Self-Assembling Systems are Distributed Systems

Aaron Sterling*

July 7, 2009

Abstract

In 2004, Klavins et al. introduced the use of graph grammars to describe—and to program—systems of self-assembly. We show that these graph grammars can be embedded in a graph rewriting characterization of distributed systems that was proposed by Degano and Montanari over twenty years ago. We apply this embedding to generalize Soloveichik and Winfree’s local determinism criterion (for achieving a unique terminal assembly), from assembly systems of 4-sided tiles that embed in the plane, to arbitrary graph assembly systems. We present a partial converse of the embedding result, by providing sufficient conditions under which systems of distributed processors can be simulated by graph assembly systems topologically, in the plane, and in 3-space. We conclude by defining a new complexity measure: “surface cost” (essentially the convex hull of the space inhabited by agents at the conclusion of a self-assembled computation). We show that, for growth-bounded graphs, executing a subroutine to find a Maximum Independent Set only increases the surface cost of a self-assembling computation by a constant factor. We obtain this complexity bound by using the simulation results to import the distributed computing notions of “local synchronizer” and “deterministic coin flipping” into self-assembly.

1 Introduction

When a global structure (or organism) forms because of the connections formed by strictly simpler structures to one another, following only local rules, we say the global structure self-assembles. The goal of the emerging field of algorithmic self-assembly is to direct (or to program) the self-assembly of desired structures, by constructing self-assembling agents, and their environment, so they combine to form a desired result. Two main research areas have studied algorithmic self-assembly: nanotechnology and robotics. Nanostructure self-assembly dates back to the 1980’s, when Seeman engineered “tiles” from DNA strands that could connect to other tiles [13]. Winfree [18] (and later Rothemund as well [10]) designed a Tile Assembly Model as a mathematical abstraction of nanotile self-assembly. This Tile Assembly Model has become a fundamental tool in both theoretical and practical research. Our focus, though, in this paper, is on a theoretical advance that came out of the field of robotics: graph assembly systems, introduced by Klavins et al. in 2004 [6]. Graph assembly systems are a special class of graph grammars, and they provide a symbolic and topological characterization of a wide variety of systems of self-assembly. For example, a graph
assembly system that simulates the Tile Assembly Model appears in \[4\]. It turns out that graph assembly systems can be embedded in a graph rewriting characterization of distributed systems that was proposed by Degano and Montanari over twenty years ago \[3\]. We explore that observation in this paper, to make rigorous the intuition that self-assembling systems are distributed systems.

The formalisms to model self-assembling systems contain the following: a finite set of distinct types of self-assembling agents, a set of local binding rules that completely determines the behavior of the agents, and an initial configuration of the system. A particular self-assembly “run” starts with an operator placing a finite seed assembly on the surface, and then allowing a “solution” containing infinitely many of each agent type to mix on the surface. Agents bind to the seed assembly, and to the growing configuration, consistent with the local rules, and in a random, asynchronous manner. In the Winfree-Rothemund Tile Assembly Model, each agent is a four-sided tile, and the assembly surface is the two-dimensional integer plane. When designing a tile assembly system (a set of tile types and an initial seed tile) within this model, it would be useful to know whether all error-free self-assembly runs terminate in the same terminal assembly. Soloveichik and Winfree proved a useful criterion for this, called local determinism \[14\]: every locally deterministic tile assembly system has a unique terminal assembly. In this paper, we use theorems of Degano and Montanari about their graph grammar characterization of distributed systems to generalize local determinism to graph assembly systems—providing, for the first time, a criterion for proving uniqueness of terminal assembly for systems that assemble in a space other than the plane, or that contain agents that are not 4-sided.

Next, we prove a partial converse of the embedding results, by showing that all distributed systems with constant-size message complexity can be simulated topologically by graph assembly systems. Of course, for real-world applications, agents need to inhabit a physical space, under geometric constraints, so we also investigate what distributed systems can be simulated by graph assembly systems whose agents are placed on the points of $\mathbb{Z}^2$ and $\mathbb{Z}^3$. We define a class of network graphs such that all distributed systems that send only constant-size messages and whose network graph is in that class, can be simulated in two dimensions by a particular type of graph assembly system. Further, for any network graph not in that class, there exists a distributed system (with constant-size message complexity) that cannot be simulated in two dimensions by such a graph assembly system. Finally, we show that the same distributed systems that can be simulated topologically, can also be simulated in $\mathbb{Z}^3$, though our proof uses graph assembly systems of much greater complexity than the topological proof did.

We then use these simulation theorems to adapt known distributed algorithms to the realm of self-assembly. In particular, we adapt a recent algorithm due to Schneider and Wattenhofer \[12\]—which finds a Maximum Independent Set in asymptotically optimal time and message complexity for “growth-bounded” network graphs—to show that a self-assembling computation can execute a subroutine to find an MIS without consuming much geometric space. More formally, we define a complexity measure, rectangular surface cost, for a self assembling computation, which in $\mathbb{Z}^3$ means the volume of the minimal rectangular solid that contains all agents that perform the computation. We simulate Schneider and Wattenhofer’s algorithm, and a local synchronizer that adapts their synchronous algorithm to the asynchrony of self-assembly, to show that, for growth-bounded network graphs, running a subroutine to compute an MIS only causes a computation’s rectangular surface cost to increase by a constant factor, regardless of the size of the graph. That concludes the results of this paper.

Several researchers have expressed the intuition that self-assembly and distributed computing
are related. Klavins, in particular, reported programming self-assembling robots with graph assembly systems, and called graph grammars “distributed algorithms” [5]. However, the rigorous application of distributed computing results to the field of self-assembly is quite recent. The first such application appeared in [17], and a tile assembly simulation of the consensus problem for some systems of distributed processes appeared in [16].

The rest of the paper is structured as follows. Section 2 provides background on graph assembly systems and Degano and Montanari’s graph grammar characterization of distributed systems. Section 3 embeds graph assembly systems into the graph grammar characterization of distributed systems and uses that to prove a generalization of Soloveichik and Winfree’s theorem for local determinism. In Section 4 we prove distributed system simulation theorems for topological and geometric graph assembly systems. Section 5 applies these simulation theorems to prove an upper bound on the surface cost of the self-assembled computation of MISes for growth-bounded graphs. In Section 6 we summarize our work and suggest directions for future research.

2 Background

2.1 Graph assembly systems

In this subsection, we provide the basic definitions of graph assembly systems. Intuitively, the vertices of a graph represent the self-assembling agents in a system, the edges of the graph represent the current bonds between the agents, and the labels on the vertices represent the current physical conformations of the agents. The graph rewriting rules state that if agents are in the correct conformations, and have the appropriate binding relationship, then they can change relationships, by associating, disassociating, and/or changing conformations. We refer the reader to [4] for several motivating examples.

All graphs in this subsection are simple labeled graphs over a finite alphabet $\Sigma$. Later in the paper we will extend these graphs to hypergraphs. For now, a graph $G = (V, E, l)$ is a triple, where $V$ is a set of vertices, $E$ a set of edges, and $l : V \rightarrow \Sigma$ a labeling function. If $G$ is a graph, we sometimes write $V_G$ and $E_G$ to denote the vertices, or edges, of $G$, respectively. A rule is a pair of graphs $r = (L, R)$ where $V_L = V_R$. $L$ is the left-hand side of $r$, and $R$, the right-hand side of $r$.

Let $G_1$ and $G_2$ be graphs. A function $h$ from $V_{G_1}$ to $V_{G_2}$ (often written $h : G_1 \rightarrow G_2$) is a label preserving embedding if (1) $h$ is injective; (2) $\{x, y\} \in E_{G_1} \Rightarrow \{h(x), h(y)\} \in E_{G_2}$; (3) $l_{G_1} = l_{G_2} \circ h$. A rule $r$ is applicable to a graph $G$ if there exists an embedding $h : L \rightarrow G$. An action on a graph $G$ is a pair $(r, h)$ such that $r$ is applicable to $G$ as witnessed by embedding $h$.

**Definition 1.** Let $G = (V, E, l)$ be a graph, $r = (L, R)$ a rule applicable to $G$, and $(r, h)$ an action. The application of $(r, h)$ to $G$ produces a new graph $G' = (V', E', l')$, defined as follows.

$$V' = V$$
$$E' = [E \setminus \{\{h(x), h(y)\} | \{x, y\} \in L\}] \cup \{\{h(x), h(y)\} | \{x, y\} \in R\}$$
$$l'(x) = \begin{cases} 
    l(x) & \text{if } x \notin h(V_L) \\
    l_R \circ h^{-1}(x) & \text{otherwise}
\end{cases}$$

**Definition 2.** A graph assembly system is a pair $(G_0, \Phi)$, where $G_0$ is the initial graph and $\Phi$ is a (finite) set of rules, called the rule set.
Intuitively, $G_0$ represents the initial configuration of self-assembling agents, before any binding rules have been applied; while $\Phi$ characterizes the binding rules of the system. We now define the notion of a language generated by a graph assembly system.

**Definition 3.** Let $G = (G_0, \Phi)$ be a graph assembly system. A connected graph $H$ is reachable in $G$ if there exists a sequence of rules in $\Phi$ that can be applied to $G_0$ in order to produce $H$. We write $R(G_0, \Phi)$ for the set of graphs reachable in $G$. A connected graph $H \subseteq R(G_0, \Phi)$ is stable if there exists no rule in $\Phi$ that can be applied to $H$. The language of $G$, written $L(G)$, is the set of stable graphs that are reachable in $G$.

### 2.2 Background on grammars for distributed systems (GDS)

In 1987, Degano and Montanari proposed a characterization of distributed systems based on graph rewriting [3], which they called “Grammars for Distributed Systems,” or GDS. A more complete introduction can be found in [15], but briefly, a GDS is a triple $(\Sigma, D_0, P)$, where $\Sigma$ is an alphabet of events and processes that can legally appear in a distributed computation, $D_0$ is an initial finite distributed system with no events, and $P$ is a set of graph productions that characterizes legal computation steps. Degano and Montanari defined an ultrametric space of temporally ordered computations, and used this to prove that any weakly fair GDS computation has a result that is final, i.e., it converges to a limit to which no graph production can be legally applied.

**Definition 4.** A computation for GDS $G$ is a (finite or infinite) sequence ${D_i} = \langle D_0, D_1, \ldots \rangle$ such that $D_i \xrightarrow{G} D_{i+1}, i = 0, 1, \ldots$. A distributed system $D$ is final if there exists no $D'$ such that $D \xrightarrow{G} D'$. A computation is successful iff its result is final. A computation is weakly fair iff any process to which a production can be applied will eventually have some production applied to it. The language $L(G)$ generated by $G$ is the set of the distributed systems that are the results of all successful computations of $G$.

Recall that an ultrametric on a set $I$ is a function $d : I \times I \to \mathbb{R}_+$ that is reflexive, symmetric, and satisfies the condition $d(x, z) \leq \max\{d(x, y), d(y, z)\}$. If $d$ is an ultrametric on $I$, then $(I, d)$ is called an ultrametric space.

Given a distributed system $D = (N, S, f, l, \leq)$ and a subsystem $s \in S$, let $\text{depth}(s)$ be the natural number defined as the cardinality of a longest chain (without repetitions) consisting of $\leq$-predecessors of $s$. Then for any distributed system $D$ and natural number $n$, we can define the truncation $[D]_n$ of $D$ at depth $n$ by $[D]_n = (N', S', f', l', \leq')$, where

- $S' = [S]_n = \{ s \in S \mid \text{depth}(s) < n \}$
- $N' = \{ n \mid (\exists s \in S)(\exists i \in \mathbb{N})f(s)\downarrow_i = n \}$
- $f', l', \leq'$ are the restrictions of $f, l, \leq$ to $S'$ respectively.

**Definition 5.** Let $D_1$ and $D_2$ be distributed systems. The distance $d(D_1, D_2)$ is defined as

$$d(D_1, D_2) = \begin{cases} 2^{-\max\{n|[D_1]_n = [D_2]_n\}} & \text{if such a maximum exists} \\ 0 & \text{otherwise.} \end{cases}$$
Let \( \mathcal{D} \) be the set of all distributed systems in which each subsystem has a finite number of predecessors, and in which only finitely many concurrent steps occur simultaneously. Let \( \text{Fin}(\mathcal{D}) \) be the set of all finite distributed systems in \( \mathcal{D} \). Degano and Montanari proved the following.

**Theorem 1.** \((\mathcal{D}, d)\) is an ultrametric space. Further, \((\mathcal{D}, d)\) is the completion of \((\text{Fin}(\mathcal{D}), d)\). Every infinite computation \( \{D_i\} \) is a (convergent) Cauchy sequence in \((\mathcal{D}, d)\).

If a computation is finite, its result is the last element of the computation. If the computation is infinite, its result is its limit in \((\mathcal{D}, d)\). This limit is guaranteed to exist by Theorem 1. Because of this, Degano and Montanari were also able to prove the next theorem.

**Theorem 2.** An infinite computation is weakly fair iff it is successful.

### 3 Graph Assembly Systems are Distributed Systems

#### 3.1 Embedding Graph Assembly Systems in GDSes

It is a fundamental observation of this paper that graph assembly systems can be embedded into grammars for distributed systems, in the following rigorous sense.

**Theorem 3.** Let \( G = (G_0, \Phi) \) be a graph assembly system. Then there is a grammar for a distributed system \( G^* = (\Sigma, D_0, P) \) and an injective mapping \( \psi : \mathcal{R}(G_0, \Phi) \to (\mathcal{D}, d) \) such that

1. \( \psi(G_0) = D_0 \). (The initial graph is the initial distributed system.)
2. \( H \in \mathcal{L}(G) \iff \psi(H) \in \mathcal{L}(G^*) \). (Each graph in the language of \( G \) corresponds to a unique distributed system that can be reached from the initial state by a legal computation.)
3. If \( (G_0, H_1, H_2, \ldots) \) is a (finite or infinite) sequence such that each graph in the sequence can be obtained by applying rules from \( \Phi \) to the graph preceding it in the sequence, then \( (D_0, \psi(H_1), \psi(H_2), \ldots) \) is a legal computation in \( G^* \).

We defer the proof of Theorem 3 to the Appendix.

#### 3.2 Generalization of local determinism

We now apply Theorem 3 to generalize a theorem of Soloveichik and Winfree about tile self-assembly. They were interested in guaranteeing that a tile assembly system would always form a unique terminal assembly, and defined a sufficient condition, *local determinism*, that would guarantee such uniqueness. For reasons of space, we defer the formal definition of the Winfree-Rothemund Tile Assembly Model—and a graph assembly system characterization of it—to the Appendix. We use the ultrametric space of legal GDS computations to extend Soloveichik and Winfree’s idea to arbitrary graph assembly systems.

First, we define a generalized notion of local determinism.

**Definition 6.** Let \( G = (\Sigma, D_0, P) \) be a GDS. We say \( G \) is *locally deterministic* if the following holds for all computations \( \{D_i\} \) generated by \( G \). For any \( k > 1 \) and any process \( s \in D_k \), let \( D_j \) be maximal such that \( D_j \in \{D_i\} \) and \( s \notin D_j \). Then there is exactly one production applicable to the parents (i.e., immediate \( \leq \)-predecessors) of \( s \), and that production produces \( s \) in the location where it appears in \( D_k \).
In words, the initial graph and the productions of $G$ are such that the local subsystems of any finite computation entirely determine their children at the future computation step when a production is applied to them. Because the space $(\mathcal{D}, d)$ is complete, we obtain the next theorem.

**Theorem 4.** Let $G = (\Sigma, \mathcal{D}_0, P)$ be a GDS that is locally deterministic. Then all (finite or infinite) weakly fair computations generated by $G$ have the same result.

As graph assembly systems—and GDS’s—can simulate a wide range of self-assembly models, including Winfree-Rothemund tile self-assembly, Soloveichik and Winfree’s theorem is a corollary to Theorem 4.

**Corollary 1** (Soloveichik and Winfree [14]). Let $T = (T, \sigma, \Sigma, \tau, R)$ be a tile assembly system that is locally deterministic (in the sense of tile self-assembly, as defined in the Appendix). Then $T$ has a unique terminal assembly.

### 4 Self-assembly simulation of distributed systems

#### 4.1 Topological simulation of distributed systems

Graph grammars—and graph assembly systems—are Turing universal. However, the weakness of the graph grammar characterization of self-assembly is that it captures only the topology, and not the geometric constraints of the system. This means that we can simulate any single processor (Turing machine, finite state machine, etc.) using self-assembling agents in the plane, but it may not be possible to simulate a network of distributed processors, because the communication between processors may interfere with the system’s ability to grow, depending on how the agents embed themselves into their geometric environment. To provide a specific example, it is not possible for the Winfree-Rothemund Tile Assembly Model to simulate a 3-consensus object in two-dimensional tile assembly—though it can be done in three dimensions—because there is no way to ensure that three independently-growing subassemblies can have wait-free access to a common decision point [16].

In this section, our objective is to use graph assembly systems to simulate “classical” distributed systems. We first present a self-assembly simulation that considers only topology, and not geometric constraints. We then include greater physical realism, by considering graph assembly systems that embed in the plane, and in 3-space.

For the rest of this paper, we will let $M$ be an asynchronous message-passing model of distributed computing with $n$ processors, such that each processor runs forever, and can send and receive messages of constant-size complexity. The assumption about constant-size message complexity is to make our proofs simpler, as we can represent each possible message with a distinct self-assembling agent, instead of a group of agents. We require that each processor run forever to ensure that our self-assembly simulations preserve a fact about “classical” distributed systems: the number of messages has no effect on a processor’s ability to perform correctly. This is an important consideration once we consider simulations embedded into a geographic environment, as the agents passing messages could conceivably block the progress of agents simulating a processor, or vice versa. Moreover, if graph assembly systems can simulate some $M$, all of whose processors run forever, then they can also simulate $M'$, with the same network graph as $M$, but containing processors that halt.

Intuitively, $M$ can be simulated by a system of self-assembly if the logic of each processor can be simulated by a distinct subsystem, and the simulation of any message sent from processor $p_i$ to
processor $p_j$ eventually arrives (with probability one) at the subsystem that is simulating $p_j$, and is incorporated into that subsystem’s execution. We formalize that intuition as follows.

**Definition 7.** Let $G$ be a graph assembly system, and $G^*$ its induced GDS. Then we say $G^*$ simulates $M$ if:

1. There is a 1-1 mapping $h$ from configurations of $M$ to distributed systems derivable from $G^*$.
2. If $C_0, \phi_0, C_1, \phi_1 \cdots C_i, \phi_i$ is a legal execution segment of $M$, then $h(C_0) \rightarrow h(C_1) \rightarrow \cdots \rightarrow h(C_i)$ is a legal computation derivable from $G^*$.
3. If there is no legal execution segment from $C_0$ to $C_1$ in $M$, then there is no legal computation derivable from $G^*$ such that $h(C_0) \rightarrow h(C_1)$.
4. Let $C$ be a configuration of $M$ and $\mathcal{C}$ be a set of configurations of $M$. If $M$ is such that, upon achieving configuration $C$, it must eventually achieve some configuration $C' \in \mathcal{C}$, then $G^*$ is such that, if it ever reaches $h(C)$ then it must achieve $h(C')$ for some $C' \in \mathcal{C}$. (Note that $\mathcal{C}$ may be an infinite set.)
5. If $\Phi$ is a legal, finite execution segment in $M$ that contains the event that $p_i$ places message $m$ in the outbuffer intended for processor $p_j$, then, with probability one, all legal computations derivable from $h(\Phi)$ will include some $D$ such that $h^{-1}(D)$ is a configuration of $M$ in which $m$ has arrived at $p_j$.
6. If in configuration $C$ of $M$ all processes have halted, then the distributed system $h(C)$ derivable from $G^*$ is terminal.

Further, we say $G$ simulates $M$ if $G^*$ does.

Ignoring geometric constraints for the moment, we can obtain the following theorem that shows graph assembly systems can topologically simulate distributed systems.

**Theorem 5.** Let $M$ be an asynchronous message-passing model of distributed computing such that all processors run forever, and each processor can send messages of size bounded by some constant $k$. Then there is a graph assembly system that simulates $M$.

### 4.2 Simulation of distributed systems by agents with geometric constraints

We turn now to self-assembly simulations when the assembling agents are constrained to embed into a particular geometric environment. Each agent has a “size”: we will work in either the two-dimensional or three-dimensional integer lattice, and at most one agent can be placed at a given lattice point. We will characterize only simulations achieved by passive self-assembly—self-assembly in which the agents take no action once they adhere to the growing configuration. Graph assembly systems can also model active self-assembly, in which the agents are “smarter,” and can run algorithms to dissolve bonds they have made. We will not consider that here. Moreover, we will only consider the following type of simulation in this paper.

**Definition 8.** Let $M$ be a system of distributed processors, and $G$ a graph assembly system that simulates $M$. We say that $G$ simulates processors independently if, for any $i \neq j$, the convex hull of the area filled by agents simulating $p_i$ is disjoint from the convex hull of the area filled by agents simulating $p_j$. 
The purpose of the preceding definition is to require that a graph assembly system not “interweave” simulation of multiple processors. On the one hand, this requirement will make our proofs easier. On the other hand, such simulations are important in their own right, as they are more resilient to certain errors: incorrect binding in one area of the assembly will only affect the simulation of one processor, as the others are simulated in disjoint areas.

Let $C = \{H \mid H$ is a directed graph and $(\forall v \in V(H))[\text{indegree}(v) + \text{outdegree}(v) \leq 2]\}$. If the network graph of a distributed system is a member of $C$, that is a sufficient condition for the distributed system to be simulated by planar self-assembly.

**Theorem 6.** Let $\mathcal{M}$ be a system of distributed processors with constant-size message complexity, such that each processor runs forever. If the network graph of $\mathcal{M}$ is an element of $C$, then there exists a graph assembly system $G$ that simulates $\mathcal{M}$, simulates processors independently, and embeds in the plane.

On the other hand, for any graph not in $C$, there is a system of distributed processors with that network graph that cannot be simulated by the restricted version of planar self-assembly we are considering.

**Theorem 7.** Let $H \notin C$ be a graph. Then there exists a system of distributed processors $\mathcal{M}$ (of constant-size message complexity), which cannot be simulated by any graph assembly system that embeds in the plane and simulates processors independently.

By contrast, if agents have all of 3-space to work with, they can simulate $\mathcal{M}$ with any network graph.

**Theorem 8.** Let $\mathcal{M}$ be a system of distributed processors with constant-size message complexity, such that each processor runs forever. There exists a (passive) graph assembly system $G$ that embeds in $\mathbb{Z}^3$ and simulates $\mathcal{M}$, simulating processors independently.

### 5 Bounding the surface cost of self-assembling computations

We now apply the ability to simulate systems of distributed processors in 3-space, to show that finding a Maximum Independent Set can be performed by self-assembling agents in a physically compact way for certain network graphs. Self-assembling agents take up space when they perform computations, and, other things being equal, it would be helpful if they took up as little space as possible. One reason for this is to promote modularity of self-assembling systems: if we know one functionality will require at most $X$ physical space, and a second functionality will require at most $Y$ physical space, then if we budget a rectangular solid that contains enough space for both computations (plus overhead for connection), we are guaranteed we can combine both functionalities into a larger system, without modifying the internal structure of either computational process.

Since our focus is on modular connection of self-assembling subsystems into a larger whole, we will focus on the minimal convex solid that contains the entirety of a subsystem, instead of the complexities of the surface of the shape assembled. That is the motivation for the following definition.

**Definition 9.** Let $A$ be an assembly obtained by a computation of a graph assembly system, and assume $A$ is embedded into geometric space $S$. The *convex surface cost of $A$* is the size of the convex hull of $A$. This cost will be either an area or a volume for the cases we are interested in:
agents embedded in the plane or 3-space. In the specific case of agents embedding in \( \mathbb{Z}^3 \), we define the rectangular surface cost of \( A \) as the volume of the minimal rectangular solid that fully contains \( A \).

While randomized algorithms have been very helpful in distributed computing, one drawback of many such algorithms is that they only terminate within a certain expected time, instead of guaranteeing termination by some point. Simulating such an algorithm could be problematic in the world of self-assembly, as agents might grow beyond an expected boundary and collide with other parts of the configuration. One alternative to randomization is “deterministic coin flipping,” which uses processor ID’s to perform “coin-flipping-like” operations. We will simulate a deterministic coin flipping algorithm for MIS due to Schneider and Wattenhofer [12], to show that computing an MIS on so-called growth bounded graphs only adds a constant factor to the surface cost of the simulation of any algorithm in which the MIS-computation appears.

**Definition 10.** Let \( H = (V, E) \) be a graph. \( H \) is growth bounded if there is a polynomial bounding function \( f(r) \) such that for each node \( v \in V \), the size of a MIS in the neighborhood \( N^r(v) \) is at most \( f(r) \), for all \( r \geq 0 \). (Here \( N^r(v) \) is the set of nodes reachable from \( v \) in \( r \) hops.)

The motivation behind this definition was to capture the notion that “vertices are connected if they are physically near one another.” Growth bounded graphs generalize well-known graph classes used to study wireless networks, such as unit disc graphs.

Schneider and Wattenhofer’s MIS algorithm has log-star time and message complexity. However, if we limit our consideration of self-assembling systems to ones that contain no more agents than there are particles in the universe, we can approximate the log-star function with a constant function, and obtain the following theorem.

**Theorem 9.** There exists a constant \( k \) such that the following holds. Let \( \mathcal{M} \) be a system of distributed processors of constant-size message complexity, such that \( \mathcal{M} \) runs algorithm \( A \) and always terminates, whose network graph is a growth-bounded graph, and such that \( \mathcal{M} \) contains no more nodes than there are atoms in the physical universe. Define algorithm \( B \) as, “Run algorithm \( A \), then compute a Maximum Independent Set of the network,” and define \( \mathcal{M}' \) as a system with the same network graph as \( \mathcal{M} \) that executes \( B \). Let \( G \) be a graph assembly system that simulates \( \mathcal{M} \) (simulating processors independently), whose surface cost for \( A \) is minimal, and is contained in a rectangular solid of dimensions \( l, w, h \). Then there is a graph assembly system \( G' \) that simulates \( \mathcal{M}' \) (simulating processors independently), whose surface cost is contained in a solid of dimensions \( l + k, w + k, h + k \).

Expressed in big-oh notation, \( O(\text{rectangular surface cost of } A) = O(\text{rectangular surface cost of } B) \), with processors simulated independently, as the number of nodes of \( \mathcal{M} \) goes to a large-but-finite number, such as \( 2^{65536} \).

To execute the MIS algorithm, we modify our simulation slightly, to allow for processors starting with unique ID’s. We model this by having each processor start with an input string that is encoded as a string concatenated to a special character concatenated to a second string. The first string is the processor ID, and the second is the initial input to the processor, which we already know how to simulate. When the algorithm executes the deterministic coin flipping steps, the wedge construction operates on the string that simulates the processor ID.

Critical to the proof of Theorem 9 is the fact that we can simulate execution of the MIS algorithm in asynchronous self-assembly, even though the algorithm itself is synchronous. To accomplish
this, we adapt the distributed computing notion of a local synchronizer to self-assembly. A local synchronizer is a distributed algorithm that allows neighboring processors to process messages in a particular order, consistent with the round structure of a synchronous algorithm. We adapted local synchronizer ALPHA, which requires constant-many messages between each pair of neighbors per round. As the whole MIS algorithm runs for only constant-many rounds, the growth of the overall assembly along any particular vector in $\mathbb{Z}^3$ is constant, and the surface cost of the total computation increases by only a constant factor.

6 Conclusion

In this paper, we built an underpinning of rigor for the intuition that self-assembling systems are distributed systems. We then used known machinery from distributed computing to generalize an existing result in self-assembly (local determinism implies unique terminal assembly), and to prove a complexity upper-bound for a subroutine (finding an MIS) that is often used in distributed algorithms.

However, our theorems were limited to distributed systems of constant-size message complexity, and simulations by passive self-assembling systems in which processors were simulated independently. Directions for future work include investigation of the simulation of other types of distributed systems, as well as the properties of active graph assembly systems, and simulations that would interweave configurations that simulate processors. Perhaps most importantly, though, we made the assumption throughout this paper that all self-assembly was error-free, both at the initial binding steps, and also that no part of the growing assembly would degrade over time. The use of distributed computing techniques to manage fault tolerance in self-assembling systems will be, we believe, a fundamental area of research.

Acknowledgements

I am grateful to Jim Aspnes and Roger Wattenhofer for helpful discussions.

References

[1] L. Adleman, Q. Cheng, A. Goel, M.-D. Huang, D. Kempe, P. M. de Espanés, and P. W. K. Rothemund, *Combinatorial optimization problems in self-assembly*, in STOC ’02: Proceedings of the thirty-fourth annual ACM symposium on Theory of computing, New York, NY, USA, 2002, ACM, pp. 23–32.

[2] H. Attiya and J. Welch, *Distributed Computing: Fundamentals, Simulations and Advanced Topics (2nd edition)*, John Wiley Interscience, March 2004.

[3] P. Degano and U. Montanari, *A model for distributed systems based on graph rewriting*, J. ACM, 34 (1987), pp. 411–449.

[4] E. Klavins, *Directed self-assembly using graph grammars*, in Foundations of Nanoscience: Self Assembled Architectures and Devices, Snowbird, UT, 2004.

[5] E. Klavins, *Programmable self-assembly*, Control Systems Magazine, 24 (2007), pp. 43–56.
A Proof of Theorem 3

**Theorem 10** (Restatement of Theorem 3). Let \( G = (G_0, \Phi) \) be a graph assembly system. Then there is a grammar for a distributed system \( G^* = (\Sigma, D_0, P) \) and an injective mapping \( \psi : \mathcal{R}(G_0, \Phi) \to (D, d) \) such that

1. \( \psi(G_0) = D_0 \). (The initial graph is the initial distributed system.)
2. \( H \in \mathcal{L}(G) \iff \psi(H) \in \mathcal{L}(G^*) \). (Each graph in the language of \( G \) corresponds to a unique distributed system that can be reached from the initial state by a legal computation.)

3. If \( (G_0, H_1, H_2, \ldots) \) is a (finite or infinite) sequence such that each graph in the sequence can be obtained by applying rules from \( \Phi \) to the graph preceding it in the sequence, then \( (D_0, \psi(H_1), \psi(H_2), \ldots) \) is a legal computation in \( G^* \).

**Proof.** Fix graph assembly system \( G = (G_0, \Phi) \). We construct \( \psi \) and \( G^* \) as follows.

We assume \( \Phi \) is finite, though \( G_0 \) may be infinite. Let \( \Sigma \) be the alphabet of \( G \). We define \( \Sigma^* \), the alphabet of events and processes for GDS \( G^* \) by: (1) For each \( L \) such that \( (\exists R)[(L, R) \in \Phi] \), place a unique event name \( L^* \) in \( \Sigma^* \); (2) for each vertex label \( \lambda \in \Sigma \), place a unique process name \( \lambda^* \in \Sigma^* \). This constructs \( \Sigma^* = (T, M) \).

Let \( k \) be the maximum degree of a node in any \( L \) or \( R \) such that \( (L, R) \in \Phi \), and fix an orientation on each rule in \( \Phi \) so the edges of each node are marked first, second, third, up to \( k \)-th. For node \( v \) with label \( \lambda \in \Sigma \), define \( \psi(v) \triangleq s \), where \( s \) is a hyperedge on \( k \) nodes with label \( \lambda^* \in \Sigma^* \). For nodes \( u, v \) with an edge \( e \) between them, let \( i, j \) be the orientation markers of where the edge connects to \( u, v \). Define \( \psi(e) \triangleq e' \), where \( e' \) is an edge from the \( i \)-th port of \( \psi(u) \) to the \( j \)-th port of \( \psi(v) \).

Define \( G^* = (\Sigma^*, D_0, P) \) so \( D_0 \triangleq \psi(G_0) \), and \( P \triangleq \{ (\psi(L), L^* \rightarrow \psi(R)) \mid (L, R) \in \Phi \} \), where \( L^* \rightarrow \psi(R) \) is the subgraph obtained by drawing an edge from the event \( L^* \) to the subsystem \( \psi(R) \). Then \( \psi(G_0) = D_0 \), and, if \( H \) is reachable from \( G_0 \), then \( \psi(H) \) can be generated from \( G^* \) by applying the appropriate rule, translated from \( \Phi \) to \( P \). So any legal application of rules of \( G \) induces a legal computation in \( G^* \) via application of \( \psi \). Finally, since we defined \( \psi \) injectively, \( H \in \mathcal{L}(G) \iff \psi(H) \in \mathcal{L}(G^*) \).

Figure 1 shows a GDS representation of the behavior of a sample graph assembly system.

## B Proofs for Section 3.2

### B.1 Winfree-Rothemund Tile Assembly Model

Winfree’s objective in defining the Tile Assembly Model was to provide a useful mathematical abstraction of DNA tiles combining in solution in a random, nondeterministic, asynchronous manner. Rothemund, and Rothemund and Winfree, extended the original definition of the model. For a comprehensive introduction to tile assembly, we refer the reader to [10]. Intuitively, we desire a formalism that models the placement of square tiles on the integer plane, one at a time, such that each new tile placed binds to the tiles already there, according to specific rules. Tiles have four sides (often referred to as north, south, east and west) and exactly one orientation, i.e., they cannot be rotated.

A tile assembly system \( T \) is a 5-tuple \( (T, \sigma, \Sigma, \tau, R) \), where \( T \) is a finite set of tile types; \( \sigma \) is the seed tile or seed assembly, the “starting configuration” for assemblies of \( T \); \( \tau : T \times \{N, S, E, W\} \rightarrow \Sigma \times \{0, 1, 2\} \) is an assignment of symbols (“glue names”) and a “glue strength” (0, 1, or 2) to the north, south, east and west sides of each tile; and a symmetric relation \( R \subseteq \Sigma \times \Sigma \) that specifies which glues can bind with nonzero strength. In this model, there are no negative glue strengths, i.e., two tiles cannot repel each other.

A configuration of \( T \) is a set of tiles, all of which are tile types from \( T \), that have been placed in the plane, and the configuration is stable if the binding strength (from \( \tau \) and \( R \) in \( T \)) at every
Figure 1: A comparison of a graph assembly system and its GDS representation. This figure is based on Example 1 in [4]. In the column on the left, we see one possible assembly sequence from the initial graph and rule set. On the right, we see the same behavior represented in GDS format.
possible cut is at least 2. An assembly sequence is a sequence of single-tile additions to the frontier of the assembly constructed at the previous stage. Assembly sequences can be finite or infinite in length. The result of assembly sequence $\alpha$ is the union of the tile configurations obtained at every finite stage of $\alpha$. The assemblies produced by $T$ is the set of all stable assemblies that can be built by starting from the seed assembly of $T$ and legally adding tiles. If $\alpha$ and $\beta$ are configurations of $T$, we write $\alpha \rightarrow \beta$ if there is an assembly sequence that starts at $\alpha$ and produces $\beta$. An assembly of $T$ is terminal if no tiles can be stably added to it.

We are, of course, interested in being able to prove that a certain tile assembly system always achieves a certain output. In [14], Soloveichik and Winfree presented a strong technique for this: local determinism. An assembly sequence $\alpha$ is locally deterministic if (1) each tile added in $\alpha$ binds with the minimum strength required for binding; (2) if there is a tile of type $t_0$ at location $l$ in the result of $\alpha$, and $t_0$ and the immediate “OUT-neighbors” of $t_0$ are deleted from the result of $\alpha$, then no other tile type in $T$ can legally bind at $l$; the result of $\alpha$ is terminal. Local determinism is important because of the following result.

**Theorem 11** (Soloveichik and Winfree [14]). If $T$ is locally deterministic, then $T$ has a unique terminal assembly.

Klavins has shown how to model a Tile Assembly System with a graph assembly system (Example 7 in [4]). We summarize his construction here, for completeness.

Let $T = (T, \sigma, \Sigma, \tau, R)$ be a tile assembly system. We will construct a graph assembly system $G = (G_0, \Phi)$ that models it as follows. To model tile edges, we extend $\Sigma$ to alphabet $\Sigma^*$ of $G$, by adding, for every $a \in \Sigma$, new symbols $(N, a), (S, a), (E, a), (W, a)$ and $(N, a)', (S, a)', (E, a)', (W, a)' to $\Sigma^*$. Intuitively, the unprimed symbols indicate that $a$ is a symbol at an unmatched (north, south, east or west) edge, while the primed symbols indicate that $a$ is a symbol at an edge that is bound to another tile. We also add four new symbols to $\Sigma^*$: $x, y, x'$ and $y'$. The symbols $x$ and $x'$ represent the “center” of a tile in either a state that is bound (unprimed) or unbound (primed) to the surface, and $y, y'$ represent points of the surface that either have a tile bound (unprimed) to them or not (primed). We also add to $\Sigma^*$ the new symbols $N, S, E$ and $W$. These symbols will be used to specify the orientation of the underlying grid.

The initial graph $G_0$ of $G$ consists of an underlying grid, and an infinite supply of tiles of each type. We can specify the initial seed assembly $\sigma$ by using symbols from $\Sigma^*$ to define a grid that has specific tiles attached to it in finitely many places. (See [4] for a depiction of a grid that represents the assembly surface.) The binding rules of $T$ can be translated directly into assembly rules of $G$, but, in addition, we must add additional rules to ensure that all legal $G$-assemblies are planar. These rules appear in Figure 2.

### B.2 Proof of Corollary 1

We now prove Soloveichik and Winfree’s theorem that a locally deterministic tileset has a unique terminal assembly (i.e., Corollary 1 in Section 3.2).

**Proof of Corollary 1.** Let $T$ be a locally deterministic tile assembly system. Let $G_T$ be a graph assembly system that simulates the behavior of $T$, constructed as described in Section B.1 and let $G_T^*$ be a GDS that simulates $G_T$, as provided by Theorem 3. In the Winfree-Rothemund Tile Assembly Model, all tile assembly sequences are assumed to be fair, so any computation generated by $G_T^*$ will be weakly fair. Note that if $T$ is locally deterministic (in the sense of tile assembly),
Figure 2: Binding rules to ensure planarity when simulating tile assembly systems with graph assembly systems (modeled after a figure in [4]). The top rule specifies how an unattached tile may bind to a tile that is already attached to the growing seed supertile. The bottom rule specifies how two adjacent tiles already bound to the seed supertile, but not to each other, may bind. We add two such rules for each \((p,q) \in R\), where \(R\) is the binding relation of the graph assembly system.

then \(G_T^\pi\) is locally deterministic (in the sense of this paper), because the predecessors of any tile \(t\) determine that \(t\) is the unique tile that can be placed at that location. Therefore, by Theorem [4], all computations generated by \(G_T^\pi\) will have the same result, which implies that \(T\) itself must have a unique terminal assembly.

B.3 Proof of Theorem 4

Theorem 12 (Restatement of Theorem 4). Let \(G = (\Sigma, D_0, P)\) be a GDS that is locally deterministic. Then all (finite or infinite) weakly fair computations generated by \(G\) have the same result.

Proof. If the successful result of a computation generated by \(G\) is finite, then suppose there are two finite computations \(\{D^1_i\}\) and \(\{D^2_i\}\), each producing a different result. Then there must be some \(n\) such that \([D^1]_n = [D^2]_n\) but \([D^1]_{n+1} \neq [D^2]_{n+1}\). Let \(s\) be a process that witnesses the difference; suppose WLOG that \(s \in [D^1]_{n+1}\). Since \(G\) is locally deterministic, and both computations are weakly fair, \(s\) will eventually appear, with exactly the same predecessors, in \(\{D^2\}\). So the results of the two computations will eventually be equal.

Now suppose \(G\) generates two infinite weakly fair computation \(\{D^1_i\}\) and \(\{D^2_i\}\). By Theorem 2, both \(\{D^1_i\}\) and \(\{D^2_i\}\) are successful. But then we can argue as above. If the results of the two computations differ, they differ at some subsystem with finitely many predecessors. As those predecessors determine their child to be the unique subsystem that supposedly causes the difference, the assumption contradicts local determinism. 

15
We conclude this section by noting that the “converse” question—given a terminal assembly $A$, construct a graph grammar whose unique result is $A$—is not well understood. An early complexity result in tile self-assembly showed that finding a minimal tileset that uniquely produces an assembly is NP-complete \[1\]. More generally, unless $P=NP$, there is no polynomial-time compression algorithm such that, given a string, the algorithm produces a grammar whose size is within a small constant factor of the minimal grammar that produces the string \[8\]. In the positive, Klavins et al. have proposed an algorithm that, given an arbitrary graph $G$, constructs a graph grammar whose unique result is $G$ \[6\]. Their algorithm has not been implemented, but Peshkin has reported implementation of a graph grammar compression algorithm to analyze DNA molecules \[9\]. The approximation ratios obtained by these algorithms are unknown.

C Proof of Theorem 5

Theorem 13 (Restatement of Theorem 5). Let $\mathcal{M}$ be an asynchronous message-passing model of distributed computing such that all processors run forever, and each processor can send messages of size bounded by some constant $k$. Then there is a graph assembly system that simulates $\mathcal{M}$.

Proof. Let $p_1, \ldots, p_n$ be the processors of $\mathcal{M}$. Since graph assembly systems are Turing universal, there are graph assembly systems $G_1, \ldots, G_n$ that simulate $p_1, \ldots, p_n$. For simplicity, we will use, as a “base” for our construction, a simulation of the $n$ processors by 4-regular self-assembling agents that can be embedded in the plane. Winfree showed (in his Tile Assembly Model) that any Turing machine could be simulated in this way \[18\], by a planar “wedge” construction. This wedge construction can be modified to simulate a processor sending a constant-size message to another location in the plane, and the angle at which the wedge grows can be modified to fit as many wedges as necessary into a single quadrant of the plane \[7\]. So if $\mathcal{M}$ is such that no processor ever sends a message to any other, we can build a GDS that simulates $\mathcal{M}$ easily, by starting with an initial graph $G_0$ that encodes wedges that simulate each $p_i$. As long as we are careful to ensure the rule set $\Phi$ constructs wedges that grow without colliding with one another—which we can always do—then $(G_0, \Phi)$ is a graph assembly system that simulates $\mathcal{M}$.

So assume that some processors in $\mathcal{M}$ send messages to one another. We will enhance the wedges that simulate the logic of each processor by defining agents that build along one side to act as an inbuffer, and along the other side to simulate the sending of a message from one processor to another.

Let $W_i$ be the graph grammar that produces the wedge that simulates $p_i$. WLOG, assume that $W_i$ grows northward, and grows in width only to the west. We will simulate an inbuffer along the east side of $W_i$, and modify it so that infinitely many rows simulate checking the value of that inbuffer. Figure 3(i) and 3(ii) shows how to modify a wedge so that the information “message $m$ received” or “no message received” can be incorporated into the overall logic of the wedge.

Let $X \subseteq \{(i, j) \mid 1 \leq i \neq j \leq n\}$ be the set of pairs of processors such that $p_i$ could send a message to $p_j$ under some legal execution of $\mathcal{M}$. For each $(i, j) \in X$, we add an “information highway” from $W_i$ to $W_j$ by adding ports to the inbuffer of Figure 3(ii); this is shown in Figure 3(iii). We then define one agent (and corresponding rules) for each message and each element of $X$, so the agent simulates “message $m$ is in transit from $p_i$ to $p_j$.” As there are constant-many possible messages, and $n$ processors, we only need to define finitely many such agents. As $p_i$ may send a second message $m_2$ to $p_j$ before a first message $m_1$ has been delivered, we add ports to the message-encoding agents that duplicate the “information highway.” This permits the creation of a
queue: messages from $p_i$ to $p_j$ will arrive in the order sent. This step of the construction is shown in Figure 3(iv).

We complete construction of the inbuffer simulation mechanism with agents and rules that behave according to Figure 4. If messages $m_1, \ldots, m_k$ from distinct processors $p_{i_1}, \ldots, p_{i_k}$ are waiting in the inbuffer, one of them will eventually enter the inbuffer with probability one. This is guaranteed because we are assuming that if more than one rule in $\Phi$ is applicable at a given time step, the rule that is applied will be chosen uniformly at random. Therefore, since it takes placement of only one agent to move a message up a wedge one row, but several tiles to build an entire wedge row, any message sent will (almost surely) eventually get to the top of the wedge. Note that this construction guarantees only that if $m_1$ and $m_2$ are sent by the same processor, and $m_1$ is sent first, then $m_1$ will enter the inbuffer before $m_2$. Messages sent by different processors have no such prioritization. Nevertheless, as the inbuffer is checked infinitely often, and there can only be $k$-many messages competing to enter the inbuffer at any time step (for some fixed $k$), each message waiting at a given time step $s$ will eventually enter with probability one, satisfying our definition of simulation of an asynchronous system. (We are glossing over the mechanism that moves $m_2$ to the front of the $p_i$-queue, if $p_i$ sent $m_1$ and $m_2$, and then $m_1$ entered the inbuffer. To construct this mechanism formally, we begin with a mechanism in the Tile Assembly Model that does this for a single processor: such a mechanism appears in Figures 7 and 8. We then define graph assembly system agents and rules for that mechanism, using Klavins’ graph grammar simulation of the Tile Assembly Model presented in the Appendix.)

To complete the proof, we note that the simulation of the sending of messages is straightforward, and illustrated in Figure 5. Since we are only considering topology, and do not care about embedding the agents into a geometric space, we can assume that each wedge grows in width by one agent at every other row, without worrying about collision with other parts of the construction. Then, since we know the exact pattern of wedge growth, we can define agent rules that move a sent message down the side of the wedge that is growing (i.e., that the inbuffer is not on), and hardcode additional binding rules into $G_0$ that send the message along ports of $G_0$ on to its destination wedge, bringing us back to the mechanism in Figure 4.

As we can build a graph assembly system that simulates $M$, by Theorem 3, we can build a GDS that simulates $M$ also.

**D Proof of Theorem 6**

**Theorem 14** (Restatement of Theorem 6). Let $M$ be a system of distributed processors with constant-size message complexity, such that each processor runs forever. If the network graph of $M$ is an element of $C$, then there exists a graph assembly system $G$ that simulates $M$, simulates $M$ in parallel, and embeds in the plane.

**Proof.** We will prove a stronger statement: the entire simulation can be constructed by tiles embedding in the plane, i.e., 4-regular agents. Our method is an extension and generalization of a construction found in [7]. The main idea behind our simulation is shown in Figures 3, 7 and 8. For each processor in $M$, we use a modified wedge construction to simulate its execution of its algorithm, and its inbuffer and outbuffer. This is essentially a planar version of the proof of the topological simulation theorem, except now, between each row of the wedge that simulates a computation step, we place two inbuffer (and/or outbuffer) rows instead of one: one row checks for an
Figure 3: This figure illustrates how to simulate a processor’s inbuffer, given a “wedge” construction (symbolized by the yellow tiles) that simulates the logic of the processor, as in (i). In stage (ii), we interleave the logic of the processor with an “inbuffer checking row,” such that the agents only bind once the blue inbuffer agent has bound to the structure. The agents in the inbuffer row copy information from south to north, so no information is lost in the construction. The dashed arrows show the direction in which agents bind at inbuffer rows. In stage (iii), we modify the inbuffer agents so they have unique ports for each processor that might potentially send a message to the processor the yellow agents are simulating. Stage (iv) shows this in action: processor $p_j$ has sent messages $m_1$ and $m_2$, in that order, and both messages are “climbing” up the edge of the wedge.
Figure 4: The conclusion of the inbuffer simulation described in Figure 3. We see three messages waiting: $m_1$ and $m_2$, both sent from processor $p_j$; and $m_3$, sent from processor $p_k$. The predecessors of the agent that simulates $m_3$ are connected to the appropriate ports along the inbuffer, like the agents that carried $m_1$, but we have represented them with a dotted line, for clarity. Three possible agents can bind to the inbuffer location at the northeast corner of the wedge: an agent (like before) that simulates no message received yet; an agent simulating that $m_1$ was received, and an agent simulating the receipt of $m_3$. (Note that $m_2$ cannot be received until $m_1$ has been delivered. Then it will be advanced to the head of the queue using a mechanism similar to the one used in tile assembly in Figure 8.)
Figure 5: As the yellow wedge grows, if the processor logic indicates a message be sent, the wedge sends a tile encoding the message, and the direction and distance it travels, down the edge of the wedge that is not in use by the inbuffer. This is represented by the red-colored tiles. The ports on the west side of the red tiles are available in case the wedge wants to send another message at a future point in the construction.
Figure 6: Schematic of a planar simulation of a three-processor distributed system. In this figure, tiles are simulating the fact that processor $p_1$ is sending a message to $p_2$, $p_2$ is sending a message to $p_3$, and $p_3$ is sending a message to $p_1$. The green tiles represent the initial seed assembly, the yellow tiles represent the growing wedge constructions $\pi_1$ through $\pi_3$ that simulate the processors, and the blue tiles represent the agents that simulate transmission of messages from one processor to the next. The black arrows show the order of tile placement in the plane. Figures 7, 8 and 9 illustrate how to simulate the sending and receiving of messages in more detail.

incoming message on one side of the wedge (or sends an outgoing message), and the second row checks for an incoming message on the other side of the wedge (or sends an outgoing message). Figure 9 shows this construction in more detail.

To simulate $M$ with an arbitrary network graph in $C$, we need to ensure that the wedges that simulate the processors never collide with one another. We accomplish this using a technique that first appeared in [7]: we attach a counter to one side of the wedge, and modify the tile assembly system so the wedge grows at a slope determined by the counter. In this way, it is possible to have infinitely many wedge constructions start building on the $x$-axis upward into the first quadrant, and yet none will ever collide. As we only need finitely many wedges to simulate any $M$, this is certainly sufficient for our purposes. Note that due to the requirement that $M$ have constant-size message complexity, for each possible message we can add a tile to the set of tiles that constructs the counter, so it will still be possible to simulate an inbuffer (or outbuffer) on the same side of the wedge as the counter. These new tiles count to the north, and transmit slope information to the wedge, while at the same time, messages “piggyback” on them from west to east, performing the function shown in Figure 9.

Putting all the elements together demonstrates existence of a tile assembly system that simulates any $M$ as given in the theorem statement. We are done.
Figure 7: This figure illustrates the simulation of a processor, its inbuffer, and the receipt of a message. The yellow tiles are the edge of a “wedge” that simulates processor 2 (in the simulation of a two-processor system). The blue tiles represent the inbuffer of processor 2. In this diagram, every other row of the wedge is a row that is built from the inbuffer tile, to the west. The arrows show the direction that tiles bind in. More generally, the wedge construction does not need to “check” the inbuffer at every other row, as long as it checks infinitely often. Figure (i) shows how message $m_1$ attaches along the east side of the inbuffer, and has opportunities due to double bonds to bind to the north of the most recent inbuffer tile. In Figure (ii), $m_1$ is successfully transferred into the inbuffer, which then means that the next row that checks the contents of the inbuffer transmits $m_1$ to (potentially) all tiles in the wedge, as shown. The farthest northeast tile in Figure (ii) changes the column to an inbuffer column, in case later messages need to be transferred from the east to the wedge growing to the west.
Figure 8: This figure continues the construction from Figure 7(ii), by showing how a second message, named $m_2$, gets transferred to the “first message” column of the wedge simulating processor 2. The mechanism shown in Figure 7 will then transfer $m_2$ to the inbuffer. Note that the furthest northeast tile sets up the configuration so any further messages sent can be transferred west, as $m_2$ was.
Figure 9: This figure shows how to modify the wedge construction so that a simulated processor can receive a message on the “jagged edge” of the wedge, as shown in Figure 6, with the west side of subassembly $\pi_1$ receiving a message sent from subassembly $\pi_3$. The yellow tiles are the ones simulating the processor and the blue tiles are the ones transmitting the message. The blue tiles crawl up the side of the yellow wedge, which now has two rows that check for receipt of messages. (The arrows show the order of tile placement.) The far west location of the upper of the two rows can accept a tile that encodes “no message received,” or, “message $m$ received,” for each of the constant-many messages $m$. If the “no message received” tile binds, it communicates its presence to the west (shown by the “N” in this figure), so the (blue) message transmitting-ray can place a tile with a double-bond to the west, so it can proceed northward in a jagged fashion, to match the edge of the processor simulation. On the other hand, if the message binds into that upper row (as shown with the red tile in this figure), it propagates to the interior of the wedge, just like a message received from the mechanism of the previous diagrams. Further, it sends a signal north and west (shown by the “X”) to stop the message ray from continuing to propagate northward. We accomplish the sending of multiple messages over time by means of a slight modification to the mechanism shown in Figure 7(ii).
E  Proof of Theorem 7

Theorem 15 (Restatement of Theorem 7). Let $H 
otin C$ be a graph. Then there exists a system of distributed processors $M$ (of constant-size message complexity), which cannot be simulated by any (passive) graph assembly system that embeds in the plane and simulates processors independently.

Proof. Fix a graph $H$ that has at least one vertex $v$ such that $\text{indegree}(v) + \text{outdegree}(v) \geq 3$. We will construct an $M$ (in which each processor runs forever) so that an adversary can produce a worst-case order of message delivery that prevents $M$ from being simulated in passive self-assembly. This proof is similar to one in [16], which showed it is not possible in two-dimensional tile self-assembly for three different locations to have wait-free access to a common point.

Let $M$ be such that each processor changes state infinitely often, regardless of the messages it receives from other processors, and further that each processor that can send messages to its neighbors does so infinitely often, regardless of the behavior of other processors. Assume there exists a graph assembly system $G$ that simulates $M$ in two dimensions, simulating processors independently. We can then assume that we start with an initial, connected, seed assembly that encodes the starting location for each independent configuration that will simulate a processor. Let $\pi_v$ be the growing subassembly that simulates the behavior of the processor at vertex $v$ in the network graph of $M$.

In the network graph of $M$ there is a (directed or undirected) edge from $p_v$ to at least three other processors; let us call them $p_0$, $p_1$ and $p_2$. We need to be able to build rays of tiles between $\pi_v$ and at least three other independent subassemblies, call them $\pi_0$ through $\pi_2$, as they simulate $p_0$ through $p_2$. However, since all four $\pi_i$ are rooted to a common assembly, at least one of these rays, say from $\pi_0$ to $\pi_v$ (we have the same problem if it is from $\pi_v$ to $\pi_0$) must traverse through the cone of either $\pi_1$ or $\pi_2$ (WLOG call it $\pi_1$).

If the ray from $\pi_0$ to $\pi_v$ builds before the agents forming $\pi_1$ are placed there, then $\pi_1$ will be blocked from making further progress, and will not have the space to simulate $p_1$ forever, a contradiction. On the other hand, if we interweave the information from the message ray from $\pi_0$, and the simulation of $p_1$, so $\pi_1$ grows in such a way that the ray from $\pi_0$ is transmitted “through” it, as agents from the ray and from $\pi_1$ communicate to build a new row of $\pi_1$ that, for example, sends information from $\pi_1$ northward, and the message from $\pi_0$ eastward, we again achieve a contradiction, because $G$ cannot simulate all legal computations of $M$. In particular, $\pi_0$ can only communicate with $\pi_v$ if $\pi_1$ cooperates. However, our assumption about $M$ requires that $p_0$ be able to send infinitely many messages to $p_v$, even if $p_1$ is very slow, or fails. Therefore, no such $G$ can exist.

It is essential to this proof that we are considering only passive self-assembly; active agents might recognize a message blockage and disassemble, then reassemble in a different conformation, in order to permit a $\pi_i$ to grow “through” a message ray that was previously placed on the surface. We leave consideration of that possibility to future work.

F  Proof of Theorem 8

Theorem 16 (Restatement of Theorem 8). Let $M$ be a system of distributed processors with constant-size message complexity, such that each processor runs forever. There exists a (passive)
graph assembly system \( G \) that embeds in 3-space and simulates \( \mathcal{M} \), simulating processors independently.

**Proof.** We want to build a graph assembly system that simulates the processors of \( \mathcal{M} \) independently. As each processor runs forever, hence potentially changes configuration infinitely many times, we need to ensure that for each processor there is a convex cone that can extend infinitely in one direction. Suppose processors \( p_0 \) and \( p_1 \) send one another messages. We need to ensure that agents carrying those messages can travel between the convex cones simulating \( p_0 \) and \( p_1 \). This means that we need to build a surface that extends infinitely in at least one direction: the cones that simulate the two processors, connected by the rays that simulate the messages. The objective of the proof will be to show that for any \( \mathcal{M} \) satisfying the theorem statement, it is possible to stitch such surfaces together so that the network graph of \( \mathcal{M} \) is correctly simulated, and there is no legal assembly sequence in which any surface blocks off the progress of any other.

We start by hardcoding an initial seed assembly, with tiles that begin a wedge construction for each processor in a line along the \( x \)-axis, similar to Figure 6. For each \( i,j \) such that, for some legal execution of \( \mathcal{M} \), \( p_i \) sends a message to \( p_j \), we build an “information highway” (a foundation for message-passing rays of agents, like the tiles between wedges in Figure 6) from wedge \( \pi_i \) to wedge \( \pi_j \) that lies in a unique plane, shared by no other information highways. This is similar to the well-known “book embedding” of a graph, except that here we are working in the lattice \( \mathbb{Z}^3 \), not \( \mathbb{R}^3 \), as we assume we can place at most one agent at each point of the integer lattice. However, as the network graph of \( \mathcal{M} \) is finite, we can place additional agents in the seed assembly between wedges, to separate out how far apart the wedges lie on the \( x \)-axis; and we can define (finitely many distinct) agent types of rational length, so we have room to place each information highway on a unique plane, even though we are working in discrete space.

In particular, if both \( p_i \) and \( p_j \) can send messages to each other, in three dimensions there is space to fit two convex cones in parallel planes “stacked on top of each other”—one for messages from \( p_i \) to \( p_j \), and the other from \( p_j \) to \( p_i \)—in such a way that those convex cones fit between the convex cones of agents that simulate the behavior of \( p_i \) and \( p_j \) themselves. There is, however, a potential bandwidth problem if multiple messages are trying to connect with a particular wedge at a given lattice point. Therefore, it suffices to show that if a processor is supposed to receive multiple incoming messages, then in the simulation, all of those messages will be received, *i.e.*, there is no worst-case assembly sequence in which a message will be dropped or blockaded by another message.

We resolve this by generalizing the multiple-message-delivery mechanism of Figure 7. Let \( m_0 \) be an upper bound on the number of distinct processors that can send any given processor a message. The inbuffer-simulating edge of the wedge will be built by agents with \( m_0 + 1 \) connections to other agents. One connection will be to the wedge, as before, and the other \( m_0 \) connections will be input locations to receive messages from sending processors. The inbuffer binding rules order the sending processors as \( p_{i_1}, \ldots, p_{i_{m_0}} \), and keeps a counter that cycles through that order. When a wedge builds a row to check whether a message has been received, it checks to see whether there is a message in the queue from processor \( p_{\text{counter}} \), and then it increments the value of the counter. If there is such a message waiting, it gets transmitted to the interior of the wedge. Otherwise, the wedge accepts no message during that row of the construction. (This can all be programmed into individual agent types, because the mechanism we are describing uses one variable of constant space.) If there are multiple messages from a single processor \( p_i \), for some \( i \), they wait behind one another in the plane of their information highway, similar to messages \( m_1 \) and \( m_2 \) in Figure 7.
This ensures that each message will get to the front of the queue after only finitely messages ahead of it enter the inbuffer. Once it is at the front of the queue, it will have infinitely-many chances to enter the inbuffer without competition from other messages, hence will eventually enter the inbuffer with probability one. This is what we needed to prove.

With tile self-assembly in the plane, the induced graph of agents and bonds is such that every edge is of length 1: it is a grid graph. The proof above, by contrast, makes use of the fact that agents in a graph assembly system can have connections (edges) to other agents that vary in length, from agent type to agent type. As a result, it is relatively “easy” to ensure there is enough physical space to accept input from up to \( m_0 \) different processors into the same inbuffer queue. We believe that, by attaching counters to both the wedge constructions and the information highways, it should be possible to ensure room for everything, and to prove the same theorem, using agents with an unvarying connection length in \( \mathbb{Z}^3 \), such as unit cubes. However, we leave that for future work.

G Proof of Theorem 9

The proof of this theorem involves the use of local synchronizer ALPHA, which we now describe. (See [2] for a textbook introduction to local synchronizers, including ALPHA.) ALPHA is a local algorithm, which for each node in the network graph executes the following two steps, so neighboring processors can simulate acting in lockstep at a given synchronous round \( i \):

1. Send message to all neighbors, include round information \( i \) and actual data of round in \( i \), if any.

2. Wait for message of round \( i \) from all neighbors, then proceed to next round.

We have written this as though \( i \) increases without bound. However, for any finite network graph, it is possible to execute ALPHA correctly such that there is some constant \( k \) so the counter for \( i \) will never exceed \( k \) (it will revert back to 0, say, and start counting upward again). (This is a standard distributed computing technique.) If \( E \) is the number of edges of the network graph, the total number of messages ALPHA will send is \( O(|E|) \). However, let us consider how many messages are sent by an individual processor. If ALPHA is aiding a system to simulate synchrony for an algorithm that terminates in \( r \) rounds, then each processor will send \( 2^r \) messages to each of its neighbors on account of ALPHA: one message to announce the beginning of each round, and one message to announce that the simulation of that round has been safely completed.

Our interest is the limitation of the surface cost of the overall computation we are simulating. We know that each wedge construction will build \( r \) rows to simulate execution of the main algorithm. We can hardcode into each wedge construction an ALPHA row that is built immediately before the row that checks the inbuffer. This new ALPHA row sends a message to each neighbor, alternating between sending the current value of the round counter, or sending the “safe termination” message.

**Theorem 17** (Restatement of Theorem 9). There exists a constant \( k \) such that the following holds. Let \( M \) be a system of distributed processors of constant-size message complexity, such that \( M \) runs algorithm \( A \) and always terminates, whose network graph is a growth-bounded graph, and such that \( M \) contains no more nodes than there are atoms in the physical universe. Define algorithm \( B \) as, “Run algorithm \( A \), then compute a Maximum Independent Set of the network,” and define \( M' \) as a system with the same network graph as \( M \) that executes \( B \). Let \( G \) be a graph assembly system...
that simulates $\mathcal{M}$ (simulating processors independently), whose surface cost for $A$ is minimal, and is contained in a rectangular solid of dimensions $l, w, h$. Then there is a graph assembly system $G'$ that simulates $\mathcal{M}'$ (simulating processors independently), whose surface cost is contained in a solid of dimensions $l + k, w + k, h + k$.

Proof. We modify our method of simulation so that (1) processors can have unique ID’s, and (2) a processor can terminate one algorithm and then call another. We hardcode into the initial seed assembly that each wedge construction simulating a processor propagates a unique ID throughout its construction. (As mentioned in the main body of the paper, we encode this by making the initial “input” to each wedge construction an ID string concatenated to a separator concatenated to the processor’s input string.) Then, when a processor terminates, we simulate this with a special agent that announces termination, and we ensure the agents propagating the processor ID bind near the agent announcing termination.

Let $G$ be a graph assembly system that simulates $\mathcal{M}$ in $\mathbb{Z}^3$ (simulating processors independently, in the manner described in the previous paragraph), such that the (worst-case) rectangular surface cost of $G$ simulating $A$ is minimal for such simulations. We extend $G$ to graph assembly system $G'$ that builds from the halting tiles and the ID’s, to simulate Schneider and Wattenhofer’s MIS algorithm, locally synchronized by ALPHA. The existence of such a $G'$ is guaranteed by modifying the proof of Theorem 8 to allow for propagation of processor ID’s.

Now we consider the rectangular surface cost of worst-case behavior of $G'$. The MIS algorithm has log-star time and message complexity. However, we are assuming the number of nodes in the network graph is well below $2^{65536}$, so we can approximate the log-star function with a constant function, and assume that the MIS algorithm executes in constant time, and sends constant-many messages, regardless of the size of the network graph. Also, as a practical matter, this constant will be small, like 5. As a result, there is some constant $c$ such that each wedge construction simulating a processor will build at most $c$ additional rows, and each information highway will need to build at most $c$ additional rows as well. Since each wedge construction, and each information highway, builds in a distinct plane, the surface cost of the worst-case terminal assembly of $G'$ is at most $c$ larger than $G$, when measured in any direction (vector) of $\mathbb{Z}^3$.

If we assume that positive and negative vectors along the $x$-, $y$- and $z$-axes each is increased by $c$, then the total rectangular surface cost of the computation simulated by $G'$ is $k = 2c$ greater in each direction than the surface cost of the computation simulated by $G$. As the existence of such a $k$ is what we set out to prove, we are done.