Breaking the Box: Simulated Protein Computing

Christopher N. Eichelberger and Mirsad Hadzikadic

UNC Charlotte, Charlotte, NC 28223 USA
christopher@uncc.edu

Abstract. Computers since the 1940s have shared the same basic architecture described by Turing and von Neumann, in which one central processor has access to one contiguous block of main memory. This architecture is challenged by modern applications that require greater parallelism, distribution, coordination, and complexity. Here we show that a model of protein interactions can serve as a new architecture, performing useful calculations in a way that provides for much greater scalability, flexibility, adaptation, and power than does the traditional von Neumann architecture. We found that even this simple simulation of protein interactions is universal, being able to replicate the calculation performed on a digital computer, yet without relying upon a central processor or main memory. We anticipate that the convergence of information- and life-sciences is poised to deliver a platform that invigorates computing as it provides insight into the complexity of living systems.

Keywords: simulation, protein, parallel, distributed, complex adaptive system, architecture.

1 Introduction

For years, our computers have relied on the von Neumann architecture [40], the endless repetition of "fetch and execute." The advantage of this serial design is that it is deterministic: The process is repeatable and predictable. Increases in performance have arisen from two main sources: increasing density on integrated circuits [29, 18], and distributing computing tasks across multiple processors [4]. Unfortunately, we are beginning to approach some hard limits with respect to circuit density, and adapting software to take advantage of multiple processors remains very difficult [17]. One approach to overcoming these limitations is to pursue different architectures. Biologically-inspired computing is one such alternative.

Life processes occur with frantic parallelism that is more widely distributed than our silicon-based computing, because there is no dependence upon a single core memory. This distribution comes at a cost: By working outside of the innately serial von Neumann architecture, chemical and biological systems give up strict determinism, and enjoy repeatability and predictability only probabilistically. Furthermore, the mechanics of neither cellular biology nor proteomics are sufficiently well understood to allow us to craft a large-scale, general purpose

J. Suzuki and T. Nakano (Eds.): BIONETICS 2010, LNICST 87, pp. 452–468, 2012.

computing machine that takes advantage of all of the parallelism implicit within single-celled organisms.

Previous work has focused on two fronts: approaching information science from within the lab; and incorporating laboratory science ideas into programming systems.

1.1 Approaching Information Science from within the Lab

Chemically-inspired computers, such as BZ machines, use a central oscillator as a synchronization method for molecular computations [1, 5]. Unforunately, the central oscillator tends to constrain the speed of the entire system (to approximately 10 cycles per second [10]), negating many of the advantages of rapidly reacting chemical species. In sum, the billions of interacting molecules are doing significantly less work than they are capable of doing, because of the way they are used in aggregate. This architecture may provide benefits in terms of being applicable in settings where silicon processing not well suited, but it does not currently appear to be a viable candidate to provide greater computing throughput.

DNA computing, in contrast, originally relied upon DNA molecules as inert data elements operated upon by lab protocol qua software [3, 14], principally taking advantage of the ability to explore a huge number of combinations of data solutions simultaneously, albeit in a very manualy-driven process. Since then, there have been projects that have used DNA in a more active manner, allowing it to participate in chemical reactions that produce behaviors that are recognizable as logical functions [35, 11, 34, 37]. MAYA-II was a system, built from more than 100 different DNA gates, that could play tic-tac-toe [23]. The team has extended this work, creating a simulation tool that helps to design and debug these networks of biological circuits [25]; this is important, because it reflects the impact of emergent complexity on even relatively small bio-chemical computers.

In vivo computing involves creating information-processing units out of chemical species that occur naturally within organisms (though not in the precise forms, configurations, or concentrations used). The goal is to use these live computing units to influence one or more of the processes within the organism. This function could include, for example, disease diagnosis, treatment, and drug delivery [2, 24]. Not only does emergence continue to play an important role in *in vivo* computing, but it is arguable that its role becomes paramount, as unintended side-effects of a computation carried on inside a living organism could be disasterous.

Where laboratory-based models excel is in making performing simple computations in settings that are not accessible to traditional processors. What they lack is an effective way to model the unintended consequences of introducing interacting chemical species into a complex environment.

1.2 Incorporating Laboratory Science Ideas into Programming Systems

One of the first significant proposals for couching computation in terms of chemical reactions was Banâtre and Le Métayer's Γ (alternately, GAMMA) [6, 7, 8, 9]. Γ portrays programming as a series of multiset transformations, in which the resident species are both data and rules. While Γ is parallelizable and probabilistic (in terms of which reactions are run on what data elements), there is one important accommodation made for halting: Each rule that fires is consumed as it runs. It is also important that reactions and data species do not support wild-cards, meaning that all inputs and rules must be enumerated explicitly. From early on, Banâtre and Le Métayer provided plenty of example programs demonstrating how Γ could be used to solve general computing tasks such as identifying an extreme value in a collection.

Following Γ came the chemical abstract machine, or CHAM [12, 13]. CHAM describes a language derived from Γ, but one that is treated in in much greater, and more formal, detail. Whereas Γ served to highlight the utility of chemistry as a computing metaphor, CHAM highlights expresses the expectations and bounds on the formal language of one chemistry-inspired computing approach. CHAM was extended to include membranes, a mechanism that is used to provide for the isolation and localization of computations. Membrane computing is a CHAM concept.

Giavitto and Michel's MGS — *(encore) un Modèle Géneral de Simulation (de système dynamique)* — superclasses both Γ and CHAM (along with cellular automata, Lindenmayer systems, and Paun systems) [19]. Though it is weakly typed, MGS is a functional language that has support for a number of programmer-friendly constructs such as sets, sequences, records, and arrays. In contrast to Γ and CHAM, though, MGS allows transformation rules to include wild-cards. The rules have such a rich syntax, in fact, that they represent a rather wide departure from real chemistry: Rather than having simply $A + B \rightarrow C$, MGS allows A and B to inspect each other, evaluate independent expressions, and incorporate evaluations into C. One consequence of this flexibility is that the number of chemical species that MGS must track can become astronomically large, depending on the program being run: An implementation of a 100-city TSP problem, for example, would likely exhaust the resources of the local machine. As an additional aid to the programmer, but what is arguably another departure from verisimilitude, MGS allows the coder to specify ordered execution within a rule via statement priority. MGS is a programming environment available for public download, but it is constrained to operate on only one computer at a time; there is no cluster-aware version of MGS available.

COPASI — *CO*mplex *PA*thway *SI*mulator — is a joint project of three universities, and is designed to perform stoichiometric analysis and simulation [22, 31]. That is, given a set of chemical reactions and a starting set of reagent concentrations, COPASI has multiple methods of predicting how the solution will change over time, from ordinary differential equations to stochastic simulations based on Gillespie's earlier work. COPASI is not a programming tool *per se* so much

as it is a tool for (bio)chemists, but it is one of the tools that some of the other research projects employ while working on chemistry-inspired computing.

Matsumaru *et al.*, for example, have used both MGS and COPASI to explore chemical organization theory [26, 27, 28]. Their work centers on how to create standard computer science constructs — such as flip-flops and oscillators — using simulated chemical reactions, and how to use graph theory on the stoichiometric description of a system to help bridge the micro-level behaviors with the macro-level outcomes. Like most of the methods within this family, chemical organization theory is not constrained by conservation of mass; in fact, violating this conservation is key to the success of the oscillator they create, as they rely upon a constant influx of new reagents to drive the oscillator. Matsumaru and colleagues echo Müller-Schloer's concerns about organic computing: Emergence is a key property of chemical systems (and simulations), but the bottom-up approach to programming is difficult to program (and to control) effectively [30].

Where chemically- and biologically-inspired computing systems excel is in exploring new methods of parallelization as wellas providing platforms for modeling and controlling complexity and emergence as they can be exhibited in real systems. What they lack is suitable verisimilitude to real molecules (and any continuous path to improve this) to allow their results to be more generally applicable to either massively-parallel programming systems or chemically-embedded systems.

1.3 Simulated Protein Computing

The model presented here is meant to be a hybrid approach between computing-inspired laboratory methods and lab-inspired computing methods. Its purpose is to help explore, simultaneously, and in a breadth-first manner, the following:

1. How might an abstract model of chemical interactions be used as the basis for a new computing architecture? Given that silicon-based models of molecular interactions are bound to be crude for now, how can we abstract the chemical model so that it can easily be upgraded over time?
2. How might we develop a (relatively) inexpensive, software-based simulation that provides the opportunity to explore, quantify, control, and re-use complexity in bottom-up systems such as we find in real, living cells?

Simulated protein computing has no more to do with real proteins [15] — at least for now — than a genetic algorithm has to do with real DNA molecules [21]. When we speak of crafting protein programs, we are still talking about writing text files that the computer will interpret and execute. Our intent, though, is not merely to drape clever coding tricks in superficial biological metaphor, but is to begin to explore the real capabilities and the real limitations that computer programs will exhibit when they are expressed as protein molecules. (See Table 1 for a comparison of traditional computing with simulated protein computing.) Neither computer science nor proteomics is yet ready for this convergence, but our experience suggests that the conjunction will be profitable to both disciplines.

Table 1. Contrasting traditional computing and protein computing

traditional computing	simulated protein computing
Advantages: – determinism – global memory: there is only one authoritative value for each variable – familiarity, and the amount of investment in the current architecture – simplicity and directness: this method is well suited for writing operating systems and word processors	Advantages: – distributed memory: data proteins are scattered across the simulation, providing for concurrent access – parallelism: functional proteins are independent enzymes, all copies of which can execute simultaneously – distribution: coding enzymes can be introduced to any location, and can diffuse to new locations – localization: proteins can have become differentially concentrated across locations, providing for location-specific computing – separability: each function is an independent particle, so computation is innately separable – emergence: differential computation supports experimentation with self-modifying approaches to solving difficult problems, such as artificial intelligence or drug design and delivery – hybridization: protein computing is inspired by, and can inform, both information science and life science
Disadvantages: – serial execution – difficulty scaling: concurrency must be handled by the programmer – inseparability: not all problems are equally separable	Disadvantages: – non-determinism – distributed memory: variable values can only be established by assay – novelty: writing code for this architecture requires a different mindset

2 Method

Programming within a living organism will mean fracturing the building blocks with which we build software. It is tempting to think of a cell as if it were one processing unit, but this would be a mistake. Though the nucleus may help to direct cellular activities, these processes occur throughout the cell, albeit in specialized forms depending on the location, and most often through the interaction of proteins. Our simulation creates a virtual cell as a collection of small, uniform volumes within which independent logical proteins interact. In a real cell, each of these spaces would contain the proteins that react; in our

simulation, each compartment maintains its own estimate of the proteins that may be present, treating each compartment as if it were a tiny reaction vessel.

There are two types of biological molecules that the simulation supports: inert, structural proteins, that are analogues to plain data in traditional programming, such as the integer 5; and enzymes, that can bind to other species, and interact with them, that are analogues to subroutines in traditional functional programming languages, such as "IF(condition, true-result, false-result)".

Each new architecture entails new assumptions. Approaching programs as if they were proteins means assuming that many of the standard tools of digital computing are no longer available. There is, for instance, no longer a CPU. Every enzyme (active protein) is its own processor, working at the same time as all others, but without any knowledge of them except through their influence on the local, shared environment. There is also no main memory. The variable "X" no longer is a unique location in memory, but may exist in thousands of copies — each with its own value — across multiple locations. This means that though performing individual calculations may be very fast, determining an consensus output value may be very time-consuming, requiring an assay to establish the distribution of values. In traditional programming, the code directly manipulates a variable's value; in simulated protein computing, the code shapes a variable's probability density function. Arguments are no longer passed into functions, but functions have binding sites into which available proteins of the right shape and pattern may bind when needed. Subroutines are written as substrates for computing. This means that there are times when an enzyme gets a chance to become active, but cannot do so, because no suitable inputs are available to satisfy the binding sites. Lastly, intermediate computations inside of any subroutine (enzyme) no longer matter. The only state changes the system recognizes involve: denaturing a protein, thereby removing it from the local environment; assembling a new particle (or new conformation) from one or more existing particles, and introducing it into the local environment; and moving chemical species from one location to another (diffusion). See Table 1 for a comparison of these two architectures.

The main event loop in this type of system changes from the von Neumann architecture's "fetch-execute" to one in which every cellular compartment does the following every time step: the compartment accepts species that infuse from neighbouring locations; generates a sample of proteins present at the physical location being simulated (from the probabilistic profile of the proteins that may be present); allows each protein the opportunity, if it has a functional form, to bind and react with the other proteins present in the sample window; takes the resulting list of proteins created, destroyed, and modified in the previous step, and updates the reaction vessel's profile of proteins likely present; computes gradients against its neighbours, and prepares a list of diffused proteins to export to each neighbour at the beginning of the next time step.

Proteins are large molecules, but small processors, so protein programs look very much like low-level routines in which every smallest step must be represented explicitly. Our simulated enzymatic programs assume that there are

functional motifs — blocks of amino acid residues — that serve as the equivalent to machine instructions. These motifs, treated as if they were indivisible units, are assembled into simple tree-shaped programs that resemble to abstract syntax trees for a functional language. (See Figure 1 and Figure 3 for example programs used in the experiments described later in this work.) Every motif, when it is evaluated, returns a single value; these values are passed up the tree, reaching the head where the final value is discarded, because — as pointed out earlier — all intermediate results are meaningless once they have been used. The only motifs that change the environment are EMIT, responsible for introducing a new protein into the local environment, and DENATURE, which is responsible for removing an existing protein from the local environment. Table 2 shows the same function implemented twice, once in traditional pseudo-code and once in the form that might be used in this biologically inspired architecture.

Table 2. Contrasting two implementations of the same function

traditional pseudo-code	simulated protein computing

`function get_minimum(A, B)`
`{`
`  if (A < B) return A;`
`  return B;`
`}`
	`(if`
	`  (and`
	`  (exists (match value (.*)))`
	`  (exists (match value (.*)))`
	`  )`
	`  (if`
	`  lt($1, $2)`
	`  (complex (emit $1) (denature $2))`
	`  (complex (emit $2) (denature $1))`
	`  )`
	`)`

One important difference between these two code samples is that the traditional version creates a new value that is a copy of whichever input parameter represents a lesser value; the protein version binds two values from the local environment, and replaces the greater value into a copy of the lesser. Whereas the former method creates a single, definitive answer, the latter method merely alters the distribution of values in the local environment. Another important difference is that the traditional version will always return a value when it is invoked; the protein version can only run when its local environment includes two VALUE proteins that can bind into its activation regions. This means that protein programs have reaction rates that are influenced by the concentration of other proteins nearby. This is a concern (and an opportunity) that programmers working in real biological systems will have, but that traditional programmers will not have.

To implement the method fully requires substantially more information about how the system is defined, constrained, and run, including: the language model,

with both syntax and operators; the modular abstraction of chemistry, including which reactions are allowed and how to compute chemical gradients; the (lossy) data compression used to handle the large volume of data about what protein species may occupy a given reaction vessel; the encoding that allows the secondary conformation of the protein programs, their tree shape, to be inferred from their primary conformation; the macro facility that allows certain proteins to be stored without loss of fidelity; the geometry of simulated cells and their constituent compartments; the extensible monitoring that allows us to probe any compartment; the graphical display of results as the simulation progresses; the XML pre-processor that simplifies writing source code for this platform; etc. These details exceed the scope of the current argument, and so are omitted.

3 Experiments and Results

Foregoing the programmers' canonical, "Hello, world!", this paper focuses on three separate experiments: two NAND experiments and a decomposition experiment.

The logical NOT AND (NAND) function compares two boolean values: If both inputs are TRUE, the result of the function is FALSE; otherwise, the function returns TRUE. To explore correctness, we present two versions of NAND. The first operates directly on raw numbers, and demonstrates the accuracy of the computation. The second version operates on labeled complexes, and demonstrates how networks of cascading NANDs can be executed reliably by a simulated protein computer. Jointly, the NAND experiments are important, because any universal binary computer can be simulated using nothing more than NAND functions.

The last experiment is a simple decomposition reaction that we use to explore the performance implications of protein programming.

3.1 Unlabelled NAND and Correctness

Assume that the simulation is roughly analogous to a flask containing various reagents. In the unlabelled NAND case, the flask contains only these reagents in equal proportions: integer value 0, an inert data species; integer value 1, an inert data species; and NAND, an enzyme that can bind to two inert data species. Once it binds to two data elements successfully, it then (and only then) applies the NAND function to its two bound inputs, and transforms one of them (selected randomly) from its bound input value to the result of the function. Once the evaluation is complete, the two bound species — one as given, and one transformed to the function result — are released back into the environment. Figure 1 illustrates this program.

The total number of particles in the system is fixed. (While the system does not enforce conservation of mass, it is recommended.) The only change in the system over time is the relative proportion of zeros and ones as the data proteins are transformed by the enzyme. Figure 2 is a scatter plot that displays the sampled concentrations of ones and zeros in the simulation over time.

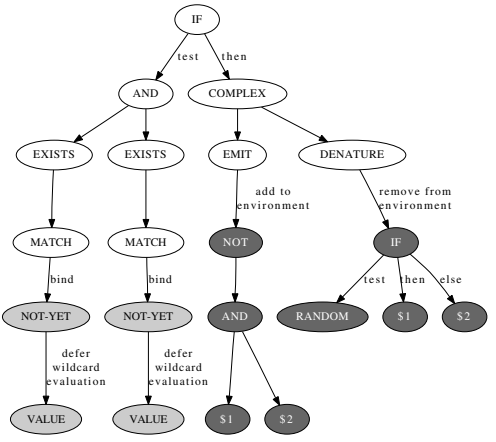

Fig. 1. Unlabeled NAND

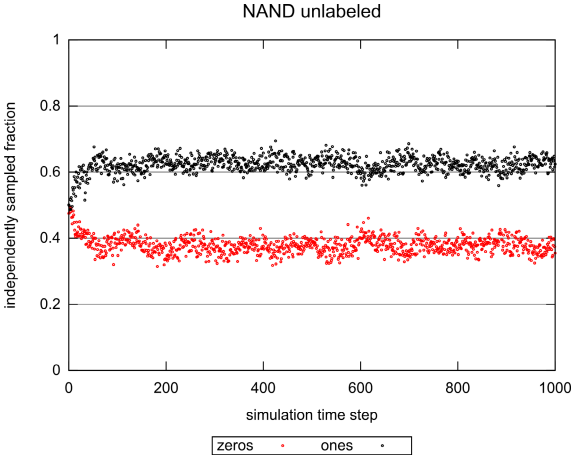

Fig. 2. Unlabeled NAND output. The black points represent the sampled concentration of ones; the red points represent the sampled fraction of zeros.

Within approximately 100 simulated time steps, the system reaches a rough equilibrium, in which the increase in the concentration of ones is offset by the increasing likelihood of the NAND function to return 0 in an environment dominated by ones. Assuming that we adopt the convention that the concentration of integer ones is [#1], and the concentration of integer zeros is [#0], we can express this equilibrium in its algebraic form as:

$$\frac{d[\#1]}{dt} = -\frac{d[\#0]}{dt} = -[\#1]^2 + [\#1][\#0] + [\#0]^2 \tag{1}$$

Given the additional constraint that, because we are preserving mass, the total number of integers (data proteins) remains fixed, the system becomes solvable:

$$[\#0] = \frac{3-\sqrt{5}}{2} \approx 0.38, \quad [\#1] = \frac{\sqrt{5}-1}{2} \approx 0.62 \tag{2}$$

The algebraic solution to the system matches the equilibrium on which the simulation fairly quickly settles, suggesting that the sampling and computation are being performed correctly.

3.2 Labelled NAND and Completeness

The initial test was artificially simple: There are very few useful applications that consist of a single, isolated calculation. It is more important to investigate whether serial computations can be performed reliably. To explore whether biologically-inspired computing can satisfy this requirement, we introduced labeled complexes (akin to tagging biological chemicals) into the system.

This second test uses labels to identify each piece of data as specific to one stage in a multi-stage computation. Each stage is tagged with a different label, and the stages together constitute a network of cascading NANDs. Being able to construct and coherently run such networks is relevant to the reach of the computing system, because NAND networks are sufficient to emulate any other function on a universal computer.

The reaction vessel is initialized with these species: complex (A,0), an inert data species, in which "A" is the label, and "0" is the value; complex (A,1), an inert data species, in which "A" is the label, and "1" is the value; NAND(A,A) $\rightarrow$ B, a function that binds to two integer values that share the "A" label, computes the NOT AND result on these two operands, and then converts one of these bound inputs to the output value, changing its label from "A" to "B" (see Figure 3); NAND(B,B) $\rightarrow$ C, similar to the function described previously, but using different labels.

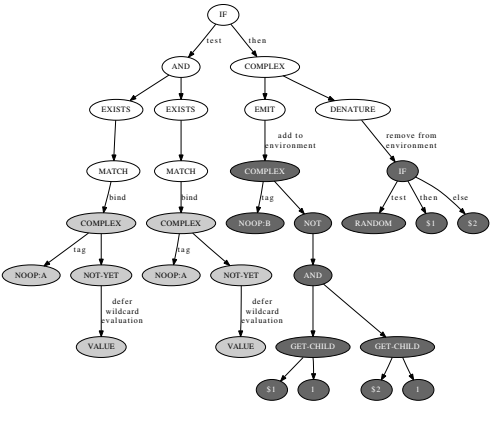

Fig. 3. Labeled NAND

Because we have already established that a single NAND performs as expected, we monitored only the total number of data elements that were labeled for each stage in the computing chain: A, B, and C. The results appear in the scatter plot in Figure 4.

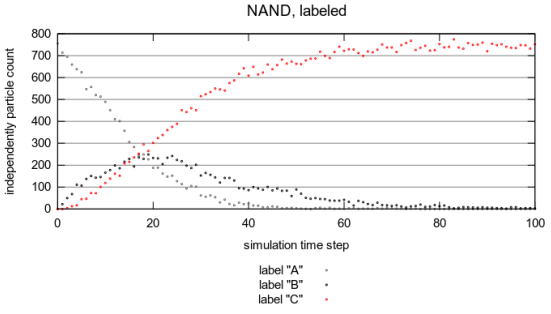

Fig. 4. Labeled NAND output. These are the particle counts, determined by assay, from the labeled NAND at each time step. The light gray points are the concentration of data element A; the black points are the concentration of data element B; and the red points are the concentration of data element C.

Note that the unreplenished inputs, labeled A, diminish over time as expected. Species B begins at a concentration of zero — because none were introduced into the reaction vessel, but must arise as a result of NAND(A,A) producing them — and increases. Species B, however, grows more slowly than species A decreases, because NAND(B,B) is consuming B to produce species C. These progressions validate an important property: the nominally serial computation is honouring the serial dependencies as expected, even while the reactions themselves are occurring in parallel. Additionally, Figure 5 shows that the simulated interactions of proteins are behaving as would be predicted by the differential equations that we would expect using Runge Kutta 4:

$$\frac{dA}{dt} = -k_A \cdot A, \quad \frac{dB}{dt} = -\frac{dA}{dt} - k_B \cdot B, \quad \frac{dC}{dt} = k_B \cdot B \tag{3}$$

As in the unlabelled case, the labeled NAND has produced behaviours that are consistent with what would have been predicted of a real, wet-lab system.

3.3 Decomposition and Performance

If protein computing is ever to be useful, it ought to provide performance advantages over traditional computing. To evaluate the expected performance increase of wet-lab computing over simulated protein computing, we focused on a simple decomposition reaction: E + S → ES → E + P. “E” is the enzyme catalyst; “S” is the substrate; and “P” is the product. As a simulated protein program, E binds to S, emits P back into the environment, and denatures S. The activity

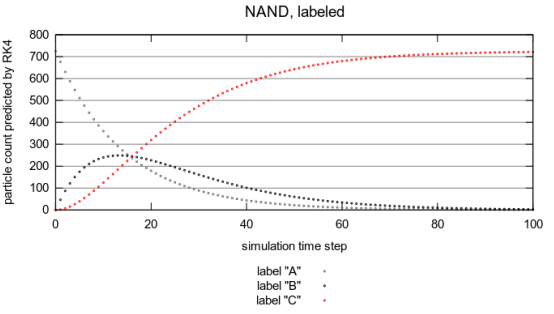

Fig. 5. Predicted output from the labeled NAND. Given the equations as written in (3), these are the curves that Runge Kutta 4 would predict. The light gray points are the concentration of data element A; the black points are the concentration of data element B; and the red points are the concentration of data element C.

of a real, yet fairly simple, enzyme can be characterized as a function of the amount of substrate present. We did this for the artificial case, fixing the enzyme at 1000 particles, and allowing the amount of substrate to range from 0 to twice the amount of enzyme present; each test case was replicated 20 times. The resulting average activity is plotted in Figure 6; the double-reciprocal of these data appear in Figure 7.

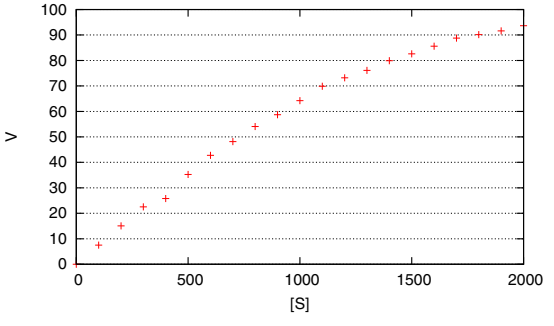

Fig. 6. The Michaelis-Menten plot of enzyme velocity (activity), V, versus substrate concentration, [S]

This double-reciprocal plot is only interesting, because the best-fit line has an R^2 value of 0.998, and a good linear fit of data on this plot happens to be typical of simple real-world enzymes. It is also simple, if nave, to estimate the Michalis-Menten constants from this plot using Equation (4), concluding that $K_m \approx 4111$ particles, and $v_{max} \approx 318$ activations per second.

$$\frac{1}{v} = \frac{K_m}{v_{max}} \cdot \frac{1}{x} + \frac{1}{v_{max}} \tag{4}$$

Inspecting Figure 6 suggests that the analysis is off, because the curve appears to be nearing a plateau much faster than the expected v_{max} of 318 would imply;

furthermore, since the K_m maps to the concentration of substrate at which the enzyme performs at half its maximum velocity, it seems unlikely that a K_m of greater than 4000 is warranted.

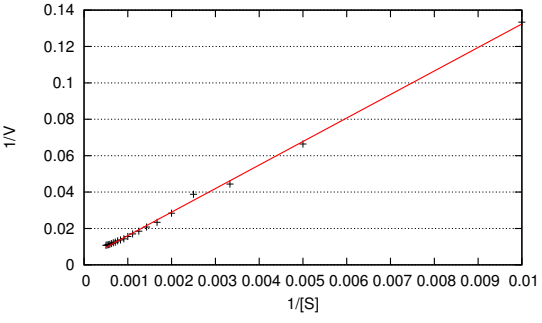

Fig. 7. The Lineweaver-Burk double-reciprocal plot of enzyme velocity (activity), 1/V, versus substrate concentration, 1/[S]

To estimate the performance of this simple decomposition if it were conducted entirely through real proteins rather than through simulation, temporarily assume that the complexity of the simulated deconstruction enzyme, E, is roughly comparable to the complexity of acetylcholinesterase (AChE). This likeness is motivated by the fact that AChE also decomposes its substrate (acetylcholine); in the experiments conducted here, E also performs only a single decomposition. Because the simulation used 1000 enzyme particles, the theoretical maximum number of times that any single E enzyme could activate per second is 0.318; AChE can react approximately 12,500 per second [20]. The implication is that real AChE outperforms the simulated E by a factor of roughly 39,000. Given the overhead of the simulation, that performance increase does not appear to be substantial, but the cursory analysis is misleading.

There are many difficulties in comparing the simulated decomposition enzyme to real AChE: there is no evidence to support the supposition that AChE's enzymatic efficiency is anywhere near that of E; there is, in fact, no basis for establishing an enzymatic efficiency for E at all; the mechanics of the artificial protein chemistry are so simple that there is almost no way to draw realistic comparisons to real proteins. Despite these difficulties, it is possible to appreciate better the increase in performance that protein computing represents by conducting a simple thought-experiment: One drop of a 0.1 mM solution of AChE, provided with a surfeit of substrate, would produce approximately 3.75×10^19 reactions in a single second. The Jaguar cluster at Oak Ridge National Laboratories is a a super-computer that, as of December 31, 2009, was at the top of the list of high-performance computers in the world [38]. Jaguar contains just over 2.7 million computing cores [32], each no more than 100 times more powerful than the desktop machine on which these tests were run. If this simulation were written so that it could occupy 100% of the Jaguar's resources, and there were

(magically) no inefficiencies introduced from the parallelization, it would still take more than 13 years to complete the equivalent computation performed by that one drop of AChE in one second.

Clearly, this is the kind of spurious extrapolation that invites ridicule, because it is clear that a considerable portion of the chemical reactions taking place represent redundant work. The obvious question to ask is: How much of the work being done is useful? Think of the computations in terms of breadth and depth, where breadth represents similar reactions occuring at an early stage in a much longer chain of computations, and depth represents progress down any single (potentially very deep) computational chain. If, out of one million enzymes, 90% are executing the equivalent of instruction #1, then they represent the breadth of computing; the 10% that are executing the equivalent of some later calculation in the larger effort represent the depth of the computing that is taking place. If one assumes that each unit time results in some (average) fraction, p, of all enzymes reacting, then the expected distribution of protein reactions will peak at $\lfloor p \cdot t \rfloor$, where t is the number of time units that have transpired. This means that the depth of computation increases linearly with the time elapsed, allowing one to conclude that — even when much of the computation in a protein system is duplicative, and seemingly wasteful — any single computational path is proceeding forward very rapidly.

This exaggerated extrapolation of expected throughput also serves another real purpose: It highlights two important properties of protein computing, both of which motivate continuing this research to find better ways to learn how to program within this paradigm:

1. Compactness: Real proteins, whether they are enzymes or inert data, can fill a small volume with large numbers, providing both for fast execution as well as significant exploration. Silicon computing, in contrast, occurs in only two dimensions.
2. Frictionless scaling: Given adequate substrate, two drops of enzyme solution will yield twice the product that one yields, because the activation of each molecule is entirely independent. Traditional computing, in contrast, imposes an increasing communication inefficiency as the number of computing units increases.

The core opportunity that real protein computing represents is to take advantage of the three-dimensional density of real containers in a way that is subject to fewer diminishing returns. The core opportunity that simulated protein computing represents is a way to explore the vagaries of programming in an entirely new way.

3.4 Conclusions

The two successful NAND experiments provide evidence that the simulated protein computing method described here is functionally complete, meaning that it can compute any binary-valued function [33]. This property makes it possible for protein computing to reproduce any function that is expressed within

a modern silicon processor. Because we assume that our traditional computing devices are universal, this same assumption now extends to include the model of protein computing presented here [39], [37], [16], even though traditional notions of recursive enumeration do not appear to apply to protein programs.

Of greater impact than its universality is the system's ability to scale up. Breaking calculations into pieces that can be sent to multiple computers in parallel is difficult, time-consuming, and prone to introduce error. Programming in the simulated protein environment presents a learning curve, because it is very different than traditional programming, yet all of the programs that get written can immediately be run across an arbitrarily large number of processors. Simulating a non-von Neumann architecture on a traditional computer is not particularly efficient, but it allows us to become familiar with the capabilities and pitfalls – including emergent properties arising from this bottom-up approach – of such a programming method until such time as a true in vivo implementation is available when it is conceivable that a solution of proteins — planned and refined in a software simulation — is introduced to a colony of generic cells; as the solution washes across the millions of living organisms, the foreign proteins invoke a chain of responses that culminate in an assay that provides the distribution of problem results. In such a system, it is not merely each of the millions of cells that is a processor, but each of the cells is a collection of millions of processing units, freeing us at last from the fetch-execute bottle-neck [36].

Notes and Comments. The germ from which this project grew — "What would it look like if we could use proteins to write programs?" — began as a project jointly conceived with Dr. Kayvan Najarian, now at Virginia Commonwealth University.

Dr. Seok-Won Lee of UNC Charlotte was instrumental in reviewing early drafts of this text. Dr. William Tolone and Dr. Zbigniew Ras also served as reviewers.

Full source code for this project (licensed under the GPLv3) is available via http://www.simulatedproteincomputing.org.

References

1. Adamatzky, A., De Lacy Costello, B.: Experimental logical gates in a reaction-diffusion medium: The XOR gate and beyond. Phys. Rev. E 66(4), 46112 (2002)
2. Adar, R., Benenson, Y., Linshiz, G., Rosner, A., Tishby, N., Shapiro, E.: Stochastic computing with biomolecular automata. Proceedings of the National Academy of Sciences 101(27), 9960–9965 (2004)
3. Adleman, L.M.: Molecular computation of solutions to combinatorial problems. Science 266(11), 1021–1024 (1994)
4. Anderson, D.P.: Public computing: Reconnecting people to science. In: Conference on Shared Knowledge and the Web, Residencia de Estudiantes (2003)
5. Bull, L., Adamatzky, A., De Lacy Costello, B.: On polymorphic logical gates in sub-excitable chemical medium (June 2010)

6. Banâtre, J.-P., Le Métayer, D.: A parallel machine for multiset transformation and its programming style. Future Generation Computer Systems 4(2), 133–144 (1988)
7. Banâtre, J.-P., Le Métayer, D.: The GAMMA model and its discipline of programming. Sci. Comput. Program. 15(1), 55–77 (1990)
8. Banâtre, J.-P., Le Métayer, D.: Programming by multiset transformation. Commun. ACM 36(1), 98–111 (1993)
9. Banâtre, J-P., Priol, T.: Chemical programming of future serviceoriented architectures
10. Bánsági, T., Leda, M., Toiya, M., Zhabotinsky, A.M., Epstein, I.R.: High-frequency oscillations in the Belousov-Zhabotinsky reaction. The Journal of Physical Chemistry A 113(19), 5644–5648 (2009)
11. Benenson, Y., Adar, R., Paz-Elizur, T., Livneh, Z., Shapiro, E.: Dna molecule provides a computing machine with both data and fuel. Proc. Natl. Acad. Sci. U.S.A. 100(5), 2191–2196 (2003)
12. Berry, G., Boudol, G.: The chemical abstract machine. Theor. Comput. Sci. 96(1), 217–248 (1992)
13. Boudol, G.: A Generic Membrane Model, Global Computing Workshop, Rovereto. In: Priami, C., Quaglia, P. (eds.) GC 2004. LNCS, vol. 3267, pp. 208–222. Springer, Heidelberg (2005)
14. Braich, R.S., Chelyapov, N., Johnson, C., Rothemund, P.W.K., Adleman, L.: Solution of a 20-variable 3-sat problem on a dna computer. Science 296(5567), 499–502 (2002)
15. Creighton, T.E.: Proteins: Structures and Molecular Properties. W. H. Freeman and Company (1993)
16. Davis, M., Sigal, R., Weyuker, E.J.: Computability, Complexity, and Languages, Fundamentals of Theoretical Computer Science, 2nd edn (1994)
17. Dijkstra, E.W.: Solution of a problem in concurrent programming control. Communications of the ACM 8(9) (1965)
18. Dubash, M.: Moore's Law is dead, says Gordon Moore. Techworld (2005)
19. Giavitto, J.-L., Michel, O.: Mgs: a rule-based programming language for complex objects and collections. Electronic Notes in Theoretical Computer Science 59(4) (2001)
20. Goodsell, D.S.: Acetylcholinesterase: molecule of the month. In: RCSB Protein Data Bank (June 2004)
21. Holland, J.H.: Adaptation in Natural and Artificial Systems. The University of Michigan (1975)
22. Hoops, S., Sahle, S., Gauges, R., Lee, C., Pahle, J., Simus, N., Singhal, M., Xu, L., Mendes, P., Kummer, U.: COPASI — a COmplex PAthway SImulator. Bioinformatics 22(24), 3067–3074 (2006)
23. Sutovic, M., Lederman, H., Pendri, K., Andrews, W.L.B.L., Stefanovic, D., Macdonald, J., Li, Y., Stojanovic, M.N.: Medium scale integration of molecular logic gates in an automaton. Nano Letters 6(11), 2598–2603 (2006)
24. Kahan, M.: Towards molecular computers that operate in a biological environment. Physica D: Nonlinear Phenomena 237(9), 1165–1172 (2008)
25. Fanning, M.L., Macdonald, J., Stefanovic, D.: Advancing the Deoxyribozyme-Based Logic Gate Design Process. In: Deaton, R., Suyama, A. (eds.) DNA 15. LNCS, vol. 5877, pp. 45–54. Springer, Heidelberg (2009)
26. Matsumaru, N., Centler, F.,, P.S.: Chemical organization theory as a theoretical base for chemical computing. In: Teuscher, C., Adamatzky, A. (eds.) Unconventional Computing 2005: From Cellular Automata to Wetware (2005)

27. Matsumaru, N., Dittrich, P.: Organization-oriented chemical programming for the organic design of distributed computing systems. In: BIONETICS 2006: Proceedings of the 1st International Conference on Bio Inspired Models of Network, Information and Computing Systems, p. 14. ACM, New York (2006)
28. Mendes, P., Hoops, S., Sahle, S., Gauges, R., Dada, J., Kummer, U.: Computational modeling of biochemical networks using copasi. In: Maly, V. (ed.) Methods in Molecular Biology, Systems Biology, vol. 500, Humana Press, a part of Springer Science + Business Media, LLC (2009)
29. Moore, G.E.: Cramming more components onto integrated circuits. Electronics 38(8) (1965)
30. Müller-Schloer, C.: Organic computing: on the feasibility of controlled emergence. In: CODES+ISSS 2004: Proceedings of the 2nd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, pp. 2–5. ACM, New York (2004)
31. di Speroni, P., Matsumaru, F.N., Centler, F., Dittrich, P.: Chemical organization theory as a theoretical base for chemical computing. International Journal of Unconventional Computing 3(4), 285–309 (2007)
32. National Center for Computational Sciences. Jaguar (2009), http://www.nccs.gov/computing-resources/jaguar/
33. Pelletier, F.J., Martin, N.M.: Post's functional completeness theorem. Notre Dame Journal of Formal Logic 31(2) (1990)
34. Sayut, D.J., Niu, Y., Sun, L.: Construction and enhancement of a minimal genetic and logic gate. Applied and Environmental Microbiology 75(3), 637–642 (2009)
35. Simmel, F.C., Yurke, B., Sanyal, R.J.: Operation kinetics of a dna-based molecular switch. J. Nanosci Nanotechnol 2, 383–390 (2002)
36. Stepney, S. (et al.) Journeys in non-classical computation : A grand challenge for computing research (2004), www.nesc.ac.uk/esi/events/Grand_Challenges/proposals/stepney.pdf
37. Su, X., Smith, L.M.: Demonstration of a universal surface DNA computer. Nucl. Acids Res. 32, 3115–3123 (2004)
38. Top500 Supercomputer Sites. ORNLs Jaguar claws its way to number one, leaving reconfigured Roadrunner behind in newest TOP500 list of fastest supercomputers (2009), http://www.top500.org/lists/2009/11/press-release
39. Turing, A.M.: On computable numbers, with an application to the Entscheidungsproblem. Proceedings of the London Mathematical Society 2(42), 230–265 (1937)
40. von Neumann, J.: First draft of a report on the edvac (1945)