SAPIENZA
UNIVERSITÀ DI ROMA

Parallel Priorities: Optimizing Priority Queues for NUMA Machines

Ingegneria dell'informazione, informatica e statistica

Corso di Laurea Magistrale in Engineering in Computer Science

Candidate

Umberto Mazziotta

ID number 1647818

Thesis Advisor

Alessandro Pellegrini

Co-Advisor

Romolo Marotta

Academic Year 2018/2019

Parallel Priorities: Optimizing Priority Queues for NUMA Machines
Master thesis. Sapienza – University of Rome

This thesis has been typeset by LaTeX and the Sapthesis class.

Author's email: mazziotta.1647818@studenti.uniroma1.it

Time marches on

Acknowledgments

Vorrei spendere qualche riga per ringraziare tutti quelli che mi hanno permesso di raggiungere questo traguardo.

Il primo ringraziamento va alla mia famiglia. Ai miei genitori, Antonio e Olga, e a mia sorella, Marcella, che mi hanno sempre sostenuto e appoggiato le mie scelte, e hanno sempre cercato di fare di tutto affinché potessi dedicarmi completamente agli studi.

Vorrei ringraziare il mio relatore, Alessandro Pellegrini, che con un giudizio su un semplice progetto mi ha dato l'input che mi ha portato a svolgere questo lavoro, e per il supporto dato durante questi mesi. Con lui, vorrei ringraziare anche il mio correlatore, Romolo Marotta, per avermi fatto da guida durante il lavoro svolto per questa tesi, per tutti i consigli dati, per la pazienza mostrata e l'immensa disponibilità che ha sempre offerto. Grazie di tutto.

Grazie ad Andrea, con cui ho legato più che mai durante questi ultimi due anni. Per esserci supportati a vicenda durante gli esami e nelle nostre crisi personali. Per esserci stato più di quanto io abbia fatto con lui, per aver condiviso ogni problema senza filtri e per essere diventato un fratello su cui poter sempre contare.

Grazie a Claire, per aver reso speciale ogni giorno di quest'ultimo anno, per aver sopportato tutti i miei malumori e per aver sempre cercato di fare di tutto per farmi stare bene.

Grazie a Leon, Gioia, Christian, Altea, Fabio, Eric, Ludovica, Davide, Carmine, Roberto e tuti gli altri con cui ho condiviso momenti fin da quando ho messo piede a Roma, grazie per tutti i bei momenti che abbiamo passato insieme, e per aver reso il tempo passato al DIAG più confortevole, rendendolo una seconda casa in cui voler sempre tornare.

Grazie a Caterina, Marika e Rosa, per essere riusciti a rimanere legati dopo più di dieci anni, per esserci sempre state nei momenti di bisogno, ognuna a modo suo. Grazie per aver sempre trovato il tempo per coltivare la nostra amicizia.

Infine, grazie ad Alberto, Antonio, Federico, Francesco, Peppe, Pia, Pietro, Stefania e Steven. Grazie per tutte le meravigliose serate che abbiamo passato insieme, per i viaggi, le feste e tutte le cazzate. Grazie per ricordarmi che dopo il lavoro c'è sempre una pausa. Grazie per aver creduto, prima di tutti, che ce l'avrei fatta.

Contents

Chapter 1

Introduction

Concurrent data structures are used everywhere, from kernels of Operation Systems to applications, and they are a fundamental building block for modern software. Since the broad use we can make of them, it is important to optimize them to prevent performance degradation.

In the last years, computer architectures have seen a continuous growth in terms of number of processors and their speed [37].

For thirty years, the processors speed growth was ruled by Moore's law, for which the number of transistors in a dense integrated circuit doubles approximately every 18 to 24 months [31]. Thanks to the MOSFET scaling rules, increasing the density of transistors for die area, it increases also their frequency of switch, resulting in a higher Clock speeds, keeping the power supplied the same.

Processors saw a continuous increase of their speed, and this gave to sequential programs for free better performance only because they were running on faster processors.

Starting from 2000s, the trend changed. In fact, the increasing transistors speed also increases the leakage power, which results in higher power dissipation. This led to higher temperatures reached from the processors. Thus, the increasing of speeds reached its limit because of the increasing dissipated power, leading to what is called the *Power Wall*.

In order to keep increasing the performance of the processors, manufacturers started to increase the number of cores of processors to take advantage of the increasing transistor density per area, instead of keep increasing their speed. Having more cores on the same processor allows to execute more tasks at the same time but doesn't allow them to go faster for free. A program that was designed to run on a single core, moved onto a multi-core processor won't see any improvement since it keeps running on one core. To fully exploit the hardware changes, there is the need of finding new way to accomplish performance gains. In the case of multi-cores and multiprocessors this is achieved by making programs concurrent, allowing them to run on multiple processors. This can be achieved by splitting computations in such a way that they can be carried out by multiple threads that can run on the different cores.

In many cases this adds the need of synchronizing the threads to manipulate shared resources in a correct way, which traditionally implies a serialization of the execution when working on them, resulting in a waste of computational power and energy. Thus, it is important to assure scalability of data structures to allow programs to take full advantage of the underlying hardware.

Classical computing architectures have a single memory access point shared among all the processors. Having an ever-increasing number of CPUs that try to access the same memory is another bottleneck for processors speeds. So, this leads to new architectural changes. Non-Uniform Memory Access (NUMA) machines first appeared in the late eighties, and in the last years, they started to become very common, especially on large multi-core machines. In this kind of architectures, the processors are grouped into *nodes*, and each node has a local memory. Nodes can access the memory local to other nodes, but it is faster for a processor accessing the memory within its node than a remote memory.

Usually, when we move traditional data structures to NUMA machines, they can easily suffer from a significant performance loss, due to the extra time needed to access remote memory and the contention on the interconnection between nodes.

To increase the scalability of applications on these systems, the number of remote accesses must be reduced. Many general solutions have been already designed to overcome this problem, often based on thread and/or memory placement [12, 39], and data replication [9]. Anyhow, the performance offered by these generic solutions are often sub-optimal, hence the programmers should modify the data structure to best fit the underlying architecture.

In this work, we try to address these problems induced by modern architectures on concurrent data structures as priority queues. Such data structures are widely used in simulation systems, in operating systems scheduler, in numerical computations and also in many other performance critical applications.

A priority queue contains a set of key-value pairs, which are ordered by their priority, represented by the key. Such data structure offers two methods: `enqueue()` for inserting items in the queue, and `dequeue()` which removes and returns the element with highest priority i.e., the element associated to the smallest key.

Traditionally, these are built on top of heaps or tree data structures. Recently, highly concurrent priority queues have been built on top of many lock-free data structures such as Linked-Lists [27] or Skip-Lists [2, 26], because with them it is relatively easy to achieve concurrent access to different parts of the data structure, and thus improving their scalability.

Many parallel data structures optimized per NUMA machines rely on *delegation* to increase the locality of operations [7, 28, 35]. Delegation allows threads to do not execute directly operations, but they delegate some other threads to execute it, while they wait for the response. Usually, a single thread executes all the pending operations on the data structure, which can be a dedicated one [28] or chosen when an operation has to be executed by acquiring a lock [18]. This technique is ideal for NUMA architectures, since it usually shows a good cache locality and low memory contention.

In this thesis, we adopt this technique in order to add NUMA-awareness to a concurrent priority queue. What we do is to use delegation, but without having a

single thread doing the work while the others wait by doing nothing. We introduced a set of communications channels between NUMA nodes, allowing threads to post operations in order to let them be executed by threads that are closer to the memory. The main difference with similar approaches is in the wait phase, since here we tried to avoid waste of time induced by waiting for the response, by allowing every thread to execute an operation with destination its NUMA node. This approach aims into reducing remote memory accesses only to the ones needed to publish the operations and their responses, while keeping the same level of concurrency of operations. As we will see this approach has some potential, since it will show some real improvements but, it might reduce the progress guarantees provided by the underlying data structure. In the end, we tried also to overcome to this problem by modifying the basic algorithms of the underlying data structure.

This thesis is organized as follows. In the next chapter we make an overview of the main covered concepts, and we describe the Conflict Resilient Calendar Queue, which is the priority queue we used as starting point for our solutions. Then, we provide different solutions to optimize the priority queue, showing also the results obtained in the performance evaluation, and the results against similar approaches that are found in literature. Finally, we conclude with some remarks on the work done, and other possible steps that can be done toward a further improvement of these solutions.

Chapter 2

Concurrent Programming

A system is said concurrent when it supports multiple actions in progress at the same time [5].

The first attempt towards concurrency was achieved with time-sharing. Even though the computer had a single core, tasks were able to run concurrently by sharing their execution time on the CPU in an interleaved fashion. The execution of a program is logically divided into time slices and when the slice ends the process is removed from the CPU to allow another task to be executed for another time slice. This is done thanks to the help of a scheduling algorithm which is provided by the Operating System.

The problem when dealing with concurrency is that, having multiple tasks running at the same time, we cannot say in which order they will be executed, and when they work on shared resources we can't know in which order they access and modify them. In this case the unpredictability of the executions can lead to unpredictable results on the resource, which is left in a state that do not comply with its specification. This problem is known as *race condition*, on which the resulting state of the resource depends on which order the tasks interact with it. This is a critical problem since the race for the resource can also affect the results of the computation of other processes that access it. This problem must be solved by properly synchronizing accesses to shared resources, namely *Critical Sections*, by

guaranteeing *mutual exclusion*.

Mutual Exclusion is a requirement in concurrent programming in order to avoid race conditions when accessing critical sections, and this is done by allowing only one thread to execute the critical section at a time. *Mutual Exclusion* was first formalized and solved by Dijkstra [13] in the sixties. Another requirement that was stated in this work is that a solution, along with mutual exclusion, it must provide also no deadlock - for which if multiple processes want to access the critical section, the decision of which one can access it is not postponed until eternity and, at least one of them can execute the critical section eventually. The solution proposed by Dijkstra is based on the use of shared variables on which processes can read and write atomically.

In particular, it prevents multiple accesses to the resource by arbitrating threads approaching the critical section. Each process must acquire an exclusive access to the shared resource before entering the critical section, and then release it when done. In other words, Dijkstra designed a synchronization primitive, which is commonly known as *lock*. However, as we will see later, there are also other way to deal with synchronization algorithms.

When developing concurrent programs, we are interested in maintaining a set of properties which are usually guaranteed by a sequential program for free. These requirements for a concurrent program can be defined in terms of *Safety* (or Correctness) and *Liveness* (or Progress) properties [25]. Safety properties imply that nothing bad happens, while Liveness that something good will happen eventually. Safety concerns about what the program is allowed to do, while Liveness is about what must happen during the computation. For example, Mutual exclusion is a Safety Property, while no deadlock is a Liveness property.

When we are in a concurrent environment, multiple threads might access shared resources. The shared resource is usually some memory area, e.g. a data structure that is accessed concurrently by threads and can be modeled as a *shared object*. A *shared object* is a container for data, with a state, on which it is possible to operate

through a set of *methods*. Each object has well defined methods (in terms of their specification) which allows threads to interact with it and modify its *state* [20]. We can model the execution of a concurrent object by a *history*, which is a finite sequence of method *invocation* and *response events*. A *subhistory* of a history H is a subsequence of events of H, and an *interval* is a subhistory consisting of contiguous events [21]. We will go back to this model in the next section, where it will be explained in more details.

2.1 Progress Conditions

Accesses to shared object are commonly synchronized through locking primitives: a lock is acquired before accessing shared data and then it is released when done. But when we an algorithm resorts to locking primitives, we can only make some weak assumptions on the progress it guarantees. In fact, when we are using locks, also the role of the scheduler has to be taken in account, in fact if it halts a thread during a critical section, it won't able to release the lock, preventing other threads to execute the critical section, limiting the progress. In particular, relying on locks, what we can guarantee is that a method can be *deadlock-free* or *starvation-free*.

We say that a method is deadlock-free if *some* thread acquires a lock eventually, while a method is starvation-free if every thread acquires a lock eventually. So, deadlock-freedom guarantees that the whole system proceeds, while with starvation-freedom we are making a stronger assumption, since every thread is able to make progress. Anyhow, to achieve stronger properties, their requirements have also to put constraints on the behavior of the scheduler. In fact, without the help of the scheduler it would be impossible to achieve stronger properties like *starvation-freedom* - we can just think of an unfair scheduler that ignores the presence of one of the threads, and never schedules it. In that case, we cannot guarantee the progress of all the threads. Usually, such drastic scenarios never happen, and it is assumed that the scheduler is benevolent, so often weaker properties are the most considered

ones by programmers, because the scheduler helps the progress of every thread.

These two properties are denoted as *blocking* correctness conditions. Thus, an algorithm that can guarantee only one of these two conditions is said to be a blocking algorithm. Dealing with blocking algorithms could not be the optimal choice in term of progress, since a thread could halt, crash or being de-scheduled while holding a lock. These kinds of scenarios become a problem since delays in critical sections will harm the progress of other threads. To cope with this, algorithms which guarantee *non-blocking* progress conditions have emerged. They do not rely on locks and critical sections are reduced to a single Read-Modify-Write [22] machine instruction, whose atomicity and termination is ensured by the hardware. By guaranteeing these conditions, we can ensure that if any thread has delays or halts, the others can still make progress. We will say a method is *lock-free* if *some* thread calling a method returns eventually, while it is *wait-free* if *every* thread that calls it returns eventually. The two definitions are similar to the ones of *deadlock-freedom* and *starvation-freedom* since one looks at the progress of the whole system, while the other ensures the progress of every thread. To these two definitions we can add a third weaker non-blocking property: *obstruction freedom*, for which a method returns if the caller thread runs long enough in isolation, i.e. without any thread taking steps that cause contention on shared resource during an interval of time.

The definitions of blocking progress conditions have been given in terms of the presence of a locking primitive, but this is not always the case. In fact, in some implementation is not simple to identify where is a lock and what are the critical sections. Moreover, putting the definitions in these terms we cannot compare the properties, since the non-blocking ones do not rely locking constructs and proper critical section.

The weakest condition we are interested in, is when the whole system progress. In this case if we consider a history H, we will say that a method provides *minimal progress* in H if, in every suffix of H, *some* method has a matching response. While the stronger conditions expect that all the thread progress. In this case we will say

that a method provides *maximal* progress in a history H, if for every suffix of H, every pending active invocation has a matching response.

We say a history is *fair* if each thread takes an infinite number of concrete steps, while it is *uniformly isolating* if, for every $k > 0$, any thread that is active, has an interval where it takes at least k contiguous steps.

Restricting the definitions to only fair histories adds for free the extra requirement that each thread eventually leaves its critical section. The properties that will be restricted to the subset of fair histories are said to be *dependent*. These properties depend on a benevolent behavior of the scheduler. We say that a property is *dependent* if it does not guarantee minimal progress in every history, and it is *independent* if it does.

So, we can redefine the properties in terms of *fair histories* and *minimal* and *maximal progress*.

A method is *deadlock-free* if it guarantees minimal progress in every fair history, and maximal progress in some fair history.

A method is *starvation-free* if it guarantees maximal progress in every fair history.

A method is *obstruction-free* if it guarantees maximal progress in every uniformly isolating history.

A method is *lock-free* if it guarantees minimal progress in every history, and maximal progress in some history. A method is *wait-free* if it guarantees maximal progress in every history.

In this scenario, we have that *deadlock freedom*, *starvation freedom* and *obstruction freedom* are dependent, while *lock freedom* and *wait freedom* are independent.

We can notice that there is a "hole" in the definitions, since we have not defined a property that guarantees minimal progress in every uniformly isolating history and maximal progress in some history. *Clash-freedom* is such property, and it is the weakest non-blocking progress condition.

These properties are summarized in Figure 2.1. The horizontal line divides

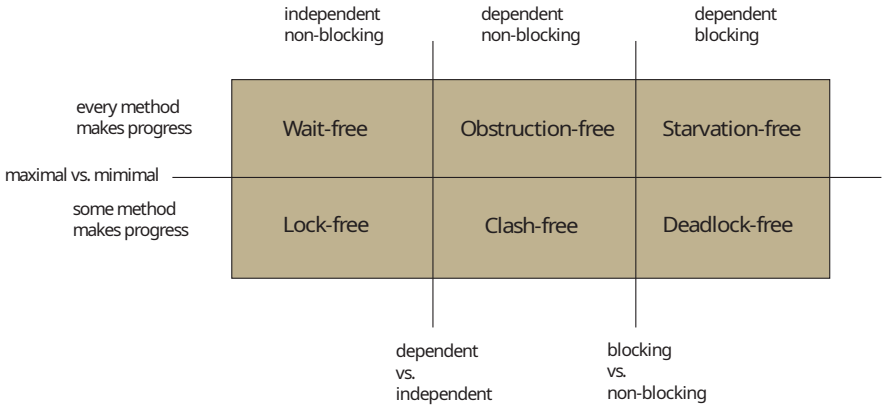

Figure 2.1. Periodic table of progress conditions.

properties that guarantee minimal progress form the ones guaranteeing maximal progress. The vertical line on the right divides blocking properties from non-blocking ones, while the one on left divides dependent properties from independent ones.

2.2 Correctness Conditions

One challenge in concurrent programming is to assure that programs behaves correctly. Consequently, we have to define what a correct behavior is. This is not trivial task for concurrent programs, since we cannot predict in which state is left the shared resource. A simplistic way to say that a concurrent program is correct, is saying that the program execution resembles a sequential one. Intuitively, when we rely on locks, what we are doing is to force the concurrent execution to be sequential in the critical section, since we made only one thread at a time to access the shared resources. We can say that a concurrent execution is correct because we can compare its execution to a sequential one. This is because a sequential execution is simpler to be modeled. The problem is that given a concurrent execution we can interleave the operations in different ways and build different sequential executions for the same program, and for this reason we have different correctness criteria. In this section we will consider only few of them, that are the most common ones: *Sequential Consistency* [24] and *Linearizability* [21].

Before describing what are the most common correctness criteria, we have to recall what is a *sequential history*. In the previous section we gave the definition of a *history* as a finite sequence of invocation and response events on a *shared object*. We will say that a *history* H is *sequential*, if its first operation is an invocation, each invocation is followed by its response, and each response is preceded by its invocation. If the history is *concurrent* if it is not *sequential*.

Since in a concurrent execution we have multiple processes/threads operating on *shared objects*, we can define the *process subhistory*, $H|P$ of a history H as the subsequence of all events in H which are performed by a process P. A *object subhistory*, $H|x$ of a history H is the subsequence of all events in H which are performed on an object x. We will have that two histories H and H' are equivalent if, for every process P, $H|P = H'|P$. Finally, a *history* is *correct* if it is equivalent to a *sequential history* which satisfies a set of correctness criteria. A concurrent object is correct with respect to a correctness condition if we can guarantee that every possible history is equivalent to a sequential history such that it satisfies the correctness criteria.

While describing correctness conditions in the next subsections, we will use a FIFO queue as working example to highlight their main aspects. In a FIFO queue, we have two methods `Enq` which insert an item into the queue and `Deq` which removes and returns the oldest inserted item. We will denote with `P Enq(n) x`, the invocation of the insertion of the element n in the queue x by a process P. Similarly, `P Deq( ) x` denotes a Dequeue operation. With `P ret(n) x`, we will denote the response to an operation, and the value between the parenthesis is the item returned value by the method (if there is any).

2.2.1 Sequential Consistency

The first correctness criterion which we present is *Sequential Consistency*, originally defined by Lamport [24].

A history is sequentially consistent if it is equivalent to a sequential history which

is correct according to the sequential definition of the object. By the definition, an object is *sequentially consistent* if every history associated to its usage is sequentially consistent.

In this case, we have that the histories associated to the object have to be equivalent to a sequential one, which is compliant with the object specification.

If we look at example in Figure 2.2, we have that the history H is equivalent to the sequential history in H', since they have same process subhistories. The history H' is correct according to the sequential definition of a FIFO queue and consequently H is correct.

```
H:  A Enq(1) x          H': B Enq(2) x
    A ret( ) x              B ret( ) x
    B Enq(2) x              A Enq(1) x
    B ret( ) x              A ret( ) x
    B Deq( ) x              B Deq( ) x
    B ret(2) x              B ret(2) x
```

Figure 2.2. Example of Sequential Consistent History. History H is sequentially consistent, since it is equivalent to the sequential history H'.

One important characteristic of sequential consistency is that if we have objects that are sequentially consistent, their composition is not guaranteed to be sequentially consistent. The example in Figure 2.3, is a demonstration of this property. The object subhistories of H, are $H|x$ and $H|y$ and are both sequentially consistent, but if we consider the history H, it is not.

```
H:  A Enq(1) x
    A ret( ) x
    A Enq(1) y
    A ret( ) y          H|x:A Enq(1) x          H|y:A Enq(1) y
    B Enq(2) y              A ret( ) x              A ret( ) y
    B ret( ) y              B Enq(2) x              B Enq(2) y
    B Enq(2) x              B ret( ) x              B ret( ) y
    B ret( ) x              A Deq( ) x              B Deq( ) y
    A Deq( ) x              A ret(2) x              B ret(1) y
    A ret(2) x
    B Deq( ) y
    B ret(1) y
```

Figure 2.3. Example of non-composability of Sequential Consistency. History H, is not sequentially consistent, but the two object subhistories are.

Consequently, Sequential Consistency is not a *local* property, namely it is not closed under composition. Locality is an important property, since it allows to build concurrent systems in a modular fashion.

2.2.2 Linearizability

The *Linearizability* property was introduced by Herlihy and Wing [21].

An object is *linearizable* if every history associated with its usage is *linearizable*. A history H is *linearizable* if it is equivalent to a sequential history H' such that H' is correct according to the sequential definition of the object (the history is sequentially consistent), and if a response precedes an invocation in the original history H, then it must precede in the sequential one H' as well.

In Figure 2.4 we have a linearizable history, H.

```
H:  A Enq(1) x                    H': B Enq(2) x
    B Enq(2) x                        B ret( ) x
    A ret( ) x                        A Enq(1) x
    B ret( ) x                        A ret( ) x
    B Deq( ) x                        B Deq( ) x
    B ret(2) x                        B ret(2) x
```

Figure 2.4. Example of linearizable history. In this case H is linearizable because, it is equivalent to the sequential History H', and we have the same order between responses and invocations.

Linearizability is stronger than sequential consistency, since we have that the history has to be sequential consistent, and for each method must exist an atomic point in the time, between its invocation and response, in which the operation appears to take place, named *linearization point*.

If we consider the history H in Figure 2.4 and we assign linearization points, we can obtain a sequential history by projecting them on the real time axis. If the resulting history is correct with respect to definition of the object, then it is linearizable (Figure 2.5).

With linearizability, differently from sequential consistency, the composition of linearizable object is still linearizable, so it is a *local* property. Herlihy and Wing

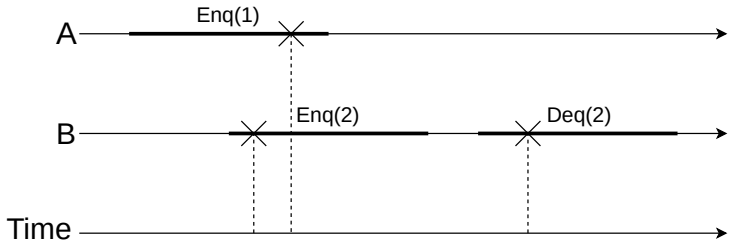

Figure 2.5. Example of Linearizability. We project the operation on time axis, and we track the linearization points of operation (dotted lines). The resulting sequential history (one of the possible) is correct w.r.t the sequential definition of FIFO, so the history is linearizable.

demonstrated that any history is Linearizable if and only if, for every object x, $H|x$ is linearizable. Thus, the property is closed with respect to composition, and so it is *local*. This is important, since we can demonstrate that a system is correct according to linearizability by simply demonstrating that the object that compose it are linearizable.

2.3 Non-Blocking Algorithms

In Section 2.1 we have said that lock-based solutions are commonly used to guarantee mutual exclusion, but blocking algorithms are not suitable for all kinds of scenarios, since they cannot guarantee strong progress guarantees. If we think of a real-time task, resorting to blocking algorithms makes delays in a single thread harm the whole system. Furthermore, the use of locking primitives (most of the time) lacks scalability. In fact, if we rely on a data structure synchronized with locks on a multi-core, we could have very low performance. In particular, locks force only one thread at a time to access the critical section, so in practice we are using only one CPU core at a time, losing the advantages of parallel hardware (and wasting energy in case of active waiting phases e.g., spinning).

We could limit this negative effect and increase parallelism by reducing the granularity of locks. But also, this approach has its drawbacks. In fact, this introduces overheads both in terms of programming effort and resource utilization (like the memory needed to instantiate the additional locks). This approach increases the

parallelism in the program, but it can easily introduce lock dependencies that might lead to a higher probability of causing deadlocks.

Another drawback of using locks is the lack of composability that, as we have said for correctness criteria, it is an important property when writing applications, since it helps into increasing modularity of applications, reducing programming effort and increasing the maintainability and reusability of applications.

Non-blocking algorithms do not rely on locking primitives to synchronize the threads, but they use atomic instructions that are provided by the computing architecture. In most of the cases a non-blocking algorithm relies on atomic Read-Modify-Write (RMW) instructions such as `CompareAndSwap` (CAS), `Fetch&Add`, and other similar primitives. These kinds of instructions are able to read a memory location and then modify it and write back on memory in an atomic fashion, thanks to hardware support.

In general, it is not easy to deal with non-blocking algorithms, since they require a certain programming effort and re-design of critical sections, caring into memory handling. These open to new possible inconsistencies.

The different RMW primitives have different "power", is it possible to describe a hierarchy of non-blocking objects such that an object of a lower level cannot be used to implement objects of higher levels [19]. This hierarchy is based on the *consensus number*, which tells what is the maximum number of processes for which a synchronization primitive is able to solve a consensus problem. `Fetch&Add` or `Test&Set` have a consensus number equal to 2, while CAS primitive has a consensus number which is infinite, so it is widely used in non-blocking algorithms, since it allows to solve consensus for an infinite number of processes, and thus it can be used to implement any kind of wait-free object.

A common problem to deal with when using the CAS instruction for the synchronization is the *ABA problem*. This is a problem that happens when a memory location is read twice, and the same value is found. We might be led into thinking that if a value read is the same, nothing has changed, but this is not true. The

CAS instruction doesn't know whether the value read has changed twice before being executed, so it is possible that reading twice the same value do not imply that nothing has changed. The problem can be summarized in the following steps:

1. A process P_1 read value A from shared memory;
2. A process P_2 is scheduled and writes B on memory;
3. Now P_3 is scheduled-in, and writes again A;
4. P_1 is scheduled again and reads A, since it is the same value it performs a CAS even if it should have not.

Since the CAS instruction doesn't know if a value has changed twice before being executed, it is possible that reading twice the same values makes the CAS succeed anyway, even though something is changed in the meanwhile. In some architectures, the ABA problem is solved using ad-hoc instructions such as `Load-Link` (LL) and `Store-Conditional` (SC). LL reads the last value written in a memory location, while SC performs the update on the same memory area only if no other updates have been performed. However, these instructions are not available on architectures that support the CAS instruction (e.g. x86), so other techniques are needed to deal with this problem.

The ABA problem is critical when we are managing pointers and dynamic memory allocation. In this scenario, when we free a memory buffer, we do not know when it can be reused, since some other thread might hold a reference to that buffer. If such thread uses it after its release, it could cause problems on the integrity of the data structure.

Most of data structure need the use of dynamic memory allocation, and there is the need to cope with safe memory reclamation. Often in literature there is no explicit dealing with memory, since the authors focus more on the algorithm. In real case scenarios, the computing systems have a limited amount of memory, and to use such algorithms we have to find a way to reuse the memory, but there is

still the problem of doing it in a safe manner. This is because we don't know how many threads are concurrently accessing the data structure, and most importantly how many of them are referencing to a specific memory location, so if we are using dynamic allocated memory, we don't know when it is safe to free and reuse it. So, to deal with this problem, some garbage collection mechanisms has been designed, such as Reference Counting [38], Hazard pointers [30], Epoch Based Memory Reclamation [16], Drop the Anchor [4] and Hyaline [33].

2.3.1 Non-blocking Linked List

One of the most used data-structure used in a lot of scenarios is the Linked List. Harris proposed a non-blocking implementation based on CAS primitive [17], which has been used as baseline approach for other non-blocking data structures such as priority queues [26, 36], and sets [16, 29]. The operations supported by the data structure are insertions, deletions and searches of a given key. These functions are shown in the C++ like pseudocode in the listings in Figure 2.6.

The nodes, namely the elements of the list, are constituted by a *key* field and a *next* field which contains a reference to the next node in the list. The data structure will rely on two dummy nodes, *head* and *tail* to delimit its boundaries.

Insertion. To perform an insertion, a new node is created, then two nodes in the list that will surround the new one are found. The search will return in $left$ the last node with key lower than or equal to the node we want to insert, while in $right$ the first node in the list with the key greater than the one we are looking for. The successor of the new node is set as the $right$ node, while the new node is added to the list using a CAS operation on the *next* field of the $left$ node (Figure 2.7). In the case the CAS fails, the procedure restarts from the search of the $left$ and $right$ nodes.

Deletion. The deletion is a little bit more complex, since we cannot simply unlink with a single CAS, skipping the to-be-deleted node and making the field next of its

```
class List<KeyType> {
    Node<KeyType> *head;
    Node<KeyType> *tail;

    List() {
        head = new Node<KeyType>();
        tail = new Node<KeyType>();
        head.next = tail;
    }
}
```

```
class Node<KeyType> {
    KeyType key;
    Node *next;

    Node(KeyType key) {
        this.key = key;
    }
}
```

```
public boolean List::insert (KeyType key) {
    Node *new_node = new Node(key);
    Node *right, *left;

    do {
        right = search (key, &left);
        if ((right != tail) && (right.key == key))
            return false;
        new_node.next = right;
        if (CAS (&(left.next), right, new_node))
            return true;
    } while (true);
}
```

```
public boolean List::delete (KeyType search_key) {
    Node *right_node, *right_node_next, *left_node;

    do {
        right_node = search (search_key, &left_node);
        if ((right_node == tail) || (right_node.key != search_key))
            return false;
        right_node_next = right_node.next;
        if (!is_marked_reference(right_node_next))
            if (CAS (&(right_node.next),
                right_node_next, get_marked_reference (right_node_next)))
                break;
    } while (true);

    if (!CAS (&(left_node.next), right_node, right_node_next))
        right_node = search (right_node.key, &left_node);
    return true;
}
```

```
private Node *List::search (KeyType search_key, Node **left_node) {
    Node *left_node_next, *right_node;

search_again:
    do {
        Node *t = head;
        Node *t_next = head.next;

        /* 1: Find left_node and right_node */
        do {
            if (!is_marked_reference(t_next)) {
                (*left_node) = t;
                left_node_next = t_next;
            }
            t = get_unmarked_reference(t_next);
            if (t == tail) break;
            t_next = t.next;
        } while (is_marked_reference(t_next) || (t.key<search_key));
        right_node = t;

        /* 2: Check nodes are adjacent */
        if (left_node_next == right_node)
            if ((right_node != tail) && is_marked_reference(right_node.next))
                goto search_again;
            else
                return right_node;

        /* 3: Remove one or more marked nodes */
        if (CAS (&(left_node.next), left_node_next, right_node))
            if ((right_node != tail) && is_marked_reference(right_node.next))
                goto search_again;
            else
                return right_node;
    } while (true); /*B2*/
}
```

Figure 2.6. Harris Linked List functions

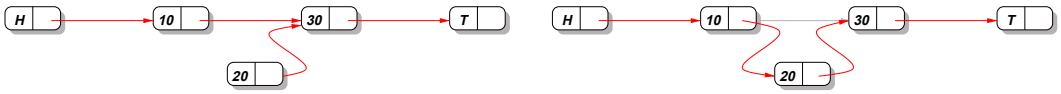

Figure 2.7. Example of Insertion, we want to insert the node 20. First, we create the new node then, with a search, we found the nodes 10 ($left$) and 30 ($right$), and we set 30 as the successor of the newly created node. Then with a CAS on the $next$ field of 10, we finalize the insertion of 20.

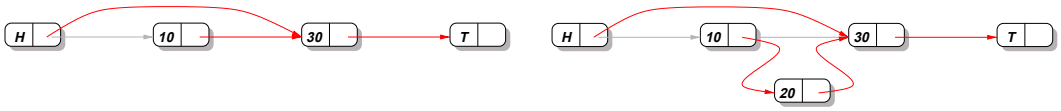

Figure 2.8. Example of incorrect deletion. If we want to try to delete the node 10, we cannot simply use a CAS on the next field of $left$ to disconnect the to-be deleted node. In doing so, a concurrent insert (in this case of the key 20), we will have that the new node could be inserted after the deleted node, being unreachable

$left$ node point to the $right$ node using a single CAS. A deletion performed in this way can cause problems when there are concurrent insertions. In more detail, the node could be inserted after the just deleted node, becoming unreachable as shown in Figure 2.8).

So, what it is done is to associate each node with a 1-bit state machine, which tells whether a node is valid, or invalid due to some concurrent delete operation or ready to be reclaimed. When we want to delete a node, we mark it as deleted - the node in this state is *logically deleted*. When the node is in this state its successor cannot be changed. Consequently, it is safe to disconnect logically deleted nodes using a CAS, since no concurrent insertion can happen on these nodes. Such operation is done by connecting the $left$ and the $right$ nodes using a CAS to make the two adjacent (Figure 2.9). In this way, the node won't be more reachable from the $left$ node and can be safely reclaimed during a grace point, using a proper garbage collection mechanism.

The unchaining of logically deleted nodes, is done either after the deletion or during a search, since the search procedure will look for the first two valid nodes that surround the searched key, if the two are not adjacent, the procedure will try to compact the list using a CAS.

Figure 2.9. Example of correct deletion: To be correctly removed, first we mark the *next* field of the node we want to delete using a CAS. Then, the node can be safely removed.

Search. The search is organized into three steps:

1. Search of first valid $left$ and $right$ node;
2. Check $left$ and $right$ node adjacency;
3. Removal of marked node in case of non-adjacent $left$ and $right$.

The search function will look for the nodes surrounding or that will surround the node associated with the given key, and while doing this it will check if the nodes are adjacent, in the case it will take care of unchaining logically deleted nodes between the two using a CAS operation.

2.3.2 CRCQ

The main work that will be described in next chapter is based on the Conflict Resilient Lock-Free Calendar Queue (from now on CRCQ) described in [27]. This priority queue is a lock-free implementation of the classical calendar queue defined by Brown [6].

A priority queue is an abstract data type which handles key-value pairs, which are events (value) with a timestamp (key). Such pairs are ordered by their priority, represented by the key. Two methods are offered: `enqueue()` for inserting items in the queue, and `dequeue()` which removes and returns the element with highest priority i.e., the element associated to the smallest key. The CRCQ is basically a hash-table, where each bucket is a Lock-Free Ordered Linked List of Harris.

The data structures consist in a pointer to a table, in which there are all the metadata needed for its management. Here we have an array of buckets (called *physical buckets*), that contain ordered sequence of items of the priority queue. To be handled in a lock free manner, each node relies on the last 2 bits of the

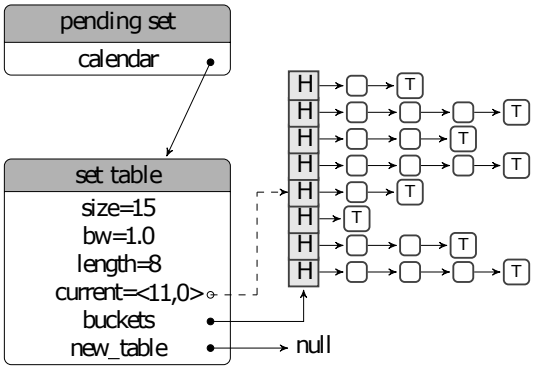

Figure 2.10. CRCQ scheme. The structure is composed of an array, physical buckets, whose length is stored in the length field. The field current will keep the index of the virtual bucket containing the highest priority element, and the current epoch. Each physical bucket is a non-blocking linked list, logically partitioned into virtual buckets using the bucket width (bw).

next field to keep information about the state of the node, a feature that will be analyzed later (differently from the Harris linked list where, excluding when a node is unlinked from the list, we have only two states: Valid and Logically Deleted). Each physical bucket of the table is logically partitioned into *virtual buckets*, which contain a range of keys determined by the *bucket width*. To keep track of the highest priority element, a variable *current* contains the index of the virtual bucket keeping the minimum key. In this variable is also kept an *epoch* number, which is used to achieve conflict resilience, avoiding aborts and to fully restart operations in case of conflicting actions.

The scheme of the data structure is summarized in Figure 2.10.

In order to achieve O(1) amortized cost, the data structure periodically performs a re-size operation, which increases or decreases the number of physical buckets and redistributes the events among them. So, two variables *length* and *size* are kept maintaining respectively the size of the array of physical buckets the number of elements within the table.

Before any operation it is performed, a call to `ReadTable()` gives an up-to-date reference to the table and, in case it is needed, it starts a resize operation that will involve all threads. The field *new_table* will be set upon a resize, and at the end of the procedure will be swapped with the old hash table via a CAS operation.

Enqueue Operation. The enqueue operation takes in input an event e. It takes a reference to the table using the `ReadTable()` procedure, then it computes the destination virtual bucket by dividing the priority with the current bucket width of the table. Then the physical bucket is determined using a modulo operation on the virtual bucket index with the length of the table.

Once the physical bucket is determined and retrieved from the array, a search is performed as done in the Harris's Linked list, retrieving the left and the right node of the one we want to insert. Then the new key is inserted using a CAS operation. With this CAS are also removed the nodes that have been logically deleted between left and right nodes.

When the node has been inserted, the operation is completed by updating *current*. The variable *current* is updated only when the element inserted has a key lower than the minimum, or it is in the same virtual bucket. In this case, we say that the insertion is in the past and, the epoch in the variable current is increased. To complete the operation, the size of the queue is increased by using a `Fetch&Add`.

This operation is described in Algorithm 1

Algorithm 1 Lock-Free Enqueue

```
1:  procedure ENQUEUE(event e)
2:      tmp ← new node(e)
3:      repeat
4:          h ← READTABLE()
5:          nc ← ⌊n / (h.bw)⌋
6:          bucket ← h.table[nc mod h.length]
7:          ⟨left, right⟩ ← bucket.SEARCH(tmp.t, VAL|MOV)
8:          tmp.next ← right
9:          old ← h.current
10:         tmp.epoch ← old.epoch
11:     until CAS(&left.next, UNMARK(right), tmp)
12:     repeat
13:         old ← h.current
14:         if nc > old.value then
15:             break
16:     until CAS(&h.current, old, ⟨nc, old.epoch + 1⟩)
17:     Fetch&Add(&h.size, 1)
```

Dequeue Operation. The pseudocode for the Dequeue operation is described in Algorithm 2.

Algorithm 2 Lock-Free Dequeue

```
1: procedure DEQUEUE
2:     while true do
3:         h ← ReadTable()
4:         oldCur ← h.current
5:         cur ← oldCur.value
6:         myEpoch ← oldCur.epoch
7:         bucket ← h.table[cur++ mod h.t_size]
8:         left ← bucket.next
9:         min_next ← left
10:        if ISMARKED(left, MOV) then
11:            continue
12:        while left.local_epoch ≤ myEpoch do
13:            right ← left.next
14:            if ¬ISMARKED(right) then
15:                if left.ts < curr · h.bw then
16:                    right ← Fextch&Or(&left.next, DEL)
17:                    if ¬ISMARKED(right) then
18:                        Fetch&Add(&h.size, −1)
19:                        return left.event
20:                else
21:                    if left = tail ∧ h.t_size = 1 ∧ ¬ISMARKED(min_next, MOV)
   then
22:                        return null
23:                    CAS(&bucket.next, min_next, left)
24:                    CAS(&h.current, oldCurr, ⟨curr, myEpoch⟩)
25:                    break
26:            if ¬ISMARKED(right, MOV) then
27:                break
28:            left ← UNMARK(right)
```

Like the enqueue, the algorithm retrieves a valid reference to a table as first step. Then it will read *current* in order to determine the virtual bucket, which is supposed to keep the minimum, and the current *epoch*. Using the value of *current*, the physical bucket that contains the minimum is retrieved. A dequeue starts a traversal in order to find the first unmarked node.

During the scan it checks if the node belongs to the current epoch and to the virtual bucket we are interested in. The check on the epoch is performed because

the operation must extract a node which has been a minimum during the lifetime of the dequeue. So, we are sure that the last enqueue operations have been correctly linearized. If the node found has an epoch bigger than the current one, it means that it belongs to an enqueue that has been executed after the current dequeue, hence the operation starts from scratch. If we reach the end the virtual bucket, it means that the bucket is empty. In this case, *current* is updated with a CAS, and the search is executed on the next virtual bucket, by restarting the procedure from the beginning.

The search for a node stops when a valid node, with a minimum timestamp and a correct epoch, is found in the current virtual bucket. Then the node is marked with a `Fetch&Or` as logically deleted. The atomic operation will return the value of the pointer to the node before the operation; thus, this is used to check whether the marking attempt has been successful or not. If the node was unmarked then the dequeue was successful and the size of the queue is decremented, otherwise a concurrent dequeue has extracted the very same item, so the search for a valid node continues.

If during the search a node has been found as marked as `MOV`, it means that a resize is taking place, so the operation restarts from scratch, and helps the resize by invoking the `ReadTable()` procedure.

Resize Operation. As first operation this procedure will check if a resize of the queue is needed and triggers the resize protocol. The resize is a crucial operation, because it guarantees that we have on average a constant number of elements in each bucket, thus ensuring an amortized O(1) complexity.

If `ReadTable()` detects that buckets are not balanced, it triggers a reshuffle protocol. The balancing condition depends on the current size of the table, and a threshold that is a function of the desired number of elements per bucket (DEPB). The version of the CRCQ we are taking in consideration is an adaptive version called ACRCQ, this version computes DEPB at each resize as a function of the

Algorithm 3 Lock-Free ReadTable

```
1: procedure ReadTable
2:     h ← array
3:     curSize ← h.size
4:     if h.new = null∧ resize is NOT required then
5:         return h
6:     compute newSize
7:     CAS(&h.new, null, new array(newSize))
8:     newH ← h.new
9:     if newH.bw ≤ 0 then
10:        begin ← Random()
11:        for j ← 0 to h.t_size − 1 do
12:            i ← (begin + j) mod curSize
13:            retry-loop to mark i−th head as MOV
14:            retry-loop to mark first node of i−th bucket as MOV
15:        MST ← compute bucket width
16:        CAS(&newH.bw, −1.0, MST)
17:    for i ← 0 to h.length − 1 do
18:        while j−th bucket of h is non-empty do
19:            i ← (begin + j) mod curSize
20:            get first node of bucket i as right
21:            get the next node of right as right_next
22:            if right = tail then
23:                break
24:            if right_next ≠ tail then
25:                retry-loop to mark it as MOV
26:            Create a copy of right node
27:            while true do
28:                search for right.ts in a virtual bucket vb of newH
29:                if found a node n with the same key then
30:                    release copy
31:                    copy ← n
32:                    break
33:                else if successful to insert copy as INV with CAS then
34:                    break
35:            if CAS(&right.replica, null, copy) then
36:                Fetch&Add(&newH.size, 1)
37:            else if right.replica ≠ copy then
38:                try-loop to mark copy as DEL
39:            retry-loop to ensure that newH.current.value ≤ vb
40:            retry-loop to mark right.replica as VAL
41:            retry-loop to mark right as DEL
42:    CAS(&q.array, h, newH)
43:    return newH
```

concurrency of performed operations. This is because the number of items per buckets impacts the probability that concurrent dequeue might conflict each other.

When a resize operation is needed, the procedure will be announced by setting the `new_table` field. In this way, when a concurrent thread will call the `ReadTable()` it will join the resize protocol. After the resize is started, the table is freezed by marking all the physical bucket heads and their first node as `MOV`. With this operation, all new upcoming dequeue and enqueue operations will join the resize procedure. Then, the new bucket width is computed by traversing the nodes in the queue and computing the average priority distance between the keys, which is then multiplied with the DEPB in order to get the new bucket width.

Before starting to migrate a node from the old table to the new one, each thread ensures that both the node and its successor are marked as `MOV`, in order to avoid interference with thread that are performing concurrent operations that are not aware of the resize operation. When the two nodes are marked as `MOV`, the thread will insert a copy of the first one marked as `INV` in the new table, waiting to be validated. In fact, multiple copies of the node might be inserted in the new table. Then one of the copies found in the new bucket is set main as replica of the original node. At this point, the replica then will be marked as valid and the original node is marked for deletion. This algorithm is shown in Algorithm 3

2.4 Shared Memory Systems

In common Shared Memory System sall processors can access any region of the memory memory at any time. This is possible because the processors access the memory through the same interconnection bus. In this scheme, all the processors have the same view and access latency of the memory. For this we refer to systems like this as Uniform Memory access (UMA).

Since the increasing number of cores in a processor, and the lack of scalability of a single interconnection bus, it becomes a hard task to keep low the access latency

to memory. For this reason, manufacturers make a portion of cores be faster while accessing a specific partition of RAM banks. In any case, each processor can still access all the memory on the system. With this solution, the access to memory isn't uniform. In particular, latencies are shorter when accessing to local memory, but they become longer when accessing remote memory (local to other cores). This happens because memory request has to pass through multiple interconnections. Since the access latency is not uniform, systems like these are referred to as Non-Uniform Memory Access (NUMA).

2.4.1 NUMA - Non-Uniform Memory Access

NUMA architectures are becoming commonly adopted, in particular in large multi-core systems. In a NUMA system the memory is divided into NUMA nodes, where each one is composed by a processor and the memory directly attached to it. Commonly, NUMA can be encountered on multi-socket machines, where typically a NUMA node is associated to a physical socket, and so composed by the processors and the memory attached to them.

All memory in one NUMA node has the same access characteristics in term of latency and bandwidth for a particular set of processors, thus their latency to access the memory in the same node will be lower than the one needed for accessing the memory of a different node. This is because we have that the memory has to be accessed by every processor and, since with this scheme we have the memory partitioned across nodes, the cost to access memory is increased by the cost of passing through the interconnections between the nodes.

The communication between the nodes has a cost denoted as *NUMA factor*, which is the extra time needed to access remote memory [14]. The number of interconnections traversed to reach the memory is the distance between the nodes, and the greater it is, the higher the cost to pay to access the memory will be(Figure 2.11). For this reason, is important the relative position of processors and memory.

In order to reduce the effects of NUMA is important where tasks are run, and

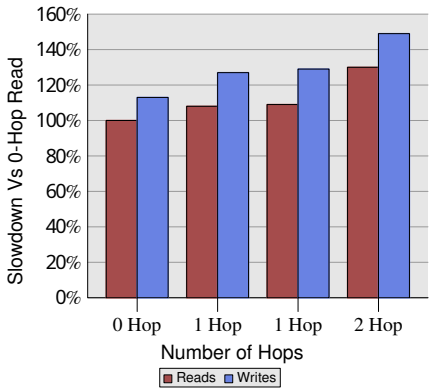

Figure 2.11. Slowdown of remote Reads and Writes in an AMD machine with 4 sockets

where their memory is allocated. The process of assigning memory from the NUMA nodes available in the system is called NUMA placement. On the one hand, having a task that uses memory from a remote node does not affect the correctness of the program. On the other hand, it has an impact on performance.

In fact, when dealing with this kind of systems, the programmer must be aware of the different performance due to remote-memory access latencies and should explicitly handle it.

Modern Operating Systems are designed to take into account NUMA, according to different policies that can be applied. If the Operating System is not NUMA aware, it will simply ignore the different costs to access memory. This is a very simple policy and it is the one used when NUMA first appeared. This does not prevent programs or the OS to run, but the performance is often sub-optimal, since they will depend on how the system is configured, and how programs run.

Another configuration performs memory striping (in some cases, this is done by mapping memory addresses to cache lines in the nodes, in a way to take adjacent cache lines from different nodes [23]). In this way the effects of latency are averaged-out, but again the performance won't never be optimal, making the interconnection across nodes be always in use.

In the case the OS is NUMA aware, it will help the allocation of memory by using policies that are designed to maximize the performance of the applications.

In most of the cases, the default policy of the OS tries to serve the memory

from the node local to the processor that requested it, reducing the utilization of the interconnection. For small applications, this policy usually works well, but this is not optimal with large scale applications, that may use more cores than the ones available in a single node or need more memory than the one available on a single NUMA node. The OS offers specific API that allow the programmer to specify the allocation policy to use for an application or a range of memory addresses and allow migrate the content of memory frames across nodes.

For example, the Linux Kernel is NUMA aware. It provided basic NUMA support from version 2.4, which was further improved in next versions, reaching a full support from version 3.8 when also the scheduler started to become NUMA aware [1, 10]. When we are under NUMA, the memory is allocated following policies that can be per process or per ranges of virtual memory addresses. The two most important policies that can be used are `NODE_LOCAL` and `INTERLEAVE`. With a `NODE_LOCAL` policy, the memory allocation is served form the node local to the one that requested it. `INTERLEAVE` policy will serve memory from all the NUMA nodes in a round robin fashion.

When we move to NUMA machines, the applications in most of the cases see a performance drop due to the non-uniform access latencies and the congestion of the interconnections. Some automatic solution has been developed to leverage the effects, some of these are based on automatic thread and/or memory placement. Some of these solution focus on the migration of task across the nodes [3], others on automatic handling of memory placement, using statistics on memory access patterns to decide how to change the allocation policy for memory pages, or enabling their replication [12]. Other approaches focus on both memory and task migration depending on the type of workload [39]. One problem associated to migrating task across different NUMA nodes is that the performance of a migrated thread could be negatively affected, since after the migration the access have to be done remotely, using the cross-connection.

In the case the performance is critical in the application, the programmer must

put effort into modifying it in order to best fit the architecture. The solution often comes by fine-tuning the allocation policies of the processes or by pinning of threads on NUMA nodes. In other cases there is a complete redesign of the data structures in order to let them work on NUMA without performance loss, by replicating them over NUMA nodes [9] or even distributing them on the different nodes [7, 8, 11]

Chapter 3

Multi-socket communication for Concurrent Priority Queues

In Section 2.3.2 we have illustrated the Conflict Resilient Calendar Queue, a lock-free priority queue. The main problem with this data structure, as happens for many others, is that when moving on NUMA machines, it has a performance drop due to the extra costs induced by remote memory accesses. To have an idea of NUMA effects, consider the chart in Figure 3.1. It shows the average throughput of a CRCQ implementation while running a benchmark for evaluating priority queues with various queue sizes and thread counts. The data structure implementation scales only when we are on up to two NUMA nodes. Then, after a drop of performance, the throughput stays stable. This is because the considered machine has four physical sockets and eight NUMA nodes, so the impact of NUMA breaks in when we are operating on multiple sockets.

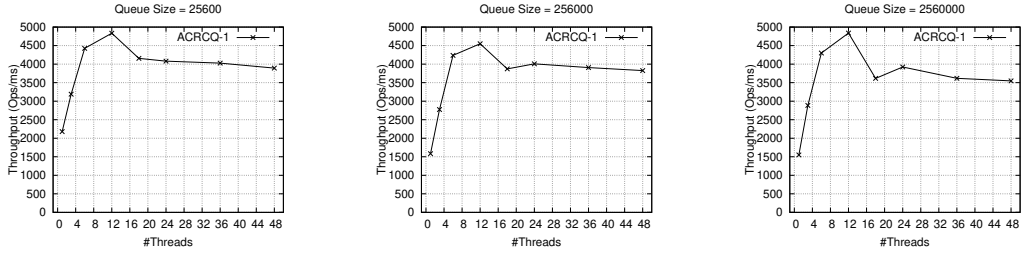

Figure 3.1. ACRCQ base performance.

As we have already discussed in the previous chapter, working on NUMA machines can induce performance degradation due to the memory layout, since remote memory accesses are costlier than local ones. With ACRCQ, there is no explicit handling of NUMA and the default OS policy is used. In this chapter, we will propose several countermeasures that could be applied to best fit NUMA architectures and, we will see their effect on performance.

Since the data structure is based on a hash table, we can partition the table by mapping physical buckets on specific NUMA nodes.

In this way, the memory is delivered only from the NUMA node on which the bucket resides, improving the operation locality on buckets for threads running on the same NUMA node. As shown in Section 2.3.2, the methods of the data structure are enqueue and dequeue of items. An enqueue operation typically involves only one physical bucket, and it spans multiple buckets only when a resize operation occurs. Having physical buckets on specific NUMA nodes is good for this operation since it works on more than one physical bucket only when a concurrent resize occurs, so the enqueue operations will be local to a single NUMA node with high probability. Conversely, a dequeue operation commonly spans over multiple buckets. In fact, the virtual bucket chosen by a dequeue for extraction might result empty due to concurrent extractions, so the operation will look to the next one which resides in the next physical bucket. Mapping physical buckets to single NUMA nodes could reduce data locality of dequeues, resulting in a performance degradation of a potential high-latency operation. In fact, when a virtual bucket is found empty, the dequeue operations restart from scratch. This restart comes with a change of *current*, which changes the target physical bucket and so the locality of the operation. So, having each bucket on a different NUMA node increases the number of remote accesses done by dequeue operations. For this reason, the mapping of physical buckets on NUMA nodes is done by binding multiple adjacent physical buckets to the same NUMA node.

In this way, if a dequeue changes the candidate virtual bucket for the extraction

(and so the physical bucket on which it operates), the operation will still lay in the same NUMA node with high probability.

We clustered multiple physical buckets also because of the resizes operations. In fact, the number of physical buckets (value stored in the field *length*) of the table can change over time after resizes operations. A number too small of physical buckets for the binding to NUMA nodes could end up into varying the destination NUMA node too often while performing an operation. But also, an excessive size for the clusters has its drawbacks. Having large clusters forces physical buckets to be located only on a subset of the NUMA nodes on which the threads are running, forcing a part of them to do only remote work.

We performed an experimental evaluation in which we run the benchmark multiple times with different sizes for the clusters, it turned out that the best size is four. So, we map incrementally each four adjacent physical buckets to the same NUMA node.

Clearly, mapping physical buckets to NUMA nodes is not enough to increase the operation locality, since the NUMA node on which a thread operates will depend on the operation performed. This scenario does not reduce the amount of work that a thread has to do remotely, but it could increase it without bringing any benefit. For this reason, we adopt a delegation technique for which one thread does not execute its own operation (unless it is on the NUMA node where it resides), but waits until another thread (which resides on the NUMA node local to the operation) executes the operation for it.

Delegation techniques have been widely explored in literature to improve locality and performance. One example of this techniques is Flat Combining [18], for which each thread publishes its operation on a shared queue of pending requests, then it tries to acquire a global lock for the structure, becoming a combiner, and then it applies all the pending operations in the queue at once. A similar technique is also used in the Hash Table CPHash [28] which is partitioned in order to maximize cache locality, and for each partition there is a specific server thread that executes

the operations for the other client threads.

Since ACRCQ is already handled in a lock-free manner, the introduction of Flat-Combining might be a waste of resources having a single thread doing the work for all. So, having delegation and non-blocking synchronization techniques in mind, we propose a novel approach such that any thread can do the work destined to the node where it resides, waiting actively for the response for its operation.

In particular, we introduced a communication channel for each NUMA node, where each thread can publish its operations to let them be executed locally from other threads. The channel allows threads to post operations towards NUMA nodes. In this way we expect that the remote accesses are performed only by posting operations on the channel.

Before executing any operation, a thread computes the destination node of its operation. If the operation is an enqueue, it checks on which NUMA node is the physical bucket in which the item must be inserted while, if it is a dequeue this is done by checking the NUMA node of the physical bucket containing the virtual one pointed by current. If the operation has the NUMA node on which the thread is running as destination, it applies the operation on the underlying data structure and returns. If the operation is on a remote NUMA node, the thread publishes its operation on the channel of the remote node and starts waiting. During the wait phase, the thread extracts operations from the channel of its local node and execute them. If there are no pending operations on the local channel, the thread checks if a reply for its operation arrived. If there is a reply it can return. Otherwise, it keeps waiting, trying to extract and execute operations from the local channel.

However, there are two aspects that must be considered. As rated before, operations can change their destination bucket during their execution or while they are pending, hence they may change their locality. This could end into making a thread execute remote operations that could be carried out by some other thread locally. If the locality of the operation changes, we could make it fail, and post it again on the channel of the new NUMA node. Such an activity will be denoted as

repost of an operation. In this way, if an operation changed its locality, a thread does not perform it remotely.

Secondly, in delegation schemes, a thread might be forced to wait for some other thread to execute its operation before it can return, independently if it does it by executing other operations or doing nothing. In particular, while a thread is trying to execute a specific operation, no one else can execute it, since it could cause inconsistencies on the data structure. And so, only threads that are holding an operation that has been extracted from the channel must apply it. Doing so, we are introducing a sort of lock on the top of a non-blocking data structure. This is not a problem for the correctness, but we are weakening the progress guarantees.

To solve this issue, we allow the requestor thread to execute its own operation if there are no pending operations for the current node, or a reply doesn't come within an acceptable amount of time. Anyhow, this cannot be done safely using original algorithms as they are, since it could be possible that an operation is applied multiple times on the data structure. For example, if a thread A is trying to execute an enqueue operation for another thread B, and the requestor decides to execute it, it is possible that they will insert the same item twice.

At first, we ensured correct manipulation of the data structure thanks to a blocking approach. In particular, we introduced a status variable on each operation, to avoid that the requestor thread tries to execute the same operation while some other thread is executing it. Therefore, if a thread does not get a reply for its request, or does not have any work to do, it tries to lock its operation and executes it remotely. Otherwise, if it is already being handled by someone else, it will wait for the reply. This clearly makes the algorithm be blocking, since the requestor has to wait that the delegated thread completes its work. Then, we solved this problem by modifying the data structure algorithms to allow multiple threads to execute the same operation by helping each other.

3.1 Descriptors scheme for delegation

To implement the operation passing scheme, we defined an object, called *operation descriptor*, that keeps all the information to execute an operation (e.g. method invoked and its parameters). Before executing any operation, a thread will instantiate and populate a descriptor, which will be then published on the queue. This will be the used to allow other threads to read the parameters of the operation and to write eventually its result. For the blocking versions of the channels, the descriptor has three fields: *type*, *event* and *state*.

The field *type* will hold the type of the operation issued by a thread. The field *event* will be used to keep the event to insert in the queue in case the operation is an enqueue or to write the extracted item from the queue in the case the operation is a dequeue. The last field *state* will be used to avoid that an operation is executed by two concurrent threads. It has three possible values: `Free`, `Done` and `inHandling`.

When an operation descriptor is instantiated, the field state is set to `Free`, meaning that the operation has to be applied. When a thread gets a descriptor from the channel, it atomically changes the field *state* from `Free` to `inHandling` before applying the method, preventing the requestor to execute the same operation. Once the operation is performed, the thread sets the state to `Done` in order to communicate that the operation has been completed. In the case the operation failed because the channel performs repost of operation, the state is set back to `Free` to let it be executed by some other threads.

3.1.1 Non-Blocking adaptation

To achieve non-blocking progress, a thread that issued an operation must complete regardless of the state of the delegated thread. However, we were forced to introduce a lock on each operation descriptors to prevent that the requestor thread and a delegated thread corrupt the state of the underlying data structure by trying to apply the same method invocation multiple times.

If the requestor wants to execute its own operation, it can happen that it cannot execute it because a delegated thread is attempting to execute it, holding a lock on the operation. Thus, if the delegated thread has delays or it is suspended, the requestor cannot return.

Consequently, if we want to ensure non-blocking progress and correctness, the following must always hold: if the thread that issued an operation and a delegated thread end up executing the same operation, either only one succeed or they agree on the returned value. In this way, if the operation is an enqueue the item is inserted only once, while if the operation is a dequeue only one item will be extracted from the queue. So, we modified the original enqueue and dequeue algorithms (and consequently the organization of nodes and od operation descriptors), with the purpose to allow two threads to execute the same operation at the same time.

For the enqueue operation, we decided to use an approach similar to the one used to migrate nodes during the resize operation of the original algorithm. We allow multiple threads to insert a node in an invalid state in such a way that it cannot be extracted from the queue or unchained during searches. Then, threads will decide which node is the one to be considered as the inserted one through a validation step. Consequently, only one node will be validated, while the others will be marked as `DEL` in order to let them to be unchained and recycled later.

For the dequeue operation the approach is different. The threads that are executing the operation must agree on which is the node to be deleted, or check if someone else has already performed the extraction. To do so, we must know, if a node has been already marked as `DEL`, for which operation it has been deleted.

So, each operation descriptor, along with its parameters, has a 64 bits integer that holds a unique identifier, called ID. If the operation is an enqueue, it will always be 1, while if it is a delete it will be an integer greater than 1. Then, we added to the descriptor a field called *candidate*, which is a pointer to an item in the queue, used by threads to agree on the result of the operation. In the case the operation is an enqueue, this field will be used to agree on the node to be

validated. For an extraction, this will be used by threads to decide which node must be dequeued.

Also the structure defining a node is changed accordingly by expanding the filed next to a 128 bit variable, which now is a pair $\langle next, opID \rangle$, where *next* is the pointer to the next node in the queue, and $opID$ is an identifier which will be set either to 0 if the node is valid, to 1 if the node is being inserted or, if the node is marked as `DEL`, to the identifier of the operation that deleted it. Then, we also added a field *requestor* in the node to hold a reference to the descriptor of the operation that requested its insertion, which will be used during enqueues when a concurrent resize occurs.

Concurrent Enqueue. The pseudocode for the enqueue algorithm is described in Algorithm 4.

As said, the enqueue allows multiple threads to insert a node in an invalid state (we will refer to this state as `TEMP`). This is done by setting the `opID` to 1. In this state, a node can be skipped during a deletion, but it cannot be unchained during the scan of the bucket, since it has to be considered as valid. Every thread that has performed the enqueue operation will try to set the candidate field of the operation descriptor to the `TEMP` node it has inserted, using a one-shot CAS (line 20). If the CAS succeeds, it means that the node the thread inserted is the one that must be validated. Otherwise, if this CAS fails, the node the thread inserted must be ignored by the other operations, so the thread that inserted it will enter in the condition at line 21 and executes a CAS loop to mark the node as `DEL`. Then, the thread will read from the descriptor the node that has been set as candidate and start to help in its validation. To make the node valid, we use again a CAS to clean the `TEMP` mark (line 28), and only the thread that succeeds will update *current* (if needed) and increment the size of the table, while the other threads return.

New Resize. Since we have added a new state for nodes in the queue (`TEMP`) we must consider them explicitly during resize phases.

Algorithm 4 NB-Enqueue

```
1: procedure DO_ENQUEUE(operation op)
2:     tmp ← new node(op.event, NID)    ▷ Take memory from the local NUMA node (NID)
3:     tmp.opID ← 1
4:     tmp.requestor ← op
5:     repeat
6:         h ← READTABLE()
7:         dest_node ← ComputeDestNode(h, op)
8:         if dest_node ≠ NID then
9:             free tmp node
10:            tmp ← new node(op.event, dest_node)
11:            tmp.opID ← 1
12:            tmp.requestor ← op
13:        nc ← ⌊n / h.bw⌋
14:        bucket ← h.table[nc mod h.length]
15:        ⟨left, right⟩ ← bucket.SEARCH(tmp.t, VAL|MOV)
16:        tmp.next ← right
17:        old ← h.current
18:        tmp.epoch ← old.epoch
19:    until CAS(&left.next, UNMARK(right), tmp)
20:    node ← VALCAS(&op.candidate, null, tmp)
21:    if node ≠ null then                    ▷ Mark tmp node as DEL
22:        repeat
23:            curr_widenext ← ⟨tmp.next, tmp.opID⟩
24:            new_widenext ← ⟨tmp.next|DEL, 0⟩
25:        until CAS(&tmp.widenext, curr_widenext, new_widenext)
26:    tmp ← op.candidate
27:    if ¬CAS(&tmp.opID, 1, 0) then          ▷ Validate the candidate
28:        return FAIL
29:    repeat
30:        old ← h.current
31:        if nc > old.value then
32:            break
33:    until CAS(&h.current, old, ⟨nc, old.epoch + 1⟩)
34:    Fetch&Add(&h.size, 1)
35:    return SUCCESS
```

In particular, instead of migrating `TEMP` nodes, we make threads complete their validation during the migration towards the new table. For this reason, when a node is being inserted, we add in it a reference to the descriptor of the operation. In this way, during the migration, if a thread encounters a node in a `TEMP` state, it reads the candidate directly from the descriptor pointed by the node, and tries to set it using a CAS and then proceed with the migration. If the candidate has been already set to a different node, the node we are trying to migrate must be deleted. Therefore, the thread marks the node as `DEL` and does not migrate it. We did this modifying the resize algorithm shown in Algorithm 3, replacing line 26 with the code in Algorithm 5.

Algorithm 5 Migration of a Node

```
26: if right.opID = 1 then
27:     requestor ← right.requestor
28:     if ¬CAS(&(requestor.candidate), null, right) then
29:         repeat
30:             old_state ← ⟨right.next, right.opID⟩
31:             new_state ← ⟨right.next|DEL, 0⟩
32:         until CAS(&right.widenext, old_state, new_state)
33:         continue
34: Create a copy of right node                          ▷ do not copy opID
```

Concurrent Dequeue. When a single dequeue operation can be handled by multiple concurrent threads, we want to ensure that only a single node is extracted from the queue. To achieve this goal the candidate field tells which is the extracted node for the operation, in order to forbid that multiple threads to extract multiple items for the same dequeue. The pseudocode is shown in Algorithm 6.

The first step a thread does is to check the candidate field from the descriptor (Line 10-21). If it is set, some other thread is already performing the same dequeue and has found the minimum, so the thread will try to help in its execution, by attempting to extract the node pointed by the candidate field. If the extraction succeeds, the thread can return without start to search in the physical bucket, while in the case the extraction fails, it reset the candidate field to `null` value.

In the case the candidate field was unset, or the extraction failed, the thread starts scanning the physical bucket as usual but skipping `TEMP` nodes. When a valid node for deletion is reached, the thread proposes it as candidate using a CAS before trying the extraction (Line 29).

If the CAS fails, the candidate is already set, it means that someone else is trying to extract a node for the same operation, so the current thread will try to help the extraction of the node pointed by the field candidate with a single CAS (Line 42-44). After this CAS the nodes is marked, and if it is deleted and the identifier `opID` matches with the ID of the operation descriptor, it means that the extraction is completed, so it can return. Otherwise, if the `opID` of the node doesn't match, it means that a concurrent dequeue which has been issued by another thread extracted

Algorithm 6 NB-Dequeue

```
1:  procedure DO_DEQUEUE(operation op)
2:      while true do
3:          h ← ReadTable()
4:          oldCur ← h.current
5:          cur ← oldCur.value
6:          myEpoch ← oldCur.epoch
7:          bucket ← h.table[cur++ mod h.t_size]
8:          left ← bucket.next
9:          min_next ← left
10:         candidate ← op.candidate
11:         if candidate ≠ null then
12:             if candidate = 1 then
13:                 return null
14:             oldState ← ⟨candidate.next, 0⟩
15:             newState ← ⟨candidate.next|DEL, op.ID⟩
16:             CAS(candidate.widenext, oldState, newState)
17:             if ISMARKED(candidate, DEL) then
18:                 if candidate.opID = op.ID then
19:                     return candidate.event
20:                 else
21:                     Reset op.candidate
22:         if ISMARKED(left.MOV) then
23:             continue
24:         while left.local_epoch ≤ myEpoch do
25:             right ← left.next
26:             opID ← left.opID
```

```
27:         if ¬ISMARKED(right) ∧ opID ≠ 1 then
28:             if left.ts < cur · h.bw then
29:                 CAS(&(op.candidate), null, left)
30:                 candidate ← op.candidate
31:                 if candidate = left then
32:                     repeat
33:                         oldState ← ⟨left.next, 0⟩
34:                         newState ← ⟨left.next|DEL, op.ID⟩
35:                     until CAS(&left.widenext, oldState, newState) ∨
    ISMARKED(left, DEL)
36:                     if left.opID = op.ID then
37:                         Fetch&Add(&h.size, −1)
38:                         return left.event
39:                 else if candidate = 1 then
40:                     return null
41:                 else
42:                     oldState ← ⟨candidate.next, 0⟩
43:                     newState ← ⟨candidate.next|DEL, op.ID⟩
44:                     CAS(candidate.widenext, oldState, newState)
45:                     if ISMARKED(candidate.next, DEL) then
46:                         if candidate.opID = op.ID then
47:                             return candidate.event
48:                         else
49:                             Reset op.candidate
50:                             break
51:             else
52:                 if left = tail ∧ h.t_size = 1 ∧ ¬ISMARKED(bucket.next, MOV)
    then
53:                     if CAS(&op.candidate, null, 1) then
54:                         return null
55:                 CAS(&bucket.next, min_next, left)
56:                 CAS(&h.current, oldCur, ⟨cur, myEpoch⟩)
57:                 break
58:         if ISMARKED(right, MOV) then
59:             Reset op.candidate
60:             break
61:         else if ISMARKED(right, DEL) then
62:             if opID = op.ID then
63:                 return left.event
64:             else
65:                 Reset op.candidate
66:         left ← UNMARK(right)
```

the node first, so the candidate field in the descriptor is reset to `null` and the search for a new minimum restarts. If the candidate was unset, the CAS of Line 29 sets candidate and, the extraction is tried with a CAS loop until the node gets marked. When the node gets marked, we check again whether the `opID` of the node matches with the ID of the operation, to detect if some thread has completed the extraction, and the queue size can be decreased and the thread returns.

The last case is when the queue is found empty. In this case the candidate is set to special value 1, signaling that some thread, executing that specific dequeue operation, has detected the queue as empty (Line 52). If the emptiness condition is not met it continues by looking for an item to be extracted in the next virtual bucket.

3.2 Evaluation Setup

We have explored two different kinds of channels, one of which has two different implementations, so we performed different tests to determine which channel behaves better.

The machine on which the test has been run is a four-socket machine running Linux, with 8 NUMA nodes, mounting four AMD Opteron 6168 processors with 12 cores each for a total of 48 cores.

For each test we run a benchmark for priority queues. The benchmark uses a synthetic workload based on the *Hold-Model* [34], where a *hold* operations is a dequeue followed by an enqueue. The benchmark gradually populates the queue with a given number of elements, which is the queue size. Once the queue reaches the target size, each thread runs for 10 seconds performing operations on the top of the queue. We run the tests varying the number of threads from 1 to 48 and all the results provided are averaged out from multiple runs with the same configuration of the benchmark.

The first set of tests is used to study the execution without the underlying logic of

the calendar queue (i.e., using only the channel) to see what its performance limit is. This is because if the channel alone performs worse than the basic queue, it makes no sense to use it, since the overhead will be too high and cannot bring any performance improvement. We designed two simple tests, the first one consists in enabling the channel but, instead of performing the real enqueue or dequeue operations, they will return directly without interacting with the queue. The second one adds a busy loop of about $12\mu s$ instead of resorting to the underlying operation. We chose this length for the loop, since it is the average time that the ACRCQ takes to perform a queue operation with maximum concurrency. The ACRCQ, when we run the benchmark with 48 cores, observes a throughput of $4000 ops/ms$. This means that a single thread on average performs $83 ops/ms$ and consequently, a thread takes $12\mu s$ to execute a single operation.

With the second set of tests, we study what is the best configuration for the channels. Threads execute real enqueue and dequeues on the calendar queue, showing how the solution work in practice. In particular, we propose different configuration for each channel, and we will use this set of tests to study which configuration is the best one.

Since the considered machine has four physical sockets and eight NUMA nodes, we propose two variation of the channel, one implemented allocating the channels per-NUMA nodes, the other by allocating the channel per-socket.

Moreover, as already said, since we are delegating operations, it may happen that one changes its destination bucket, and thus can also change its locality. So, we add two other implementations of the channels. In the first one, once a thread decides to apply a method, it executes it until success as done with the original algorithm of the calendar queue. In the second one, the thread can perform a repost of an operation in case it changes its destination memory zone. Every time a thread takes a reference of the table to apply a method to the queue, it computes the destination NUMA node of the operation. If the operation is no longer local to the cluster of cores on which the thread is running, it aborts the method execution and

publishes it again on the new destination channel. While, if the requestor thread takes in handling its own operation, it will execute it until it succeeds, ignoring any locality change of the operation.

Previously we illustrated a variation of the algorithms in order to avoid blocking behavior. So, the third type of test is used to study the execution of the different configuration of the queue with this non-blocking version.

Finally, the last set of tests is used to study how the different channels and algorithms behave in time-sharing by executing the evaluation program with one running at the same time using all cores available on the machine. We chose as program the same benchmark run on an unrelated priority queue and we set it up to run for the same time of the tests. This interferes with threads running the channel algorithm, allowing to evaluate scenarios where a delegated thread might be often non-dispatched on any CPU core.

3.3 Nomenclature Used

In this section we introduce the nomenclature that will be used in the next sections where we will illustrate three different implementations of the communication channel used for the delegation scheme. In Section 3.4 we present a channel based on FIFO queues, in Section 3.5 we describe an optimization to the FIFO queue-based channel, by using two distinct FIFO queues, one for enqueue and one for dequeue operations. Finally, in Section 3.6 we present the last communication channel. Here we used a set of shared registers, to implement a dedicated communication channel between pairs of remote threads.

In the end, we have three possible configurations for each channel, and the experimental evaluation uses of all the possible combinations of them:

1. Whether to repost the operation in case of locality change;
2. Whether the channel is NUMA node or per socket;
3. Whether the blocking or non-blocking version is used.

In order to distinguish between the different variants, we will rely on the following conventions.

Type of Channel. The first suffix after the name "NUMA" describes the type of channel used. We have 3 suffices -Q, -2Q, -P. The suffix -Q indicates that we used as channel a FIFO queue, while the suffix -2Q means the we use the two distinct FIFO queues, one for enqueue and one for dequeue operations. The suffix -P indicates a per thread channel.

Range of Channel. The second suffix tells whether the channel is per-socket or per-NUMA node. If there is a suffix -SKT the channel is per socket, while if it is not present, the channel is per NUMA-node.

Type of Operation. The last suffix in the name tells if in the test are performed real operations or not. A suffix -FK will say that the logic of the underlying queue is turned-off, and so the operations on the queue will return immediately. A suffix -BL instead stands for the busy loop of $12\mu s$ before returning, instead of executing the operation.

Type of Policy. A prefix R- will say that an operation is re-posted whenever it changes its locality. Conversely, if it is not present, we assume that an operation is executed till success.

Type of Algorithm. An extra prefix NB- will tell that we are working with the non-blocking version of the algorithm, otherwise the channel has to be assumed as configured with its blocking version.

3.4 NUMAQ: a FIFO-queue based channel

In this solution, we use a non-blocking FIFO queue as communication channel. We chose the LCRQ [32] for the implementation, since it is both lock-free and

linearizable. This FIFO queue relies on `Fetch&Add` and 128-bit CAS instructions. It is a linked list of cyclic arrays of R items each (CRQs), which uses two indexes *head* and *tail* to perform its operations. Each entry of a CRQ is a triple of 128-bit, so when enqueuing the index is obtained through `Fetch&Add` on the tail, while the element of the array is inserted using a CAS operation on the slot of the array, until success. The dequeue operation is similar, the index of the head is increased using `Fetch&Add`, and the value is removed using again CAS. The use of the linked list allows to use other CRQs if the current becomes full while, if it becomes empty, it can be removed. We used it as black box, using it to allow threads to pass each other references to operation descriptors.

3.4.1 Description of the Algorithm

For the implementation of this solution, we allocated one LCRQ for each NUMA node (or for each physical socket), then the threads will publish their operations there. When an operation is published and a thread extracts it from the queue, it is either kept until the operation is completed or until it changes of locality (in the case we are using a repost policy). In both cases the enqueue and dequeue operation becomes very similar, since they perform the same steps: publishing an operation on a queue and executing an operation of other threads until the requested operation is executed by someone else.

For the blocking version, we rely on the *state* field of the operation descriptor to lock an operation and avoid that two threads try to execute it at the same time, causing inconsistency on the state of the queue. What we want to avoid is that: either we have multiple insertions of the very same element, or that for a single dequeue we remove more than one item from the priority queue.

The pseudocode of the operations is described in Algorithm 7. Here we wrap the call to the real operation, which now are called `do_Enqueue()` and `do_Dequeue` (both in the case we are using the original algorithm with the blocking scheme, or we are using the new no-blocking versions). When a thread calls a method, it calls

this wrapper which enables the delegation scheme designed.

As first step a thread allocates the descriptor for the operation it wants to be executed. After that, it enters the waiting loop, where it takes the reference to the table and computes the destination of the operation. If the operation is local, it doesn't enqueue it, skipping the condition and trying to execute the operation by itself. Instead, if the operation is remote, it enqueues it on the channel of the destination cluster. During waiting phases, the threads execute operations from the local FIFO queue. When the local queue is found empty, the thread will try to execute its own operation. Before trying to execute an operation, a thread tries to mark the operation as `InHandling` to avoid it from being executed by the requestor at the same time. If the operation cannot be set as `inHandling`, the thread do not proceed to re-enqueue it, since either the operation is already completed, or it is being handled by its requestor (in this last case it will be executed till success). If the requestor thread fails into taking in handling its own operation, it does not enqueue it, to avoid inserting again the same operation in the queues. So, in the next iteration the requestor will check the result of the operation and, if its operation is not done, it will try to extract another one to do, or in the case the channel is still empty, to execute again its own operation.

The code is designed to work both with the version with repost and without repost, so after executing the operation the thread will check whether the operation terminated correctly or failed because it changed its locality. If the operation taken in handling ends correctly, the thread marks the operation as `Done` and doesn't re-enqueue it. In the case the operation fails because the channel uses repost, it is marked as `Free` and, in the next iteration, the thread will compute the new destination of the operation. If the new destination is remote, it will be posted on the right channel, otherwise it is kept, and the thread performs a new attempt of execution.

The check on the return value of the operation is made before trying to extract an operation descriptor from the FIFO queue. This is because if a thread holds the

Algorithm 7 NUMAQ wrapper

```
1:  procedure ENQUEUE(Event e)                    ▷ or Dequeue without argument
2:      myOperation ← new Operation(enq, e) ▷ for dequeue argument is only deq
3:      operaton ← myOperation
4:      mine ← false
5:      loop
6:          h ← ReadTable()
7:          if operation ≠ null ∧ ¬mine then
8:              dest_node ← ComputeDestination(h, operation)
9:              if dest_node ≠ local_node then
10:                 queue[dest_node] ← enqueue(operation)
11:                 operation ← null
12:         if operation = null then
13:             if myOperation.state = Done then
14:                 return          ▷ In case of dequeue return myOperation.event
15:             operation ← dequeue(queue[local_node])
16:             if operation = null then
17:                 mine ← true
18:                 operation ← myOperation
19:             else
20:                 mine ← false
21:         if ¬CAS(op.state, Free, InHandling) then
22:             operation ← null
23:             continue
24:         if operation.type = Enqueue then
25:             ret ← do_Enqueue(operation.event)
26:             if ret ≠ FAIL then
27:                 mark operation as Done
28:                 operation ← null
29:                 continue
30:         else
31:             ret ← do_Dequeue(&event, operation)
32:             if ret ≠ FAIL then
33:                 mark operation as Done
34:                 operation.event ← ret
35:                 operation ← null
36:                 continue
37:         If operation not null mark it as Free
```

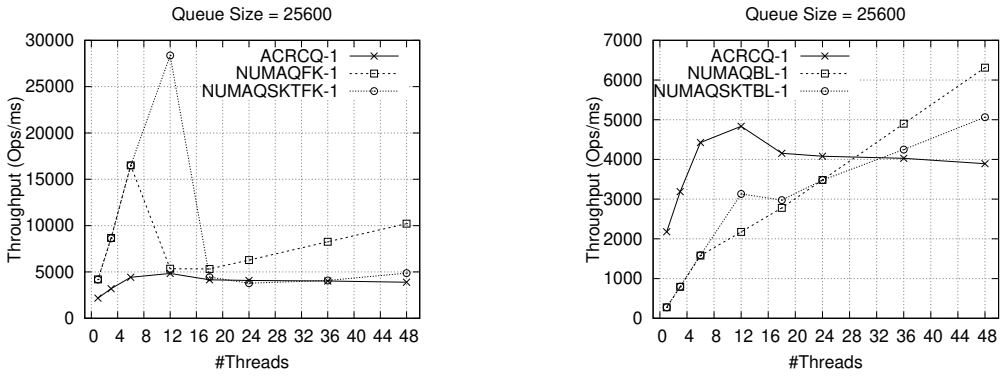

Figure 3.2. Results of Channel Q without operations.

descriptor of an operation which is still local, it might be convenient to complete it. Consequently, a thread checks whether the operation has been completed only when it does not hold an operation descriptor.

3.4.2 Tests on NUMAQ

Figure 3.2 shows the maximum channel performance achievable by the channel at different contention levels.

When running a single NUMA node (or a single socket), the throughput is maximized, since we are not using any channel, and the operations do nothing. When it works on multiple NUMA nodes, the channel starts to be used, thus we see a decrease of throughput since now we are posting and extracting the operations. The channel latency per NUMA node is lower than the mean access time of the underlying priority queue, while in the case the channel is per socket, it is almost the same. Looking at this plot, we expect that when we are in a single NUMA node/socket, the performance is equal to the one without the channel.

When using the busy loop instead of the real operation, we can see that is it possible that the channel can scale. Anyhow it is important to observe that the duration of a real operation can vary depending on the number and the type of concurrent operations that are occurring on the same bucket, so it obvious that when we are on a single NUMA node, with lower concurrency the operations are

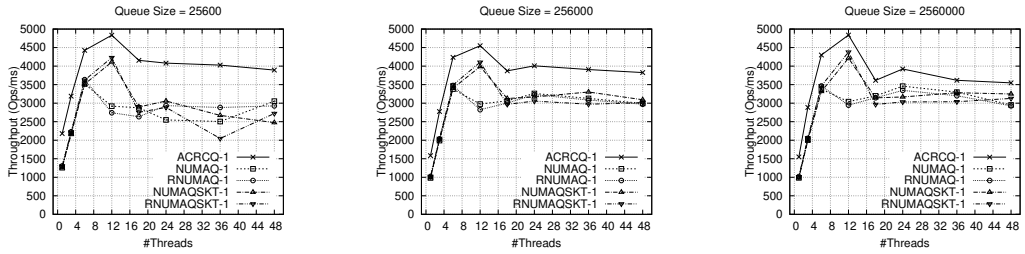

Figure 3.3. Results of Blocking Version using channel Q.

executed faster and all the memory accesses are local, so the estimated duration for the loop is much higher than the actual one when we are in the left part of the plot.

In Figure 3.3 we can see how the channel performs when deployed on top of the real priority queue.

When we are on a single NUMA node, the throughput grows when we are within a cluster but with a lower throughput than the ACRCQ. Conversely, when we are on many clusters the throughput decreases and then stay stable.

The performance drop introduced with this solution can be explained with to the inefficiency of the channel used. Even though it is theoretically a scalable solution, the performance of the channel alone is too close to the one of ACRCQ. In the end, it just adds an extra overhead to the computation.

Furthermore, the usage of the descriptors to pass the operations has a negative impact on performance. In fact, each time a thread wants to read the parameters of the operation it pays an extra remote access, since the memory required to hold the descriptor resides on the node local to the thread that issued the request. In case we are using a repost (R-) policy the situation is even worse, since we are further increasing the number of remote accesses publishing multiple times the operation.

The non-blocking version do not use the three possible states of operations, since in this case is it possible for an operation to be safely executed simultaneously by more threads, since they will agree on a common result. Therefore, the enqueue and dequeue algorithm observes a small difference, since we don't have to care about the *state*. Therefore, we will have the same algorithm, with the lines from 22 to 24 and line 38 removed. The results are shown in Figure 3.4.

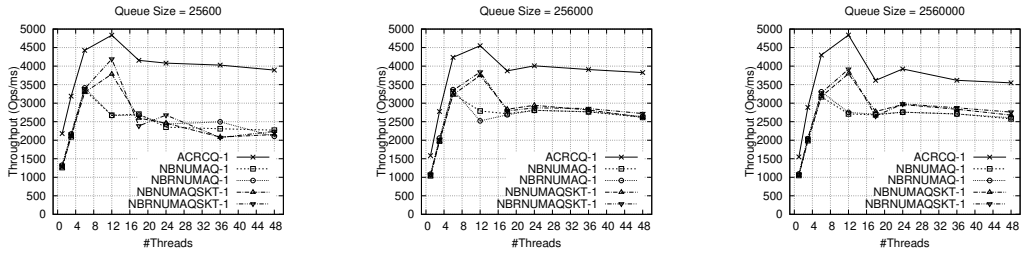

Figure 3.4. Results of Non-Blocking version using channel Q.

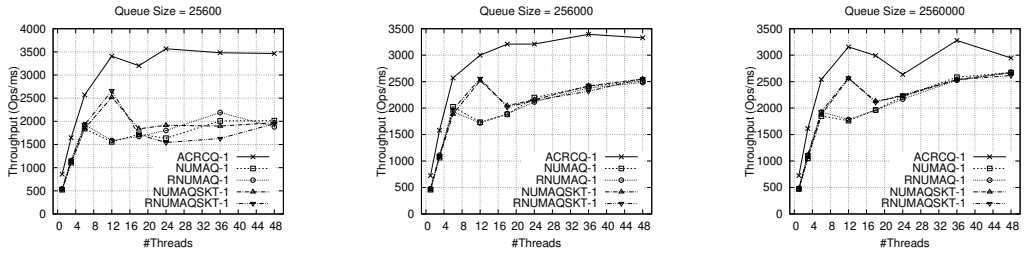

Figure 3.5. Results in Time-Sharing of Blocking Version using channel Q.

We can see that the plot follows the same trend, but with a further decrease of performance with respect the blocking solution. This is because of the additional overhead to the basic algorithm used to perform enqueues and dequeues in order to allow multiple instances of the same operation to be executed at the same time. Such overhead is visible also when we are on a single socket/NUMA node, but when we are in scenarios were we are on many clusters, the use of the *candidate node* becomes heavier since its descriptor has to be one, and it resides on the node where the thread that issued the operation is running, causing additional remote accesses.

Figure 3.5 and Figure 3.6 show the performance in time-sharing with another application. We can see that we have a performance drop which is associated to the lower CPU time assigned to threads. As for the previous test, the performance is better with the blocking solution, but the distance with the performance with the ACRCQ is bigger. It follows that providing non-blocking progress to the channel algorithm do not provide advantages in terms of performance.

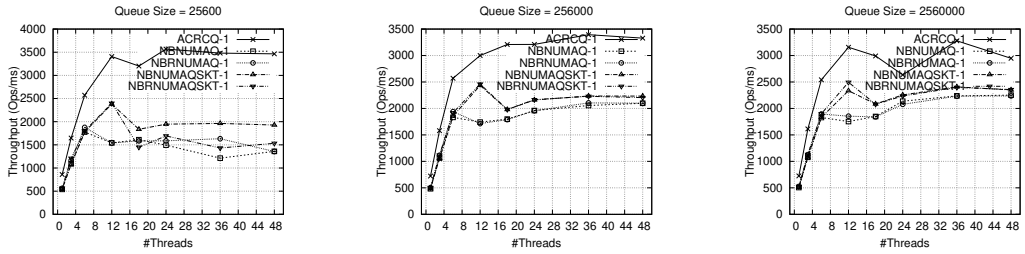

Figure 3.6. Results in Time-Sharing of Non-Blocking version using channel Q.

3.5 NUMA2Q: optimizing NUMAQ

A natural way to optimize the channel based on FIFO queue is reducing the pressure on the queue by using two distinct FIFOs per cluster, one for enqueue operations and the other one for dequeues. This should reduce the contention on the channel queues and boost performance.

3.5.1 Description of the Algorithm

In this scenario, when a thread executes an operation, it will be published to the relative queue on the destination cluster, and then the thread will be waiting for a response performing the same kind of operations for other threads, taking it from the respective queue of same type. The resulting operations are shown in Algorithm 8.

The algorithm is similar to the one used by the Q channel. The only difference is that a thread will execute only operations of the same type of its requests.

3.5.2 Tests on NUMA2Q

As we can see from Figure 3.7, the channel 2Q behaves lightly better than the Q channel.

We can see an enhancement of performance when we are using many clusters, since the use of two separated channels allows to reduce concurrency level on the FIFO queues, resulting in a higher throughput. When we are moving to an execution with busy loops, we can see the behavior is similar to the Q channel.

When we enable the operations, we can see a small improvement in performance

Algorithm 8 NUMA2Q wrapper

```
1:  procedure ENQUEUE(Event e)                          ▷ or Dequeue without argument
2:      h ← ReadTable()
3:      myOperation ← new Operation(enq, e) ▷ for dequeue argument is only deq
4:      operaton ← myOperation
5:      mine ← false
6:      loop
7:          h ← ReadTable()
8:          if operation ≠ null ∧ ¬mine then
9:              dest_node ← ComputeDestination(h, operation)
10:             if dest_node ≠ local_node then
11:                 enq_queue[dest_node] ← enqueue(operation)    ▷ deq_queue for
    Dequeue operation
12:                 operation ← null
13:         if myOperation is Done then
14:             if myOperation.state = Done then
15:                 return          ▷ In case of dequeue return myOperation.event
16:             operation ← enq_dequeue(queue[local_node])      ▷ deq_queue for
    Dequeue operation
17:             if operation = null then
18:                 mine ← true
19:                 operation ← myOperation
20:             else
21:                 mine ← false
22:         if ¬CAS(op.state, Free, InHandling) then
23:             operation ← null
24:             continue
25:         ret ← do_Enqueue(operation.event) ▷ do_Dequeue() in case of dequeue
26:         if ret ≠ FAIL then
27:             mark operation as Done
28:             operation ← null
29:             continue
30:         If operation not null mark it as Free
```

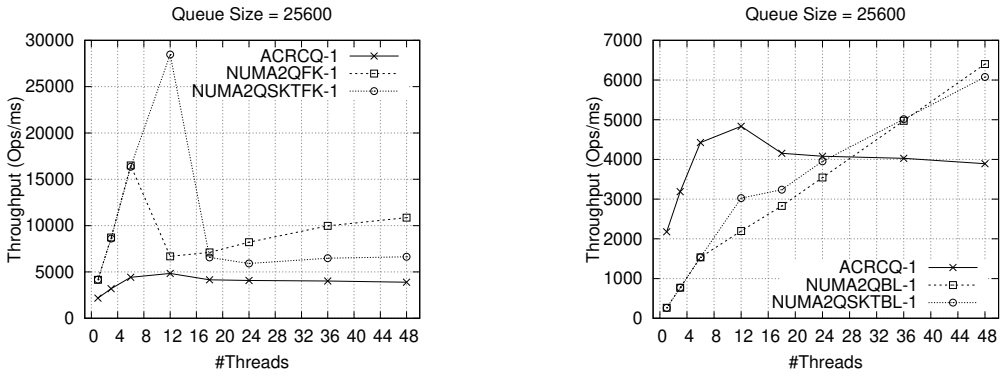

Figure 3.7. Results of Channel 2Q without operations.

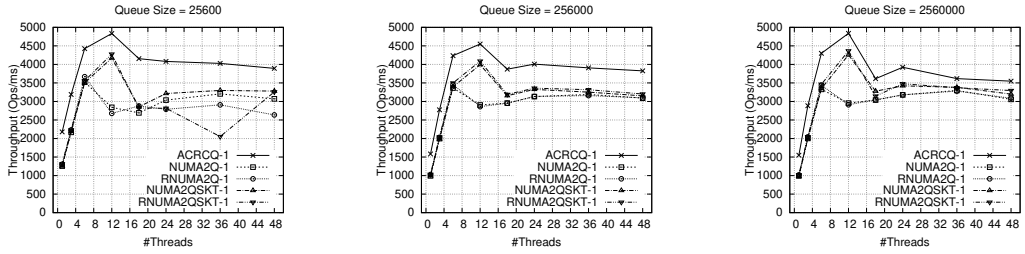

Figure 3.8. Results of Blocking Version using channel 2Q.

with small queue size, but no noticeable difference on medium-big sizes (Figure 3.8). We can also see that the performance is higher when we are on the per-NUMA channel and with the versions using the repost policy, while for a bigger size of the queue we observe that the per-socket channel is always faster than the per-NUMA node channel.

Also, in this case, when we move to the non-blocking version (Figure 3.9), the performance is worse than the blocking one. The worst performance is reached with small queue sizes. The throughput increases as the queue size, but lower than one provided by the previous solution. This happens because splitting queues makes threads find more often the channels empty and so, it tries to execute its own operation (which is carried out remotely), slowing down the execution. This behavior is less frequent with the blocking solution since the operation cannot be carried out remotely by the requestor unless it acquires the lock on the operation.

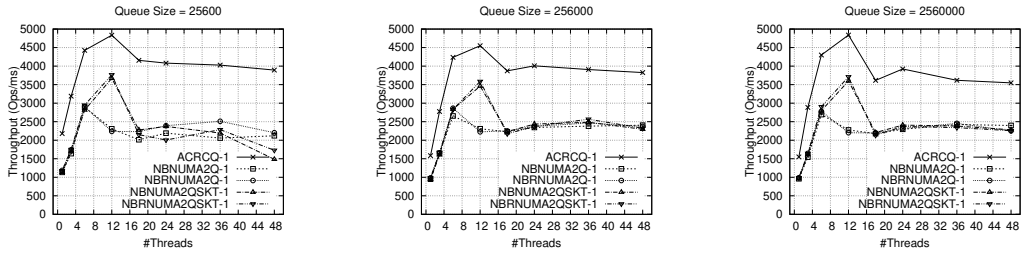

Figure 3.9. Results of Non-Blocking Version using channel 2Q.

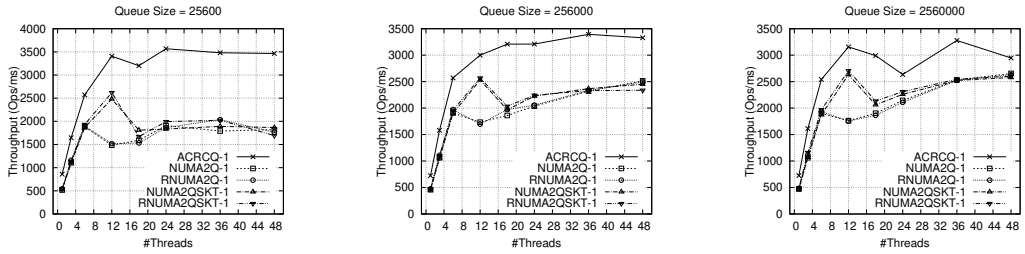

Figure 3.10. Results in Time-Sharing of Blocking Version using channel 2Q.

3.6 NUMAP

Even though FIFO queues have been proven to be scalable, using them as communication channels introduce high overhead Consequently, we tried to simplify the operation passing scheme. To this goal, we introduce a channel that does not use a shared pool of operations between all the threads running on the same cluster, but we use a set of registers such that each core has a dedicated communication channel with one core on each remote cluster. This channel will be called as P-channel.

3.6.1 Description of the Algorithm

In order to implement this scheme, we assumed that the threads operating on the priority queue don't go in time-sharing, and that they will populate the cores one NUMA node at a time in increasing order. In this way, each thread will run on a single core, and we are sure that each thread has one twin for each NUMA node.

In this way the number of NUMA nodes that are needed to serve memory request is minimized, and we are sure that each register will hold communications between two threads that run on two cores of different clusters (we will refer to these pair of

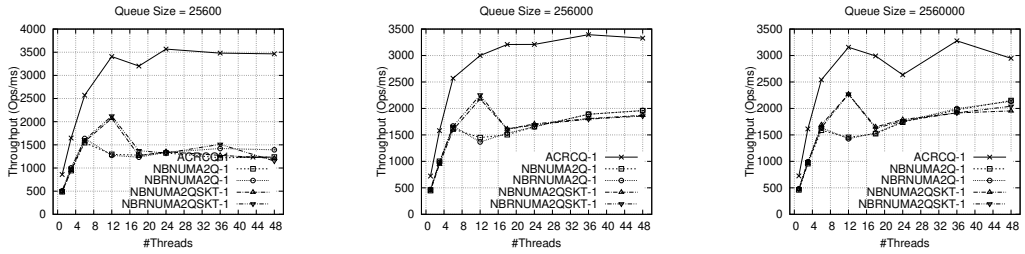

Figure 3.11. Results in Time-Sharing of Non-Blocking version using channel 2Q.

threads as *twin threads*).

We arranged the registers in a $n \times m$ matrix where the number of columns m is equal to the number of threads, and the number of rows n is equal to the number of NUMA nodes with active threads. Such matrix will be allocated by row, in such a way that each row is on a different NUMA node, specifically the index of the row will determine which is the NUMA node on which it is allocated.

To post an operation towards one of its twins, a thread will write in the register on the column with index equal to its thread id, using the row index to specify which is the NUMA node where the twin resides. Then, a thread can read the request operations by reading from the row relative to the NUMA node where it resides, taking care of reading registers only from the columns that are of its twin threads. With this scheme, the reads from the channel will always be local, while the only remote accesses are the one to post on the channel.

Since threads have to send operations and receive responses, we use two of these matrices, one to handle the requests and the other to handle the responses.

The allocation and the interface of the channel are described in Algorithm 9 and 10, while the operation to read and write on it are described in Algorithm 11.

We associated a counter to each register entry in order to determine whether it has been already read or not. This avoids that a thread reads and executes the same operation multiple times, taking place of the *state* field of the operation descriptor. In fact, when an operation is read from the channel the counter is increased. If the counter is equal or greater than 1, it means that the twin has read the operation from the channel, so it is trying to execute it.

Algorithm 9 Channel Initialization

Require: Number of Numa Nodes n , Number of Threads t
1: $Channel \leftarrow$ Array of n pointers to slot
2: **for** i from 0 to n **do**
3: $Channel[i] \leftarrow$ Array of t slots Allocated on NumaNode i
4: **for** j from 0 to t **do**
5: $Channel[i][j]$.Status $\leftarrow FREE$
6: $Channel[i][j]$.ReadCount $\leftarrow 1$
7: **return** $Channel$

Algorithm 10 Get slots methods

Require: Per thread variables: Thread Id tid, Local NUMA Node Id nid, Local thread id $ltid$
1: **function** GETSLOTTO(destNode $node$)
2: **return** $Channel[node][tid]$

3: **function** GETSLOTFROM(sourceNode $node$)
4: **return** $Channel[nid][ltid + (node * num\_cpu\_per\_node)]$

With this channel, the enqueue and dequeue operations are deeply changed, since we don't have a pool with a single access primitive as in the previous case, and here the threads have to manually check whether a request arrived. The operations now are executed according Algorithm 12.

In this case, as first operation the thread acquires a reference to the table using `ReadTable()` and then it computes the destination node. If the operation is local, it is executed till success. If the operation is remote, it acquires a reference to the slot to pass the operation to the respective twin thread, and a reference to the slot where it will receive the reply. Then the thread starts looping waiting for the response. In one iteration it calls the routine `ReadAndExec()`, described in algorithm 13, in which it iterates over the slots from the twin threads, and executing their operations. In the case all the slots are empty, it increases the *attempts* counter otherwise it resets it. This counter is used to determine how many loops the thread has done without doing any operation. If the counter goes above a certain threshold, the thread attempts to read the operation from the slot where it published it, revoking its request, to post it then on another slot if the operation changed its locality, or

Algorithm 11 Read and Write Operation

```
1: function WRITE(Slot slt, Operation op)
2:     repeat
3:     until CAS(&(slt.status), FREE, WRITE)
4:     Copy op into slt.op
5:     slt.ReadCount = 0
6:     slt.state = FREE
7:     return True
1: function READ(Slot slt, Operation *op)
2:     if ¬CAS(&(slt.status), FREE, READ) then
3:         return False
4:     Copy slt.op into *op
5:     val ← F&A(&(slt.ReadCount), 1)
6:     slt.state = FREE
7:     if val = 0 then
8:         return True
9:     else
10:        return False
```

to execute it by itself.

This threshold is set to 100 but, evaluating the performance running the benchmark trying with higher thresholds both for enqueue and dequeue operations we didn't see any remarkable change in performance.

The revoking of the operation happens only if the twin thread didn't read the slot, otherwise the operation is considered as taken in handling by the twin. In this case the requestor will keep waiting for a response. The thread after one iteration across its slots, done with the call to `ReadAndExec()` will check if a reply arrived. If there is one, it can read the returned value and return.

In case we are using a channel with repost, the repost of the operation is not done by the thread that performed the attempt to execute the operation. In this case, the thread that attempted to execute the operation communicates it to the requestor by writing on the response slot the failure of the operation using a new *ret* field. The requestor thread when reads the negative response from the slot performs the repost by itself.

When we move to the non-blocking version of this channel, we have to do some changes in order to use the new algorithms.

Algorithm 12 P operation

```
1: procedure ENQUEUE(Event e)                          ▷ nothing in case of deq
2:     Operation ← newOperation(enq, e)                ▷ argument is only deq in case of dequeue
3:     h ← ReadTable()
4:     destNode ← ComputeOpDestination(h, Operation)
5:     if destNode = NID then
6:         return do_enq(e)/do_deq()
7:     FromMeSlot ← RequestCh.getSlotTo(destNode)
8:     RespSlot ← ResponseCh.getResponseSlotFrom(destNode)
9:     Write(FromMeSlot, Operation)
10:    Attempts ← 0
11:    loop
12:        if Attempts > MAXAttempts then
13:            if Read(FromMeSlot, &Op) then
14:                h ← ReadTable()
15:                NewdestNode ← ComputeOpDestination(h, Operation)
16:                if NewdestNode ≠ destNode ∧ NewdestNode ≠ NID then
17:                    destNode ← NewdestNode
18:                    FromMeSlot ← RequestCh.getSlotTo(destNode)
19:                    RespSlot ← ResponseCh.getResponseSlotFrom(destNode)
20:                    Write(FromMeSlot, Operation)
21:                    Attempts ← 0
22:                else
23:                    return do_enq(e)/do_deq()
24:            Attempts ← 0
25:        if ReadAndExec() then
26:            Attempts ← 0
27:        else
28:            Attempts + +
29:        if Read(RespSlot, &Op) then
30:            if Op.ret = FAIL then
31:                h ← ReadTable()
32:                destNode ← ComputeOpDestination(h, Operation)
33:                FromMeSlot ← RequestCh.getSlotTo(destNode)
34:                RespSlot ← ResponseCh.getResponseSlotFrom(destNode)
35:                Write(FromMeSlot, Operation)
36:                Attempts ← 0
37:            else
38:                break
39:    return Op.ret/Op.event
```

Algorithm 13 ReadAndExec()

```
1: ret ← False
2: Op ← EmptyOp
3: for each NumaNode n do
4:     SlotToMe ← RequestCh.getSlotFrom(n)
5:     if Read(SlotToMe,Op) then
6:         SlotFromMe ← ResponseCh.getSlotTo(n)
7:         if Op.Type = enq then
8:             Op.ret ← do_enq(Op.event)
9:         else if Op.Type = deq then
10:            Op.event ← do_deq()
11:        Write(SlotFromMe, Op)
12:        ret ← True
13: return ret
```

We decide to fall back to the same descriptor passing scheme used in the previous solution. This is because we must have a unique copy of the operation descriptor for each operation to handle correctly the synchronization of threads through the candidate field. In this case, the slots are used to pass between threads the reference to the descriptor.

3.6.2 Tests on NUMAP

As for Q channels, we have evaluated the performance of the P-channel under different scenarios. Figure 3.12 shows the throughput running without interacting with the underlying data structure. Here, we can see that this kind of channel is much faster than the previous ones. This is because we are on a single cluster the channel has almost no overhead, while in the previous case a thread checks the locality of the operation even in the case the operation is local. Another factor of improvement is given by the fact that (at least in the blocking version) the operations are not passed through a reference to the descriptor, but it is copied across the register.

In the end, when we pass to an execution with busy loops the expected performance is quite similar to the one seen with the other two types of channels.

When we apply the real operations and with this channel, we can effectively

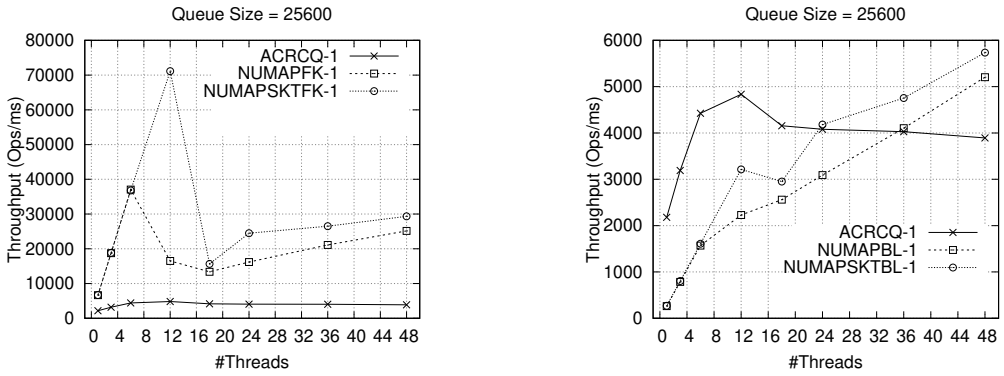

Figure 3.12. Results of Channel P without operations

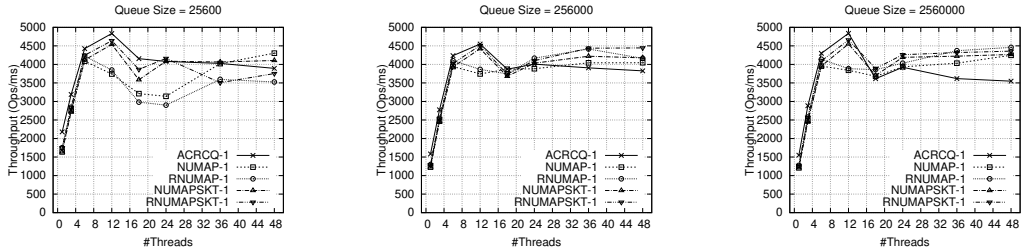

Figure 3.13. Results of Blocking Version of P Channel with operations.

increase the performance of the priority queue. When we are on medium and large queue sizes, we can see that the channel provides an enhanced scalability. As shown in Figure 3.13, it almost reaches the same performance of a single socket when running at maximum thread count. Furthermore, its looks like that it is possible that performance can increase further with larger thread counts.

As said this enhancement can be explained by the fact that in this way, we are not passing reference to operation descriptor, but the content of the descriptor itself. This effectively reduces the number of remote accesses, since the only ones are now used for sending operations requests and their responses and enqueue and dequeue operations are executed mostly local.

Then when we try to move to the Non-blocking version.

As we can see from Figure 3.14 there is a performance drop with respect to the blocking version and we are also below the baseline implementation, but we are above all the previous solution. This is because of the overhead of the algorithm,

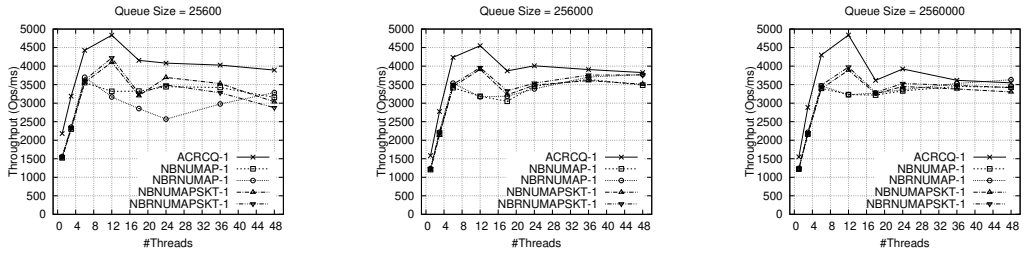

Figure 3.14. Results of Non-Blocking Version of P Channel with operations.

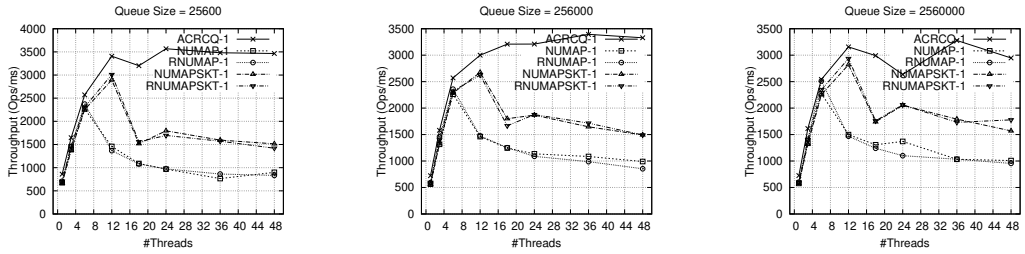

Figure 3.15. Results in Time-Sharing of Blocking Version using channel P

and because we are using again descriptors to pass the operations. But we are close the performance of the base ACRCQ.

Finally, we can see in in Figure 3.15 and 3.16 that, when we go in time sharing, this solution is the one which sees the biggest performance drop. This is expected because in this solution, a thread delegates only one of its twins to execute its operations. In time-sharing the thread are dispatched with less frequency, thus an operation is delayed more and so a thread is forced into revoking the operation to a twin more often.

3.7 Comparison with similar approaches

In this final section, we show the difference in performance with solutions that are available in the literature.

Our first competitor is Flat Combining [18], because of the similar nature of the solutions. In fact, Flat Combining is a well-known example of delegation mechanism, adopting this scheme outperforms many state-of-the-art non-blocking data structures. Although, it is not specifically designed for NUMA, but its idea is often

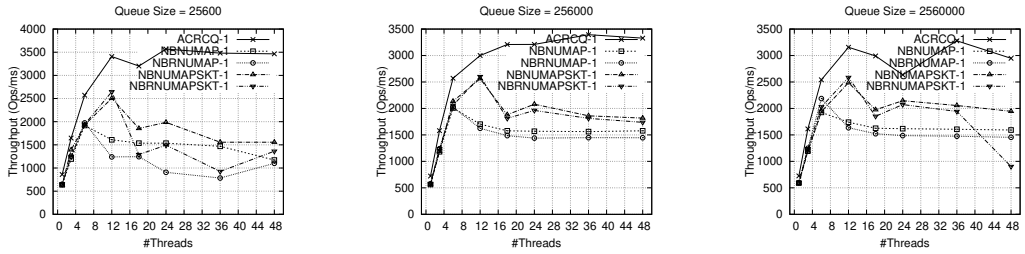

Figure 3.16. Results in Time-Sharing of Non-Blocking version using channel P

used behind many data structures optimized per NUMA, since it allows to improve cache locality. So, we adapted a priority queue using flat combining to run with our benchmark.

The second one is based on the work of Fatourou and Kallimanis [15]. This work addresses the weak points of combining techniques. Here, they proposed to split processors into clusters, each one applying a flat-combining mechanism on data structures. Then, when a thread becomes a combiner for its cluster, it tries to acquire a queue lock in order to modify the data structure by applying the request only from the threads of its cluster. This scheme claims to reduce the contention on the interconnect across NUMA nodes. Here, accesses to the underlying data structure are periodically granted to one cluster of cores (i.e., from the same socket or NUMA node) at a time. This solution was adapted by Morrison and Afek to optimize the LCRQ [32] by relying on a shared variable, which keeps the current cluster enabled to access the data structure. Before executing an operation, a thread will check if it is running on the enabled cluster. In this case, it performs its operation, otherwise it waits for a while, then it tries to update the shared variable to the id of its cluster, and finally it executes its operation. This solution does not introduce any lock, since in the end every operation is executed. We embedded this scheme into the CRCQ, creating the ACRCQH.

We compared these two solutions with the one presented in the previous section. We do not compare with all the possible variants, but we choose the best perfroming ones. In particular, we choose a socket as CPU cluster and we enabled repos repost, expect for the blocking version of NUMAQ, which performs better without repost.

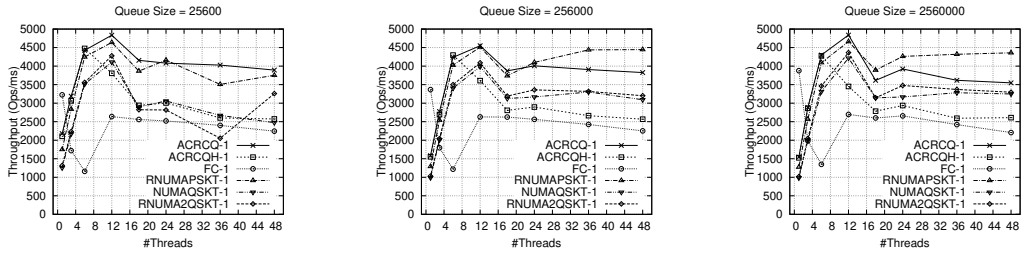

Figure 3.17. Best of Blocking solutions versus FlatCombining (FC) and hierarchical optimization of the ACRCQ (ACRCQH)

We can see the results in Figure 3.17 and 3.18 and Figure 3.19 and 3.20 the results with time-sharing.

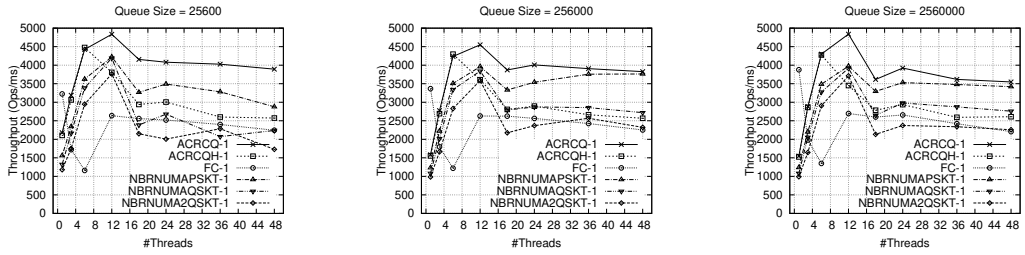

Figure 3.18. Best of Non-Blocking solutions versus FlatCombining (FC) and hierarchical optimization of the ACRCQ (ACRCQH)

As we can look from Figure 3.17, when we are running on many clusters, the blocking version of channel P is always faster than ACRCQH and FC. The Q and 2Q channels scale better when the queue sizes are medium and medium-large.

When running with the non-blocking arrangement of our delegation mechanism (Figure 3.18), only the channel P is above the two solutions for small queue sizes, while the channel 2Q never goes above the flat combining.

For medium and medium-large queue sizes, the P channel is the one showing the best performance over the other solutions, while Q channel shows performance lightly better than ACRCQH. The 2Q channel instead is the one behaving worse, since cannot reach Flat Combining.

In the end, we can see that the ACRCQH is not a real improvement. This is because operations do not have a specific locality, and the items of the queue are not distributed across NUMA nodes, since the ACRCQH still relies on the default

allocation policy. It can be that this add extra remotes accesses, that ends up increasing the contention on the interconnections instead of decreasing it.

Also, flat combining does not improve the performance, but this because the implementation taken was not designed to work on NUMA machines, causing most of the work to be done remotely by the combiner thread.

In Figure 3.19 shows the solutions behavior in time sharing. Flat Combining keeps almost the same performance that it had in the case not in timesharing instead of seeing a drop because of its blocking nature. What happens is that in the waiting phase, the threads of flat combining instead of spinning, they yield their time slice in order to allow other threads to run on the CPU, so the solution is not affected too much by the presence of the other process running.

In the end, for small queue sizes the channel-based solutions do not reach the same performance of the previous cases, with the performances staying below the ones of all the competitors. With bigger queue sizes, at least the channel Q can perform better than both Flat Combining and ACRCQ. When we are with the non-blocking solution (Figure 3.20) both the channel Q and 2Q show a higher throughput than flat combining.

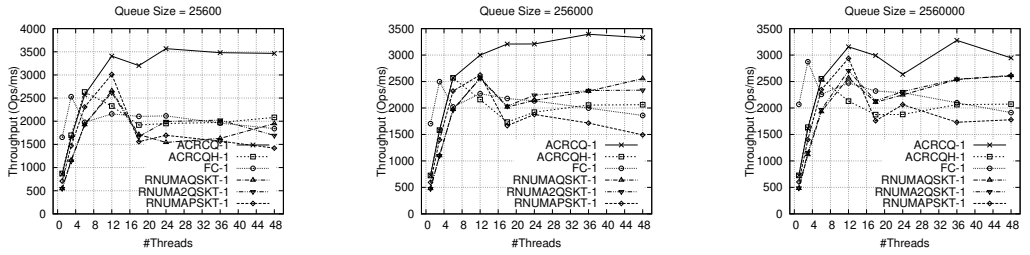

Figure 3.19. Best of Blocking solutions versus Flat Combining (FC) and hierarchical optimization of the ACRCQ (ACRCQH) in Time-Sharing

As we can see, the different solutions behave differently depending on the size of the workload. If we are in an environment where we don't have concurrent programs that require much CPU time, and we don't have the need of strong progress guarantees the P-channel is the one that achieve the best results. Instead, if we need stronger progress the P-channel is again the best solution among the ones explored

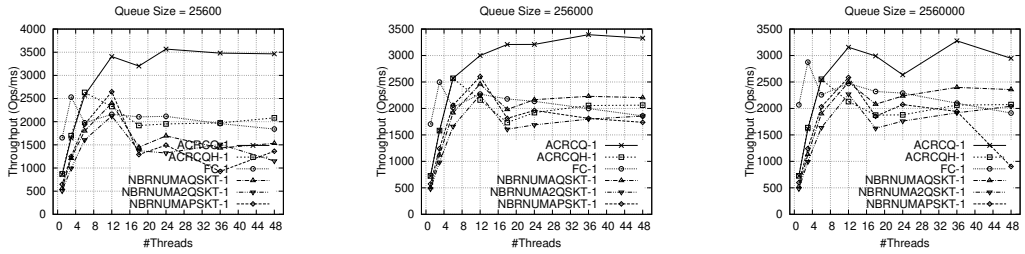

Figure 3.20. Best of Non-Blocking solutions versus Flat Combining (FC) and hierarchical optimization of the ACRCQ (ACRCQH) in Time-Sharing

but, in that case, the basic ACRCQ is still a better solution.

Chapter 4

Conclusions

In this thesis, we have illustrated which are the main challenges a programmer has to tackle when working on NUMA systems and when dealing with concurrent data structures. We have shown what happens when we move an application on such architectures, and what are the problems that this imply. We have presented a brief analysis and possible solutions that could be applied to a high-performance priority queue, to overcome the drop of performances when we move to a NUMA machine.

We started from the simple observation that the reference solution is based on a hash table and we have re-arranged it in a way that matches the underlying architecture, distributing it over NUMA nodes. We then introduced a delegation technique in this environment, to allow threads that work on the priority queue to do not execute work that could be on a remote NUMA node, but to pass it to other threads, and then "wait" for a response.

State-of-the-art delegation techniques usually put a limitation on the progress guarantees, since they usually rely on locking primitives to allow only one thread to access the shared data structure. We also proposed a solution to avoid this blocking behavior, by modifying the basic data structure in order to allow multiple instances of the same operation to be executed at the same time.

We tried with three different implementations for the operation-passing mechanism, two implementations use a lock-free, linearizable FIFO queue and a third

one in which we set up a mapping between threads. In this last solution we have introduced dedicated communication channels between pairs of threads residing on different NUMA nodes, using a simple register.

For at least one solution, the P-channel, we were able to increase effectively the performance of the baseline solution, but with the main drawback of becoming blocking.

We adapted and tested the same solutions in order to achieve lock-freedom, letting the thread to avoid to wait for a response in certain cases, but since the excessive overhead of the found solution we weren't able to increase the performance further, getting at best performance similar to the baseline solution.

All these analyzed solutions are able to outperform similar solutions, like Flat Combining, which is currently one of the state-of-the-art delegation techniques, in all the tested scenarios.

Anyhow, a lot of work can be still done on these solutions: although these have shown good result, they could be further improved. In fact, there are some aspect that we left uncovered. For example, in the P-channel we didn't consider explicitly the case where some threads do not have any twin. This does not prevent these threads to work, but they are forced to carry out their own operations independently of their locality. Also, a re-design of the non-blocking solution here illustrated is needed, since it introduces an excessive overhead, that holds us to get any real improvement. A possible step in this direction could be done by introducing other synchronizations mechanisms based on Hardware Transactional Memory.

Anyhow, it is still possible to find other solutions for an implementation of the channel which could guarantee a higher scalability, since we do not need to give a strict ordering to pending operations in the channel.

Since we have already weakened the progress condition, another possible step toward a better improvement could be weakening the correctness criterion. So, instead of trying to be Linearizable, we could pass to a Sequential Consistent data structure. With sequential consistency we could avoid performing part of the en-

queue operation, since only the enqueue close to the minimum must be executed when requested. In this way, the execution of the enqueue can be done in a second moment, and this could bring improvement with many different workloads.

Bibliography

[1] Linux Kernel documentation numa memory policy. `https://www.kernel.org/doc/html/latest/admin-guide/mm/numa_memory_policy.html`.

[2] Alistarh, D., Kopinsky, J., Li, J., and Shavit, N. The spraylist: A scalable relaxed priority queue. In *ACM SIGPLAN Notices* (2015), vol. 50, ACM, pp. 11–20.

[3] Blagodurov, S., Zhuravlev, S., Fedorova, A., and Kamali, A. A case for numa-aware contention management on multicore systems. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques* (2010), ACM, pp. 557–558.

[4] Braginsky, A., Kogan, A., and Petrank, E. Drop the anchor: lightweight memory management for non-blocking data structures. In *Proceedings of the twenty-fifth annual ACM symposium on Parallelism in algorithms and architectures* (2013), ACM, pp. 33–42.

[5] Breshears, C. *The art of concurrency: A thread monkey's guide to writing parallel applications.* " O'Reilly Media, Inc.", 2009.

[6] Brown, R. Calendar queues: a fast 0 (1) priority queue implementation for the simulation event set problem. *Communications of the ACM 31*, 10 (1988), 1220–1227.

[7] Calciu, I., Gottschlich, J., and Herlihy, M. Using elimination and delegation to implement a scalable numa-friendly stack. In *Presented as part of the 5th {USENIX} Workshop on Hot Topics in Parallelism* (2013).

[8] Calciu, I., Mendes, H., and Herlihy, M. The adaptive priority queue with elimination and combining. In *International Symposium on Distributed Computing* (2014), Springer, pp. 406–420.

[9] Calciu, I., Sen, S., Balakrishnan, M., and Aguilera, M. K. Black-box concurrent data structures for numa architectures. *ACM SIGOPS Operating Systems Review 51*, 2 (2017), 207–221.

[10] Corbet, J. Numa in a hurry. `https://lwn.net/Articles/524977/`, nov 2012.

[11] Daly, H., Hassan, A., Spear, M. F., and Palmieri, R. Numask: high performance scalable skip list for numa. In *32nd International Symposium on Distributed Computing (DISC 2018)* (2018), Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[12] Dashti, M., Fedorova, A., Funston, J., Gaud, F., Lachaize, R., Lepers, B., Quema, V., and Roth, M. Traffic management: a holistic approach to memory placement on numa systems. *ACM SIGARCH Computer Architecture News 41*, 1 (2013), 381–394.

[13] Dijkstra, E. W. Solution of a problem in concurrent programming control. *Commun. ACM 8*, 9 (sep 1965), 569–.

[14] Drepper, U. What every programmer should know about memory. *Red Hat, Inc 11* (2007), 2007.

[15] Fatourou, P., and Kallimanis, N. D. Revisiting the combining synchronization technique. In *ACM SIGPLAN Notices* (2012), vol. 47, ACM, pp. 257–266.

[16] Fraser, K. Practical lock-freedom. Tech. rep., University of Cambridge, Computer Laboratory, 2004.

[17] Harris, T. L. A pragmatic implementation of non-blocking linked-lists. In *International Symposium on Distributed Computing* (2001), Springer, pp. 300–314.

[18] Hendler, D., Incze, I., Shavit, N., and Tzafrir, M. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures* (2010), ACM, pp. 355–364.

[19] Herlihy, M. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS) 13*, 1 (1991), 124–149.

[20] Herlihy, M., and Shavit, N. On the nature of progress. In *International Conference On Principles Of Distributed Systems* (2011), Springer, pp. 313–328.

[21] Herlihy, M. P., and Wing, J. M. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS) 12*, 3 (1990), 463–492.

[22] Kruskal, C. P., Rudolph, L., and Snir, M. Efficient synchronization of multiprocessors with shared memory. *ACM Transactions on Programming Languages and Systems (TOPLAS) 10*, 4 (1988), 579–601.

[23] Lameter, C., et al. Numa (non-uniform memory access): An overview. *Acm queue 11*, 7 (2013), 40.

[24] Lamport, L. How to make a multiprocessor computer that correctly executes multiprocess programm. *IEEE transactions on computers*, 9 (1979), 690–691.

[25] Lamport, L. A simple approach to specifying concurrent systems. *Communications of the ACM 32*, 1 (1989), 32–45.

[26] Lindén, J., and Jonsson, B. A skiplist-based concurrent priority queue with minimal memory contention. In *International Conference On Principles Of Distributed Systems* (2013), Springer, pp. 206–220.

[27] Marotta, R., Ianni, M., Pellegrini, A., and Quaglia, F. A conflict-resilient lock-free calendar queue for scalable share-everything pdes platforms. In *Proceedings of the 2017 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (2017), ACM, pp. 15–26.

[28] Metreveli, Z., Zeldovich, N., and Kaashoek, M. F. *Cphash: A cache-partitioned hash table*, vol. 47. ACM, 2012.

[29] Michael, M. M. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures* (2002), ACM, pp. 73–82.

[30] Michael, M. M. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems 15*, 6 (2004), 491–504.

[31] Moore, G. E., et al. Cramming more components onto integrated circuits, 1965.

[32] Morrison, A., and Afek, Y. Fast concurrent queues for x86 processors. In *ACM SIGPLAN Notices* (2013), vol. 48, ACM, pp. 103–112.

[33] Nikolaev, R., and Ravindran, B. Hyaline: Fast and transparent lock-free memory reclamation. *arXiv preprint arXiv:1905.07903* (2019).

[34] Rönngren, R., and Ayani, R. A comparative study of parallel and sequential priority queue algorithms. *ACM Transactions on Modeling and Computer Simulation (TOMACS) 7*, 2 (1997), 157–209.

[35] Strati, F., Giannoula, C., Siakavaras, D., Goumas, G., and Koziris, N. An adaptive concurrent priority queue for numa architectures. In *Pro-*

ceedings of the 16th ACM International Conference on Computing Frontiers (2019), ACM, pp. 135–144.

[36] Sundell, H., and Tsigas, P. Fast and lock-free concurrent priority queues for multi-thread systems. *Journal of Parallel and Distributed Computing 65*, 5 (2005), 609–627.

[37] Sutter, H. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's journal 30*, 3 (2005), 202–210.

[38] Valois, J. D. Lock-free linked lists using compare-and-swap. In *PODC* (1995), vol. 95, Citeseer, pp. 214–222.

[39] van Riel, R., and Chegu, V. Automatic numa balancing. *Red Hat Summit* (2014).