Why Non-Blocking Operations Should Be Selfish

Joel Gibson[1] and Vincent Gramoli[1,2]

[1] University of Sydney, Sydney, Australia
[2] NICTA, Sydney, Australia
jgib4447@uni.sydney.edu.au
vincent.gramoli@sydney.edu.au

Abstract. Non-blocking data structures are often analysed by giving an upper *amortised* running time bound in terms of the size of the data structure and a measure of contention. The two most commonly used measures are the point contention c_P, the maximum number of processes active at any one time during an operation, and the interval contention c_I, the number of operations overlapping with a given operation. In this paper, we show that when summed across every operation in an execution, the interval contention c_I is within a factor of 2 of the point contention c_P. Our proof relies on properties of interval graphs where at least one simplicial vertex exists, and uses it to construct a lower bound on the overall point contention. We show that this bound is tight.
This result contradicts the folklore belief that point contention leads to a tighter bound on complexity in an amortised context, and provides some theoretical grounds for recent observations that using less helping in non-blocking data structures can lead to better performance. We also propose a linked list algorithm based on Fomitchev and Ruppert's algorithm but with *selfish* operations: read-only operations that do not help others but rather execute wait-free. The higher performance of our approach compared to the original list confirms that reducing helping can increase performance, with the same asymptotic amortised complexity.

Keywords: Helping, lock-freedom, wait-freedom, point contention, process contention, interval contention, overlapping-interval contention

1 Introduction

Non-blocking algorithms guarantee that some process completes an operation in every sufficiently long execution[3]. They are appealing because they (i) do not suffer from the preemption imposed by the scheduler, and (ii) do not require the high cost of helping needed by common *wait-free* algorithms to make all operations complete in a finite number of steps [2]. As an example, existing non-blocking data structures that make use exclusively of compare-and-swaps (CASes) for synchronisation have recently been shown to outperform other state-of-the-art data structures [3]. A non-blocking update typically reads a memory

[3] Note that non-blocking is sometimes used to refer to a larger class of progress conditions, instead we use the less general definition from [1].

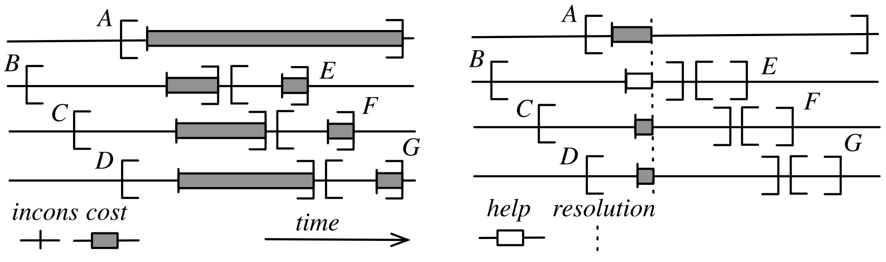
location, then takes steps before executing a CAS that compares the previously seen value to the current value. A difference in these values indicates an *inconsistency* due to a concurrent modification: the CAS fails and the steps are typically re-executed. In a lock-free algorithm the *step complexity* (i.e., the worst-case number of steps taken in an execution) of a single operation cannot be bounded because it can fail and retry arbitrarily many times. Instead we consider the step complexity across every operation in the execution, usually stated as the *amortised* cost of an operation in terms of the size of the data structure and some contention parameter c.

Two notions of contention have attracted lots of interest in the recent years [4–8]. First, the *interval contention* c_I is the number of operations overlapping with a given operation. An amortised bound on the step complexity in terms of the interval contention is relatively easy to obtain by making operations charge each other for steps that would usually be unnecessary in a sequential execution. Provided an operation charges any other overlapping operation a constant amount and at most a constant number of times, this leads naturally to an amortised additive $O(c_I)$ term. Unfortunately, the interval contention of an operation can be arbitrarily large in a system involving as few as two processes. Second, the *point contention* c_P is the maximum number of processes active at any time during the operation [4]. An amortised bound on the step complexity in terms of point contention is usually harder to achieve, for example the proofs in [5, 8] rely on reasoning about the relative ordering of individual CAS steps inside operations. Some authors [6, 7] have provided modifications of their algorithms which perform extra helping so that they can tighten what would otherwise be a bound in terms of the interval contention to a bound in terms of the point contention. Since the point contention is bounded above by the maximum number of concurrent processes, it appears to be a tighter and more realistic bound than the interval contention.

In this paper, we show that when summed across every operation in an execution, c_I is within a factor of 2 of c_P. Our proof relies on properties of interval graphs where the point (resp. interval) contention of an operation *op* is the size of one of the largest cliques containing *op* (resp. *op*'s degree + 1). The presence of at least one special vertex called simplicial, in any interval graph, allows us then to construct a lower bound on the overall point contention. In other words, we show that point contention and interval contention are interchangeable additive factors of the amortised complexity of concurrent algorithms. We show that this bound is tight, in the sense that given any $0 < \epsilon < 1/2$, there are executions in which the ratio of the overall interval contention to the overall point contention is arbitrarily close to $2 - \epsilon$. Finally, we also compare these contention definitions to two other definitions, the original interval contention definition [9] that we call *process contention*, and the *overlapping-interval contention* [6]. We believe our result to be important as it shows that point contention does not give a tighter amortised complexity bound than interval contention.

Our result challenges a popular belief that increasing the amount of helping done by operations can reduce the asymptotic step complexity of data structures.

For example, in [6], the amortised complexity of a skip list operation is given as $O(c_I)$, as an inconsistency caused by an update may be encountered by every traversal concurrent with that update, and so the update may be charged $O(c_I)$ times. The authors argue that by making the traversals perform extra helping so as to resolve this inconsistency, the update will be charged at most $O(c_P)$ times. So it seems that by introducing more helping into the algorithm, an additive term of $O(c_I)$ can be reduced to $O(c_P)$. The authors of [7] argue similarly for their lock-free binary search tree.



(a) Interval contention without extra help (b) Point contention with extra help

Fig. 1. It was recently noted [6,7] that if an update A creates a transient inconsistent state, *incons*, that is encountered by concurrent traversals B, C, D, E, F, G then making the other operations *help* another when encountering an inconsistency can reduce the interval contention $O(c_I)$ to the point contention $O(c_P)$ by simply resolving the inconsistency as early as the first operation, say B, encounters the inconsistency

To illustrate how helping seemingly reduces the cost due to contention, consider the execution of an algorithm depicted in Figure 1 representing the intervals of seven operations, A, B, C, D, E, F, G, where time increases from left to right. Operation A updates shared data, as indicated by the vertical bar. Before operation A completes, other operations observe a transient inconsistent state produced by A that incurs a cost. We say that operations, whose interval overlaps A's interval, charge A for the cost of observing this inconsistency. Since the inconsistency was introduced by A, and must be resolved for A to finish, at most c_I operations can charge A for this. If, however, the first operation that observes this inconsistency, namely B, *helps* the update by resolving it, then subsequent operations E, F, G will no longer observe this inconsistency. As long as every operation eagerly attempts to remove this inconsistency, at most c_P operations can charge A for observing the inconsistency. Since $c_P \leq c_I$ for each operation, up until now it was believed to lead to a tighter bound on complexity, even in an amortised context.

By contrast, extra helping could intuitively lower the performance of a data structure, especially by forcing an operation, which would otherwise be read-only, to write. Such a behaviour was recently observed on the Harris' list-based set [10] and led researchers to prevent read-only operations from writing: they implemented what they called an "optimised" variant of Harris' with a contains operation that traverses logically deleted nodes without physically unlinking them. The authors compared these two implementations on up to 40 hardware threads and observed that their optimised version increases performance. Former implementation variants of Harris' algorithm also suggested the appeal of a wait-free contains [11,12].

In the light of our result, we designed a new non-blocking list-based set algorithm based on Fomitchev and Ruppert's [5], but with a wait-free contains operation which performs no writes. Their original list has amortised complexity $O(n + c_P)$ per operation, whereas the modification is easily seen as $O(n + c_I)$, which our result tells us is equivalent. We implemented Fomitchev and Ruppert's linked list in C. We compared the performance of our linked list against the performance of Fomitchev and Ruppert's linked list on a 64-core machine with various sizes and update ratios. The results show that our linked list is more efficient in all tested settings than Fomitchev and Ruppert's. While we do not claim that helping is always detrimental to the performance, these results confirm empirically that limiting helping *can* lead to better performance, at no increase of the amortised cost of an operation.

To conclude, we believe our result is not only insightful for theoreticians but also for practitioners. First, an amoritised complexity with an additive $O(c_I)$ term is equivalent to having an $O(c_P)$ term instead, without the burden of modifying the algorithm or complicating the analysis required to measure the point contention. Second, as modern chip multiprocessors offer more and more cores, it is likely that enough processes (or threads) accessing a concurrent data structures can proceed without being preempted by the operating system. In this case, it seems that an implementation with *selfish* operations, which are wait-free read-only operations that do not help other operations, could lead to higher performance, simply because (i) read-only operations would avoid writing, hence limiting cache invalidations and (ii) update operations would fix any inconsistency they introduce without being arbitrarily delayed via preemption. Our result gives theoretical grounds to support favouring selfish operations over helpful operations when all that distinguishes them is an additive $O(c_P)$ or $O(c_I)$ term.

In Section 2 we define existing contention measures and explain how to reason about them in terms of interval graphs. In Section 3, we show that in any finite execution the overall point contention cannot be twice as large as the point contention and that this bound is tight, and briefly show related results for the process contention and the overlapping-interval contention. In Section 4 we show empirically that replacing a helpful contains operation by a selfish one increases the performance of Fomitchev and Ruppert's linked list. We list directions to explore new contention metrics in Section 5. We present the related work in Section 6 and we conclude in Section 7.

2 Preliminaries

Let a finite execution α involving P processes be a finite set $\mathcal{O}$ of *operations* with two mappings I and π. $I : \mathcal{O} \to \mathbb{R} \times \mathbb{R}$ maps operations to compact real intervals, and $\pi : \mathcal{O} \to \{1, \ldots, P\}$ maps operations to the processes that executed them. If for two operations $op, op' \in \mathcal{O}$ we have $I(op) \cap I(op') \neq \emptyset$, we say that op and op' *overlap*. Furthermore, I should be injective[4], and the execution should be *well-formed*: any two operations mapping to the same process should not overlap. Figure 2 shows an example of a finite execution.

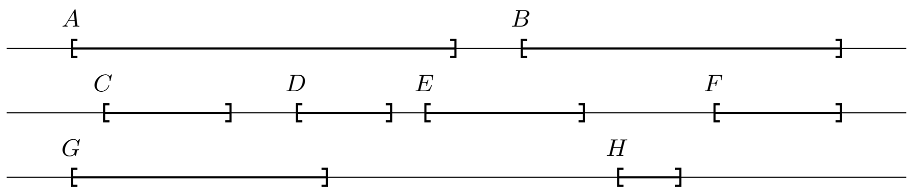


Fig. 2. An example execution involving 3 processes and 8 operations. The point contentions of A, C, D, and G are 3, while E, B, H, and F are 2. The process contention of E is 2, and of B is 3. The interval contentions of A, B, and C are 5, 4, and 3 respectively. The overlapping-interval contention of E is 5.

Definition 1. *In a finite execution $\alpha = (\mathcal{O}, P, I, \pi)$, the point contention c_P, process contention c_K, interval contention c_I, and overlapping-interval contention c_{OI} are functions $\mathcal{O} \to \mathbb{Z}_+$ defined by:*

$$
\begin{aligned}
c_P(op) &= \max_{x \in I(op)} |\{op' \in \mathcal{O} \colon x \in I(op')\}| \\
c_K(op) &= |\{\pi(op') \colon op' \in \mathcal{O} \wedge op' \text{ overlaps } op\}| \\
c_I(op) &= |\{op' \in \mathcal{O} \colon op' \text{ overlaps } op\}| \\
c_{OI}(op) &= \max_{\substack{op' \in \mathcal{O} \\ op' \text{ overlaps } op}} c_I(op')
\end{aligned}
$$

Proposition 1. *For any operation op, $1 \le c_P(op) \le c_K(op) \le c_I(op) \le c_{OI}(op)$, and $c_K(op) \le P$.*

Proof. Let $S = \{op' \in \mathcal{O} \colon op' \text{ overlaps } op\}$, and for any $x \in I(op)$, let $S_x = \{op' \in \mathcal{O} \colon x \in I(op')\}$. The definitions of contention for the operation op now become:

$$
c_P(op) = \max_{x \in I(op)} |S_x| \qquad c_I(op) = |S|
$$

$$
c_K(op) = |\pi(S)| \qquad c_{OI}(op) = \max_{op' \in S} c_I(op')
$$

[4] This is not restrictive: any finite execution in which two intervals are identical may be perturbed slightly such that they are not, without affecting contention.

By these characterisations we find $c_{OI}(op) \geq c_I(op)$ because $op \in S$, and $c_I(op) \geq c_K(op)$ because a set is at least as large as its image under a map. Note that since the execution is well-formed, $|S_x| = |\pi(S_x)|$, and since we have $S_x \subseteq S$ for all $x \in I$, it follows that $c_P(op) \leq c_K(op)$. Finally, all of these bounds are tight by considering an execution containing one operation and one process.

All that is needed to calculate c_I and c_{OI} is information about which pairs of operations overlap. In fact, this is the case for c_P as well. Hence a natural setting to analyse these measures of contention is an *interval graph*, which retains the information of which operations overlap, while hiding the complications of processes and exact points in time. First we introduce some terminology.

Any graphs $G = (V, E)$ considered here are finite, undirected, and without multiple edges or loops. V denotes the vertex set and E denotes the edge set. For any vertex subset $U \subseteq V$, $G[U] = (U, E \cap (U \times U))$ is called the *subgraph induced by* U. A vertex subset $U \subseteq V$ forms a *clique* if the subgraph $G[U]$ is complete. For any vertex v, its *neighbourhood* $N(v)$ consists of all vertices incident to v. A vertex v is called *simplicial* if the subgraph induced by its neighbours and itself $G[\{v\} \cup N(v)]$ is complete.

Definition 2. *The interval graph of a finite set of real intervals S is the graph with vertex set S, with an edge between two intervals $I, J \in S$ if $I \neq J$ and $I \cap J \neq \emptyset$.*

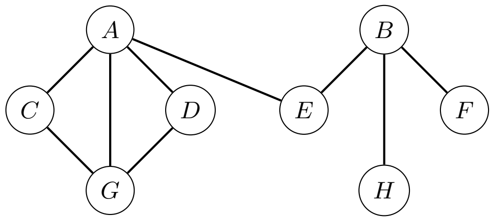


Fig. 3. The interval graph corresponding to the execution in Figure 2. The vertices C, D, H and F are simplicial in this graph.

Definition 3. *A perfect elimination order is an ordering $\{v_i\}_{i=1}^{n}$ of vertices in a graph such that for all $1 \leq i \leq n$, v_i is simplicial in $G[v_1, \ldots, v_i]$.*

Interval graphs belong to a larger class of graphs called *chordal* graphs, which are graphs in which any cycle of length 4 or more always contains a *chord*, an edge connecting two non-adjacent vertices of the cycle. It is a well-known fact that chordal graphs are characterised completely by the existence of a perfect elimination order. In the case of interval graphs the existence of such an order is easy to see, and so a short proof is given below.

Lemma 1. *Every interval graph on n vertices admits a perfect elimination order.*

Proof. The case for $n = 1$ is clear. We proceed by induction: assume that the claim holds for interval graphs with $n-1$ vertices. Take the vertex v corresponding to the interval with earliest finishing time: this vertex is simplicial since the finishing time intersects every interval which overlaps v. We already have a perfect elimination order $\{v_i\}_{i=1}^{n-1}$ of $G - v$ by assumption, and so setting $v_n = v$ gives a perfect elimination order $\{v_i\}_{i=1}^{n}$ of G.

Finally, in the interval graph of an execution, the point contention of an interval is equal to the size of the largest clique containing the vertex corresponding to that interval. This can be seen by considering the mapping $\mathbb{R} \to 2^{\mathcal{O}}$, $x \mapsto \{op \in \mathcal{O} \mid x \in I(op)\}$ which maps points to the operations active at that point in time: it will always map a point to an empty set, or a maximal clique. For this reason we consider the following lemma, which will allow us to put a lower bound on the overall point contention in terms of the number of vertices n and the number of edges m.

Lemma 2. *In an interval graph G, let $M(v)$ be the size of a maximum clique containing the vertex v. Then $\sum_{v \in V} M(v) \geq n + m$.*

Proof. Take a perfect elimination order $\{v_i\}_{i=1}^{n}$ of the vertices of G, and define $G_i = G[v_1, \ldots, v_i]$. This gives a family of graphs $G_n, \ldots, G_1$, such that $G_n = G$, G_1 is a single vertex, and $G_j = G_{j+1} - v_{j+1}$ for all $1 \leq j < n$. Let d_i be the degree of v_i in G_i. Since v_i is simplicial in G_i, $\{v_i\} \cup N(v_i)$ forms a clique in G_i and hence also in G, so $1 + d_i \leq M(v_i)$. Finally, note that d_i is the number of edges removed when removing v_i from G_i, so $\sum_{i=1}^{n} d_i = m$. So $\sum_{v \in V} M(v) \geq \sum_{i=1}^{n}(1 + d_i) = n + m$.

3 Equivalence of amortised measures of contention

For any finite execution α, let $c_P(\alpha) = \sum_{op \in \mathcal{O}} c_P(op)$, and likewise for the other measures of contention.

Theorem 1. *In any finite execution α, $c_P(\alpha) \leq c_I(\alpha) < 2c_P(\alpha)$.*

Proof. Form the interval graph using the intervals $I(op)$ for each $op \in \alpha$. By the definitions of contention given in Section 2, we can see that the interval contention of a single operation op is $c_I(op) = 1 + \deg op$, where deg denotes the degree of the operation's interval in the interval graph. Summing across all operations, $c_I(\alpha) = n + 2m$, where n is the number of vertices and m the number of edges in the interval graph. As discussed previously, the point contention $c_P(op)$ is the size of the largest clique containing op in the interval graph. So by Lemma 2 we have $n + m \leq c_P(\alpha)$.

Putting these together with the inequality in Proposition 1, we find that

$$n + m \leq c_P(\alpha) \leq c_I(\alpha) \leq n + 2m$$

and so by taking the ratio of $c_I(\alpha)$ to $c_P(\alpha)$,

$$1 \leq \frac{c_I(\alpha)}{c_P(\alpha)} \leq \frac{n+2m}{n+m} = 1 + \frac{m}{n+m} < 2$$

Although this fact alone tells us that in amortised terms, $c_P = \Theta(c_I)$ and so the point contention and interval contention are equivalent, it is interesting to examine what a "worst-case" execution is. Intuitively, we want to keep the point contention small, while making intervals overlap as many times as possible. Such a construction is given in the proof of Theorem 2 and illustrated in Figure 4, and shows that the bound given above is tight.

Theorem 2. *For any $0 < \epsilon < 1/2$, there exists a family of executions $\{\alpha_n\}_{n\geq 1}$ where each α_n has n operations and ϵn processes, such that*

$$\lim_{n\to\infty} \frac{c_I(\alpha_n)}{c_P(\alpha_n)} = 2 - \epsilon.$$

Proof. Let α_n be an execution containing n operations labelled op_i for $0 \leq i < n$, and let $1 \leq k \leq n/2$. We define the mappings $\pi(op_i) = i \pmod{k}$ and $I(op_i) = [i, i + k - \frac{1}{2}]$ for all $0 \leq i < n$. It is easy to check that at the start or end point of each operation there are k operations active at that point in time and so $c_P(op) \geq k$ for all operations. Since there are only k processes, $c_P(op) = k$ for all operations, so $c_P(\alpha_n) = nk$.

By the length and placement of operations, for every operation op the set of operations intersecting its left endpoint is disjoint to the set of operations intersecting its right endpoint, and the union of these is every operation concurrent with op. Hence every operation but the first $k-1$ and the last $k-1$ operations have interval contention $2k-1$. The first operation has interval contention k, the next $k+1$, and so on until the kth operation has interval contention $2k-1$, and similarly for the last k operations. By overcounting the interval contention overall and subtracting off the start and end deficits, we find that $c_I(\alpha_n) = n(2k-1) - 2(0 + 1 + \ldots + (k-1)) = 2nk - n - k(k-1)$. Letting $k = \epsilon n$ and taking the ratio of interval to point contention, we get $c_I(\alpha_n)/c_P(\alpha_n) = 2 - \epsilon - \frac{1-\epsilon}{\epsilon n}$.

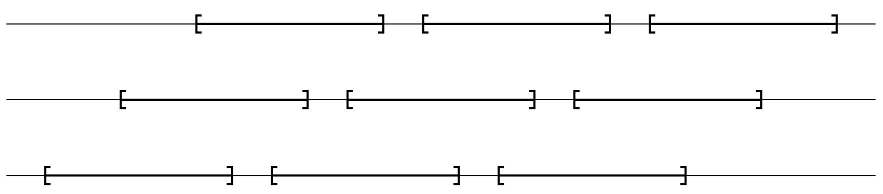


Fig. 4. An example worst-case construction with $n = 9$ and $k = 3$.

Finally, Theorem 1 and Proposition 1 give the chain of inequalities $c_P(\alpha) \leq c_K(\alpha) \leq c_I(\alpha) < 2c_P(\alpha)$ and so the process contention is also amortised equivalent to the point contention. The overlapping-interval contention, on the other hand, cannot be bounded within a constant factor of the point contention. Consider an execution of two processes, where the first process has one long-running operation, and the second process runs $n-1$ short operations, all of which execute inside the interval of the long-running operation. In this execution α, we have $c_P(op) = 2$ and $c_{OI}(op) = n$ for every operation, so $c_P(\alpha) = 2n$ and $c_{OI}(\alpha) = n^2$.

4 Evaluation of the Selfish linked list

A key application of Theorem 1 is that algorithms which had very strict helping requirements in order to attain a $O(c_P)$ amortised additive complexity term may be able to be modified to have weaker helping requirements, without any change in asymptotic complexity. It has been observed in practice [10] that having wait-free read-only operations on concurrent data structures often gives an increase in performance, regardless of asymptotic complexity. Here we modify an existing non-blocking linked-list algorithm by Fomitchev and Ruppert [5] to have a wait-free contains operation and show that the resulting algorithm, namely the *Selfish linked list*, gives better performance.

4.1 The Selfish linked list algorithm

First, we recall Fomitchev and Ruppert's construction of the list. Each node stores three fields: a *key* field, indicating the value represented by that node in the set, a *backlink* field, used when traversal is interrupted by a concurrent modification, and a *successor* field. The *successor* field stores a *right* pointer to the next node in the list and two booleans *flag* and *mark*. The list contains two dummy *head* and *tail* nodes, with keys $-\infty$ and ∞ respectively.

When a node is to be deleted, its predecessor's *flag* bit is set, indicating that the predecessor's successor field may not be modified until the node has been removed. Following this, the node's *mark* bit is set, indicating its successor field may not be modified from now on, and the node's *backlink* field is set to point to the predecessor. Finally, the predecessor's *successor* field is modified to remove the *flag* bit and swing the *right* pointer over the node being deleted.

Every operation in the algorithm performs eager helping: as soon as a traversal encounters a node with a *mark* set, it attempts to help remove that node from the list. Removes and inserts which encounter nodes with *flag* bits set must help the concurrent remove operation to physically remove those nodes. In addition, any attempts to flag nodes may have to backtrack through chains of backlinks in order to reach nodes which are not logically deleted. The existence of the backlinks means that nodes do not ever have to restart from the beginning of the list and is key to an amortised $O(n + c_P)$ time per operation.

Our modification is very similar to the modification of the Harris list presented in [11,12] to replace the contains operation, which would usually attempt to help remove marked nodes from the list, with a read-only operation which makes a single pass through the list. The pseudocode of the contains operation is

Algorithm 1 The wait-free Contains operation

```
procedure CONTAINS($k$)
    $current \leftarrow head$
    $marked \leftarrow false$
    while $current.key < k$ do
        $succ \leftarrow current.succ$
        $marked \leftarrow succ.mark$
        $current \leftarrow succ.right$
    end while
    return $(current.key = k) \wedge (marked = false)$
end procedure
```

depicted in Algorithm 1. The operation is linearisable and wait-free, and replaces the original lock-free contains operation: all other operations of the list remain as in the original. Since the operations in the original list had an amortised complexity of $O(n + c_P)$ and we change only the read-only operation, we conclude our new list has an amortised complexity of $O(n + c_I)$ for each operation: in the presence of only updates, the complexity is $O(n + c_P)$ as originally shown by Fomitchev and Ruppert, and introducing contains operations means that at most c_I more is billed to each concurrent update by a contains operation that traverses a logically deleted node. We implemented both the original algorithm and our modification in C and we did not include any memory reclamation technique. Implementing a memory reclamation technique is not straightforward and can substantially impact performance [13].

4.2 Experimental evaluation

We performed the experiment with Synchrobench [3] on a 4 socket AMD Opteron 6378 2.4 GHz 16 cores (64 cores in total) machine running Fedora Linux 18. GCC 4.9.2 was used to compile the C code. The benchmarking program initialises the data structures with N elements randomly selected from $\{1, \ldots, 2N\}$, and spawns from 1 to 78 threads. Each test runs for 10 seconds. Each thread chooses of the three operations contains, insert, or remove based on the *update ratio*. In the data shown here, the update ratio is always 10%, meaning that 10% of operations are contains operations, 45% are insert, and 45% are remove. Each datapoint shown is an average of 20 runs, and the error bars are the sample standard deviation of those runs.

As shown in Figure 5, the list with our modification has much higher throughput than the original algorithm, especially in the small case of a 128 element list, where there is a 20% throughput improvement. We conclude that limiting helping can increase performance in concurrent data structures, and our result in Theorem 1 gives a guarantee that our new $O(n + c_I)$ algorithm is asympotically equivalent to the old $O(n + c_P)$ algorithm.

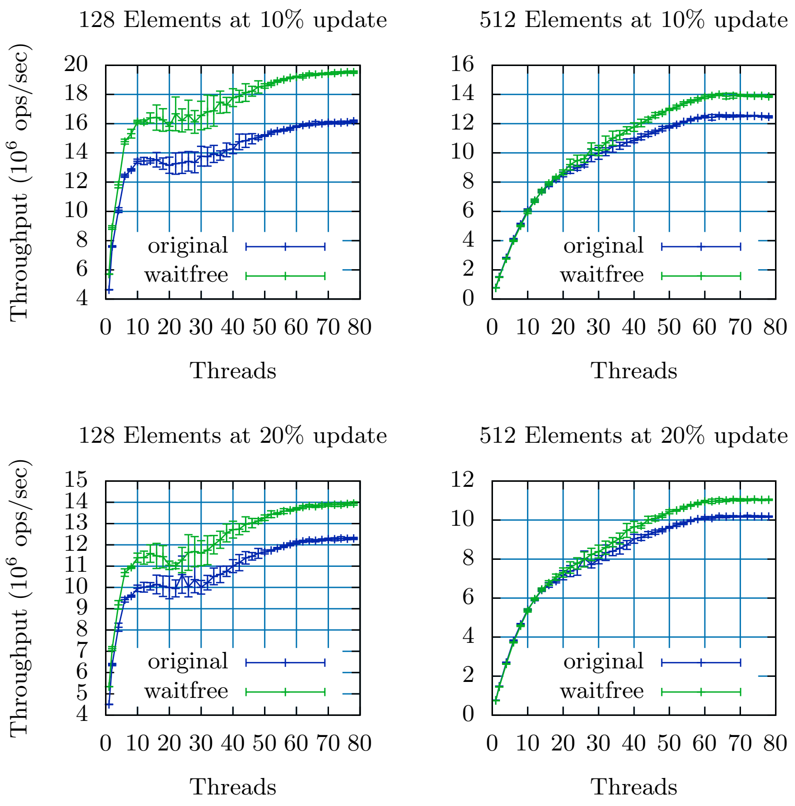


Fig. 5. A comparison of Fomitchev and Ruppert's original algorithm to our modified version with a wait-free contains implementation. Both algorithms have the same asymptotic worst-case complexity. Each datapoint shown is an average of 20 runs of 10 seconds each. The vertical bars represent the sample standard deviation.

5 Towards a more refined notion of contention

As discussed previously, there has been a view that by introducing more helping into an algorithm, the amortised step complexity can be "tightened" from an additive term of $O(c_I)$ to $O(c_P)$. By Theorem 1, we know these two quantities to be amortised equivalent, so clearly this cannot be distinguished by the point or interval contention. However, this does not rule out the existence of a more refined measure of contention that does separate these cases.

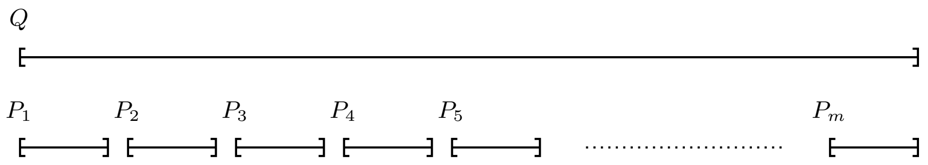


Fig. 6. Suppose Q causes an inconsistency in the data structure and then gets suspended for a long time. If every operation performs eager helping, an inconsistency caused by Q will only be observed once. If no operations perform helping, it will be observed m times.

Consider an execution of one long-running process Q and m short-running processes $P_1, \ldots, P_m$. Suppose we have a structure featuring logical deletions, and the first step in a removal is to mark a node logically deleted. The long-running process Q marks an element as logically deleted before being suspended for a long time, while the short-running processes repeatedly access the data structure. This is illustrated in Figure 6. If the short running processes P_i perform no helping to try to physically remove the node, each will spend extra time traversing a node not present in the set, and incur a constant cost. The total cost of these extra steps is $\Theta(m)$. On the other hand, if every operation eagerly tries to help finish any partially completed delete operation it comes across, the first short operation P_1 will suffer this (constant) cost, and the rest will traverse with no extra cost. So without helping, there are $\Theta(m)$ extra steps that need to be carried out, and with helping there is only $O(1)$.

The current measures we have of contention, the point and interval contention, will not separate these cases. Some finer measure of contention is needed to capture this case and show when helping can really benefit an algorithm in an asymptotic sense. We believe that some of the theory developed here could be useful in determining and analysing a new measure of contention that separates these two cases.

6 Related work

For more than two decades, contention has been known to be an important complexity factor of concurrent algorithms [14]. This observation motivated the definitions of various contention measures: hot-spot contention [14], process contention (originally called interval contention) [9], interval and point contentions [4], step contention [15] and overlapping-interval contention [6]. An overview of some of these properties was given in [16].

To diminish contention several techniques were adopted. Adaptive algorithms [4] were designed to implement applications that could adapt to the point contention during the execution of an operation. Contention-sensitive data structures [17] propose to reduce the cost of lock-based data structures in the absence of contention.

A pragmatic way of reducing contention in data structures is to split operations into abstract updates and structural updates. The speculation-friendly binary search tree was the first algorithm to generalize this decoupling by both keeping logically deleted nodes and relaxing the balance constraints [18]. The contention-friendly binary search tree adopts the same decoupling but synchronises with locks rather than transactional memory [19]. A non-blocking chromatic tree exploits this decoupling up to a constant number k of violations hence upper-bounding the imbalance at time t by $k + c$ where the contention c represents the number of updates in progress at time t [20]. Finally, this decoupling was used to implement efficient non-blocking skip lists that do not suffer traditional contention hotspots [21,22].

Many linked list algorithms have been proposed over the last two decades [5,11,23–25]. Although potentially very efficient, lock-based linked lists [11,25] generally do not perform as well as non-blocking ones [5,24] when the number of processes exceeds the number of available computing resources. This is typically due to the contention induced on locks. Non-blocking linked lists are thus particularly interesting. Harris' linked list [24] is still one of the fastest [3] but its cost per operation can be $\Omega(nc_P)$ in some execution. Fomitchev and Ruppert provide a linked list with an amortized complexity of $O(n+c_P)$ per operation [5]. Our Selfish list, of asymptotically equivalent complexity $O(n+c_I)$, shows better performance.

Both wait-free and non-blocking properties guarantee progress regardless of the way the operating system schedules threads [26]. Our algorithm is non-blocking but not wait-free as it guarantees wait-freedom only of the contains operation. Recent results showed that under a stochastic scheduler, some non-blocking algorithms, also called single CAS universal, are wait-free with probability 1 [27], however, our algorithm does not fall in this category. There exist both methodologies [28] and simulation techniques [29] to obtain wait-freedom with a slight performance loss: one can run a lock-free fast path and start a wait-free slow path if the fast path was unsuccessful.

7 Conclusion

When summed across every operation in an execution, the point contention cannot be twice larger than interval contention. Our proof is interesting in its own right as it draws a natural relation between the theory of contention and the theory of interval graphs, where the point contention of an operation is its degree plus one in the graph and the interval contention is the size of one of the largest cliques the operation belongs to in the graph.

The execution α of several non-blocking data structure algorithms [5–8] is known to have an asymptotic amortised complexity with an additive contention term of either $c_I(\alpha)$ or $c_P(\alpha)$. Our result shows that these terms are equivalent, hence contradicting the folklore knowledge that operations should necessarily help each other to achieve a tighter complexity bound even in an amortised context.

We evaluated the performance of a non-blocking list and a new variant of it that consists of the same algorithm but with a selfish contains operation. Their complexities $O(n + c_P)$ and $O(n + c_I)$, respectively, are known now to be equivalent. Our results on a 64-core machine show that selfishness increases performance in all settings we tested, confirming the practical relevance of our bound.

We believe that our result will be useful to simplify the analysis of non-blocking data structures in terms of amortised complexity as deriving the complexity based on interval contention seems easier than point contention. As part of future work, we would like to analyse existing algorithms in the light of our new result.

Availability. The source code used in this paper is available in Synchrobench: `https://sites.google.com/site/synchrobench`.

Acknowledgements. NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

References

1. Herlihy, M.: Wait-free synchronization. ACM Trans. Program. Lang. Syst. **13**(1) (January 1991) 124–149
2. Censor-Hillel, K., Petrank, E., Timnat, S.: Help! In: PODC. (2015) 241–250
3. Gramoli, V.: More than you ever wanted to know about synchronization: Synchrobench, measuring the impact of the synchronization on concurrent algorithms. In: PPoPP. (2015) 1–10
4. Attiya, H., Fouren, A.: Algorithms adapting to point contention. J. ACM **50**(4) (2003) 444–468
5. Fomitchev, M., Ruppert, E.: Lock-free linked lists and skip lists. In: PODC. (2004) 50–59
6. Oshman, R., Shavit, N.: The SkipTrie: low-depth concurrent search without rebalancing. In: PODC. (2013) 23–32
7. Chatterjee, B., Nguyen, N., Tsigas, P.: Efficient lock-free binary search trees. In: PODC. (2014) 322–331
8. Ellen, F., Fatourou, P., Helga, J., Ruppert, E.: The amortized complexity of non-blocking binary search trees. In: PODC. (2014) 332–340

9. Afek, Y., Stupp, G., Touitou, D.: Long lived adaptive splitter and applications. Distributed Computing **15**(2) (2002) 67–86
10. David, T., Guerraoui, R., Trigonakis, V.: Asynchronized concurrency: The secret to scaling concurrent search data structures. In: ASPLOS. (2015) 631–644
11. Heller, S., Herlihy, M., Luchangco, V., Moir, M., Scherer III, William N.and Shavit, N.: A lazy concurrent list-based set algorithm. In: OPODIS. Volume 3974 of LNCS. (2005) 3–16
12. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2008)
13. Michael, M.M.: High performance dynamic lock-free hash tables and list-based sets. In: SPAA. (2002) 73–82
14. Dwork, C., Herlihy, M., Waarts, O.: Contention in shared memory algorithms. J. ACM **44**(6) (November 1997) 779–805
15. Attiya, H., Guerraoui, R., Kuznetsov, P.: Computing with reads and writes in the absence of step contention. In: DISC. Volume 3724 of LNCS. (2005) 122–136
16. Hendler, D.: Non-blocking algorithms. In: Encyclopedia of Parallel Computing. Springer (2011) 1321–1329
17. Taubenfeld, G.: Contention-sensitive data structures and algorithms. In: DISC. Volume 5805 of LNCS. (2009) 157–171
18. Crain, T., Gramoli, V., Raynal, M.: A speculation-friendly binary search tree. In: PPoPP. (2012) 161–170
19. Crain, T., Gramoli, V., Raynal, M.: A contention-friendly binary search tree. In: Euro-Par. Volume 8097 of LNCS. (2013) 229–240
20. Brown, T., Ellen, F., Ruppert, E.: A general technique for non-blocking trees. In: PPoPP. (2014) 329–342
21. Crain, T., Gramoli, V., Raynal, M.: No hot spot non-blocking skip list. In: ICDCS. (2013) 196–205
22. Dick, I., Fekete, A., Gramoli, V.: Logarithmic data structures for multicores. Technical Report 697, University of Sydney (2014)
23. Valois, J.D.: Lock-free linked lists using compare-and-swap. In: PODC. (1995) 214–222
24. Harris, T.: A pragmatic implementation of non-blocking linked-lists. In: DISC. Volume 2180 of LNCS. (2001) 300–314
25. Gramoli, V., Kuznetsov, P., Ravi, S., Shang, D.: Brief announcement: A concurrency-optimal list-based set. In: DISC. LNCS (2015)
26. Herlihy, M., Shavit, N.: On the nature of progress. In: OPODIS. Volume 7109 of LNCS. (2011) 313–328
27. Alistarh, D., Censor-Hillel, K., Shavit, N.: Are lock-free concurrent algorithms practically wait-free? In: STOC. (2014) 714–723
28. Kogan, A., Petrank, E.: A methodology for creating fast wait-free data structures. In: PPoPP. (2012) 141–150
29. Timnat, S., Petrank, E.: A practical wait-free simulation for lock-free data structures. In: PPoPP. (2014) 357–368