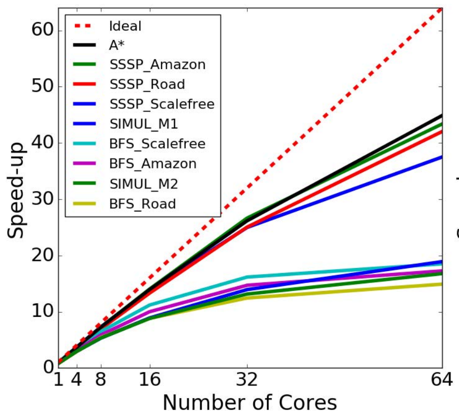
Figure 10 visualizes the degree of priority relaxation by the *PickHead* instruction. Specifically, we plot the difference between the priority returned by *PickHead* and the best priority in the snapshot, across time. The figure shows data for SSSP and BFS, on the NY graph, and for Global, Reuse, and Local accesses. We see that the only accesses that get tasks far from the best are Local accesses in SSSP. Local accesses greedily exploit the local queue and can dequeue low-quality tasks.

## 6.4 SNUG Scalability

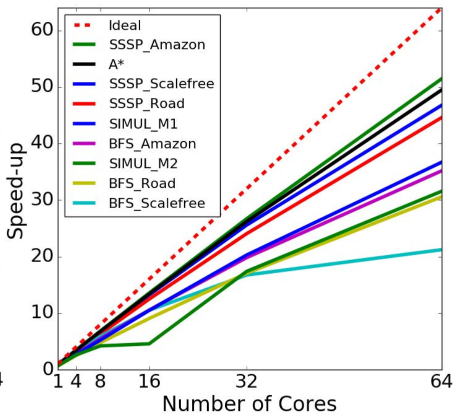
To understand the scalability of SNUG with the number of cores, Figure 11 shows the speed-ups of our applications and inputs as we change the core count from 1 to 64. The figure shows data for our baseline *SW-SK* (Figure 11a) and for *HW-D* (Figure 11b). In both figures, the speed-ups are relative to a 1-core sequential skip-list PQ design which, unlike 1-core *SW-SK*, does not use CAS instructions. Figure 11a shows that *SW-SK* does not scale for about half of the configurations, namely the BFS and SIMUL applications. These applications push many tasks with identical priorities, leading to contention in skip list nodes, which leads to high Push times (Figure 6). In contrast, Figure 11b shows that the speed-ups in *HW-D* scale better for most configurations. The distributed queues in *HW-D* mitigate the push contention in BFS and SIMUL (Figure 6), as different tasks are pushed to different queues. Hence, *HW-D* is more scalable.

## 6.5 Sensitivity Analysis

**Sensitivity to $L$ and $U$.** SNUG's algorithm for picking the next node to process (Section 4.1) has two parameters: following a global access, there are $U$ snapshot reuses and $L$ local accesses. *HW-D*



(a) *SW-SK*

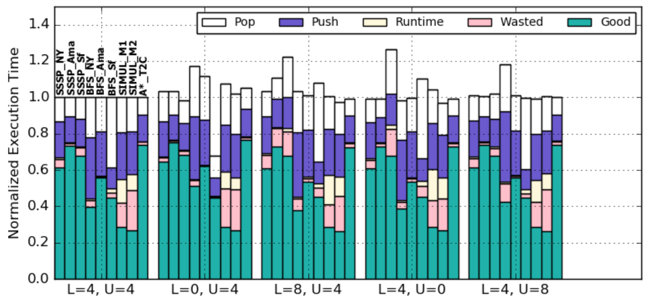


(b) *HW-D*

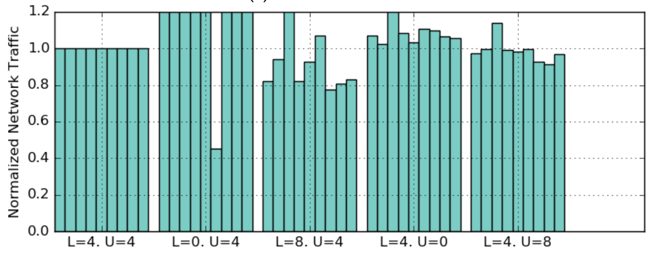
**Figure 11: Speed-ups of *SW-SK* (a) and *HW-D* (b).**

sets both parameters to 4. We do sensitivity analysis varying $L$ and $U$, and measure the execution time and the network traffic. Specifically, we fix $U$ to 4 and change $L$ to 0 and 8. Then, we fix $L$ to 4 and change $U$ to 0 and 8.

Figure 12a shows the execution time, and Figure 12b the network traffic, as we vary $L$ and $U$ from 0 to 4 and 8. Specifically, the first set of bars is *HW-D* ($L$=4, $U$=4). In the next two sets of bars, $U$ is fixed to 4 and $L$ varies to 0 and 8. In the final two sets of bars, $L$ is fixed to 4 and $U$ varies to 0 and 8. For each ($L$, $U$) setting, we have a bar for each application and input. For a given application and input, the bars are normalized to the ($L$=4, $U$=4) setting.



**(a) Execution time.**



**(b) Network traffic. Values for truncated bars are 2.3, 1.8, 1.3, 2.7, 2.0, 1.6, 1.9, 2.2, 1.8, and 2.0 from left to right.**

**Figure 12: Sensitivity to the number of local accesses ($L$) and reuses of a snapshot ($U$).**

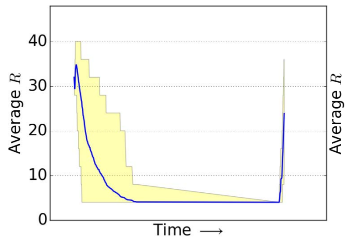
We see that changing $L$ alters the behavior of the system. When $L$=0, SNUG dequeues better tasks, by issuing more Global accesses. However, the execution time does not generally go down because the traffic increases. At the other end, when $L$=8, SNUG dequeues more low-priority tasks from the local queue. While the traffic generally decreases, the execution time does not go down because there is more wasted work. Overall, $L$=4 is slightly better.

In the final two sets of bars, we change $U$. When $U$=0, the traffic and contention is higher because there is no snapshot reuse. This results in an increase in the execution time. With $U$=8, the execution time also goes slightly up, likely due to the fact that the snapshot is reused past the point when it is useful. In summary, $U$=4 is a good design point.

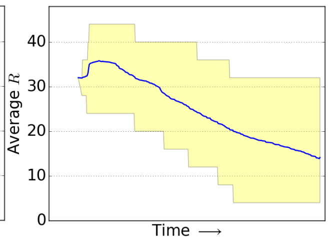
**Sensitivity to application input.** The characteristics of distributed PQs and how SNUG interacts with them could change appreciably depending on the size of input graphs or matrices. Although the computational costs for cycle-level gem5 simulation of 64 cores prohibit us from using datasets much larger than those in Table 3, we measure their impact indirectly by downsizing the L1, L2 and L3 caches instead. Generally, we find that SNUG does not unreasonably exploit any effects due to caching of inputs. For example, for BFS_NY, we reduce all the caches to half (or a quarter) of their original capacities. We find that SNUG is 2.13× (or 2.08× for the quarter cache) faster than SW-SK, compared to 2.05× in the unmodified architecture.

## 6.6 Characterizing R Adaptivity

The *PickHead* module in each core adjusts the Relaxation Count $R$ based on the rate of synchronization failures, which indicate the degree of contention. Figure 13 shows two representative examples of $R$'s behavior as the application executes: SSSP_NY and SIMUL_-M2. In the figure, the X-axis represents time. At each point in time, the figures show the average value of $R$ across all cores. The shaded area around the mean shows the minimum and maximum values across cores. In both cases, we start with our default $R_0$ = 32.



**(a) SSSP_NY**



**(b) SIMUL_M2**

**Figure 13: Average value of $R$ as execution progresses.**

SSSP_NY demonstrates how Snug automatically adjusts $R$. At the beginning of the application, there are few tasks and, as processors contend for them, $R$ goes up. Gradually, the queues obtain tasks. Dequeuing occurs concurrently across many queues, and synchronizations succeed. This triggers a decrease in $R$. The descent continues as long as the cores do not observe enough dequeue conflicts. Finally, as the application runs out of work and most queues become empty again, the contention increases, and $R$ has a final spike.

SIMUL_M2 (Figure 13b) shows a different behavior. There is an initial hump in $R$ as in SSSP_NY. However, the rate of descent of $R$ is slow. The reason is that SIMUL_M2 has a smaller average task size than SSSP_NY (Table 4). This increases the frequency of CASes attempting to dequeue the same task. The result is higher CAS failures. Snug adapts to this contention by keeping $R$ sufficiently high to avoid hotspots, while dequeuing high-quality tasks.

## 6.7 Area and Power Analysis

We estimate the area and power overhead of the *PickHead* module by considering the overhead of the Sorter & Selector and the Snapshot memory units. We use the Synopsys DC with the 32nm technology library for logic, and CACTI [44] with the 32nm ITRS-hp technology for memory components. For the design in this paper, the Snapshot memory holds 64 entries, and each entry is 16 bytes (head pointer and queue ID). It is implemented as a single-ported RAM structure. The Sorter & Selector is composed of 64 registers and 64 comparators, to process 70-bit data (priority and tag).

Our estimates show that the area of the *PickHead* module is about 0.23mm$^2$ and its power consumption 61.1mW in 32 nm. To put these numbers in context, we compare to existing processors. Publicly-available information indicates that the core area (processing unit, L1 and L2) of a Sandy Bridge processor and a Power7+ processor at 32nm are 18.18mm$^2$ and 21.14mm$^2$, respectively. Using data from [16], we estimate that the per-core power of a Sandy Bridge core is 7.54W. Therefore, we estimate the area of the *PickHead* module to be 1.27% of a Sandy Bridge core and 1.09% of a Power7+ core. We estimate the power consumption of the *PickHead* module to be 0.81% of a Sandy Bridge core.

## 7 RELATED WORK

Providing hardware support for queueing in multiprocessors has received much interest [1, 24, 26, 28, 40]. Carbon [28] is an architecture for efficient task queueing and scheduling in hardware. It focuses on applications with fine-grained tasks. It provides cores with dedicated hardware queues and orchestrates the movement of tasks between the queues. ADM [40] accelerates fine-grained task scheduling by providing a per-core hardware messaging module. Bypassing the memory hierarchy, the module enables communication between threads, which manage task queues in software. Snug focuses on speeding-up concurrent priority queueing.

Swarm [23, 24] mines parallelism from sequential programs written in a task-based programming model. Swarm builds on prior TLS and HTM schemes. It executes tasks speculatively and out of order, and efficiently speculates thousands of tasks ahead of the earliest task to uncover ordered parallelism. In contrast, Snug targets parallel programs, which already synchronize explicitly, and thus has simpler hardware mechanisms that are not based on speculation. Swarm also supports task queueing in hardware. In Swarm, cores enqueue tasks to a randomly chosen remote queue, and dequeue tasks from their local queue. In Snug, cores enqueue locally and dequeue either remotely or locally.

The Galois [35] graph analytics system contains the OBIM priority-based scheduler, which has been shown to outperform PQ designs based on Intel's TBB library [29]. Like relaxed PQs, OBIM can execute tasks out of priority order. In the Galois system, OBIM has been highly tuned to multi-socket NUMA platforms, with sizable synchronization costs. As indicated in Section 5, we evaluated OBIM but found it not competitive in the single-chip architecture with low synchronization costs that we consider.

## 8 CONCLUSIONS

Priority-based task scheduling algorithms pose a tradeoff: alleviating the synchronization hotspot by using relaxed PQs leads to wasted work. To address this tradeoff, we introduce the Snug architecture. Snug distributes the PQ into subqueues, maintains a set of Work registers that point to the highest-priority task in each subqueue, and provides an instruction that picks a high-quality task to execute.

We evaluated Snug on a simulated 64-core chip. We compared the execution time of graph and discrete event simulation applications under Snug and several other PQ algorithms. We found that Snug selects high-quality tasks while avoiding hotspots, minimizes wasted work, and consumes acceptable network bandwidth. Snug reduces the average execution time of the applications by 1.4×, 2.4× and 3.6× compared to concurrent skip list, SprayList, and software-distributed PQs, respectively. Moreover, compared to the latter two relaxed PQ designs, Snug reduces the number of wasted tasks by 3.3× and 36.8×, respectively.

## ACKNOWLEDGMENTS

This research was funded in part by the National Science Foundation under grants CNS-1763658 and CNS-1705047, and the Israel Science Foundation under grant 2005/17.

## APPENDIX: CONCURRENT PQ DETAILS

**Logical Deletions.** Deleting a node $x$ by updating only its predecessor's *next* pointer might undo the effect of a concurrent operation that updates $x$. To prevent such an atomicity violation, an operation should only update the predecessor's *next* pointer (physical deletion) after first *logically* deleting the target node, which prevents concurrent operations from updating the node. Logical deletion is performed by atomically setting a flag bit co-located with the LSB of the *next* field [17, 34]. Updates fail (and retry) if they observe that their target node has this flag bit set.

**Supporting Skip Lists.** A skip list is a sorted linked list where individual nodes have an array of *next* pointers, linked into multiple sorted lists, in an effort to speed-up searches by "skipping over" irrelevant nodes. Thanks to skipping, a skip list search traverses $n$ nodes in $O(\log n)$ time (expected). Concurrent skip list algorithms support scalable read-mostly searches and insertions, and can access the highest-priority node in $O(1)$ time [14, 18, 19, 43]. Snug's mechanisms are compatible with a skip list-based PQ. In this case, the Work register stores the head of the bottom-level list. The heads of the other, higher-level lists, remain located in memory and are maintained by the software PQ library.

## REFERENCES

[1] Ghiath Al-Kadi and Andrei Sergeevich Terechko. 2009. A hardware task scheduler for embedded video processing. In *International Conference on High-Performance Embedded Architectures and Compilers*. Springer, 140–152.

[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)*.

[3] Ron Bekkerman, Mikhail Bilenko, and John Langford. 2011. *Scaling Up Machine Learning: Parallel and Distributed Approaches*. Cambridge University Press, New York, NY, USA.

[4] D. P. Bertsekas, F. Guerriero, and R. Musmanno. 1996. Parallel Asynchronous Label-correcting Methods for Shortest Paths. *Journal of Optimization Theory and Applications* 88, 2 (Feb. 1996).

[5] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *SIGARCH Comput. Archit. News* 39, 2 (Aug. 2011).

[6] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Julian Shun. 2012. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP)*.

[7] Ulrik Brandes. 2001. A Faster Algorithm for Betweenness Centrality. *Journal of Mathematical Sociology* 25, 2 (2001).

[8] Irina Calciu, Hammurabi Mendes, and Maurice Herlihy. 2014. The Adaptive Priority Queue with Elimination and Combining. In *Proceedings of the 28th International Symposium on Distributed Computing (DISC)*.

[9] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A Recursive Model for Graph Mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining*. SIAM.

[10] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms* (3rd ed.). MIT Press.

[11] E. W. Dijkstra. 1959. A Note on Two Problems in Connexion with Graphs. *Numer. Math.* 1, 1 (Dec. 1959), 269–271.

[12] DIMACS. 2010. 9th DIMACS Implementation Challenge. http://www.dis.uniroma1.it/challenge9/download.shtml.

[13] Gal Elidan, Ian McGraw, and Daphne Koller. 2006. Residual Belief Propagation: Informed Scheduling for Asynchronous Message Passing. In *Proceedings of the Twenty-Second Conference on Uncertainty in Artificial Intelligence (UAI)*.

[14] Keir Fraser. 2004. *Practical lock-freedom*. Ph.D. Dissertation. University of Cambridge, Computer Laboratory, University of Cambridge, Computer Laboratory.

[15] Richard M. Fujimoto. 1990. Parallel Discrete Event Simulation. *Communications of ACM (CACM)* 33, 10 (1990), 30–53. https://doi.org/10.1145/84537.84545

[16] Jawad Haj-Yihia, Ahmad Yasin, Yosi Ben Asher, and Avi Mendelson. 2016. Fine-Grain Power Breakdown of Modern Out-of-Order Cores and Its Implications on Skylake-Based Systems. *ACM Trans. Archit. Code Optim.* 13, 4, Article 56 (Dec. 2016), 25 pages. https://doi.org/10.1145/3018112

[17] Timothy Harris. 2001. A Pragmatic Implementation of Non-blocking Linked-lists. In *Proceedings of the 15th International Symposium on Distributed Computing (DISC)*.

[18] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. 2007. A Simple Optimistic Skiplist Algorithm. In *14th Colloquium on Structural Information and Communication Complexity (SIROCCO)*.

[19] Maurice Herlihy and Nir Shavit. 2008. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[20] Matthew Holly and Carl Tropper. 2011. Parallel Discrete Event N-Body Dynamics. In *Proceedings of the 2011 IEEE Workshop on Principles of Advanced and Distributed Simulation (PADS '11)*. 1–10.

[21] Galen C. Hunt, Maged M. Michael, Srinivasan Parthasarathy, and Michael L. Scott. 1996. An Efficient Algorithm for Concurrent Priority Queue Heaps. *Information Processing Letters (IPL)* 60, 3 (1996), 151–157.

[22] David R. Jefferson. 1985. Virtual Time. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 7, 3 (July 1985).

[23] Mark C. Jeffrey, Suvinay Subramanian, Maleen Abeydeera, Joel Emer, and Daniel Sanchez. 2016. Data-Centric Execution of Speculative Parallel Programs. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[24] Mark C. Jeffrey, Suvinay Subramanian, Cong Yan, Joel Emer, and Daniel Sanchez. 2015. A Scalable Architecture for Ordered Parallelism. In *48th IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[25] Jonathan Karnon, James Stahl, Alan Brennan, J Jaime Caro, Javier Mar, and Jörgen Möller. 2012. Modeling Using Discrete Event Simulation: A Report of the ISPOR-SMDM Modeling Good Research Practices Task Force–4. *Medical decision making* 32, 5 (2012), 701–711.

[26] Behram Khan, Daniel Goodman, Salman Khan, Will Toms, Paolo Faraboschi, Mikel Luján, and Ian Watson. 2015. Architectural Support for Task Scheduling: Hardware Scheduling for Dataflow on NUMA Systems. *J. Supercomput.* 71, 6 (June 2015), 2309–2338. https://doi.org/10.1007/s11227-015-1383-2

[27] Khronos Group. 2019. OpenCL. http://www.khronos.org/opencl/.

[28] Sanjeev Kumar, Christopher J. Hughes, and Anthony Nguyen. 2007. Carbon: Architectural Support for Fine-grained Parallelism on Chip Multiprocessors. *SIGARCH Comput. Archit. News* 35, 2 (June 2007), 162–173. https://doi.org/10.1145/1273440.1250683

[29] Andrew Lenharth, Donald Nguyen, and Keshav Pingali. 2015. Priority queues are not good concurrent priority schedulers. In *Proceedings of the 21st International Conference on Parallel and Distributed Computing (Euro-Par)*.

[30] 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 (OPODIS)*.

[31] Yujie Liu and Michael Spear. 2012. Mounds: Array-Based Concurrent Priority Queues. In *Proceedings of the 41st IEEE International Conference on Parallel Processing (ICPP)*.

[32] Itay Lotan and Nir Shavit. 2000. Skiplist-Based Concurrent Priority Queues. In *Proceedings of the 14th International Parallel & Distributed Processing Symposium (IPDPS)*.

[33] Janarbek Matai, Dustin Richmond, Dajung Lee, Zac Blair, Qiongzhi Wu, Amin Abazari, and Ryan Kastner. 2016. Resolve: Generation of high-performance sorting architectures from high-level synthesis. In *Proceedings of the 24th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. 195–204.

[34] Maged M. Michael. 2002. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the 14th ACM Symposium on Parallel Algorithms and Architectures (SPAA '02)*.

[35] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A Lightweight Infrastructure for Graph Analytics. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*.

[36] NVIDIA. 2019. CUDA Programming Guide v3.1. http://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[37] R. C. Prim. 1957. Shortest Connection Networks And Some Generalizations. *Bell System Technical Journal* 36, 6 (1957).

[38] William Pugh. 1990. Skip Lists: A Probabilistic Alternative to Balanced Trees. *CACM* 33, 6 (June 1990), 668–676.

[39] Hamza Rihani, Peter Sanders, and Roman Dementiev. 2015. Brief Announcement: MultiQueues: Simple Relaxed Concurrent Priority Queues. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'15)*.

[40] Daniel Sanchez, Richard M. Yoo, and Christos Kozyrakis. 2010. Flexible Architectural Support for Fine-grain Scheduling. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*.

[41] SNAP. 2003. SNAP Datasets. https://snap.stanford.edu/data/amazon0302.html.

[42] N. Sturtevant. 2012. Benchmarks for Grid-Based Pathfinding. *Transactions on Computational Intelligence and AI in Games* 4, 2 (2012), 144 – 148. http://web.cs.du.edu/~sturtevant/papers/benchmarks.pdf

[43] Håkan Sundell and Philippas Tsigas. 2005. Fast and Lock-free Concurrent Priority Queues for Multi-thread Systems. *JPDC* 65, 5 (2005), 609–627.

[44] Shyamkumar Thoziyoor, Naveen Muralimanohar, Jung Ho Ahn, and Norman P. Jouppi. 2008. *CACTI 5.1*. Technical Report HPL-2008-20. HP Labs.

[45] Wikipedia. 2019. Moving Average. https://en.wikipedia.org/wiki/Movingaverage.

[46] M. Wimmer, J. Gruber, J. Larsson Träff, and P. Tsigas. 2015. The Lock-free $k$-LSM Relaxed Priority Queue. *ArXiv e-prints* (March 2015). arXiv:cs.DS/1503.05698

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

[48] Martin Wimmer, Francesco Versaci, Jesper Larsson Träff, Daniel Cederman, and Philippas Tsigas. 2014. Data Structures for Task-based Priority Scheduling (Poster). In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*.

[49] Erjia Yan and Ying Ding. 2009. Applying Centrality Measures to Impact Analysis: A Coauthorship Network Analysis. *Journal of the American Society for Information Science and Technology* (Oct. 2009).

[50] Haiyuan Yu, Philip M. Kim, Emmett Sprecher, Valery Trifonov, and Mark Gerstein. 2007. The Importance of Bottlenecks in Protein Networks: Correlation with Gene Essentiality and Expression Dynamics. *PLoS Computational Biology* 3, 4 (April 2007).