Parallel computation using active self-assembly*

Moya Chen Doris Xin Damien Woods

California Institute of Technology
Pasadena, CA 91125, USA

Abstract. We study the computational complexity of the recently proposed nubots model of molecular-scale self-assembly. The model generalizes asynchronous cellular automaton to have non-local movement where large assemblies of molecules can be moved around, analogous to millions of molecular motors in animal muscle effecting the rapid movement of large arms and legs. We show that nubots is capable of simulating Boolean circuits of polylogarithmic depth and polynomial size, in only polylogarithmic expected time. In computational complexity terms, any problem from the complexity class NC is solved in polylogarithmic expected time on nubots that use a polynomial amount of workspace. Along the way, we give fast parallel algorithms for a number of problems including line growth, sorting, Boolean matrix multiplication and space-bounded Turing machine simulation, all using a constant number of nubot states (monomer types). Circuit depth is a well-studied notion of parallel time, and our result implies that nubots is a highly parallel model of computation in a formal sense. Thus, adding a movement primitive to an asynchronous non-deterministic cellular automation, as in nubots, drastically increases its parallel processing abilities.

1 Introduction

We study the theory of molecular self-assembly, working within the recently-introduced *nubots* model by Woods, Chen, Goodfriend, Dabby, Winfree and Yin [43]. Do we really need another new model of self-assembly? Consider the biological process of embryonic development: a single cell growing into an organism of astounding complexity. Throughout this active, fast and robust process there is growth and movement. For example, at an early stage in the development of the fruit fly Drosophila, the embryo contains approximately 6,000 large cells arranged on its ellipsoid-shaped surface. Suddenly, within 4-minutes, the embryo changes shape to become invaginated, creating a large structure that becomes the mesoderm, and ultimately muscle. How does this fast rearrangement occur? A large fraction of these cells undergo a rapid, synchronized and highly parallel rearrangement of their internal structure where, in each cell, one end of the cell

* Supported by National Science Foundation grants CCF-1219274, 0832824 (The Molecular Programming Project), and CCF-1162589. mpchen@caltech.edu, doris.s.xin@gmail.com, woods@caltech.edu. A full version of this paper will appear on the arXiv.

contracts and the other end expands. This is achieved by a mechanism that seems to crucially involve thousands of molecular-scale motors known as myosin pulling and pushing the cellular cytoskeleton to quickly effect this rearrangement [25]. At an abstract level one can imagine this as being analogous to how millions of molecular motors in a muscle, each taking a tiny step but acting in a highly parallel fashion, effect rapid long-distance muscle contraction. This rapid parallel movement, combined with the constraint of a fixed cellular volume, as well as variations in the elasticity properties of the cell membrane, can explain this key step in embryonic morphogenesis. Indeed, molecular motors that together, in parallel, produce macro-scale movement are a ubiquitous phenomenon in biology.

We wish to understand, at a high level of abstraction, the ultimate limitations and capabilities of such molecular scale rearrangement and growth. We do this by studying a theoretical model that includes these capabilities. As a first step towards such understanding, we show in this paper that large numbers of tiny motors (that can each pull or push a tiny amount) coupled with local state changes on a grid, are sufficient to quickly solve problems deemed to be inherently parallelizable. This result, described formally below in Section 1.2, demonstrates that our model, the nubots model, is a highly parallel computer in a computational complexity-theoretic sense.

Another motivation, and potential test-bed for our theoretical model and results, is the fabrication of active molecular-scale structures. Examples include DNA-based walkers, DNA origami that reconfigure, and simple structures called molecular motors [45] that transition between a small number of discrete states [43]. In these systems the interplay between structure and dynamics leads to behaviors and capabilities that are not seen in static structures, nor in other unstructured but active, well-mixed chemical reaction network type systems. Our theoretical results here, and those in [43], provide a sound basis to motivate the experimental investigation of large-scale active DNA nanostructures.

There are a number of theoretical models of molecular-scale algorithmic self-assembly processes [33]. For example, the abstract Tile Assembly Model, where individual square DNA tiles attach to a growing assembly lattice one at a time [41, 36, 17], or the two-handed (hierarchical) model where large multi-tile assemblies come together [1, 8, 12, 15], or the signal tile model where DNA origami tiles that form an "active" lattice with DNA strand displacement signals running along them [20, 30, 31], as well as models where one can program tile geometry [13, 18], temperature [1, 22, 39], concentration [6, 9, 16, 23] mixing stages [12, 14] and connectivity/flexibility [21].

The well-studied abstract Tile Assembly Model [41] is an asynchronous, and nondeterministic, cellular automaton with the restriction that state changes are irreversible and happen only along a crystal-like growth frontier. The nubots model is a generalization of an asynchronous and nondeterministic cellular automaton, where we have a non-local movement primitive. Nubots is intended to be a model of molecular-scale phenomena so it ignores friction and gravity, allows for the creation/destruction of monomers (we assume an invisible "fuel" source) and has a notion of Brownian motion (called agitation, but not used in this paper).

Instances of the model evolve as continuous time Markov processes, hence time is modeled as in stochastic chemical kinetics [5, 38]. The style of movement in nubots is analogous to that seen in reconfigurable robotics [7, 37, 26], and indeed results in these robotics models show that non-local movement can be used to effect fast global reconfiguration [4, 3, 35]. The nubots model includes features seen in cellular automata, Lindenmayer systems [34] and graph grammars [24]. See [43] for more detailed comparisons with these models.

1.1 Previous work on active self-assembly with movement

Previous work on the nubots model [43] showed that it is capable of building large shapes and patterns exponentially quickly: e.g. lines and squares in time logarithmic in their size. Reference [43] goes on to describe a general scheme to build arbitrary computable (connected, 2D) size-n shapes in time and number of monomer states (types) that are polylogarithmic in n, plus the time and states required for Turing machine simulation due to the inherent algorithmic complexity of the shape. Furthermore, 2D patterns with at most n colored pixels, where the color of each pixel is computable in time $\log^{O(1)} n$ (i.e. polynomial in the length of the binary description of pixel indices), are nubots-computable in time and number of monomer types polylogarithmic in n [43]. The latter result is achieved without going outside the pattern boundary and in a completely asynchronous fashion. Many other models of self-assembly are not capable of this kind of parallelism. The goal of the present paper is to formalize the kind of parallelism seen in nubots via computational complexity of classical decision problems.

Dabby and Chen [11] study an insertion-based model, where monomers insert between, and push apart, other monomers. In this nice simplification of nubots they build length-n lines in $O(\log^3 n)$ expected time and $O(\log^2 n)$ monomer types in 1D. They also show relationships with regular and context-free languages, and give a design for implementation with DNA.

1.2 Our results

In the nubots model a program is specified as a set of nubots rules $\mathcal{N}$ and is said to decide a language $L \subseteq \{0,1\}^*$ if, beginning with a word $x \in \{0,1\}^*$ encoded as a sequence of $|x|$ "binary monomers" at the origin, the system eventually reaches a configuration with the 1 monomer at the origin if $x \in L$, and 0 otherwise. Let NC denote the (well-known) class of problems solved by uniform polylogarithmic depth and polynomial size Boolean circuits.[1] Our main result is stated as follows.

Theorem 1 *For each language $L \in \mathrm{NC}$, there is a set of nubots rules $\mathcal{N}_L$ that decides L in polylogarithmic expected time, constant number of monomer states, and polynomial space in the input string length. Moreover, for $i \geq 1$, NC^i is contained in the class of languages decided by nubots running in $O(\log^{i+3} n)$ expected time, $O(1)$ monomer states, and polynomial space in input length n.*

[1] NC, or Nick's class, is named after Nicholas Pippenger.

NC problems are solved by circuits of shallow depth, hence they can be thought of as those problems that can be solved on a highly parallel architecture (simply run each layer of the circuit on a bunch of parallel processors, after polylog parallel steps we are done). NC is contained in P—problems solved by polynomial time Turing machines (this follows from the fact that NC circuits are of polynomial size). Problems in NC (or the analogous function class) include sorting, Boolean matrix multiplication, various kinds of maze solving and graph reachability, and integer addition, multiplication and division. Besides its circuit depth definition, NC has been characterized by a large number of other parallel models of computation including parallel random access machines, vector machines, and optical computers [19, 44, 42]. It is widely conjectured, but unproven, that NC is strictly contained in P. In particular, problems complete for P (such as Turing machine and cellular automata [29] prediction, context-free grammar membership and many others [19]) are believed to be "inherently sequential"—it is conjectured that these problems are not solvable by parallel computers that run for polylogarithmic time on a polynomial number of processors [19, 10].

Thus our main result gives a formal sense in which the nubots model is highly parallel: our proof gives a nubots algorithm to efficiently solve any highly parallelizable (NC) problem in polylogarithmic expected time and constant states. This stands in contrast to sequential machines like Turing machines, that cannot read all of an n-bit input string in polylogarithmic time, and "somewhat parallel" models like cellular automata and the abstract Tile Assembly Model, which can not have all of n bits influence a single bit decision in polylogarithmic time.

In order to obtain this result we give a number of novel nubots constructions. We show how to simulate function-computing logarithmic space deterministic Turing machines in only polylogarithmic expected time on nubots. We also show how to sort numbers in polylogarithmic expected time. Our sorting routine is used throughout our construction and is inspired by mechanisms such as gel electrophoresis that sort based on physical quantities (e.g. mass) [27]. We give a polylogarithmic expected time Boolean matrix multiplication algorithm, as well as a new line growing routine and a new synchronization (fast message passing) routine. All of these constructions are carried out using only a constant number of nubot monomers states and rules.

Previous results [43] on nubots were of the form: for each $n \in \mathbb{N}$ there is a set of nubot rules $\mathcal{N}_n$ (i.e. the number of rules is a function of n) to carry out some task parametrized by n (examples: quickly grow a line of length n, or an $n \times n$ square, grow some complicated computable pattern or shape whose size is parametrized by n, etc.). For each NC problems our main result here gives a *single* set of rules (i.e. of constant size), that works for all problem instances.

1.3 Future work and open questions

The line growth algorithm in [43] runs in expected time $O(\log n)$, uses $O(\log n)$ states and space $O(n) \times O(1)$ from a single seed monomer. In our construction (see full paper) we give another line growth algorithm that runs in expected time $O(\log^2 n)$, uses $O(1)$ states and space $O(n) \times O(1)$ from a size $O(\log n)$ seed. Is

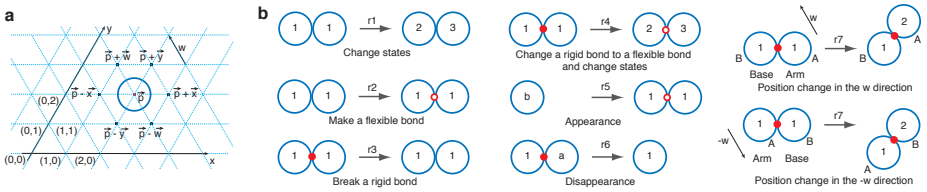

Fig. 1. Overview of nubots model. (a) A nubot configuration showing a single nubot monomer on the triangular grid. (b) Examples of nubot monomer rules. Rules r1-r6 are local cellular automaton-like rules, whereas r7 effects a non-local movement. A flexible bond is depicted as an empty red circle and a rigid bond is depicted as a solid red disk.

it possible to find a line-growth algorithm that does better than time $\times$ space $\times$ states $= \Omega(n \log^2 n)$?

Theorem 1 gives a lower bound on nubots power. What is the upper bound on confluent[2] polylogarithmic expected time nubots? One challenge involves finding better Turing machine space, or circuit depth, bounds on computing the movable set (see Section 2), and iterating this for many moves on a polynomial size (or larger) nubots grid.

Can we tighten our NC lower bound? Is the case that NC^k is contained in, say, the class of problems solved in $O(\log^{k+1} n)$ expected time on nubots? Our constructions make a lot of use of "synchronization" (where many monomers are simultaneously signaled to transition to a single common state), one way to improve our lower bound would be to see if it is possible to simulate circuits efficiently without using synchronization. The proof of Theorem 7.1 in [43] contains an example construction of a wide class of patterns that can be grown without synchronization. What conditions are necessary and sufficient for composition of arbitrary (unsynchronized) systems?

Is it possible to grow a structure of size $\Omega(n)$, in expected time $o(n)$ but without using the movement rule? Here the only source of movement comes from the "agitation" rule, which models the fact that in a liquid molecules are bombarding each other and jiggling all around. Every self-assembled molecular-scale structure was made under such conditions! Our question asks if we can *programmably exploit* this random molecular motion to build structures quicker than without it. Other open problems and further directions can be found in [43].

2 The nubots model and other definitions

In this section we formally define the nubots model. Figure 1 gives an overview of the model and rules, and Figure 2 gives an example of the movement rule.

The model uses a two-dimensional triangular grid with a coordinate system using axes x and y as shown in Figure 1(a). In the vector space induced by this

[2] By confluent we mean a kind of determinism where the system (rules with the input) is assumed to always make a unique single terminal assembly.

coordinate system, the *axial directions* $\mathcal{D} = \{\pm\vec{w}, \pm\vec{x}, \pm\vec{y}\}$ are the unit vectors along the grid axes. A pair $\vec{p} \in \mathbb{Z}^2$ is called a *grid point* and has the set of six *neighbors* $\{\vec{p} + \vec{u} \mid \vec{u} \in \mathcal{D}\}$. Let S be a finite set of monomer states. A nubot *monomer* is a pair $X = (s_i, p(X))$ where $s_i \in S$ is a state and $p(X) \in \mathbb{Z}^2$ is a grid point. Two monomers on neighboring grid points are either connected by a *flexible* or *rigid* bond, or else have no bond (called a *null* bond). Bonds are described in more detail below. A *configuration* C is a finite set of monomers along with the bonds between them.

One configuration *transitions* to another via the application of a single *rule*, $r = (s1, s2, b, \vec{u}) \to (s1', s2', b', \vec{u}')$ that acts on one or two monomers.[3] The left and right sides of the arrow respectively represent the contents of the two monomer positions before and after the application of rule r. Here $s1, s2 \in S \cup \{\mathsf{empty}\}$ are monomer states where at most one of $s1, s2$ is empty (denotes lack of a monomer), $b \in \{\mathsf{flexible}, \mathsf{rigid}, \mathsf{null}\}$ is the bond type between them, and $\vec{u} \in \mathcal{D}$ is the relative position of the $s2$ monomer to the $s1$ monomer. If either of $s1$ or $s2$ (respectively $s1'$ or $s2'$) is empty then b (respectively b') is null. The right is defined similarly, although there are some further restrictions on valid rules (involving $\vec{u}'$) described below. A rule is only applicable in the orientation specified by $\vec{u}$, and so rules are not rotationally invariant.

A rule may involve a movement (translation), or not. First, in the case of no movement: $\vec{u} = \vec{u}'$. Thus we have a rule of the form $r = (s1, s2, b, \vec{u}) \to (s1', s2', b', \vec{u})$. From above, at most one of $s1, s2$ is empty, hence we disallow spontaneous generation of monomers from empty space. *State change* and *bond change* occurs in a straightforward way, examples are shown in Figure 1(b). If $s_i \in \{s1, s2\}$ is empty and s_i' is not, then the rule induces the *appearance* of a new monomer at the empty location specified by $\vec{u}$ if $s2 = \mathsf{empty}$, or $-\vec{u}$ if $s1 = \mathsf{empty}$. If one or both monomers go from non-empty to empty, the rule induces the *disappearance* of monomer(s) at the orientation(s) given by $\vec{u}$.

For a *movement* rule it must be the case that $\vec{u} \neq \vec{u}'$ and $d(\vec{u}, \vec{u}') = 1$, where $d(u, v)$ is Manhattan distance on the triangular grid, and $s1, s2, s1', s2' \in S \backslash \{\mathsf{empty}\}$. If we fix $\vec{u} \in \mathcal{D}$, then there are two $\vec{u}' \in \mathcal{D}$ that satisfy $d(\vec{u}, \vec{u}') = 1$. A movement rule is applied both (i) locally and (ii) globally, as follows.

(i) Locally, one of the two monomers is chosen nondeterministically to be the *base* (which remains stationary), the other is the *arm* (which moves). If the $s2$ monomer, denoted X, is chosen as the arm then X moves from its current position $p(X)$ to a new position $p(X) - \vec{u} + \vec{u}'$. After this movement $\vec{u}'$ is the relative position of the $s2'$ monomer to the $s1'$ monomer, as illustrated in Figure 1(b). Analogously, if the $s1$ monomer, Y, is chosen as the arm then Y moves from $p(Y)$ to $p(Y) + \vec{u} - \vec{u}'$. Again, $\vec{u}'$ is the relative position of the $s2'$ monomer to the $s1'$ monomer. Bonds and states may change during the movement.

[3] In reference [43] the nubots model includes "agitation": each monomer is repeatedly subjected to random movements that are intended to model Brownian motion and other uncontrolled fluid flows and movement. Our constructions work with or without agitation, hence they are robust to random uncontrolled movements, but we choose to ignore this issue and not formally define agitation for ease of presentation.

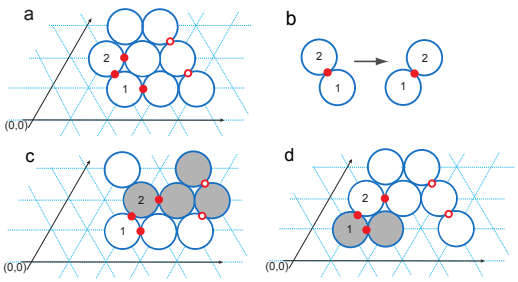

Fig. 2. An example of a movement rule with two results depending on the choice of arm or base. (a) Initial configuration. (b) Movement rule. (c) Result if the monomer with state 1 is the base. (d) Result if the monomer with state 2 is the base. We can think of (c) as pushing and (d) as pulling. Also, the affect on a flexible bonds (hollow red circles) and null bonds are shown.

(ii) Globally, the movement rule may push, or pull other monomers, or if it can do neither then it is not applicable. This is formalized as follows, and an example is shown in Figure 2. Let $\overrightarrow{v} \in \mathcal{D}$ be a unit vector. The $\overrightarrow{v}$-boundary of a set of monomers S is defined to be the set of grid points outside S that are unit distance in the $\overrightarrow{v}$ direction from monomers in S. Let C be a configuration containing adjacent monomers A and B, and let C' be C except that the bond between A and B is null in C' if not null in C. The *movable set* $M = \mathcal{M}(C, A, B, \overrightarrow{v})$ is the smallest subset of C' that contains A but not B and can be translated by $\overrightarrow{v}$ to give the set $M_{+\overrightarrow{v}}$ where the new configuration $C'' = (C' \setminus M) \cup M_{+\overrightarrow{v}}$ is such that: (a) monomer pairs in C' that are joined by rigid bonds have the same relative position in C'', (b) monomer pairs in C' that are joined by flexible bonds are neighbors in C'', and (c) the $\overrightarrow{v}$-boundary of M contains no monomers.

If $\mathcal{M}(C, A, B, \overrightarrow{v}) \neq \{\}$, then the movement where A is the arm (which should be translated by $\overrightarrow{v}$) and B is the base (which should not be translated) is applied as follows: (1) the movable set $\mathcal{M}(C, A, B, \overrightarrow{v})$ moves unit distance along $\overrightarrow{v}$; (2) the states of, and the bond between, A and B are updated according to the rule; (3) the states of all the monomers besides A and B remain unchanged and pairwise bonds remain intact (although monomer positions and flexible/null bond orientations may change). If $\mathcal{M}(C, A, B, \overrightarrow{v}) = \{\}$, the movement rule is inapplicable (the rule is "blocked" and thus A is prevented from translating).

An *assembly system* $T = (C_0, \mathcal{N})$ is a pair where C_0 is the initial configuration, and $\mathcal{N}$ is the set of rules. If configuration C_i transitions to C_j by some rule $r \in \mathcal{N}$, we write $C_i \vdash_{\mathcal{N}} C_j$. A *trajectory* is a finite sequence of configurations $C_1, C_2, \ldots, C_k$ where $C_i \vdash_{\mathcal{N}} C_{i+1}$ and $1 \leq i \leq k-1$. An assembly system evolves as a continuous time Markov process. The rate for each rule application is 1. If there are k applicable transitions for C_i then the probability of any given transition being applied is $1/k$, and the time until the next transition is applied is an exponential random variable with rate k (i.e. the expected time is

$1/k$).[4] The probability of a trajectory is then the product of the probabilities of each of the transitions along the trajectory, and the expected time of a trajectory is the sum of the expected times of each transition in the trajectory. Thus, $\sum_{t \in \mathcal{T}} \Pr[t]\text{time}(t)$ is the expected time for the system to evolve from configuration C_i to configuration C_j, where $\mathcal{T}$ is the set of all trajectories from C_i to any configuration isomorphic to C_j, that do not pass through any other configuration isomorphic to C_j, and time(t) is the expected time for trajectory t.

2.1 Nubots and decision problems

Let $\mathbb{N} = \{0, 1, 2, \ldots\}$. Given a binary string $x \in \{0,1\}^*$, written $x = x_0 x_1 \ldots x_{k-1}$, we let $\widetilde{x}$ denote a horizontal line of k nubot monomers that represent x using one of two "binary" monomer states. We let $|\widetilde{x}| \in \mathbb{N}$ denote the number of monomers in $\widetilde{x}$. Given a line of monomers A composed of m (previously defined) line segments, the notation $[A, i]$ means segment i of A, and $[A, i]_j$ means bit j of segment i of A. We next define what it means to decide a language (or problem) with nubots.

Definition 1. *A finite set of nubot rules $\mathcal{N}_L$ decides a language $L \subseteq \{0,1\}^*$ if for all $x \in \{0,1\}^*$ there is an initial configuration C_0 consisting of exactly the line $\tilde{x}$ of monomers, positioned so that the left extent of $\tilde{x}$ is at the origin $(0,0)$, where by applying the rule set $\mathcal{N}_L$, the system always eventually reaches a configuration where there is an "answer" monomer at the origin in one of two states: (a) "*accept*" if $x \in L$, or (b) "*reject*" if $x \notin L$. Further, from the time it first appears, the answer monomer never changes state.*

2.2 Boolean circuits and the class NC

We define a Boolean circuit to be a directed acyclic graph, where the nodes are called gates and each node has a label that is one of: input (with in-degree 0), constant 0 (in-degree 0), constant 1 (in-degree 0), $\vee$ (OR, in-degree 1 or 2), $\wedge$ (AND, in-degree 1 or 2), $\neg$ (NOT, in-degree 1). One of the gates is also identified as the output gate. The *depth* of a circuit is the length of the longest path from an input gate to the output gate. The *size* of a circuit is the number of gates it contains. A circuit computes a Boolean (yes/no) function on a fixed number of Boolean variables, by the inputs and constants defining the output gate value in the standard way. In order to compute functions over an arbitrarily number of variables, we define (usually, infinite) families of circuits. We say that a family of circuits $\mathcal{C}_L = \{c_n \mid c_n \text{ is a circuit with } n \in \mathbb{N} \text{ input gates}\}$ decides a language $L \subseteq \{0,1\}^*$ if for each $x \in \{0,1\}^*$ circuit $c_{|x|} \in \mathcal{C}_L$ on input x outputs 1 if $w \in L$ and 0 if $w \notin L$.

In a *non-uniform* family of circuits there is no required similarity, or relationship, between family members. We use a *uniformity function* that algorithmically specifies some similarity between members of a circuit family. Roughly speaking,

[4] For simplicity, when counting the number of applicable rules for a configuration, a movement rule is counted twice, to account for the two choices of arm and base.

a *uniform circuit family* $\mathcal{C}$ is an infinite sequence of circuits with an associated function $f : \{1\}^* \to \mathcal{C}$ that generates members of the family and is computable within some resource bound. Here we care about logspace-uniform circuit families:

Definition 2 (L-uniform circuit family). *A circuit family $\mathcal{C}$ is* L*-uniform, if there is function $f : \{1\}^* \to \mathcal{C}$ that is computable on a deterministic logarithmic space Turing machine, and where $f(1^n) = c_n$ for all $n \in \mathbb{N}$, and $c_n \in \mathcal{C}$ is a description of a circuit with n input gates.*

Without going into details, we assume reasonable encodings of circuits as strings. There are stricter, but more technical to state, notions of uniformity used in the literature [2, 46, 19, 28] (which we do not require since we are giving a lower bound on power), and circuit classes are reasonably robust under these more restrictive definitions.

Define NC^i to be the class of all languages $L \subseteq \{0,1\}^*$ that are decided by $O(\log^i n)$ depth, polynomial size L-uniform Boolean circuit families. Define $\mathrm{NC} = \bigcup_{i=0}^{\infty} \mathrm{NC}^i$, in other words NC is the class of languages decided by polylogarithmic depth and polynomial size L-uniform Boolean circuit families. Since NC circuits are of polynomial size, they can be simulated by polynomial time Turing machines, and so $\mathrm{NC} \subseteq \mathrm{P}$. It remains open whether this containment is strict [19]. See [40] for more on circuits.

3 Proof overview of Theorem 1

Here we give a high-level overview of the proof of Theorem 1. The full paper contains the detailed proof, which includes novel parallel nubots algorithms for line growth, sorting, Boolean matrix multiplication, space bounded function-computing Turing machine simulation, parallel function evaluation for functions of a certain form, Boolean circuit generation, and Boolean circuit simulation.

For each language $L \in \mathrm{NC}$, we show that there exists a finite set of nubots rules $\mathcal{N}_L$ that decides L in the sense of Definition 1. Let $\mathcal{C}_L$ be the circuit family that decides L. We begin with the observation that since L is in logspace-uniform NC, there is a deterministic Turing machine $\mathcal{M}_L$ that uses logarithmic space (in its input size) such that on unary input 1^n, $\mathcal{M}_L(1^n) = c_n$, where c_n is a description of the unique circuit in $\mathcal{C}_L$ that has n input gates.

Our initial nubots configuration consists of a length-n line of binary nubots monomers denoted $\widetilde{x}$ that represents some $x \in \{0,1\}^*$, and is located at the origin. From this we create another length-n line of monomers that encode the unary string 1^n to be used for the creation of the circuit c_n. The rule set $\mathcal{N}_L$ includes a description of $\mathcal{M}_L$. At a very abstract level, the system first generates a circuit by simulating the computation of $\mathcal{M}_L$ on input 1^n, and producing a nubots configuration (collection of monomers in a connected component) that represents the circuit c_n. The circuit is then simulated on input x. Both of these tasks present a number of challenges.

3.1 Circuit Generation

Here we describe the fast parallel simulation of the logspace machine $\mathcal{M}_L$. Logspace Turing machines have a read-only input tape with n input symbols, a read-write worktape of length $O(\log n)$, and a write-only output tape where the output tape head is assumed to always move right after writing a symbol. A configuration consists of the input tape head position, worktape contents, and worktape head position. There are at most $O(n^c)$ distinct configurations of this form, for some $c \in \mathbb{N}$, which comes from the $O(\log n)$ bound on the worktape length. Hence $\mathcal{M}_L$ runs in time $O(n^c)$. We assume that $\mathcal{M}_L$ stops in a *halt* state (we are simulating a halting, function-computing, deterministic machine, so it can be assumed to always halt in a special *halt* state). As noted, $\mathcal{M}_L$ runs in time $O(n^c)$, however we require a nubots simulation that runs in expected time that is merely polylogarithmic in n. To achieve this our simulation of $\mathcal{M}_L$ works in a highly parallel fashion, described below.

First, we describe the adjacency matrix of the configuration graph G of $\mathcal{M}_L$ on input 1^n. A configuration graph G is a directed graph, where each node represents a configuration of $\mathcal{M}_L$ on (the fixed) input 1^n [32]. There is an edge from node i to node j if and only if $\mathcal{M}_L$ moves from configuration i to configuration j in a single step. From the previously-mentioned basic facts about logspace machines, the number of nodes in G is at most polynomial in n. Further, nodes in G have out degree 0 or 1 ($\mathcal{M}_L$ is deterministic), the "halt" node has out degree 0 (we assume there are no transitions out of the halt state), and there a unique halt configuration ($\mathcal{M}_L$ completes its computation by wiping the worktape, returning all tape heads to the beginning of their tapes, and entering the halt state). The nubots system $\mathcal{N}_L$ begins by generating a representation of the adjacency matrix of graph G of machine $\mathcal{M}_L$ on input 1^n. This is achieved by building a "counter," that grows from the n monomers (that encode 1^n) to become an $O(n^c) \times O(\log n)$ rectangle, the rows of which enumerate the syntactically correct configurations of the machine via the known time ($O(n^c)$) and space ($O(\log n)$) bounds (some of these configurations are reachable on this input, and some are not). The list of configurations are grown in expected time $O(\log^2 n)$, polynomial space and only $O(1)$ monomer states. We then make a copy of this list, and pairwise compare every entry in the copy to that of the original—a process achieved via iterative copying of the list along with some geometric rearrangement tricks. The comparisons are done in parallel, where for each i, j it is checked whether configuration j is reachable from configuration i in one step on $\mathcal{M}_L$ (each such comparison depends only on configurations i, j and so can be computed in expected time $O(\log n)$ since the nubot rules $\mathcal{N}_L$ directly encode the program $\mathcal{M}_L$). The result of this process is quickly (in parallel) rearranged to form a new list (a line of monomers) that encodes the result of all of these comparisons, and thus represents the entire binary adjacency matrix M_G.

After the adjacency matrix M_G is constructed, the nubots system computes reachability on the graph G. Specifically, the rules $\mathcal{N}_L$ compute whether a path exists from the node representing the initial configuration of $\mathcal{M}_L$ on input x to the node representing the unique halting configuration in the halt state.

However, this directed graph is of size polynomial in n, so a sequential algorithm would be too slow for our purposes. We quickly (in polylog expected time) solve this reachability problem by parallel iterated matrix squaring of the adjacency matrix M_G. More precisely, we iterate $M_G := M_G^2 + M_G$ a total of $O(\log n)$ times to give the matrix M'_G. The column in M'_G that represents the halt node of graph G contains non-zero entries for exactly those nodes that have a path to the halt node [32]. The Boolean matrix squaring is carried out as follows. M_G is represented as a line of monomers, this line iscopied, and every entry of the two copies is pairwise ANDed, this involves further copying and geometric arrangement. The results are rearranged (using a novel nubots sorting algorithm discussed below) and then ORed in parallel to give the Boolean matrix M_G^2.

This parallel matrix multiplication algorithm constitutes the main part of a construction to simulate a logspace Turing machine that *decides* some language (if we also take account of accept/reject states). However, here we wish to simulate a machine that computes a *function*: $\mathcal{M}_L$'s output is a description of the circuit c_n, so we are not yet done. We add the following assumption to $\mathcal{M}_L$: it has a counter on its worktape that starts at 0 and is immediately incremented each time $\mathcal{M}_L$ writes to the output tape (this counter merely adds a $O(\log n)$ term to the worktape length). Thus only the "output-producing" configurations involve a counter incrementation. We extract from the matrix M'_G exactly those configurations that satisfy the following two criteria: (1) they are on a path from the input configuration to the halt configuration (2) they produce output. To find (1) we simply filter out those nodes (configurations) that correspond to non-zero entries in both the row of the initial node, and the column of the halt node. To get (2) we sort, via a novel, fast parallel sorting algorithm (discussed below), these configurations in increasing order of the values on their workspace counters. Then we take this sorted list, and delete everything (in parallel) except the encoded output tape write symbol from each configuration. We use the counter to sort the write symbols and are left with a line of $\ell = O(n^k)$ monomers that represent the length-ℓ output tape of $\mathcal{M}_L$ on input 1^n. This line of monomers, which we denote $\widehat{c_n}$, is an encoding of the circuit c_n.

The line of monomers $\widehat{c_n}$ is next geometrically rearranged for fast parallel circuit simulation. Here, $\widehat{c_n}$ reorganizes itself into a ladder-like form as shown in Figure 3(c) via fast parallel folding. Each layer i of the circuit c_n as shown in Figures 3(a) is encoded as a row of nubot monomers, as shown in Figure 3(c) (our circuits are assumed to be layered [40]). The circuit is now ready to be simulated.

3.2 Circuit Simulation

Recall that the circuit input bits (encoded as binary monomers) are located at the origin, and that the entire circuit was "grown" from them. These monomers move to the first (bottom) row of the encoded circuit (Figure 3(c)) and position themselves so that each gate can "read" its 1 or 2 input bits. The jth gate on layer $i \geq 1$, is simulated by a single nubot monomer that reads its adjacent 1 or 2 input bits and then sends its "result bit" to the blue "wire address" regions directly above it (Figure 3(d), in blue). After each gate computes its bit, layer i

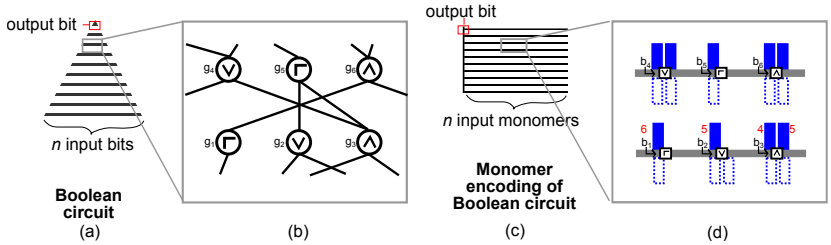

Fig. 3. Encoding of a Boolean circuit as a nubots configuration. (a) Boolean circuit with (b) detailed zoom-in. (c) Nubots configuration encoding the circuit, with zoom-in shown in (d). A wire leading out of a gate in (b) has a destination gate number encoded in (d) as strips of $O(\log n)$ blue binary monomers (indices in red). After a gate computes some Boolean function (one of $\vee$, $\wedge$, $\neg$) the resulting bit is tagged onto the relevant blue strip of monomers that encode the destination addresses (red numbers). Circuits are not necessarily planar, so to handle wire crossovers these result bits are first sorted in parallel based on their wire address, and then pushed up to the next layer of gates.

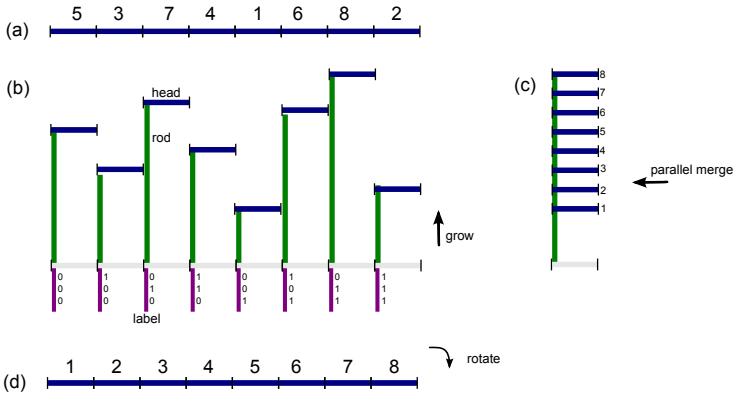

Fig. 4. High-level overview of the sorting algorithm. (a) A line of $m\lceil \log m \rceil$ monomers, split into m blue line segments ("heads") each is the binary representation of a natural number $i \leq m$. (b) A blue head that encodes value i is grown to height i by a green rod. Purple "labels" are also grown at the bottom. (c) The heads are horizontally merged, using the labels to synchronize, to be vertically aligned. (d) Merged heads rotate down into a line configuration, giving the sorted list. Each stage occurs in expected time polylogarithmic in m. See full paper for details.

"synchronizes" via a logarithmic in n expected time message passing algorithm [43]. Next, we wish to send the "result" bits from layer i to layer $i+1$. Circuits are not necessarily planar, so we need to deal with wire crossings.

Wire crossings are handled via a fast parallel sorting routine (also used in earlier parts of the construction) that is loosely inspired by Murphy et al [27] who show that physical techniques, such as gel electrophoresis, can be used to sort numbers that are represented as the magnitude of some physical quantity. The

sorting routine is illustrated in Figure 4. It takes as input a line of $m\lceil \log_2 m \rceil$ monomers, which is composed of m line segments each encoding a number in $\lceil \log_2 m \rceil$ binary monomers. Each segment grows to a height equal to its value, segments are merged horizontally, and rotated down to vertical to give a sorted list of segments, all in expected time polylogarithmic in m.

The blue "wire address" regions in the circuit (Figure 3(d)) are sorted in increasing order from left to right, then appropriately padded with empty space in between (using counters), and are passed up to the next level. After the "output gate" monomer computes its Boolean function, it signals to the rest of the circuit to destroy itself. It then moves itself to the origin and the system halts (no more rules are applicable). This completes the overview of the simulation.

This overview ignores many details. In particular the nubots model is asynchronous, that is, rule updates happen independently via stochastic chemical kinetics. The construction includes a large number of synchronization steps and signal passing to ensure that all parts of the construction are appropriately staged, but yet the construction is free to carry out many fast, asynchronous, parallel steps between these "sequential" synchronization steps.

Acknowledgments We thank Erik Winfree for valuable discussion and suggestions on our results, Paul Rothemund for stimulating conversations on molecular muscle, and Niall Murphy for informative discussions on circuit complexity theory. Damien thanks Beverley Henley for introducing him to developmental biology many moons ago.

References

1. G. Aggarwal, Q. Cheng, M. H. Goldwasser, M.-Y. Kao, P. M. de Espanes, and R. T. Schweller. Complexities for generalized models of self-assembly. *SIAM Journal on Computing*, 34:1493–1515, 2005.
2. E. Allender and M. Koucký. Amplifying lower bounds by means of self-reducibility. *Journal of the ACM*, 57:14:1–14:36, March 2010.
3. G. Aloupis, S. Collette, M. Damian, E. Demaine, R. Flatland, S. Langerman, J. O'rourke, V. Pinciu, S. Ramaswami, V. Sacristán, and S. Wuhrer. Efficient constant-velocity reconfiguration of crystalline robots. *Robotica*, 29(1):59–71, 2011.
4. G. Aloupis, S. Collette, E. D. Demaine, S. Langerman, V. Sacristán, and S. Wuhrer. Reconfiguration of cube-style modular robots using $O(\log n)$ parallel moves. In *ISAAC: Proceedings of the 19th Annual International Symposium on Algorithms and Computation*, pages 342–353, 2008.
5. D. Angluin, J. Aspnes, and D. Eisenstat. Fast computation by population protocols with a leader. *Distributed Computing*, pages 61–75, 2006.
6. F. Becker, E. Remila, and I. Rapaport. Self-assemblying classes of shapes, fast and with minimal number of tiles. In *Proceedings of the 26th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2006)*, volume 4337 of *LNCS*, pages 45–56. Springer, Dec. 2006.
7. Z. Butler, R. Fitch, and D. Rus. Distributed control for unit-compressible robots: goal-recognition, locomotion, and splitting. *IEEE/ASME Transactions on Mechatronics*, 7:418–430, 2002.

8. S. Cannon, E. D. Demaine, M. L. Demaine, S. Eisenstat, M. J. Patitz, R. T. Schweller, S. M. Summers, and A. Winslow. Two hands are better than one (up to constant factors): Self-assembly In the 2HAM vs. aTAM. In *STACS: 30th International Symposium on Theoretical Aspects of Computer Science*, pages 172–184, 2013.
9. H. Chandran, N. Gopalkrishnan, and J. Reif. Tile complexity of approximate squares. *Algorithmica*, pages 1–17, 2012.
10. A. Condon. A theory of strict P-completeness. *Computational Complexity*, 4(3):220–241, 1994.
11. N. Dabby and H.-L. Chen. Active self-assembly of simple units using an insertion primitive. In *SODA: Proceedings of the Twenty-fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1526–1536, Jan. 2012.
12. E. Demaine, M. Demaine, S. Fekete, M. Ishaque, E. Rafalin, R. Schweller, and D. Souvaine. Staged self-assembly: nanomanufacture of arbitrary shapes with $O(1)$ glues. *Natural Computing*, 7(3):347–370, 2008.
13. E. D. Demaine, M. L. Demaine, S. P. Fekete, M. J. Patitz, R. T. Schweller, A. Winslow, and D. Woods. One tile to rule them all: Simulating any Turing machine, tile assembly system, or tiling system with a single puzzle piece. Dec. 2012. Arxiv preprint `arXiv:1212.4756` [cs.DS].
14. E. D. Demaine, S. Eisenstat, M. Ishaque, and A. Winslow. One-dimensional staged self-assembly. In *DNA: 17th Intl Conf on DNA Computing and Molecular Programming*, volume 6937 of *LNCS*, pages 100–114, Pasadena, California, 2011.
15. E. D. Demaine, M. J. Patitz, T. A. Rogers, R. T. Schweller, S. M. Summers, and D. Woods. The two-handed tile assembly model is not intrinsically universal. In *ICALP: 40th International Colloquium on Automata, Languages and Programming*, Riga, Latvia, July 2013.
16. D. Doty. Randomized self-assembly for exact shapes. *SICOMP*, 39:3521, 2010.
17. D. Doty, J. H. Lutz, M. J. Patitz, R. T. Schweller, S. M. Summers, and D. Woods. The tile assembly model is intrinsically universal. In *Proceedings of the 53rd Annual IEEE Symposium on Foundations of Computer Science*, pages 439–446, Oct. 2012.
18. B. Fu, M. Patitz, R. Schweller, and B. Sheline. Self-assembly with geometric tiles. In *ICALP: The 39th International Colloquium on Automata, Languages and Programming*, volume 7391 of *LNCS*, pages 714–725. Springer, 2012.
19. R. Greenlaw, H. J. Hoover, and W. L. Ruzzo. *Limits to parallel computation: P-completeness theory*. Oxford University Press, USA, 1995.
20. N. Jonoska and D. Karpenko. Active tile self-assembly, self-similar structures and recursion. 2012. Arxiv preprint `arXiv:1211.3085 [cs.ET]`.
21. N. Jonoska and G. L. McColm. Complexity classes for self-assembling flexible tiles. *Theoretical Computer Science*, 410(4):332–346, 2009.
22. M. Kao and R. Schweller. Reducing tile complexity for self-assembly through temperature programming. In *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pages 571–580. ACM, 2006.
23. M. Kao and R. Schweller. Randomized self-assembly for approximate shapes. *Automata, Languages and Programming*, pages 370–384, 2008.
24. E. Klavins. Directed self-assembly using graph grammars. In *Foundations of Nanoscience: Self Assembled Architectures and Devices*, Snowbird, UT, 2004.
25. A. C. Martin, M. Kaschube, and E. F. Wieschaus. Pulsed contractions of an actin–myosin network drive apical constriction. *Nature*, 457(7228):495–499, 2008.
26. S. Murata and H. Kurokawa. Self-reconfigurable robots. *Robotics & Automation Magazine, IEEE*, 14(1):71–78, 2007.

27. N. Murphy, T. J. Naughton, D. Woods, B. Henley, K. McDermott, E. Duffy, P. J. van der Burgt, and N. Woods. Implementations of a model of physical sorting. *International Journal of Unconventional Computing*, 4(1):3–12, 2008.
28. N. Murphy and D. Woods. AND and/or OR: Uniform polynomial-size circuits. In *MCU: Machines, Computations and Universality*, 2013. Accepted.
29. T. Neary and D. Woods. P-completeness of cellular automaton rule 110. In *ICALP: The 33rd International Colloquium on Automata, Languages and Programming*, LNCS, pages 132–143. Springer, 2006.
30. J. Padilla, W. Liu, and N. Seeman. Hierarchical self assembly of patterns from the Robinson tilings: DNA tile design in an enhanced tile assembly model. *Natural Computing*, pages 1–16, 2011.
31. J. Padilla, M. Patitz, R. Pena, R. Schweller, N. Seeman, R. Sheline, S. Summers, and X. Zhong. Asynchronous signal passing for tile self-assembly: Fuel efficient computation and efficient assembly of shapes. In *Unconventional Computation and Natural Computation*, volume 7956 of *LNCS*, pages 174–185. Springer, 2013.
32. C. M. Papadimitriou. *Computational complexity*. Addison-Wesley Publishing Company, Inc., 1st edition, 1994.
33. M. J. Patitz. An introduction to tile-based self-assembly. In *Unconventional Computation and Natural Computation*. 7445: 34–62, LNCS, Springer, 2012.
34. P. Prusinkiewicz and A. Lindenmayer. *The algorithmic beauty of plants*. Springer-Verlag, 1990.
35. J. Reif and S. Slee. Optimal kinodynamic motion planning for 2D reconfiguration of self-reconfigurable robots. *Robot. Sci. Syst*, 2007.
36. P. W. K. Rothemund and E. Winfree. The program-size complexity of self-assembled squares (extended abstract). In *STOC: Proceedings of the thirty-second annual ACM symposium on Theory of computing*, pages 459–468. ACM Press, 2000.
37. D. Rus and M. Vona. Crystalline robots: Self-reconfiguration with compressible unit modules. *Autonomous Robots*, 10(1):107–124, 2001.
38. D. Soloveichik, M. Cook, E. Winfree, and J. Bruck. Computation with finite stochastic chemical reaction networks. *Natural Computing*, 7(4):615–633, 2008.
39. S. Summers. Reducing tile complexity for the self-assembly of scaled shapes through temperature programming. *Algorithmica*, pages 1–20, 2012.
40. H. Vollmer. *Introduction to Circuit Complexity: A Uniform Approach*. Springer-Verlag New York, Inc., 1999.
41. E. Winfree. *Algorithmic Self-Assembly of DNA*. PhD thesis, California Institute of Technology, June 1998.
42. D. Woods. Upper bounds on the computational power of an optical model of computation. In *Algorithms and Computation*, pages 777–788. Springer, 2005.
43. D. Woods, H.-L. Chen, S. Goodfriend, N. Dabby, E. Winfree, and P. Yin. Active self-assembly of algorithmic shapes and patterns in polylogarithmic time. In *ITCS'13: Proceedings of the 4th conference on Innovations in Theoretical Computer Science*, pages 353–354. ACM, 2013. Full version: `arXiv:1301.2626` [cs.DS].
44. D. Woods and T. J. Naughton. Parallel and sequential optical computing. In *Optical supercomputing*, pages 70–86. Springer, 2008.
45. B. Yurke, A. J. Turberfield, A. P. Mills, Jr., F. C. Simmel, and J. L. Nuemann. A DNA-fuelled molecular machine made of DNA. *Nature*, 406:605–608, 2000.
46. C. lvarez and B. Jenner. A very hard log-space counting class. *Theoretical Computer Science*, 107(1):3–30, 1993.