A NEW KIND OF NEW KIND OF SCIENCE by Owen T. Cunningham

INTRODUCTION

As a science amateur and software professional, I find the digital physics community's fixation on *the bit* disheartening. To the extent that J. A. Wheeler's "It from bit" [0] or MIT's "Center for Bits and Atoms" represent mere whimsical euphemism for computation as a whole, that is one thing; but I get the impression that a good many of the thought leaders in this area genuinely believe that a computational model of everything is rooted in the bit. I think there are several compelling reasons why those thought leaders should at least consider abandoning this belief. This paper discusses those reasons and then presents an alternative conception of digital physics that I call *computational ontology*.

A BIT ON THE SIDE

The first objection to the centrality of the bit is its cardinality. The *b* in "bit" comes from *binary*, and the first thing anyone learns when using, writing, or studying software is that all information can be represented as zeroes and ones. And it is true that binary notation and powers of two do exhibit a number of deep, marvelous, and useful properties. But the truth of this is largely coincidental. If the universe really is built on a notion of a digit, it is not automatically a given that that digit must be binary.

We built our conception of bits on the number two not because of two's deep and marvelous properties but because it was the easiest way to encode information in an electrical circuit. The precise amount of electricity flowing through a given circuit fluctuates measurably, even if it never strays from the ranges designated as *on*/*off* or *high*/*low*. The current flowing through an *on* circuit is not fixed, and even an *off* circuit has some nonzero amount trickling through it. The decision to restrict the number of possible values in a digital computer to two owes much more to the electrical engineering constraints of the 1940s than to any intrinsic information-theoretic necessity.

Even if we grant that the universe is best notated in base 2, there is still a deeper problem with the bit notion. Man-made computers divide bits into two categories: code and data. Certain relatively uncommon circumstances involve treating a given sequence of bits as code in some cases and data in others, but for the most part, they're distinct. Simply declaring that the universe must be a computer because everything in it can be represented as bits completely sidesteps the aspect of that notion that would actually be interesting and insightful, i.e. which portions of the universe are code and which are data.

But there is yet a deeper problem with the bit notion. Suppose we *were* somehow able to discern the universe's code bits from its data bits. In man-made computers, code bits are not intrinsically meaningful — they have meaning only with respect to the code's target CPU architecture. The reason that the owner of a PC has equal freedom to install MacOS, Linux, or Windows is that all three of those operating systems have been coded against the Intel x86 architecture. Motorola 68000 chips have a completely different architecture (the Apple Macintoshes of pre-OSX yore, older Sun Microsystems machines), as do the RISC chips used by IBM's AS/400 and RS/6000 midrange computers. OS/390 mainframes have yet a different chipset. These chipsets differ in register bases, endianness, and other fundamentals. Merely discerning the data-versus-code status of certain bits gets us no closer to figuring out what kind of CPU architecture the universe runs on.

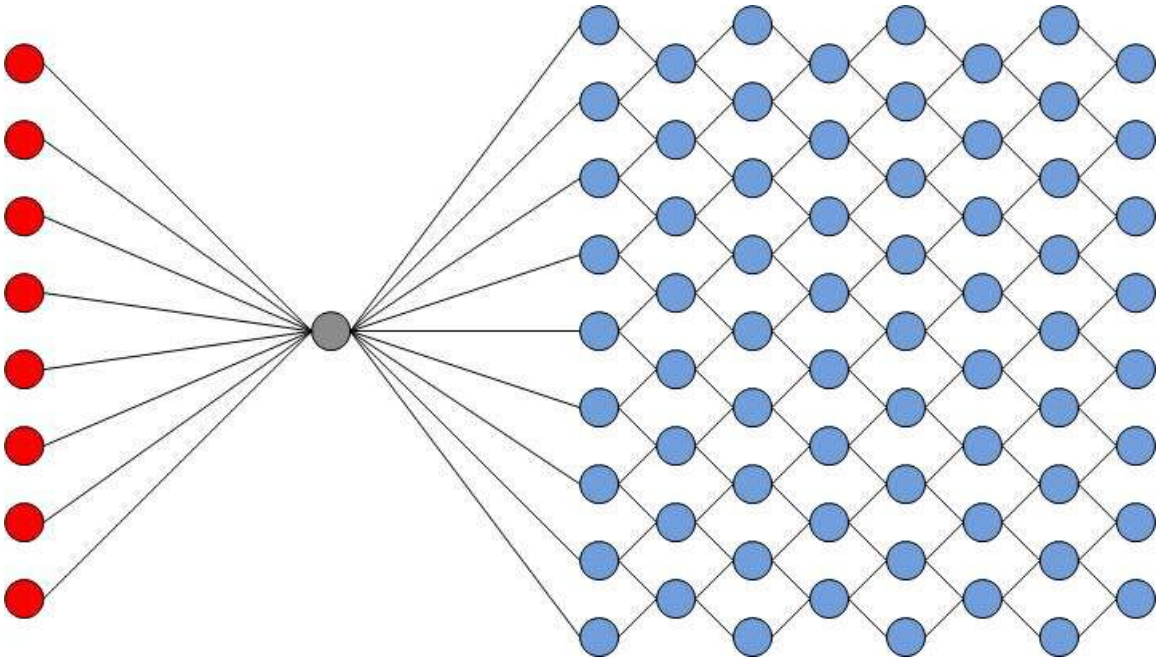
Our understanding of computer science is rooted in the study of software that runs on only a single conceptual hardware model; but the whole reason physics is interested in computer science in the first place is to explain why the underlying hardware behaves the way it does. Physics and computer science are, to borrow a term from the latter, deadlocked. So, let's see if there's a way we can remake the conceptual hardware model to break the deadlock.

THE HARDWARE ARCHITECTURE OF THE UNIVERSE

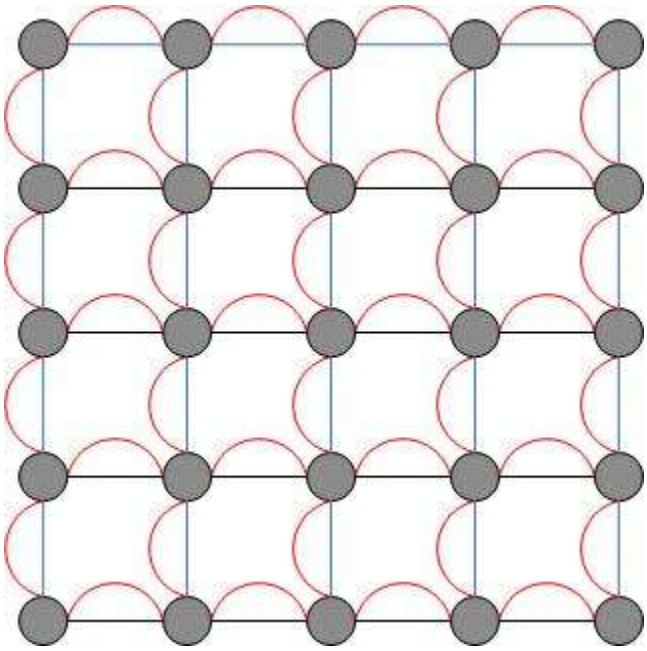
At its core, the underlying hardware architecture of nearly every electronic digital computer ever made consists of three things: processors, memory, and the bus. Processors are the things that perform well-defined operations on small chunks of data; memory is where those chunks of data are stored; and the bus is the thing between them that the chunks of data have to ride to get back and forth between the CPU and the memory.

Modern computers have lots of memory and lots of CPUs. $1,000 can buy you a machine with 1 CPU and 1,000,000,000 individually addressable locations in memory. $2,000 can get you 4 CPUs and 8,000,000,000 memory locations. $10,000 yields 32 CPUs and 32 billion bytes of memory. But all three of those machines will each have only a single bus.

To a graph theorist, this arrangement looks like this:



Red nodes are CPUs, blue nodes are individually addressable memory locations, and the one gray node is the bus. But what would happen if we got rid of the bus, and merged the two graphs, such that each node could both store *and* compute, like this:



It turns out that to properly model the behavior of such an architecture requires us to ratchet up the level abstraction away from memory and CPUs and buses, away from bits, up to the realm of pure software — to object-oriented development.

Introducing the Object Class

Imagine that each node in this graph is an instance of a single O-O class — imaginatively named `Object`. `Object`s can interact with each other in two modes: as a *thread* or as a *register*. All `Object`s act in register mode (i.e. can have data saved to or fetched from them), but only some — those that have been *activated* — can also function in thread mode.

Despite the foregoing use of the phrases "thread mode" and "register mode," it is important to realize that an *active* `Object` functions in both modes simultaneously. The mode of an `Object` is a relative proposition — one can specify an `Object`'s mode only with respect to another `Object`. The modes characterize the interactions among instances, not the instances themselves.

An `Object` that can function only in register mode is called *passive*. Active `Object`s cannot serve as registers for themselves — they must use other `Object`s to store the data needed by their threads. The process by which an active `Object` finds another `Object` to use as a register for its own computation is called *recruitment* (by analogy to cells/fibers being recruited by a muscle to execute a contraction). Active `Object`s that need computational resources recruit other `Object`s to provide those resources. Recruited `Object`s can be passive or active. (The passive-vs.-active designation pertains only to an `Object`'s status with respect to thread mode, not register mode.)

`Object`s contain other `Object`s — if you zoom in on a single `Object` instance and peer inside it, you'll find other `Object` instances. The number of times that you have to perform such zooming before you finally reach "the bottom" (i.e. an `Object` that contains no other `Object`s) is known as *rank*. Rank-0 `Object`s are at the bottom.

`Object`s are created by other `Object`s. There are two mechanisms by which such creation can occur. One is *coalescence*, whereby groups of rank-*n* `Object`s agree to coalesce into a new `Object` of rank *n*+1. There is no conservation principle governing coalescence — the rank-*n* `Object`s all continue to exist, functioning as graph nodes in their own right. Note that the rank-*n* `Object`s that initiate coalescence can be either active or passive, but the rank-*n*+1 `Object` that they create is always active.

The other `Object` creation mechanism is *copying*, whereby precisely one rank-*n* `Object` creates precisely one new rank-*n*-1 `Object`. (If a rank-0 `Object` initiates copying, the new `Object` is also rank-0.) Note that the rank-*n* `Object` initiating copying can be only active, and the rank-*n*-1 `Object` that it creates is always passive.

Regardless of the mechanism by which an `Object` is created, its initial state from a graph-theoretic perspective is the same — it's a node with a single edge connecting it to its parent. (Even in coalescence, there is only a single parent — coalescence is discussed in more detail below.) The number of edges emanating from an `Object` can grow over time, but always starts at 1.

The last major characteristic of `Object`s is that each has an address. This address is a natural number that is established at creation time and never changes. If an `Object` was created via coalescence, then its address is equal to one more than the product of the addresses of all the `Object`s that contributed to its coalescence. If it was created by expansion, then its address is simply one greater than the address of the expanding `Object`.

A Closer Look at Recruitment

Suppose an active `Object` is performing computation. It needs a register to store data for this computation. Even though it has the capability to act as a register, its rules prohibit it from using itself for this. So, it has to recruit another `Object` to act as its register.

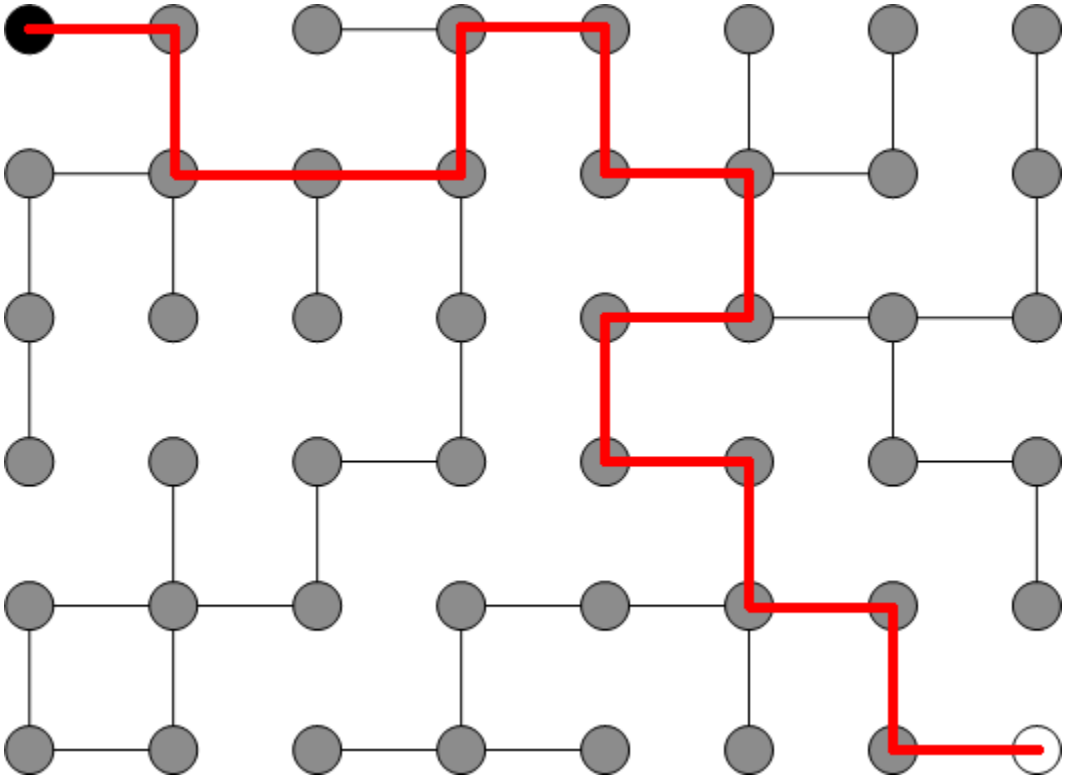
Our `Object`, being a node in a graph, has neighbors — other `Object`s connected to it by an edge. If it's a young `Object`, it probably has only one neighbor, its parent. If it's an older `Object`, it probably has multiple neighbors, of which its parent is one. So, the first thing our `Object` does is try to recruit one of its neighbors.

Our `Object` takes a "walk" and visits each of its neighbors, checking to see if there exists a neighbor that is (a) not already recruited by someone else, and (b) of lower rank than our `Object`. An `Object` cannot recruit `Object`s of equal or greater rank.

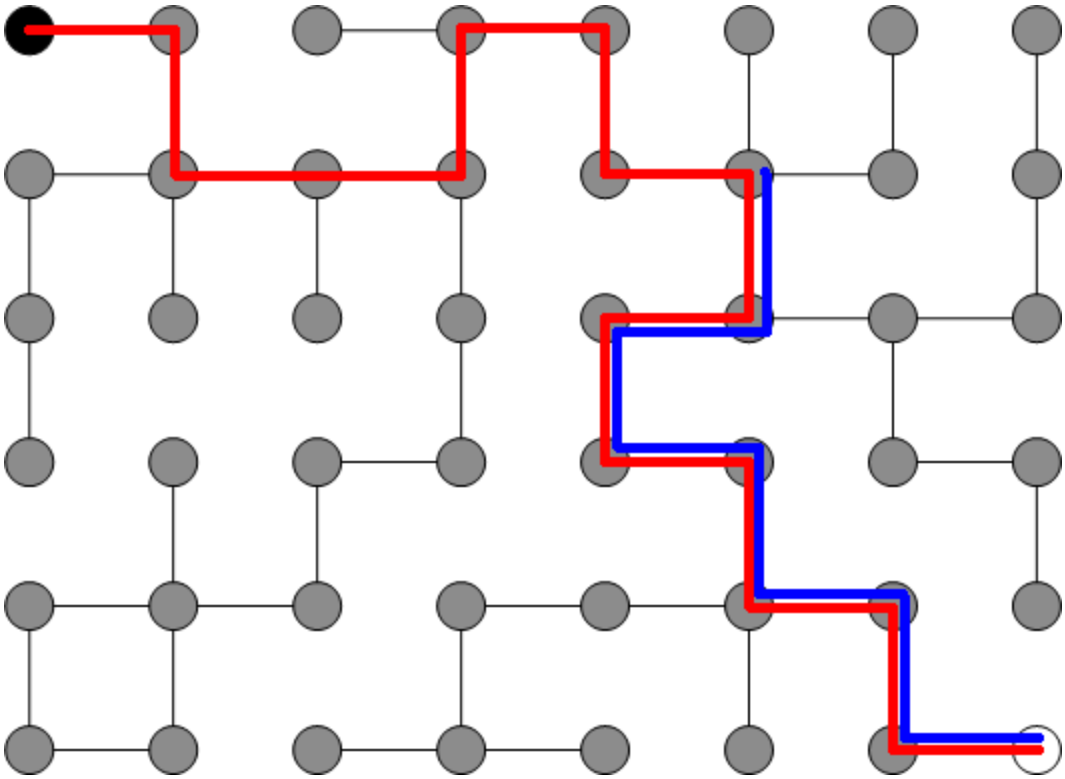
If our `Object` is able to find such a neighbor, then that neighbor becomes its recruit — our `Object` has successfully allocated the register it needed, and its thread resumes execution. If not, though, it takes another walk around the neighborhood, and this time asks its neighbors to each take a walk around *their* neighborhood, trying to recruit one of *their* neighbors, and so on. It's a breadth-first recursive search that originates from the recruiting `Object` and propagates throughout the graph.

Eventually this search terminates. If it terminates successfully, the address of the freshly recruited `Object` is returned to our `Object`. If it terminates unsuccessfully, our `Object` shrugs its shoulders and creates a new `Object` via the aforementioned process of copying.

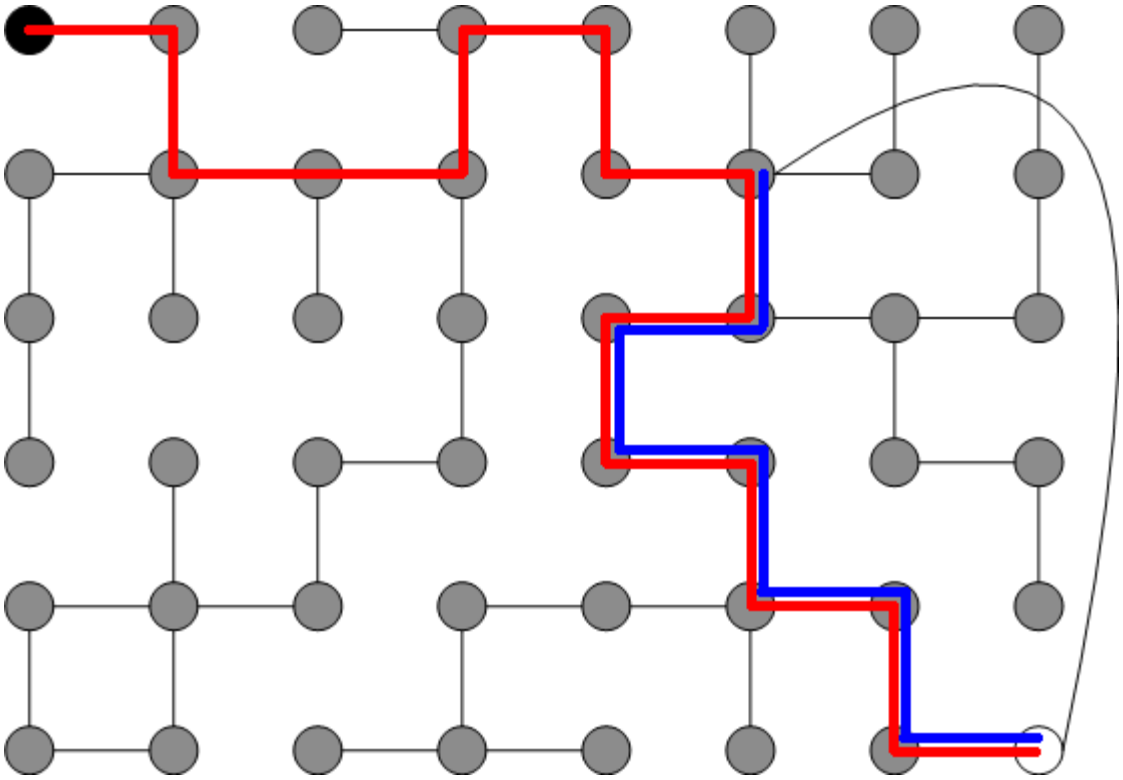
On average, it is less likely that an `Object` will be able to recruit one of its direct neighbors than that it will end up recruiting an `Object` that is at least several degrees of separation away — in other words, there will be several intermediate nodes in the graph that have to be traversed for the two `Object`s to reach each other:



In this case, something interesting happens at the very end of the recruitment process. Once the breadth-first recursive search has located a suitable recruit, that recruit *climbs back up the stack of the calling thread* by a number of stack frames equal to its own rank. To return to the earlier analogy, if our `Object` had to take a "walk" through the graph to arrive at the recruit, the recruit now retraces as much of that walk as it can based on its rank:



This retracing either brings it all the way back to our recruiting `Object` (if the degree of separation between `Object`s is less than or equal to the rank of the recruit) or brings it to some intermediate `Object` along the way (if the degree of separation is greater than the rank of the recruit). Regardless of which `Object` the recruit stops at, it inserts a new edge in the graph between itself and that `Object`. The recruit and this `Object` are now neighbors:



Over time, as active `Objects` engage in this process, not only do they accumulate recruits, but the overall graph gets denser with new edges.

These Threads Are Made for Walkin'

Active `Objects` initiate recruitment because their threads need computational resources, but one might ask, what exactly are those threads computing? It turns out they are computing another walk through the graph, but a walk subject to rules different from those governing recruitment. Recruitment allows a thread to walk through the graph without regard to the rank of the `Objects` being traversed, whereas active walks do care about the rank of the `Objects` they traverse. Recruitment walks can be thought of as "errands" — walks with a goal in mind. Active walks, by contrast, are goal-free, aimless strolls.

Threads compute their active walks by referring to two arrays embedded in the `Object` class definition; although every `Object` instance has its own copy of these arrays, the active `Object` doing the walking refers not to its own arrays but instead to those attached to whichever `Object` the thread's active walk has just brought it. The first array is the *neighbor array* — an array of addresses of adjacent nodes. The second array is the *recruit array* – an array of addresses of lower-rank nodes that have been recruited. Each of these arrays has a corresponding pointer variable that keeps track of which member is the "current" one.

As soon as a walking thread arrives at a node, it computes the arithmetic modulo of *m* by *n*, where *m* is the sum of the walking node's address and the current node's neighbor array pointer value, and *n* is the size of that same array. This new pointer value gets compared to the old pointer value; if they aren't equal, then the active thread walks over to whichever neighbor the new value now points to. If they are equal, the same address-modulo-pointer logic gets repeated, but this time for the *recruit array* – and the thread directs its active walk to whichever *recruit* node falls out of this procedure. This will cause the path of a given thread's walk to alternate between confining itself to a local region of the graph and jumping "down" to a lower rank, recursively plunging all the way down to a rank-0 `Object` and then bouncing back up through the ranks to its originating `Object`.

Regardless of which node the walk has been directed to, the next step is for the thread to *interfere* with that newly current node. If this current node is functioning in register mode and has a value stored in it, then the thread adds to its register

contents the value *a* mod *r* where *a* is the walking node's address and *r* is the current node's rank. It's an act of total caprice that, while deterministic and perfectly "ordered" from the standpoint of the thread doing the twiddling, introduces an element of "disorder" to the thread that recruited the current node.

It's helpful to think of these walks as the way active `Object`s "refresh" or "repaint" themselves. Imagine the black-and-white TVs of yore: an electron gun projects all the objects in an image onto the screen by walking boustrophedonically across the screen. On a given row of a given walk, it might paint a portion of Lassie, a portion of Timmy, and a portion of the well. Now imagine that same TV changing so that it has *many* electron guns, one responsible for painting just Lassie, another for painting just Timmy, a third for painting just the well, and so on; each object in the image gets rendered independently by its own electron gun. This analogy breaks down when we consider the entity onto which the electron guns are painting their image contributions – in the TV case, it's the screen, but in the active `Object` case, it's *all the other electron guns*.

COALESCENCE

The preceding two sections describe how `Object`s behave, but gloss over the most frequent way they come into being — coalescence. To explain this, we need to revisit the earlier discussion of recruitment.

Recall that, once an active `Object`'s breadth-first recursive search has located a suitable recruit, that recruit climbs back up the stack of the calling thread by a number of stack frames equal to its own rank, in order to insert a new edge in the graph between itself and whatever node it can reach that is closest to its recruiter. But it turns out that the recruit is paying attention to the other nodes it traverses along the way.

As the recruit traverses each intermediate `Object`, it incorporates that `Object`'s address into a multiplicative "running total," such that by the end of the stack climb, a single number is produced that can be factored into the addresses of every traversed `Object`. This number, called a *coalescence candidate*, is checked to see if the recruit has encountered it before. If so, nothing happens. But if the recruit perceives the candidate as new, it notifies its neighbors of the discovery. This forces the neighbors, whether active or passive, to recruit another `Object` to store away the newly discovered value; the address of this recruit is kept in an internal *coalescence candidate history array*, or *state array* for short. (*State* in this context is meant in the software engineering sense of *statefulness vs. statelessness.*)

If an `Object` receives a coalescence candidate notification from a neighbor, and finds that candidate already in its own state array, then *it* sends a notification to all *its* neighbors, including the original `Object`. If the original `Object` receives these response notifications in a quantity equal to or greater than $\log_2 n$ where n is its number of neighbors (rounded up), then it *initiates coalescence* — it creates a new `Object` instance, inserts an edge between itself and that instance, assigns it a rank of one more than its own rank, and gives it an address equal to one more than the sum of the addresses of all the `Object`s that participated in this coalescence "broadcast storm," including its own. All the `Object`s involved in this process then remove the original candidate from their state arrays so that a second, duplicately-numbered, `Object` will not be produced.

RUNTIME BEHAVIOR: LOGICAL INTERPRETATION

It can be difficult even for software professionals to visualize the runtime behavior of such a system, but a few existing systems can be invoked as partial metaphors to help sculpt one's mental images. One is Core War, a computational game, popularized by A. K. Dewdney in the early 1980s [1], in which two or more "battle programs" compete for dominance over a shared set of memory. Since programs are made of instructions, which occupy memory addresses, programs can attempt to "kill" others by scanning the shared memory looking for the instructions of their enemies and then overwriting them with a meaningless, destructive pattern. This is reminiscent of how active `Object`s twiddle values stored in other recruited `Object`s' registers (although such twiddling is less severe and presumably not malicious).

Another somewhat analogous system is a genetic programming environment (emulating Darwinian natural selection) that has been modified so programs seek not to replicate their own algorithm, but the algorithms of their competitors. This would fractally (or at least chaotically) intertwine our familiar biological notions of heredity, fitness, symbiosis, and altruism: the morphology of a current generation of replicator would be largely, but not completely, untethered from the emergent morphology of the next. This is reminiscent of how passive `Object`s can notice patterns in recruitment interactions (e.g. random mutations) and turn them into new active `Object`s via coalescence.

Runtime Behavior: Philosophical Interpretation

Computational ontology reverses the formalistic assumption of physics, which seeks first to enumerate the various entities in a system and then determine how they behave. Here we have defined an algorithm for generating increasingly complex entities out of nothing but natural numbers (which can safely be assumed to have existed prior to the Big Bang). In this interpretation of reality, instead of objects exhibiting behaviors, we say that *behavior is literally the substance of which objects are made* — matter and energy both are emergent properties of a vast concordance of "behavioral atoms."

Computational ontology relates to mathematics in a way that may offend the sensibilities of the hardcore "gauge theory chauvinists" in the physics community; it can be thought of as the creamy filling in the middle of a numerical Oreo. One wafer of the Oreo is (a) the set of natural numbers, which serves as an input, and the other is (b) the abstract mathematics regarded by the aforementioned chauvinists as the only proper expression of "real physics," which serves as an output.

To elaborate:

(a) In man-made object-oriented systems, there is a finite quantity known as "the heap," which can be thought of as the raw material from which object instances are manufactured. Every time an object is instantiated, that act of creation is offset by a consumption of heap; too many instances exhaust the heap. In computational ontology, the set of natural numbers serves the role of heap, and since that set is infinite, there is no limit on how many `Objects` can be created.

(b) In man-made computers, most operating systems expose "performance counters" for purposes of measuring how quickly and efficiently specific programs perform. The exact set of available counters depends on the design of the operating system, and the exact values of those counters depends on the design of the program running within it. It is my contention that *the mathematical formalisms preferred by the aforementioned chauvinists can provide only as nuanced a description of the workings of the universe as what performance counters can provide of a complex piece of software*. Fiber bundles and performance counters are both very useful, and not to be knocked by any means; but in both cases what we're really after is the source code.

Runtime Behavior: Physical Interpretation

A reasonable starting point for mapping the computational behavior of the system to the physical behavior of the universe is to ascribe computational meaning to physical primitives like mass, gravity, energy, and time. The next section deals with time; we'll deal with the rest here.

The quantity colloquially known as "mass" is conjectured to be *stack depth*. In any multithreaded programming environment, the number of threads and their total stack depth is a reasonable answer to the somewhat abstract question "how much computation is going on?" If matter is really an emergent property of behavior, then *mass* is an analogously reasonable answer to "how much *behavior* is going on?"

Equating mass with stack depth also aligns neatly with the computational interpretation of gravity. The recruitment process has the effect of increasing the edge density of the universal graph in certain places but not others. `Objects` that "need" each other end up forging more direct paths to each other across the graph. This could be perceived as the warping of space attributed to gravity by Einstein's general relativity. The tendency for high-rank `Objects` to recruit low-rank `Objects`, and consequently remake the universal graph to bring those low-rank `Objects` closer to the high-rank ones, constitutes an attractive behavior like that of gravity.

The most speculative proposal here is that energy is equivalent to *state* — the bigger an `Object`'s state array (see *Coalescence* above), the more energetic it is. The reasoning behind this is that the most energetic particles observed in the wild originate from entities like quasars and supernovae — presumably computationally busy `Objects` indeed. Such a morass of tightly packed graph nodes, all capturing each other's recruitment patterns — and retaining those that never trigger coalescence — would swell the involved `Objects`' state arrays to a much bigger size than in low-energy environments. Such `Objects` are also correspondingly more likely to contribute to coalescence attempts that begin near them, since they have such a large repository of known patterns.

TIME AND CAUSALITY

Suggesting that the universe is made of cellular automata is far from groundbreaking, but it is worth pointing out that, unlike previous models of cellular automata, computational ontology allows for relativistic interpretations of time. Traditional CA approaches to digital physics, such as that offered by Stephen Wolfram [2], still fundamentally assume a single clock governing the interactions of each cell — the n^{th} state of a given cell may depend on the n-1^{th} state of its neighbors, but all cells march forward from the n-1^{th} state to the n^{th} state in synchrony.

Computational ontology suggests that, while all `Object`s view time as a succession of discrete moments (much like Julian Barbour's notion of *durations* [3]), the determination of what constitutes such a moment is made individually, with regard only to their own rank, and without regard to the "clock speeds" of their neighbors. A single "clock tick" is rigorously defined, for active `Object`s, as *the completion of a single thread walk, all the way down to a rank 0 recruit and back*; for passive `Object`s, the definition is *a visitation by such a thread walk*. Quarks have shorter ticks of the clock than atoms, which have shorter ticks than bacteria, which have shorter ticks than hedgehogs, which have shorter ticks than planets, which have shorter ticks than galaxies — note how every `Object` in this series has a greater rank than the previous, and a correspondingly longer active walk length than the previous.

Computational ontology cannot rely on entropy as the solution to the arrow-of-time problem, because the `Object` class is essentially an entropy-conservation machine. (The system responds to an increase in disorder on one level by creating a new level — wherever something looks ordered, simply zooming into it or out from it will allow one to see concealed disorder, and vice versa.) However, *aggregate rank increase* provides a reasonable replacement. Order and disorder may oscillate into and through each other endlessly, but maximum rank increases monotonically, corresponding to time's forward march.

Coalescence allows for a conception of causality in which the driving force behind a phenomenon seems to shift from a "bottom-up" causal pressure to a "top-down" causal pressure. For example, a gust of wind might begin because a preponderance of air molecules in a given locality all happen to start moving in the same direction, but then once that inadvertently coordinated motion begins, it actively attracts contributions from neighboring air molecules that, in the absence of such a gust, would have continued doing their own thing. At some point the causal pressure shifts from *a behavioral pattern among instances on a lower level* ("pushing" a relatively small number of air molecules in the present toward the gust in the future) to *an isomorphic instantiation on a higher level* (the inchoate gust gaining a foothold in the present and "pulling" a much greater number of air molecules to join it in the future).

THE ONTOLOGICAL INERTIA THRESHOLD

At very low ranks, the behavior of the universal graph conjures to mind Wheeler's "quantum foam," a complex turbulent storm-tossed sea. At these ranks `Object`s coalesce and then very quickly deactivate, corresponding to the short-lived, self-annihilating "virtual particles" thought to exist at the Planck scale. This turbulence is endless because these `Object`s lack ontological inertia, or *measurably long life as a single coherent pattern of objecthood*.

It seems reasonable that `Object` lifetime has a roughly exponential relationship to rank; the ontological inertia threshold is simply the "elbow"/tipping point in that exponential curve. (I make no guess as to the threshold's rank value.) `Object`s with a rank below the threshold appear just long enough to bleed into other such inertialess `Object`s, and behave very much as the eponymous "vibrating strings" of string theory are qualitatively imagined to. Ranks at or near this threshold coalesce into `Object`s that we would recognize as some kind of actual particle (although perhaps as yet still undiscovered).

COSMOLOGICAL EVIDENCE

Here are some reasons even for the gauge theory chauvinists not to dismiss computational ontology out of hand:

- **Dark matter**. Since mass is stack depth, and stack depth is a characteristic of threads, and threads originate from active `Object`s but can traverse passive ones, all `Object`s contribute to stack depth, even if only some originate it. This could be perceived as a distinction between two kinds of matter, with active `Object`s playing the role of baryonic matter, and passive `Object`s playing the role of dark matter. It seems likely that the proportion of active to inactive `Object`s in the universal graph would be close to the observed proportion of baryonic to dark matter in the universe, although this may not always have been the case (see next point).

- **Cosmic inflation without dark energy**. The unexpected acceleration of the expansion of the universe could be explained by low-rank and high-rank `Object`s taking turns increasing exponentially. Each additional `Object`, regardless of type, represents a very slight expansion of space. It seems reasonable that an `Object`-based universe would experience an S-shaped growth curve — the infant universe experienced explosive growth, as there weren't yet enough low-rank `Object`s to support mass-scale coalescence above the ontological inertia threshold, leaving copying as the dominant `Object` creation mechanism. Later, the S-curve inflected as copying slowed down and coalescence sped up. The ramp-up of coalescence drove a jump in the number of high-rank `Object`s, which increased demand for the lower-rank `Object`s provided by copying, and the curve again steepened.
- **Baryon asymmetry**. Network-savvy computer professionals may note that the graph proposed here has no central numbering authority. The mechanisms for assigning addresses to `Object` instances provide a reasonably high degree of local uniqueness, but have no hope of providing global uniqueness. It is possible that two identically-numbered `Object`s could interact with each other, although this presumably would not be the usual case. When such interactions do occur, it is possible that the two identically-numbered `Object`s merge into one — one of them transfers all its graph neighbors to the other, and then winks out of existence. This could explain not only the phenomenon of matter/antimatter annihilation, but also why there is so much more matter than antimatter in the universe. Computational ontology suggests that the proper distinction isn't matter vs. antimatter, but *uniquely-numbered matter vs. identically-numbered matter*. Even that distinction subtly misses the mark, since identically-numbered `Object`s can peacefully coexist in the graph as long as they never develop an edge connecting them; perhaps a better label would be "matter that thinks it is numbered uniquely" vs. "matter that has just discovered it isn't numbered uniquely."
- **Black holes**. Computational ontology would explain black holes as active `Object` instances whose threads got caught in a recursive infinite loop. If mass is actually stack space, then an infinitely recursing thread would appear to other `Object`s as an entity of infinite mass. Mathematical similarities have already led several to conjecture that black holes are actually a type of particle that, while ordinarily subatomic, has been "inflated" in some way to a cosmic scale. Infinite stack space accumulation could be the force driving such "inflation."
- **Quantum vacuum zero-point energy**. Since "energy" is actually *state array size*, and even passive `Object`s in a sparsely connected graph locality (such as deep space) have nonempty state arrays, they would have correspondingly nonzero energy levels.
- **Hierarchy problem**. Since gravity is an innate property of the universal graph, it is active at all ranks, whereas the strong and electroweak forces cannot manifest below the ontological inertia threshold. It seems reasonable that `Object`s above the ontological inertia threshold would perceive these forces as being stronger (due to their "nearness" in rank) than the more "rank-distant" gravity, whose effects are "stuck way down" at rank 0.

CONCLUSION

Although this paper draws on elements of physics, computer science, and mathematics, it does not attempt to definitively answer any questions or solve any problems in any of those areas. It is meant to encourage professional academics with an interest in digital physics to broaden their conception of computation beyond the binary von Neumann model, and provides a reasonably thorough example of how to begin.

REFERENCES

[0] John A. Wheeler, 1990, "Information, physics, quantum: The search for links" in W. Zurek (ed.) *Complexity, Entropy, and the Physics of Information*, Addison-Wesley

[1] A. K. Dewdney, 1984, "Core War Guidelines," *Scientific American*

[2] Stephen Wolfram, 2002, *A New Kind of Science*, Wolfram Media

[3] Julian Barbour, 2008, "The Nature of Time," Foundational Questions Institute