On the Computational Power of Shared Objects

Gadi Taubenfeld

The Interdisciplinary Center, P.O.Box 167, Herzliya 46150, Israel,
tgadi@idc.ac.il,
http://www.faculty.idc.ac.il/gadi/

Abstract. We propose a new classification for evaluating the strength of shared objects. The classification is based on finding, for each object of type o, the strongest progress condition for which it is possible to solve consensus for *any* number of processes, using any number of objects of type o and atomic registers. We use the strongest progress condition to associate with each object a number call the *power number* of that object. Objects with higher power numbers are considered stronger. Then, we define the *power hierarchy* which is an infinite hierarchy of objects such that the objects at level i of the hierarchy are exactly those objects with power number i. Comparing our classification with the traditional one which is based on fixing the progress condition (namely, wait-freedom) and finding the largest number of processes for which consensus is solvable, reveals interesting results. Our equivalence and extended universality results, provide a deeper understanding of the nature of the relative computational power of shared objects.

Keywords: Shared objects, consensus numbers, power numbers, wait-freedom, k-obstruction-freedom, wait-free hierarchy, power hierarchy, universality.

1 Introduction

Motivation

Alice, the CTO of MultiBrain Inc., was excited when she told her spouse Bob about the new multi-core computer that her company has just bought. “It has 128 cores and it is extremely powerful” she said, “we are already writing concurrent applications that will take advantage of its high level of parallelism”.

Bob, a theoretician, has asked her if they intend to use locks in their concurrent applications. “No” said Alice, “we have decided to avoid using locks”. “And what are the type of atomic operations the machine supports?” asked Bob. “Swap, and reads and writes” answered Alice.[1] “What about compare-and-swap, and load-link/store-conditional?” asked Bob. “No” answered Alice, “only atomic swap, and reads and writes are supported”. “Well” said Bob, “for many interesting applications you will not be able to use more than two of these 128 cores simultaneously, so do not expect to get too much speedup”. “Why?” asked Alice, “we have paid so much for that machine, it must be able to do better”.

[1] A swap operation takes a shared register and a local register, and atomically exchange their values.

"There is a fundamental result" explained Bob, "that using the type of atomic operations that your machine supports, it is possible to solve consensus and many other important problems for two processes only, assuming that the required progress condition is wait-freedom". "You got me worried for a minute" said Alice, "for all our practical purposes, we can do with a much weaker progress condition than wait-freedom". "Such as?" asked Bob. "You tell me" answered Alice, "what is the *strongest* progress condition for which it is possible to solve consensus and all the other important problems for *any* number of processes using only atomic swap, and reads and writes?". "Interesting question",said Bob .

We propose a new classification for evaluating the strength of shared objects, which is based on finding the strongest progress condition for which it is possible to solve consensus for *any* number of processes. Such a classification enables to answer Alice's question for any type of object. Comparing our classification with the traditional one reveals interesting results.

Consensus numbers and the wait-free hierarchy

The traditional approach for understanding the relative computational power of shared objects is to classify them according to their consensus numbers. Objects with higher numbers are considered stronger. In order to define the notion of a consensus number, we first define the consensus problem and two known progress conditions.

The (binary) consensus problem is to design an algorithm in which all correct processes reach a common decision based on their initial opinions. We assume that a process may fail only by crashing (i.e., fail-stop). The problem is defined as follows. There are n processes $p_1, p_2, \ldots, p_n$. Each process p_i has an input value $x_i \in \{0, 1\}$. The requirements of the consensus problem are that there exists a *decision value* v such that: (1) each non-faulty process eventually decides on v, and (2) $v \in \{x_1, x_2, \ldots, x_n\}$. In particular, if all input values are the same, then that value must be the decision value.

Two of the most extensively studied progress conditions, in order of decreasing strength, are wait-freedom [5] and obstruction-freedom [6]. *Wait-freedom* guarantees that every process will always be able to complete its pending operations in a finite number of its own steps, regardless of the behavior of the other processes. *Obstruction-freedom* guarantees that a process will be able to complete its pending operations in a finite number of its own steps, if all the other processes "hold still" (i.e., do not take any steps) long enough. That is, obstruction-freedom guarantees progress for any process that eventually executes in isolation long enough, however, it do not guarantee progress under contention.

The *consensus number* of an object of type o, denoted $CN(o)$, is the largest n for which it is possible to solve consensus for n processes, using any number of objects of type o and atomic registers, assuming that the required progress condition is *wait-freedom*. If no largest n exists, the consensus number of o is infinite (denoted ∞). The *wait-free hierarchy* is an infinite hierarchy of objects such that the objects at level i of the hierarchy are exactly those objects with consensus number i.

It is known that, in the wait-free hierarchy, for any positive i, in a system with i processes: (1) no object at level less than i together with atomic registers can implement any object at level i; and (2) each object at level i together with atomic registers can implement any object at level i or at a lower level [5]. If instead of wait-freedom, only obstruction-freedom is required, the hierarchy collapses.

An object o is *universal* for n processes if any object which has sequential specification has a wait-free linearizable implementation using atomic registers and objects of type o in a system with at most n processes.[2] In [5], it is proven that consensus for n processes is universal in a system with n processes, for any positive n. An immediate implication of this result is that, an object o is universal in a system with n processes if and only if the consensus number of o is at least n.

Power numbers and the power hierarchy

Instead of the traditional classification of fixing a single progress condition (namely, wait-freedom) and finding the largest number of processes for which consensus is solvable, we propose a classification which is based on finding the strongest progress condition for which it is possible to solve consensus for *any* number of processes. For a set of processes P, let $|P|$ denotes the size of P. Consider the following generalization of the notion of obstruction-freedom:

> For any $k \geq 1$, the progress condition k*-obstruction-freedom* guarantees that for every set of processes P where $|P| \leq k$, every process in P will be able to complete its pending operations in a finite number of its own steps, if all the processes not in P do not take steps for long enough.

The progress condition k-obstruction-free does not guarantee progress under contention of more than k processes. These infinitely many progress conditions cover the spectrum between obstruction-freedom and wait-freedom. The progress condition 1-obstruction-freedom is the same as obstruction-freedom. When the maximum number of processes is a fixed number, say n, k-obstruction-freedom is the same as wait-freedom, for all $k \geq n$.

> The *power number* of an object of type o, denoted $PN(o)$, is the largest k for which it is possible to solve consensus for any number processes, using any number of objects of type o and atomic registers, assuming that the required progress condition is k*-obstruction-freedom*. If no largest k exists, the power number of o is infinite (denoted ∞). The *power hierarchy* is an infinite hierarchy of objects such that the objects at level i of the hierarchy are exactly those objects with power number i.

In the above definitions, the objects of type o and the atomic registers are all assumed to be wait-free. That is, each operation that is invoked by a process on these basic objects always terminates, regardless of the behavior of the other processes. Next we generalize the notion of universality which was defined earlier in the context of the wait-free hierarchy.

[2] Sequential specification specifies how the object behaves when operations are applied sequentially. Linearizability implies that each operation should appear to take place instantaneously at some point in time, and that the relative order of non-concurrent operations is preserved.

An object o is *k-universal* if any object which has sequential specification has a *k-obstruction-free* linearizable implementation using atomic registers and objects of type o for any number of processes.

Clearly, if an object is k-universal its is also universal for k processes. An interesting question, that we resolve later, is whether an object that is universal for k processes is also k-universal in a system with *more than* k processes.

Summary of contributions

A new approach. We propose a new classification for evaluating the strength of shared objects, which is based on the new notions of *power numbers* and the *power hierarchy*. The new classification together with the technical results mentioned below, provide a deeper understanding into the nature of the relative computational power of shared objects.

An equivalence result. We show that the traditional approach which is based on determining the consensus number of an object and our approach, are two sides of the same coin. We prove that the wait-free hierarchy and power hierarchy are equivalent. That is, the consensus number of an object equals to its power number. The new classification together with the equivalence result, does not yet fully enables to answer Alice's question.

An extended universality result. We extend the known universality result for the wait-free hierarchy in the following non-trivial way. If the consensus number (or the power number) of an object o is k, then not only o is universal for k processes, o is also k-universal in a system with *any* number of processes. Put another way, an object is universal for k processes if and only if it is k-universal for any number of processes. Now, we can fully answer Alice's question. The consensus number of a swap object is 2. Thus, using the type of atomic operations that Alice's machine supports, any object or problem which has sequential specification has a 2-obstruction-free linearizable implementation for any number of processes; and this claim does not hold for 3-obstruction-freedom.

Related work

The consensus problem was formally defined in [14]. The impossibility result that there is no consensus algorithm that can tolerate even a single crash failure in an asynchronous model was first proved for the message-passing model in [3], and later has been extended for the shared memory model in which only atomic read/write registers are supported, in [12]. A recent survey which covers many related impossibility results can be found in [2].

The power of various shared objects has been studied extensively in shared memory environments where processes may fail benignly, and where every operation is wait-free. In [5], Herlihy classified shared objects by their consensus number and defined the wait-free hierarchy. He found the consensus number of many objects and, in particular, proved that the consensus number of an atomic swap object is 2. Additional results regarding the wait-free hierarchy can be found in [8, 10].

Objects that can be used, together with atomic registers, to build wait-free implementations of any other object are called *universal objects*. Previous work provided methods, called universal constructions, to transform sequential specifications of arbitrary shared objects into wait-free concurrent implementations that use universal objects [5, 15]. In [15] it is proved that sticky bits are universal, and independently, in [5] it is proved that wait-free consensus objects are universal. A bounded space version of the universal construction from [5] appears in [9]. The universal construction that we use to prove Theorem 3 conceptually mimics the original construction from [5].

As already mentioned, two extensively studied conditions are wait-freedom [5], and obstruction-freedom [6]. It is shown in [6] that obstruction-free consensus is solvable using atomic registers. Wait-free consensus algorithms that use read and write operations in the absence of (process) contention, or even in the absence of step contention, and revert to using strong synchronization operations when contention occurs, are presented in [1, 11, 13]. Linearizability is defined in [7].

The notion of k-obstruction-freedom is presented in [17], as part of a transformation that is used to fuse objects which avoid locking and locks together in order to create new types of shared objects. A weaker set of progress conditions, called k-obstacle-freedom, which cover the spectrum between obstruction-freedom and non-blocking (sometimes called lock-freedom) is defined in [17]. Results similar to those presented in this paper for k-obstruction-freedom, can also be proved w.r.t. k-obstacle-freedom.

2 An Equivalence Result

We show below that the traditional approach which is based on determining the consensus number of an object and our approach, are two sides of the same coin. Our first technical result is that the wait-free hierarchy and power hierarchy are equivalent. Put another way, we show that the consensus number of an object equals to its power number.

To simplify the presentation, it is sometimes convenient to use the notion of a consensus object instead of consensus algorithm (the two notions are essentially the same). A consensus object o supports one operation: $o.propose(v)$ satisfying: (1) *Agreement*: In any run, the $o.propose()$ operation returns the same value, called the *consensus value*, to every process that invokes it. (2) *Validity*: In any run, if the consensus value is v, then some process invoked $o.propose(v)$. When the value $v \in \{0, 1\}$ the object is called a binary consensus object. Throughout the paper, unless otherwise stated, by a consensus object we mean a binary consensus object; and by *n-consensus* we mean a multi-valued consensus object where $v \in \{0, 1, ..., n-1\}$.

Theorem 1 (Equivalence). *For any object of type o, $PN(o) = CN(o)$.*

Proof. In the sequel, the term k-obstruction-free consensus algorithm, means a consensus algorithm which, for *any* number of processes, is correct assuming that the required progress condition is k-obstruction-freedom. It follows immediately from the definitions that, for any $k \geq 1$, a k-obstruction-free consensus algorithm is also a wait-free consensus algorithm for k processes. This simple observation implies that, for any object type o, $PN(o) \leq CN(o)$. The difficult part of the proof is to show that, for any $k \geq 1$, it is possible to implement a k-obstruction-free consensus algorithm using only wait-free consensus objects for k processes and atomic read/write registers. Such an implementation together with the above observation implies the theorem. Below we present such an implementation of a k-obstruction-free consensus algorithm and prove its correctness.

In the implementation we use a function, called $set_k()$, from the positive integers into sets of size k of process ids. We do not care how exactly the function is implemented, we only care that, for every positive integer k, there exists a function $set_k()$ which satisfies the following property: For every set of k process ids P and every positive integer s there exists $t \geq s$ such that $P = set_k(t)$. That is, every set of k process ids appears infinitely often.

The algorithm proceeds in rounds. The notion of a *round* is used only for the sake of describing the algorithm. We do *not* assume a synchronous model of execution in which all the processes are always executing the same round, and where no process can move to the next round before all others have finished the previous round. Each process has a preference for the decision value in each round; initially this preference is the input value of the process. If no decision is made in a round then the processes advance to the next round, and try again to reach agreement.

Algorithm 1. k-OBSTRUCTION-FREE CONSENSUS FOR ANY NUMBER OF PROCESSES USING ATOMIC REGISTERS AND WAIT-FREE CONSENSUS OBJECTS FOR k PROCESSES:
`program for process` p_i `with input` in_i `(where` $in_i \in \{0,1\}$`).`

shared registers
$x[0..\infty, 0..1]$ infinite array of bits, initially $x[0,0] = x[0,1] = 1$ and all other entries are 0
$con[1..\infty]$ infinite array of wait-free consensus objects for k processes
decide ranges over $\{\perp, 0, 1\}$, initially $\perp$

local registers
r_i integer, initially 1
v_i bit, initially in_i

```
1 while decide $=\perp$ do
2     if $x[r_i, v_i] = 0$ then if $x[r_i, 1-v_i] = 1$ then $v_i := 1 - v_i$ else $x[r_i, v_i] := 1$ fi fi
3     if $x[r_i - 1, 1 - v_i] = 0$ then decide $:= v_i$ /*no conflict in prev round */
4         else if $p_i \in set_k(r_i)$ then $v_i :=$ con$[r_i]$.propose$(v_i)$ fi    /*update pref */
5     fi
6     $r_i := r_i + 1$
7 od
8 decide(decide)
```

In round $r \geq 1$, process p_i first checks if the flag of its preference v_i is already set. If it is not set and the flag of $1 - v_i$ is set, p_i changes its preference to $1 - v_i$. If both flags are not set, p_i flags its preference v_i by writing 1 to $x[r, v_i]$ (line 2). Then, p_i reads the flag $x[r-1, 1-v_i]$. If the flag $x[r-1, 1-v_i]$ is not set, then every process that reaches round r with the conflicting preference $1 - v_i$ will find that only $x[r, v_i]$ is set to 1, and will change its preference to v_i. Consequently, process p_i can safely decide on v_i, and it writes v_i to *decide* (line 3). Otherwise, p_i checks if it belongs to the set $set_k(r)$. If it does, p_i proposes its current preference v_i to $con[r]$ and updates its preference to be the value agreed upon in $con[r]$. Then, p_i proceeds to round $r+1$ (line 4).

If only up to k processes with conflicting preferences participate in round r, and all them are in $set_k(r)$, then all of them will reach round $r+1$ with the same preference which is the value agreed upon in $con[r]$. When all processes reach a round with the same preference, a decision is reached either in that round or the next round.

Theorem 2. *Algorithm* 1 *is a correct k-obstruction-free consensus algorithm for any number of processes, using atomic registers and wait-free consensus objects for k processes.*

Before we prove the theorem, we make the following observations:

– Let Algorithm 2 be a modified version of Algorithm 1, where line 4 is omitted. Then, Algorithm 2 is a correct 1-obstruction-free consensus algorithm for any number of processes using atomic registers only.
– Let $y[1..\infty]$ be an infinite array of swap objects, which range over $\{\bot, 0, 1\}$, initially all set to $\bot$, and let $temp_i$ be a local register of process p_i. Let Algorithm 3 be a modified version of Algorithm 1, where line 4 is replace with:
"**else** $temp_i := v_i$; $swap(y[i], temp_i)$; **if** $temp_i \neq \bot$ **then** $v_i := temp_i$ **fi**"
Then, Algorithm 3 is a correct 2-obstruction-free consensus algorithm for any number of processes using atomic registers and swap objects.
– Let $y[1..\infty]$ be an infinite array of test&set bits, initially all set to 0, and let Algorithm 4 be a modified version of Algorithm 1, where line 4 is replace with:
"**else if** $test\&set(y[r_i]) = 1$ **then if** $x[r_i, 1-v_i] = 1$ **then** $v_i := 1 - v_i$ **fi fi**"
Then, Algorithm 4 is a correct 2-obstruction-free consensus algorithm for any number of processes using atomic registers and test&set bits[3].

Below we present a correctness proof of the algorithm. Let $r \geq 1$ and $v \in \{0, 1\}$. Process p_i *reaches* round r, if it executes Statement 2 with $r_i = r$. Process p_i *prefers* the value v in round r, if $v_i = v$ when p_i reaches round r. Process p_i *commits* to the value v in round r, if it executes the assignment *decide* $:= v$ with $r_i = r$.

Lemma 1. *If all processes reaching round r have the same preference v for round r, then all nonfaulty processes reaching round r commit to v either in round r or in round $r+1$.*

Proof. Suppose all processes reaching round r have the same preference v for round r. Then, whenever some process p_i sets the bit $x[r, v_i]$ to 1, v_i equals v. Consequently, no process ever sets $x[r, 1-v]$ to 1, and hence $x[r, 1-v]$ always equals 0. Now consider a process p reaching round r. Assuming that p continues to take steps in round r, p will either (1) finds $x[r-1, 1-v] = 0$ at Statement 3, and commits to the value v in

[3] A test&set bit, say r, is a shared bit that supports two operations: (1) $test\&set$, which writes 1 to r, and returns the old value (which is either 0 or 1); and (2) *reset*, which writes 0 into r (and does not return a value).

round r, or (2) will continue to round $r+1$ with preference v. In the second case, all processes reaching round $r+1$ have the same preference v for round $r+1$, thus, p will find $x[r, 1-v] = 0$ at round $r+1$, and will commit to the value v in round $r+1$. □

Lemma 2 (validity). *If p_i decides on a value v then $in_j = v$ for some p_j.*

Proof. If there are two processes that have different inputs then the lemma holds trivially. Suppose all processes start with the same input *in*. Then, by Lemma 1, all nonfaulty processes will commit to in in the first round or the second round (actually the second round in this case), will execute the statement **decide**(*decide*), at the end of one of these two rounds, and will decide on *in*. □

Lemma 3. *If some process commits to v in round r then all processes reaching round $r+1$ prefer v and commit to v in round $r+1$.*

Proof. Suppose some process p commits to v in round r. Since p finds $x[r-1, 1-v] = 0$ at Statement 3, it follows that every process with preference $1-v$ for round r, will find in Statement 2 that $x[r, 1-v] = 0$ and $x[r, v] = 1$, and will change its preference to v. This implies that for a committed value v, no process ever sets $x[r, 1-v]$ to 1 in round r. It follows that all processes reaching round $r+1$ prefer v in round $r+1$, and since they will find in round $r+1$ that $x[r, 1-v] = 0$, they will all commits to v in round $r+1$. □

Lemma 4 (agreement). *No two processes decide on conflicting values.*

Proof. We show that no two processes commit to conflicting values. This implies that no two processes decide on conflicting values. Assume to the contrary that two processes commit to conflicting values. This means that there exist nonfaulty processes p_0 and p_1 such that p_0 commits to 0 in round r and p_1 commits to 1 in round r'. First suppose that $r \neq r'$. Without loss of generality, let $r < r'$. Since p_0 commits to 0 in round r, from Lemma 3 all processes reaching round $r+1$, and in particular p_1, commit to 0 in round $r+1$; a contradiction. Now suppose that $r = r'$. In round r, process p_0 commits to 0, and process p_1 commits to 1. Since process p_0 finds $x[r-1, 1] = 0$ at Statement 3, process p_1 must find $x[r, 1] = 0$ and $x[r, 0] = 1$ at Statement 2 and change it preference to 0 in round r. Consequently, it is not possible that both commit in round r. □

Lemma 5. *Let P be an arbitrary non empty set of at most k processes, and let r be a positive integer such that $P \subseteq set_k(r)$. If the processes in P complete the execution of round r and round $r+1$, before any of the other processes reach round r, then all nonfaulty processes reaching round $r+1$ (1) have the same preference, say v, for round $r+1$, and (2) commit to v either in round $r+1$ or in round $r+2$.*

Proof. Assume first the processes in P all have the *same* preference v, and complete the execution of round r and round $r+1$ before any of the other process reaches round r. Since it is assumed that they execute round r without interference from the other processes, they will reach also round $r+1$ with the same preference v, and with $x[r, 1-v] = 0$. No process that will arrive later will be able to change the value of $x[r, 1-v]$.

Thus, in round $r+1$ they will commit to v. Every other process that will reach round r later, will find in Statement 2 that $x[r, 1-v] = 0$ and $x[r, v] = 1$, and will change its preference to v in case it was $1-v$. This implies that no process ever sets $x[r, 1-v]$ to 1 in round r. It follows that all processes reaching round $r+1$ prefer v in round $r+1$, and since they will find in round $r+1$ that $x[r, 1-v] = 0$, they will all commit to v in round $r+1$.

Now, assume that the processes in P reach round r with *different* preferences (i.e., some prefer 0 and some 1), and complete the execution of round r and round $r+1$ before any of the other process reaches round r. Clearly, it is not possible that each one of the processes in P changes its preferences while executing Statement 2 in round r. If only the processes with input 0 (resp. input 1) change their preference while executing Statement 2 in round r, then we are in a case, similar to a one already covered, where all the processes in P reach round r with the same preference.

Thus, let us assume that not all the processes change their preference while executing Statement 2, in round r, and thus both $x[r, v]$ and $x[r, 1-v]$ will be set to 1. In such a case, Statement 4 ensures that all the processes in P will reach round $r+1$ with the same preference with the same preference v, which is the value agreed upon in $con[r]$. Since it is assumed that they execute round $r+1$ without interference from the other processes, they will reach also round $r+2$ with the same preference v, and with $x[r+1, 1-v] = 0$. No process that will arrive later will be able to change the value of $x[r+1, 1-v]$. Thus, in round $r+2$ they will commit to v. Every other process that will reach round $r+1$ later, will find in Statement 2 that $x[r+1, 1-v] = 0$ and $x[r+1, v] = 1$, and will change its preference to v in case it was $1-v$, and will later reach round $r+2$ with preference v. This implies that no process ever sets $x[r+1, 1-v]$ to 1 in round $r+1$. It follows that all processes reaching round $r+2$ prefer v, and since they will find in round $r+2$ that $x[r+1, 1-v] = 0$, they will all commit to v in round $r+2$. $\square$

Lemma 6 (liveness with k-obstruction-freedom). *Let P be an arbitrary non-empty set of at most k processes. Each nonfaulty process eventually decides and terminates, regardless whether the other processes are faulty or not, in every run in which from some point on all the processes, except those in P, do not take any steps until the nonfaulty processes in P decide.*

Proof. Assume that from some point on, say from time t, all the processes, except those in P, "hold still" (i.e., do not take any steps) until the nonfaulty processes in P decide. Let $r_i(t)$ be the value of the local register r_i at time t. Define the maximum round reached at time t, denoted $r(t)$, as: $r(t) = \text{maximum}(\{r_j(t) \mid j \in \text{set of all process' identifiers}\})$. Let r' be the smallest integer such that $r(t) \leq r'$ and $P \subseteq set_k(r')$. Then, it follows from Lemma 5 that each nonfaulty process commits before or during round $r'+2$, and later executes the statement **decide**(*decide*) and decides. $\square$

This completes the proof of Theorem 1. $\square$

3 An Extended Universality Result

An object o is *universal* for k processes if any object which has sequential specification has a wait-free linearizable implementation using atomic registers and objects of type o

in a system with at most k processes [5]. Lets assume that we know that o is universal in a system with k processes, what can we say about the computational power of o in a system with more than k processes? As we prove below, in such a case it follows from our extended universality result that o is also k-universal in a system with *any* number of processes. That is, any object which has sequential specification has a *k-obstruction-free* linearizable implementation using atomic registers and objects of type o for any number of processes.

Theorem 3 (Extended Universality). *For any object o and positive integer $k \geq 1$,*

1. *A k-obstruction-free consensus object is k-universal.*
2. *o is k-universal if and only if $PN(o) \geq k$.*
3. *o is k-universal if and only if $CN(o) \geq k$.*
4. *o is k-universal if and only if o is universal for k processes.*

Proof. Proving Part 1 of the theorem is difficult. So, before proving it, we first explain why all the other three statements are simple consequences of Theorem 1 and Part 1 of Theorem 3. **Part 2:** If o is k-universal then, by definition, k-obstruction-free consensus can be implemented using atomic registers and objects of type o, and hence $PN(o) \geq k$. If $PN(o) \geq k$, then by Theorem 2, k-obstruction-free consensus can be implemented using atomic registers and objects of type o, and thus, by Part 1 of Theorem 3, o is k-universal. **Part 3:** This item follows immediately from Part 2 and the fact that $PN(o) = CN(o)$ (i.e., Theorem 1). **Part 4:** In a system with at most k processes, k-obstruction-free is the same as wait-freedom, thus, if an object is k-universal its is clearly also universal for k processes. If an object, say o, is universal for k processes, by definition, o can implement consensus objects for k processes. Thus, by Theorem 2, k-obstruction-free consensus algorithm can be implemented using atomic registers and objects of type o. Thus, by Part 1 of Theorem 3, o is k-universal.

To prove Part 1 of Theorem 3, we present below a universal construction that implements any k-obstruction-free object o from k-obstruction-free consensus objects and atomic registers. The construction conceptually mimics the original construction for the wait-free model from [5]. The basic idea behind the construction is as follows: an object o is implemented as a linked list which is represented as an unbounded array. The entries of the array represent a sequence of invocations applied to the object. A process invokes an operation by threading a new invocation onto the end of the list. The current state of the objects corresponds to applying the sequence of invocations to the object.

First we need to generalize Algorithm 1. Recall that by *n-consensus* we mean a multi-valued consensus object where the input value taken from the set $\{0, 1, ..., n-1\}$.

Theorem 4. *For any positive integers k and n, it is possible to implement a k-obstruction-free n-consensus object for any number of processes, using atomic registers and k-obstruction-free binary consensus objects for any number of processes.*

Proof. Starting from k-obstruction-free binary consensus objects for any number of processes, we can trivially get wait-free binary consensus objects for k processes. It is well know that using atomic registers and wait-free binary consensus objects for k processes, it is simple to implement wait-free n-consensus objects for k processes ([16], page 329). Below we show that using atomic registers and wait-free n-consensus objects for k processes, it is possible to implement a k-obstruction-free n-consensus object for any number of processes. To do that, we present below Algorithm 2 which is a simple modification of Algorithm 1.

Algorithm 2. k-OBSTRUCTION-FREE n-CONSENSUS FOR ANY NUMBER OF PROCESSES USING ATOMIC REGISTERS AND WAIT-FREE n-CONSENSUS OBJECTS FOR k PROCESSES:
`program for process` p_i `with input` in_i `(where` $in_i \in \{0, 1, ..., n-1\}$`) .`

shared registers
$x[0..\infty, 0..n-1]$ infinite array of bits, initially entries of $x[0..n-1]$ are 1, all other entries are 0
$con[1..\infty]$ infinite array of wait-free n-consensus objects for k processes
decide ranges over $\{\perp, 0, 1, ..., n-1\}$, initially $\perp$

local registers
r_i integer, initially 1 ; v_i integer, initially in_i ; j integer

```
1  while decide =⊥ do
2      if $x[r_i, v_i] = 0$ then
3          $j := 0$; while $j < n$ and $x[r_i, j] = 0$ do $j := j + 1$ od
4          if $j < n$ then $v_i := j$ else $x[r_i, v_i] := 1$ fi fi
5      $j := 0$; while($v_i = j$) or ($j < n$ and $x[r_i - 1, j] = 0$) do $j := j + 1$ od
6      if $j = n$ then $decide := v_i$        /* no conflict in previous round */
7          else if $p_i \in set_k(r_i)$ then $v_i := con[r_i].propose(v_i)$ fi  /* update pref */
8      fi
9      $r_i := r_i + 1$
10 od
11 decide(decide)
```

In round $r \geq 1$, process p_i first checks if the flag of its preference v_i is already set (line 2). If it is not set and a flag for some other value is set, p_i changes its preference to the smallest such value. If non of the flags are set, p_i flags its preference v_i by writing 1 to $x[r, v_i]$ (line 4). Then, p_i reads all the flags from round $r-1$ (line 5). If non of the flags from round $r-1$ (excluding its own) is set, then every process that reaches round r with a conflicting preference, will find that only $x[r, v_i]$ is set to 1, and will change its preference to v_i. Consequently, process p_i can safely decide on v_i, and it writes v_i to *decide* (line 6).

Otherwise, p_i checks if it belongs to the set $set_k(r)$. If it does, p_i proposes its current preference v_i to $con[r]$ and updates its preference to be the value agreed upon in $con[r]$ (line 7). Then, p_i proceeds to round $r+1$. If only up to k processes with conflicting preferences participate in round r, and all of them are in $set_k(r)$, then all of them will reach round $r+1$ with the same preference which is the value agreed upon in $con[r]$. When all processes reach a round with the same preference, a decision is reached either in that round or the next round.

The assumption that n is finite and a priori known, can be removed by replacing the while loops in lines 3 and 5, with a known snapshot algorithm for unbounded number of processes. from [4]. □

We assume any shared object, o, is specified by two relations:

$$apply \subset \text{INVOKE} \times \text{STATE} \times \text{STATE},$$

$$\text{and } reply \subset \text{INVOKE} \times \text{STATE} \times \text{RESPONSE},$$

where INVOKE is the object's domain of invocations, STATE is its domain of states (with a designated set of start states), and RESPONSE is its domain of responses.

1. The *apply* relation denotes a state change based on the pending invocation and the current state. Invocations do not block: it is required that for every invocation and current state there is a target state.
2. The *reply* relation determines the calculated response, based on the pending invocation and the updated state. It is required that for any pair INVOKE $\times$ STATE there is a target state and a response.

Let o be an an arbitrary k-obstruction-free object which can be specified as described above. We present a universal construction that implements o from k-obstruction-free consensus objects and atomic registers. Since, by Theorem 4, k-obstruction-free n-consensus objects can be implemented from k-obstruction-free binary consensus objects and atomic registers, we will use in the construction below only k-obstruction-free n-consensus objects. The construction is similar to the one for the wait-free model from [16], where the wait-free n-consensus objects are replaced with k-obstruction-free n-consensus objects.

In the actual implementation there are two principal data structures:

1. For each process i there is an unbounded array, *Announce*$[i][1..\infty]$, each element of which is a *cell* which can hold a single invocation. The *Announce*$[i][j]$ entry describes the j-th invocation (operation name and arguments) by process i on o.
2. The object is represented as an unbounded array *Sequence*$[1..\infty]$ of process-id's, where for each positive integer g, *Sequence*$[g]$ is a k-obstruction-free n-consensus object. Intuitively, if *Sequence*$[k] = i$ and *Sequence*$[1], \ldots,$ *Sequence*$[k-1]$ contains the value i in exactly $j-1$ positions, then the k-th invocation on o is described by *Announce*$[i][j]$. In this case, we say that *Announce*$[i][j]$ has been *threaded.*

The universal construction of any k-obstruction-free object o is described below as the code process i executes to implement an operation on o with invocation *invoke*. For simplicity, we will assume that the input values for an *n-consensus* object are taken from the set $\{1, ..., n\}$ (instead of $\{0, 1, ..., n-1\}$).

In outline, the construction works as follows: process i first announces its next invocation, and then threads unthreaded, announced invocations onto the end of *Sequence*. It continues until it sees that its own operation has been threaded, computes a response, and returns. To ensure that each announced invocation is eventually threaded, the correct processes first try to thread any announced, unthreaded cell of process ℓ into entry *Sequence*$[k]$, where $\ell = k \pmod n + 1$. This "helping" technique guarantees that once process ℓ announces an operation, at most n other operations can be threaded before the operation of process ℓ is threaded.

ALGORITHM 3. A UNIVERSAL CONSTRUCTION:
`program for process` $i \in \{1,\ldots,n\}$ `with invocation` *invoke*

```
shared
    Announce[1..n][1..∞] array of cells which range over INVOKE ∪ {⊥},
        initially all cells are set to ⊥
    Sequence[1..∞] array of k-obstruction-free n-consensus objects
local to process i
    MyNextAnnounce integer, initially 1                     /* next vacant cell */
    NextAnnounce[1..n] array of integers, initially 1
                                                            /* next operation */
    CurrentState ∈ STATE, initially the initial state of o        /* i's view */
    NextSeq integer, initially 1                      /* next entry in Sequence */
    Winner range over {1,...,n}                      /* last process threaded */
    ℓ range over {1,...,n}                                 /* process to help */
                          /* write invoke to a vacant cell in Announce[i] */
1   Announce[i][MyNextAnnounce] := the invocation invoke
2   MyNextAnnounce := MyNextAnnounce + 1
3   while ((NextAnnounce[i] < MyNextAnnounce) do
                                           /* continue until invoke is threaded */
                                        /* each iteration threads one operation */
4       ℓ := NextSeq (mod n) + 1                       /* select process to help */
5       while Announce[ℓ][NextAnnounce[ℓ]] = ⊥                        /* valid? */
7       do
6           ℓ := ℓ + 1                        /* not valid; help next process */
7       od
9       Winner := Sequence[NextSeq].propose(ℓ)                     /* propose ℓ */
                                       /* a new cell has been threaded by Winner */
                                                       /* update CurrentState */
10      CurrentState := apply(Announce[Winner][NextAnnounce[Winner]], CurrentState)
11      NextAnnounce[Winner] := NextAnnounce[Winner] + 1
12      NextSeq := NextSeq + 1
13  od
14  return(reply(invoke, CurrentState))
```

Process i keeps track of the first index of *Announce*$[i]$ that is vacant in a variable denoted *MyNextAnnounce*, and first writes the invocation into *Announce*$[i]$[*MyNextAnnounce*], and (line 2) increments *MyNextAnnounce* by 1. To keep track of which cells it has seen threaded (including its own), process i keeps n counters in an array *NextAnnounce*$[1..n]$, where each *NextAnnounce*$[j]$ is one plus the number of times i has read cells of j in *Sequence*. Hence *NextAnnounce*$[j]$ is the index of *Announce*$[j]$ where i looks to find the next operation announced by j. We notice that, having incremented *MyNextAnnounce*:

> *NextAnnounce*$[i]$ $=$ *MyNextAnnounce*-1 until the current operation of process i has been threaded.

This inequality is thus the condition (line 3) in the while loop (lines 3 – 13) in which process i threads cells. Once process i's invocation is threaded (and *NextAnnounce*$[i]$ = *MyNextAnnounce*), it exits the loop and returns the associated response value (line

14). Process i keeps an index *NextSeq* which points to the next entry in *Sequence*$[1..\infty]$ whose element it has not yet accessed.

To thread cells, process i proposes (line 9) the id of process ℓ to the k-obstruction-free consensus object *Sequence*[*NextSeq*], and after a decision is made, records the consensus value for *Sequence*[*NextSeq*] in the local variable *Winner* (line 9). The value in *Sequence*[*NextSeq*] is the identity of the process whose cell has just been threaded. After choosing to help process ℓ (line 4), process i checks that *Announce*$[\ell]$[*NextAnnounce*$[\ell]$] contains a valid operation invocation. As discussed above, process i gives preference (line 4) to a different process for each cell in *Sequence*. Thus, all active processes will eventually agree to give preference to any pending invocation, ensuring it will eventually be threaded.

Once process i knows the id of the process whose cell has just been threaded, as recorded in *Winner*, it can update (line 10) its view of the object's state with the winner invocation, and increment its records of process *Winner*'s successfully threaded cells (line 11) and the next unread cell in *Sequence* (line 12). Having successfully threaded a cell, process i returns to the top of the while loop (line 3). Eventually, the invocation of process i will be threaded and the condition at the while loop (line 3) will be *false*. At this point, the value of the variable *CurrentState* is the state of the object after process i's invocation has been applied to the object. Based on this state, process i can return the appropriate response. This completes the proof of Theorem 3. □

4 Discussion

"Please explain me" requested Bob once he understood the new results, "when you write a concurrent application for your new machine, and contention goes above two, what do you do?". "In such cases, contention resolution schemes such as exponential backoff, are used" explained Alice, "and since with a machine which supports an atomic swap, we can easily deal with contention of two threads, the contention resolution scheme can kick-in only once three threads are contending, and not before. Furthermore, once the contention resolution scheme kicks-in, it is enough to use it until two, and not just one, contending threads stay alive. From a practical point a view, this is a big performance gain". "I see" said Bob, "so there is a tradeoff between how strong the progress condition should be, and how often the contention resolution scheme will kick-in". "Exactly" said Alice, "this is one of the reasons why the new classification and results are so helpful".

References

1. H. Attiya, R. Guerraoui, and P. Kouznetsov. Computing with reads and writes in the absence of step contention. *Proceedings of the 19th International Symposium on Distributed Computing*, LNCS 3724, 122–136, 2005.
2. F. E. Fich and E. Ruppert. Hundreds of impossibility results for distributed computing. *Distributed Computing*, 16(2-3):121–163, 2003.
3. M.J. Fischer, N.A. Lynch, and M.S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

4. E. Gafni, M. Merritt, and G. Taubenfeld. The concurrency hierarchy, and algorithms for unbounded concurrency. In *Proc. 20th ACM Symp. on Principles of Distributed Computing*, pages 161–169, August 2001.
5. M. P. Herlihy. Wait-free synchronization. *ACM Trans. on Programming Languages and Systems*, 13(1):124–149, January 1991.
6. M. P. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proc. of the 23rd Int. Conf. on Dist. Computing Systems*, 2003.
7. M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *toplas*, 12(3):463–492, 1990.
8. P. Jayanti. Robust wait-free hierarchies. *Journal of the ACM*, 44(4):592–614, 1997.
9. P. Jayanti and S. Toueg. Some results on the impossibility, universality, and decidability of consensus. In *Proc. of the 6th Int. Workshop on Distributed Algorithms: LNCS 674*, pages 69–84, 1992.
10. Wai-Kau Lo and Vassos Hadzilacos. All of us are smarter than any of us: Nondeterministic wait-free hierarchies are not robust. *SIAM Journal on Computing*, 30(3):689–728, 2000.
11. V. Luchangco, M. Moir and N. Shavit. On the uncontended complexity of consensus. *Proc. of the 17th International Symp. on Distributed Computing*, LNCS 2848, 45–59, 2003.
12. M.C. Loui and H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, 4:163–183, 1987.
13. M. Merritt and G. Taubenfeld. Resilient consensus for infinitely many processes. *Proc. of the 17th International Symp. on Distributed Computing*, LNCS 2848, 1–15, 2003.
14. M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, 1980.
15. S. A. Plotkin. Sticky bits and universality of consensus. In *Proc. 8th ACM Symp. on Principles of Distributed Computing*, pages 159–175, 1989.
16. G. Taubenfeld. Synchronization Algorithms and Concurrent Programming. *Pearson / Prentice-Hall*, ISBN 0-131-97259-6, 423 pages, 2006.
17. G. Taubenfeld. Contention-sensitive data structures and algorithms. *Proc. of the 23rd International Symp. on Distributed Computing*, September 2009. To appear.