Computer Science and Artificial Intelligence Laboratory
Technical Report

MIT-CSAIL-TR-2007-030 June 1, 2007

Amorphous Computing

Hal Abelson, Jacob Beal, and Gerald Jay Sussman

Amorphous Computing

Hal Abelson, Jacob Beal, Gerald Jay Sussman
Computer Science and Artificial Intelligence Laboratory, MIT

Contents

Glossary

Amorphous computer:
A collection of computational particles dispersed irregularly on a surface or throughout a volume, where individual particles have no *a priori* knowledge of their positions or orientations.

Computational Particle:
A (possibly faulty) individual device for an amorphous computer. Each particle has modest computing power and a modest amount of memory. The particles are not synchronized, although they are all capable of

operating at similar speeds, since they are fabricated by the same process. All particles are programmed identically, although each particle has means for storing local state and for generating random numbers.

Field:
A function assigning a value to every particle in an amorphous computer.

Gradient:
A basic amorphous computing primitive that estimates the distance from each particle to the nearest particle designated as a source of the gradient.

I Definition of the Subject and Its Importance

The goal of amorphous computing is to identify organizational principles and create programming technologies for obtaining intentional, pre-specified behavior from the cooperation of myriad unreliable parts that are arranged in unknown, irregular, and time-varying ways. The heightened relevance of amorphous computing today stems from the emergence of new technologies that could serve as substrates for information processing systems of immense power at unprecedentedly low cost, if only we could master the challenge of programming them.

II Introduction

Even as the foundations of computer science were being laid, researchers could hardly help noticing the contrast between the robustness of natural organisms and the fragility of the new computing devices. As John von Neumann remarked in 1948 [56]:

> "With our artificial automata we are moving much more in the dark than nature appears to be with its organisms. We are, and apparently, at least at present, have to be much more 'scared' by the occurrence of an isolated error and by the malfunction which must be behind it. Our behavior is clearly that of overcaution, generated by ignorance."

Amorphous computing emerged as a field in the mid-1990s, from the convergence of three factors:

- inspiration from the cellular automata models for fundamental physics [37, 16]
- hope that understanding the robustness of biological development could both help overcome the brittleness typical of computer systems and also illuminate the mechanisms of developmental biology.
- the prospect of nearly free computers in vast quantities

Microfabrication

One technology that has come to fruition over the past decade is micro-mechanical electronic component manufacture, which integrates logic circuits, micro-sensors, actuators, and communication on a single chip. Aggregates of these can be manufactured extremely inexpensively, provided that not all the chips need work correctly, and that there is no need to arrange the chips into precise geometrical configurations or to establish precise interconnections among them. A decade ago, researchers envisioned *smart dust* elements small enough to be borne on air currents to form clouds of communicating sensor particles [29].

Airborne sensor clouds are still a dream, but networks of millimeter-scale particles are now commercially available for environmental monitoring applications [43]. With low enough manufacturing costs, we could mix such particles into bulk materials to form coatings like "smart paint" that can sense data and communicate its actions to the outside world. A smart paint coating on a wall could sense vibrations, monitor the premises for intruders, or cancel noise. Bridges or buildings coated with smart paint could report on traffic and wind loads and monitor structural integrity. If the particles have actuators, then the paint could even heal small cracks by shifting the material around. Making the particles mobile opens up entire new classes of applications that are beginning to be explored by research in *swarm robotics* [38] and *modular robotics* [52].

Cellular engineering

The second disruptive technology that motivates the study of amorphous computing is microbiology. Biological organisms have served as motivating metaphors for computing since the days of calculating engines, but it is only over the past decade that we have begun to see how biology could literally be a substrate for computing, through the possibility of constructing digital-logic circuits within individual living cells. In one technology logic signals are represented not by electrical voltages and currents, but by concentrations of DNA binding proteins, and logic elements are realized as binding sites where proteins interact through promotion and repression. As a simple example, if A and B are proteins whose concentrations represent logic levels, then an "inverter" can be implemented in DNA as a genetic unit through which A serves as a repressor that blocks the production of B. [59, 58, 57, 32] Since cells can reproduce themselves and obtain energy from their environment, the resulting information processing units could be manufactured in bulk at a very low cost.

There is beginning to take shape a technology of cellular engineering that can tailor-make programmable cells to function as sensors or delivery vehicles for pharmaceuticals, or even as chemical factories for the assembly of nanoscale structures. Researchers in this emerging field of *synthetic biology* are starting to assemble registries of standard logical components implemented in DNA that can be inserted into *E. coli* [51]. The components have been engineered to permit standard means of combining them, so that biological logic designers can assemble circuits in a mix-and-match way, similar to how electrical logic designers create circuits from standard TTL parts. There's even an *International Genetically Engineered Machine Competition* where student teams from universities around the world compete to create novel biological devices from parts in the registry [27].

Either of these technologies—microfabricated particles or engineered cells—provides a path to cheaply fabricate aggregates of massive numbers of computing elements. But harnessing these for computing is quite a different matter, because the aggregates are unstructured. Digital computers have always been constructed to behave as precise arrangements of reliable parts, and almost all techniques for organizing computations depend upon this precision and reliability. Amorphous computing seeks to discover new programming methods that do not require precise control over the interaction or arrangement of the individual computing elements and to instantiate these techniques in new programming languages.

III The Amorphous Computing Model

Amorphous computing models the salient features of an unstructured aggregate through the notion of an *amorphous computer*, a collection of *computational particles* dispersed irregularly on a surface or throughout a volume, where individual particles have no *a priori* knowledge of their positions or orientations. The particles are possibly faulty, may contain sensors and effect actions, and in some applications might be mobile. Each particle has modest computing power and a modest amount of memory. The particles are not synchronized, although they are all capable of operating at similar speeds, since they are fabricated by the same process. All particles are programmed identically, although each particle has means for storing local state and for generating random numbers. There may also be several distinguished particles that have been initialized to particular states.

Each particle can communicate with a few nearby neighbors. In an electronic amorphous computer the particles might communicate via short-distance radio, whereas bioengineered cells might communicate by chemical signals. Although the details of the communication model can vary, the maximum distance over which two particles can communicate effectively is assumed to be small compared with the size of the entire amorphous computer. Communication is assumed to be unreliable and a sender has no assurance that a message has been received.[1]

We assume that the number of particles may be very large (on the order of 10^6 to 10^{12}). Algorithms appropriate to run on an amorphous computer should be relatively independent of the number of particles: the performance should degrade gracefully as the number of particles decreases. Thus, the entire amorphous computer can be regarded as a massively parallel computing system, and previous investigations into massively parallel computing, such as research in cellular automata, are one source of ideas for dealing with amorphous computers. However, amorphous computing differs from investigations into cellular automata, because amorphous mechanisms must be independent of the detailed configuration, reliability, and synchronization of the particles.

[1]Higher-level protocols with message acknowledgement can be built on such unreliable channels.

IV Programming Amorphous Systems

A central theme in amorphous computing is the search for programming paradigms that work within the amorphous model. Here, biology has been a rich source of metaphors for inspiring new programming techniques. In embryonic development, even though the precise arrangements and numbers of the individual cells are highly variable, the genetic "programs" coded in DNA nevertheless produce well-defined intricate shapes and precise forms. Amorphous computers should be able to achieve similar resuls.

One technique for programming an amorphous computer uses diffusion. One particle (chosen by some symmetry-breaking process) broadcasts a message. This message is received by each of its neighbors, which propagate it to their neighbors, and so on, to create a wave that spreads throughout the system. The message contains a count, and each particle stores the received count and increments it before re-broadcasting. Once a particle has stored its count, it stops re-broadcasting and ignores future count messages. This *count-up wave* gives each particle a rough measure of its distance from the original source. One can also produce regions of controlled size, by having the count message relayed only if the count is below a designated bound.

Two such count-up waves can be combined to identify a chain of particles between two given particles A and B. Particle A begins by generating a count-up wave as above. This time, however, each intermediate particle, when it receives its count, performs a handshake to identify its "predecessor"—the particle from which it received the count (and whose own count will therefore be one less). When the wave of count messages reaches B, B sends a "successor" message, informing its predecessor that it should become part of the chain and should send a message to *its* predecessor, and so on, all the way back to A. Note that this method works, even though the particles are irregularly distributed, provided there is a path from A to B.

The motivating metaphor for these two programs is chemical gradient diffusion, which is a foundational mechanism in biological development [3]. In nature, biological mechanisms not only generate elaborate forms, they can also maintain forms and repair them. We can modify the above amorphous line-drawing program so that it produces self-repairing line: first, particles keep re-broadcasting their count and successor messages. Second, the status of a particle as having a count or being in the chain decays over time unless it is refreshed by new messages. That is, a particle that stops hearing successor messages intended for it will eventually revert to not being in the chain and will stop broadcasting its own successor messages. A particle that stops hearing its count being broadcast will start acting as if it never had a count, pick up a new count from the messages it hears, and start broadcasting the count messages with the new count. Clement and Nagpal [15] demonstrated that this mechanism can be used to generate self-repairing lines and other patterns, and even re-route lines and patterns around "dead" regions where particles have stopped functioning.

The relationship with biology flows in the other direction as well: the amorphous algorithm for repair is a model which is not obviously inconsistent with the facts of angiogenesis in the repair of wounds. Although the existence of the algorithm has no bearing on the facts of the matter, it may stimulate systems-level thinking about models in biological research. For example, Patel et al. use amorphous ideas to analyze the growth of epithelial cells [46].

V Amorphous Computing Paradigms

Amorphous computing is still in its infancy. Most of linguistic investigations based on the amorphous computing model have been carried out in simulation. Nevertheless, this work has yielded a rich variety of programming paradigms that demonstrate that one can in fact achieve robustness in face of the unreliability of individual particles and the absence of precise organization among them.

Marker propagation for amorphous particles

Weiss's Microbial Colony Language [59] is a marker propagation language for programming the particles in an amorphous computer. The program to be executed, which is the same for each particle, is constructed as a set of rules. The state of each particle includes a set of binary markers, and rules are enabled by boolean combinations of the markers. The rules, which have the form (*trigger*, *condition*, *action*) are triggered by the receipt of labelled messages from neighboring particles. A rule may test conditions, set or clear various markers, and it broadcast further messages to its neighbors. Each message carries a count that determines how far it will diffuse, and each marker has a lifetime that determines how long its value lasts. Supporting these language's rules is a runtime system that automatically propagates messages and manages the lifetimes of markers, so that the programmer need not deal with these operations explicitly.

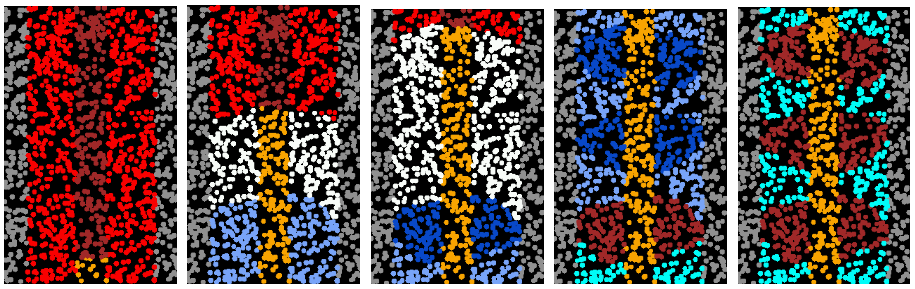


Figure 1: A Microbial Colony Language program organizes a tube into a structure similar to that of somites in the developing vertebrate. (From [59])

Weiss's system is powerful, but the level of abstraction is very low. This is because it was motivated by cellular engineering—as something that can be directly implemented by genetic regulatory networks. The language is therefore more useful as a tool set in which to implement higher-level languages such as GPL (see below), serving as a demonstration that in principle, these higher-level languages can be implemented by genetic regulatory networks as well.

Figure 1 shows an example simulation programmed in this language, that organizes an initially undifferentiated column of particles into a structure with band of two alternating colors: a caricature of somites in developing vertebrae.

The Growing Point language

Coore's *Growing Point Language* (GPL) [18] demonstrates that an amorphous computer can be configured by a program that is common to all the computing elements to generate highly complex patterns, such as the pattern representing the interconnection structure of an arbitrary electrical circuit as shown in Figure 2.

GPL is inspired by a botanical metaphor based on growing points and tropisms. A growing point is a locus of activity in an amorphous computer. A growing point propagates through the computer by transferring its activity from one computing element to a neighbor. As a growing point passes through the computer it effects the differentiation of the behaviors of the particles it visits. Particles secrete "chemical" signals whose count-up waves

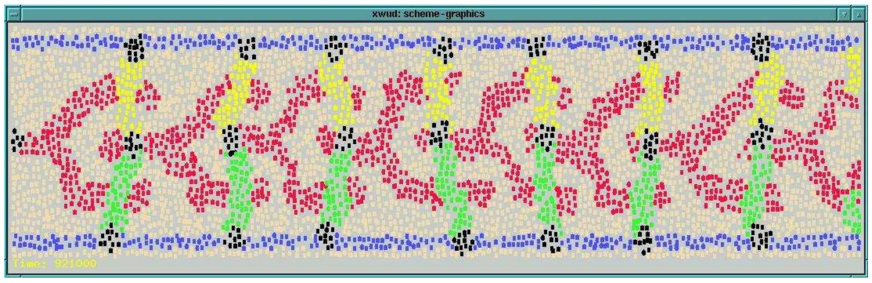


Figure 2: A pattern generated by GPL whose shape mimics a chain of CMOS inverters (from [18]).

define gradients, and these attract or repel growing points as directed by programmer-specific "tropisms. Coore demonstrated that these mechanisms are sufficient to permit amorphous computers to generate any arbitrary pre-specified graph structure pattern, up to topology. Unlike real biology, however, once a pattern has been constructed, there is no clear mechanism to maintain it in the face of changes to the material. Also, from a programming linguistic point of view, there is no clear way to compose shapes by composing growing points. More recently, Gayle and Coore have shown how GPL may be extended to produce arbitrarily large patterns such as arbitrary text strings[24]. D'Hondt and D'Hondt have explored the use of GPL for geometrical constructions and its relations with computational geometry [22, 21].

Origami-based Self-Assembly

Nagpal [41] developed a prototype model for controlling programmable materials. She showed how to organize a program to direct an amorphous sheet of deformable particles to cooperate to construct a large family of globally-specified predetermined shapes. Her method, which is inspired by the folding of epithelial tissue, allows a programmer to specify a sequence of folds, where the set of available folds is sufficient to create any origami shape (as shown by Huzita's axioms for origami [26]). Figure 3 shows a sheet of amorphous particles, where particles can cooperate to create creases and folds, assembling itself into the well-known origami "cup" structure.

Nagpal showed how this language of folds can be compiled into a a low-level program that can be distributed to all of the particles of the amorphous sheet, similar to Coore's GPL or Weiss's MCL. With a few differences of initial state (for example, particles at the edges of the sheet know that they

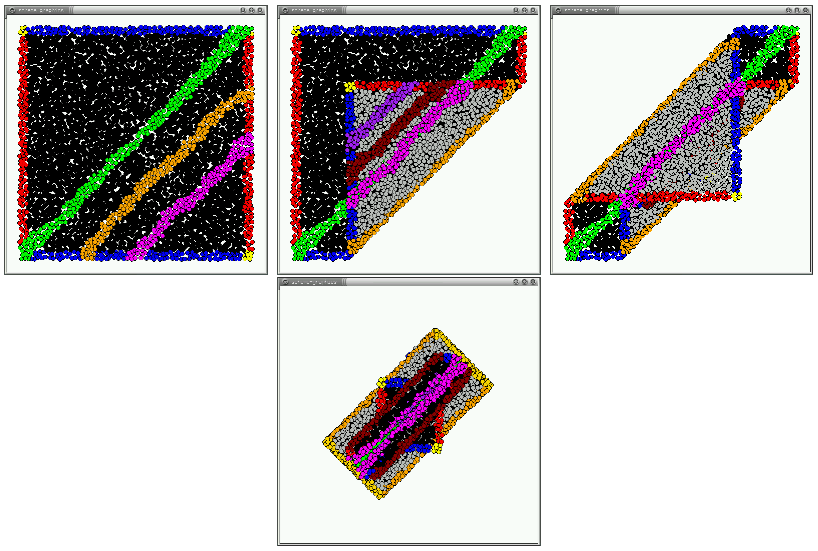


Figure 3: Folding an envelope structure (from [41]). A pattern of lines is constructed according to origami axioms. Elements then coordinate to fold the sheet using an actuation model based on epithelial cell morphogenesis. In the figure, black indicates the front side of the sheet, grey indicates the back side, and the various colored bands show the folds and creases that are generated by the amorphous process. The small white spots show gaps in the sheet cause by “dead” or missing cells—the process works despite these.

are edge particles) the particles run their copies of the program, interact with their neighbors, and fold up to make the predetermined shape. This technique is quite robust. Nagpal studied the range of shapes that can be constructed using her method, and on their sensitivity to errors of communication, random cell death, and density of the cells.

As a programming framework, the origami language has more structure than the growing point language, because the origami methods allow composition of shape constructions. On the other hand, once a shape has been constructed, there is no clear mechanism to maintain existing patterns in the face of changes to the material.

Dynamic Recruitment

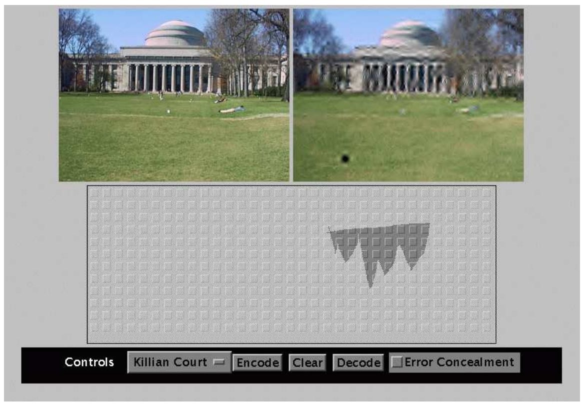
In Butera [13]'s "paintable computing," processes dynamically recruit computational particles from an amorphous computer to implement their goals. As one of his examples Butera uses dynamic recruitment to implement a robust storage system for streaming audio and images (Figure 4). Fragments of the image and audio stream circulate freely through the amorphous computer and are marshaled to a port when needed. The audio fragments also sort themselves into a playable stream as they migrate to the port.

To enhance robustness, there are multiple copies of the lower resolution image fragments and fewer copies of the higher resolution image fragments. Thus, the image is hard to destroy; with lost fragments the image is degraded but not destroyed.

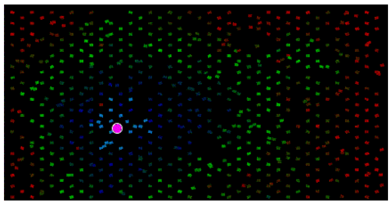
Clement and Nagpal [15] also use dynamic recruitment in the development of active gradients, as described below.

Growth and Regeneration

Kondacs [33] showed how to synthesize arbitrary two-dimensional shapes by growing them. These computing units, or "cells," are identically-programmed and decentralized, with the ability to engage in only limited, local communication. Growth is implemented by allowing cells to multiply. Each of his cells may create a child cell and place it randomly within a ring around the mother cell. Cells may also die, a fact which Kondacs puts to use for temporary scaffolding when building complex shapes. If a structure requires construction of a narrow neck between two other structures it can be built precisely by laying down a thick connection and later trimming it to size.



(a) Robust Storage



(b) Self-Organization

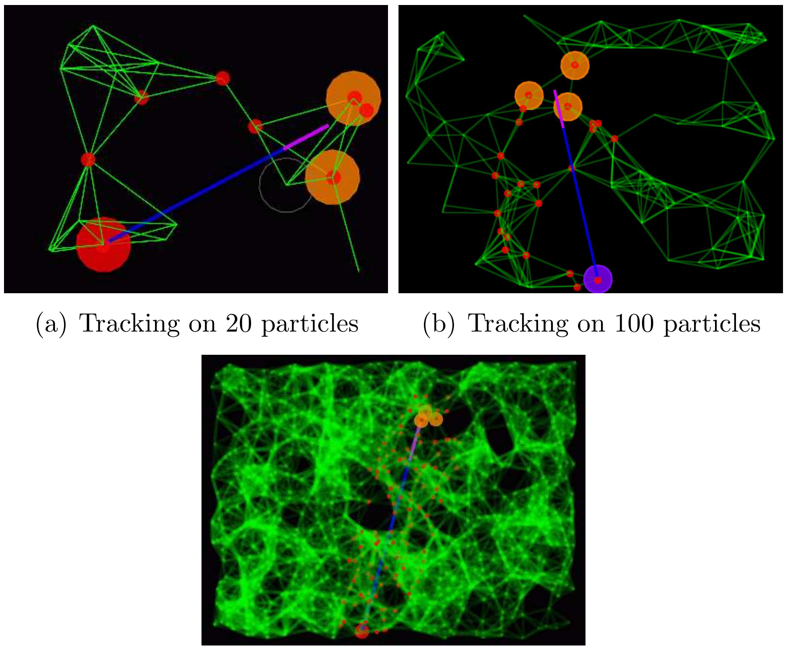
Figure 4: Butera dynamically controls the flow of information through an amorphous computer. In (a) image fragments spread through the computer so that a degraded copy can be recovered from any segment; the original image is on the left, the blurry copy on the right has been recovered from the small region shown below. In (b) audio fragments sort themselves into a playable stream as they migrate to an output port; cooler colors are earlier times. (From [13])

Attributes of this system include scalability, robustness, and the ability for self-repair. Just as a starfish can regenerate its entire body from part of a limb, his system can self-repair in the event of agent death: his sphere-network representation allows the structure to be grown starting from any sphere, and every cell contains all necessary information for reproducing the missing structure.

Abstraction to Continuous Space and Time

The amorphous model postulates computing particles distributed throughout a space. If the particles are dense, one can imagine the particles as actually filling the space, and create programming abstractions that view the space itself as the object being programmed, rather than the collection of particles. Beal and Bachrach [10, 4] pursued this approach by creating a language, Proto, where programmers specify the behavior of an amorphous computer as though it were a continuous material filling the space it occupies. Proto programs manipulate fields of values spanning the entire space. Programming primitives are designed to make it simple to compile global operations to operations at each point of the continuum. These operations are approximated by having each device represent a nearby chunk of space. Programs are specified in space and time units that are independent of the distribution of particles and of the particulars of communication and execution on those particles (Figure 5). Programs are composed functionally, and many of the details of communication and composition are made implicit by Proto's runtime system, allowing complex programs to be expressed simply. Proto has been applied to applications in sensor networks like target tracking and threat avoidance, to swarm robotics and to modular robotics, e.g., generating a planar wave for coordinated actuation.

Newton's language Regiment [45, 44] also takes a continuous view of space and time. Regiment is organized in terms of stream operations, where each stream represents a time-varying quantity over a part of space, for example, the average value of the temperature over a disc of a given radius centered at a designated point. Regiment, also a functional language, is designed to gather streams of data from regions of the amorphous computer and accumulate them at a single point. This assumption allows Regiment to provide region-wide summary functions that are difficult to implement in Proto.


(a) Tracking on 20 particles (b) Tracking on 100 particles

(c) Tracking on 1000 particles

Figure 5: A tracking program written in Proto sends the location of a target region (orange) to a listener (red) along a channel (small red dots) in the network (indicated by green lines). The continuous space and time abstraction allows the same program to run at different resolutions.

VI Primitives for Amorphous Computing

The previous section illustrated some paradigms that have been developed for programming amorphous systems, each paradigm building on some organizing metaphor. But eventually, meeting the challenge of amorphous systems will require a more comprehensive linguistic framework. We can approach the task of creating such a framework following the perspective in [2], which views languages in terms of primitives, means of combination, and means of abstraction.

The fact that amorphous computers consist of vast numbers of unreliable and unsynchronized particles, arranged in space in ways that are locally unknown, constrains the primitive mechanisms available for organizing cooperation among the particles. While amorphous computers are naturally massively parallel, the kind of computation that they are most suited for is parallelism that does not depend on explicit synchronization and the use of atomic operations to control concurrent access to resources. However, there are large classes of useful behaviors that can be implemented without these tools. Primitive mechanisms that are appropriate for specifying behavior on amorphous computers include gossip, random choice, fields, and gradients.

Gossip

Gossip, also known as epidemic communication [20, 23], is a simple communication mechanism. The goal of a gossip computation is to obtain an agreement about the value of some parameter. Each particle broadcasts its opinion of the parameter to its neighbors, and computation is performed by each particle combining the values that it receives from its neighbors, without consideration of the identification of the source. If the computation changes a particle's opinion of the value, it rebroadcasts its new opinion. The process concludes when the are no further broadcasts.

For example, an aggregate can agree upon the minimum of the values held by all the particles as follows. Each particle broadcasts its value. Each recipient compares its current value with the value that it receives. If the received value is smaller than its current value, it changes its current value to that minimum and rebroadcasts the new value.

The advantage of gossip is that it flows in all directions and is very difficult to disrupt. The disadvantage is that the lack of source information makes it difficult to revise a decision.

Random Choice

Random choice is used to break symmetry, allowing the particles to differentiate their behavior. The simplest use of random choice is to establish local identity of particles: each particle chooses a random number to identify itself to its neighbors. If the number of possible choices is large enough, then it is unlikely that any nearby particles will choose the same number, and this number can thus be used as an identifier for the particle to its neighbors. Random choice can be combined with gossip to elect leaders, either for the entire system or for local regions.[2]

To elect a single leader for the entire system, every particle chooses a value, then gossips to find the minimum. The particle with the minimum value becomes the leader. To elect regional leaders, we instead use gossip to carry the identity of the first leader a particle has heard of. Each particle uses random choice as a "coin flip" to decide when to declare itself a leader; if the flip comes up heads enough times before the particle hears of another leader, the particle declares itself a leader and broadcasts that fact to its neighbors. The entire system is thus broken up into contiguous domains of particles who first heard some particular particle declare itself a leader.

One challenge in using random choice on an amorphous computer is to ensure that the particulars of particle distribution do not have an unexpected effect on the outcome. For example, if we wish to control the expected size of the domain that each regional leader presides over, then the probability of becoming a leader must depend on the density of the particles.

Fields

Every component of the state of the computational particles in an amorphous computer may be thought of as a field over the discrete space occupied by those particles. If the density of particles is large enough this field of values may be thought of as an approximation of a field on the continuous space.

We can make amorphous models that approximate the solutions of the classical partial-differential equations of physics, given appropriate boundary conditions. The amorphous methods can be shown to be consistent,

[2]If collisions in choice can be detected, then the number of choices need not be much higher than then number of neighbors. Also, using gossip to elect leaders makes sense only when we expect a leader to be long-lived, due to the difficulty of changing the decision to designate a replacement leader.

convergent and stable.

For example, the algorithm for solving the Laplace equation with Dirichlet conditions is analogous to the way it would be solved on a lattice. Each particle must repeatedly update the value of the solution to be the average of the solutions posted by its neighbors, but the boundary points must not change their values. This algorithm will eventually converge, although very slowly, independent of the order of the updates and the details of the local connectedness of the network. There are optimizations, such as over-relaxation, that are just as applicable in the amorphous context as on a regular grid.

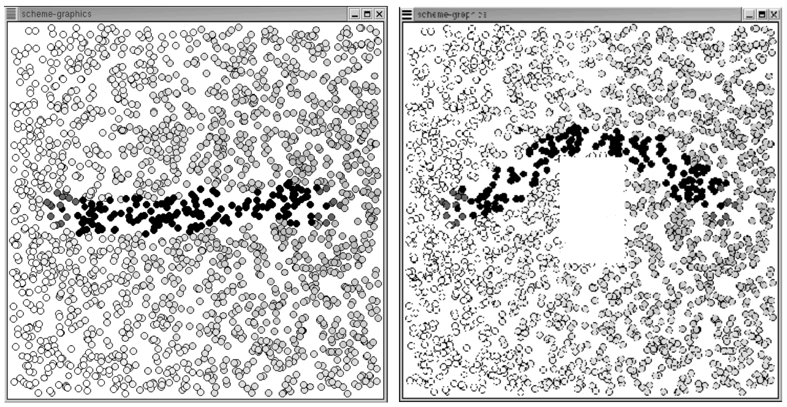
Katzenelson [30] has shown similar results for the diffusion equation, complete with analytic estimates of the errors that arise from the discrete and irregularly connected network. In the diffusion equation there is a conserved quantity, the amount of material diffusing. Rauch [50] has shown how this can work with the wave equation, illustrating that systems that conserve energy and momentum can also be effectively modeled with an amorphous computer. The simulation of the wave equation does require that the communicating particles know their relative positions, but it is not hard to establish local coordinate systems.

Gradients

An important primitive in amorphous computing is the gradient, which estimates the distance from each particle to the nearest particle designated as a source. The gradient is inspired by the chemical-gradient diffusion process that is crucial to biological development. Amorphous computing builds on this idea, but does not necessarily compute the distance using diffusion because simulating diffusion can be expensive.

The common alternative is a linear-time mechanism that depends on active computation and relaying of information rather than passive diffusion. Calculation of a gradient starts with each source particle setting its distance estimate to zero, and every other particle setting its distance estimate to infinity. The sources then broadcast their estimates to their neighbors. When a particle receives a message from its neighbor, it compares its current distance estimate to the distance through its neighbor. If the distance through its neighbor is less, it chooses that to be its estimate, and broadcasts its new estimate onwards.

Although the basic form of the gradient is simple, there are several ways in which gradients can be varied to better match the context in which they



(a) Before Failure (b) After Repair

Figure 6: A line being maintained by active gradients, from [15]. A line (black) is constructed between two anchor regions (dark grey) based on the active gradient emitted by the right anchor region (light grays). The line is able to rapidly repair itself following failures because the gradient actively maintains itself.

are used. These choices may be made largely independently, giving a wide variety of options when designing a system. Variations which have been explored include:

Active Gradients An active gradient [17, 19, 15] monitors its validity in the face of changing sources and device failure, and maintains correct distance values. For example, if the supporting sources disappear, the gradient is deallocated. A gradient may also carry version information, allowing its source to change more smoothly.

Active gradients can provide self-repairing coordinate systems, as a foundation for robust construction of patterns. Figure 6 shows the "count-down wave" line described above in the introduction to this article. The line's implementation in terms of active gradients provides for self-repair when the underlying amorphous computer is damaged.

Polarity A gradient may be set to count down from a positive value at the source, rather than to count up from zero. This bounds the distance that the gradient can span, which can help limit resource usage, but may limit the scalability of programs.

Adaptivity As described above, a gradient relaxes once to a distance estimate. If communication is expensive and precision unimportant, the gradient can take the first value that arrives and ignore all subsequent values. If we want the gradient to adapt to changes in the distribution of particles or sources, then the particles need to broadcast at regular intervals. We can then have estimates that converge smoothly to precise estimates by adding a restoring force which acts opposite to the relaxation, allowing the gradient to rise when unconstrained by its neighbors. If, on the other hand, we value adaptation speed over smoothness, then each particle can recalculate its distance estimate from scratch with each new batch of values.

Carrier Normally, the distance value calculated by the gradient is the signal we are interested in. A gradient may instead be used to carry an arbitrary signal outward from the source. In this case, the value at each particle is the most recently arrived value from the nearest source.

Distance Measure A gradient's distance measure is, of course, dependent on how much knowledge we have about the relative positions of neighbors. It is sometimes advantageous to discard good information and use only hop-count values, since it is easier to make an adaptive gradient using hop-count values. Non-linear distance measures are also possible, such as a count-down gradient that decays exponentially from the source. Finally, the value of a gradient may depend on more sources than the nearest (this is the case for a chemical gradient), though this may be very expensive to calculate.

Coordinates and Clusters

Computational particles may be built with restrictions about what can be known about local geometry. A particle may know that it can reliably communicate with a few neighbors. If we assume that these neighbors are all within a disc of some approximate communication radius then distances to others may be estimated by minimum hop count [31]. However, it is possible that more elaborate particles can estimate distances to near neighbors. For example, the Cricket localization system [48] uses the fact that sound travels more slowly than radio, so the distance is estimated by the difference in time of arrival between simultaneously transmitted signals. McLurkin's swarmbots [38] use the ISIS communication system that gives bearing and range information. However, it is possible for a sufficiently dense amorphous computer to produce local coordinate systems for its particles with even the crudest method of determining distances. We can make an atlas of overlapping coordinate systems, using random symmetry breaking to make new starting baselines [6]. These coordinate systems can be combined and made consistent to form a manifold, even if the amorphous computer is not flat or simply connected.

One way to establish coordinates is to choose two initial particles that are a known distance apart. Each one serves as the source of a gradient. A pair of rectangular axes can be determined by the shortest path between them and by a bisector constructed where the two gradients are equal. These may be refined by averaging and calibrated using the known distance between the selected particles. After the axes are established, they may source new gradients that can be combined to make coordinates for the region near these axes. The coordinate system can be further refined using further averaging. Other natural coordinate constructions are bipolar elliptical. This kind of construction was pioneered by Coore [17] and Nagpal [40]. Katzenelson [30] did early work to determine the kind of accuracy that can be expected from such a construction.

Spatial clustering can be accomplished with any of a wide variety of algorithms, such as the clubs algorithm [19], LOCI [39], or persistent node partitioning [7]. Clusters can themselves be clustered, forming a hierarchical clustering of logarithmic height.

VII Means of Combination and Abstraction

A programming framework for amorphous systems requires more than primitive mechanisms. We also need suitable means of combination, so that programmers can combine behaviors to produce more complex behaviors, and means of abstraction so that the compound behaviors can be named and manipulated as units. Here are a few means of combination that have been investigated with amorphous computing.

Spatial and Temporal Sequencing

Several behaviors can be strung together in a sequence. The challenge in controlling such a sequence is to determine when one phase has completed and it is safe to move on to the next. Trigger rules can be used to detect completion locally.

In Coore's Growing Point Language [18], all of the sequencing decisions are made locally, with different growing points progressing independently. There is no difficulty of synchronization in this approach because the only time when two growing points need to agree is when they have become spatially coincident. When growing points merge, the independent processes are automatically synchronized.

Nagpal's origami language [41] has long-range operations that cannot overlap in time unless they are in non-interacting regions of the space. The implementation uses barrier synchronization to sequence the operations: when completion is detected locally, a signal is propagated throughout a marked region of the sheet, and the next operation begins after a waiting time determined by the diameter of the sheet.

With adaptive gradients, we can use the presence of an inducing signal to run an operation. When the induction signal disappears, the operation ceases and the particles begin the next operation. This allows sequencing to be triggered by the last detection of completion rather than by the first.

Pipelining

If a behavior is self-stabilizing (meaning that it converges to a correct state from any arbitrary state) then we can use it in a sequence without knowing when the previous phase completes. The evolving output of the previous phase serves as the input of this next phase, and once the preceding behavior has converged, the self-stabilizing behavior will converge as well.

If the previous phase evolves smoothly towards its final state, then by the time it has converged, the next phase may have almost converged as well, working from its partial results. For example, the coordinate system mechanism described above can be pipelined; the final coordinates are being formed even as the farther particles learn that they are not on one of the two axes.

Restriction to Spatial Regions

Because the particles of an amorphous computer are distributed in space it is natural to assign particular behaviors to specific spatial regions. In Beal and Bachrach's work, restriction of a process to a region is a primitive[10]. As another example, when Nagpal's system folds an origami construction, regions on different faces may differentiate so that they fold in different patterns. These folds may, if the physics permits, be performed simultaneously. It may be necessary to sequence later construction that depends on the completion of the substructures.

Regions of space can be named using coordinates, clustering, or implicitly through calculations on fields. Indeed, one could implement solid modelling on an amorphous computer. Once a region is identified, a particle can test whether it is a member of that region when deciding whether to run a behavior. It is also necessary to specify how a particle should change its behavior if its membership in a region may vary with time.

Modularity and Abstraction

Standard means of abstraction may be applied in an amorphous computing context, such as naming procedures, data structures, and processes. The question for amorphous computing is what collection of entities is useful to name.

Because geometry is essential in an amorphous computing context, it becomes appropriate to describe computational processes in terms of geometric entities. Thus there are new opportunities for combining geometric structures and naming the combinations. For example, it is appropriate to compute with, combine, and name regions of space, intervals of time, and fields defined on them [10, 45]. It may also be useful to describe the propagation of information through the amorphous computer in terms of the light cone of an event [5].

Not all traditional abstractions extend nicely to an amorphous computing context because of the challenges of scale and the fallibility of parts and interconnect. For example, atomic transactions may be excessively expensive in an amorphous computing context. And yet, some of the goals that a programmer might use atomic transactions to accomplish, such as the approximate enforcement of conservation laws, can be obtained using techniques that are compatible with an amorphous environment, as shown by Rauch [50].

VIII Supporting Infrastructure and Services

Amorphous computing languages, with their primitives, means of combination, and means of abstraction, rest on supporting services. One example, described above, is the automatic message propagation and decay in Weiss's

Microbial Colony Language [59]. MCL programs do not need to deal with this explicitly because it is incorporated into the operating system of the MCL machine. Experience with amorphous computing is beginning to identify other key services that amorphous machines must supply.

Particle Identity

Particles must be able to choose identifiers for communicating with their neighbors. More generally, there are many operations in an amorphous computation where the particles may need to choose numbers, with the property that individual particles choose different numbers.

If we are willing to pay the cost, it is possible to build unique identifiers into the particles, as is done with current macroscopic computers. We need only locally unique identifiers, however, so we can obtain them using pseudorandom-number generators. On the surface of it, this may seem problematic, since the particles in the amorphous are assumed to be manufactured identically, with identical programs. There are, however, ways to obtain individualized random numbers. For example, the particles are not synchronized, and they are not really physically identical, so they will run at slightly different rates. This difference is enough to allow pseudorandom-number generators to get locally out of synch and produce different sequences of numbers. Amorphous computing particles that have sensors may also get seeds for their pseudorandom-number generators from sensor noise.

Local Geometry and Gradients

Particles must maintain connections with their neighbors, tracking who they can reliably communicate with, and whatever local geometry information is available. Because particles may fail or move, this information needs to be maintained actively. The geometry information may include distance and bearing to each neighbor, as well as the time it takes to communicate with each neighbor. But many implementations will not be able to give significant distance or bearing information. Since all of this information may be obsolete or inaccurately measured, the particles must also maintain information on how reliable each piece of information is.

An amorphous computer must know the dimension of the space it occupies. This will generally be a constant—either the computer covers a surface or fills a volume. In rare cases, however, the effective dimension of a computer may change: for example, paint is three-dimensional in a bucket and two-dimensional once applied. Combining this information with how the number of accessible correspondents changes with distance, an amorphous process can derive curvature and local density information.

An amorphous computer should also support gradient propagation as part of the infrastructure: a programmer should not have to explicitly deal with the propagation of gradients (or other broadcast communications) in each particle. A process may explicitly initiate a gradient, or explicitly react to one that it is interested in, but the propagation of the gradient through a particle should be automatically maintained by the infrastructure.

Implementing Communication

Communication between neighbors can occur through any number of mechanisms, each with its own set of properties: amorphous computing systems have been built that communicate through directional infrared [38], RF broadcast [25], and low-speed serial cables [49, 12]. Simulated systems have also included other mechanisms such as signals superimposed on the power connections [14] and chemical diffusion [59].

Communication between particles can be made implicit with a neighborhood shared memory. In this arrangement, each particle designates some of its internal state to be shared with its neighbors. The particles regularly communicate, giving each particle a best-effort view of the exposed portions of the states of its neighbors. The contents of the exposed state may be specified explicitly [13, 38, 62] or implicitly [10]. The shared memory allows the system to be tuned by trading off communication rate against the quality of the synchronization, and decreasing transmission rates when the exposed state is not changing.

IX Lessons for Engineering

As von Neumann remarked half a century ago, biological systems are strikingly robust when compared with our artificial systems. Even today software is fragile. Computer science is currently built on a foundation that largely assumes the existence of a perfect infrastructure. Integrated circuits are fabricated in clean-room environments, tested deterministically, and discarded if even a single defect is uncovered. Entire software systems fail with single-line errors. In contrast, biological systems rely on local computation, local communication, and local state, yet they exhibit tremendous resilience.

Although this contrast is most striking in computer science, amorphous computing can provide lessons throughout engineering. Amorphous computing concentrates on making systems flexible and adaptable at the expense of efficiency. Amorphous computing requires an engineer to work under extreme conditions. The engineer must arrange the cooperation of vast numbers of identical computational particles to accomplish prespecified goals, but may not depend upon the numbers. We may not depend on any prespecified interconnect of the particles. We may not depend on synchronization of the particles. We may not depend on the stability of the communications system. We may not depend on the long-term survival of any individual particles. The combination of these obstacles forces us to abandon many of the comforts that are available in more typical engineering domains.

By restricting ourselves in this way we obtain some robustness and flexibility, at the cost of potentially inefficient use of resources, because the algorithms that are appropriate are ones that do not take advantage of these assumptions. Algorithms that work well in an amorphous context depend on the average behavior of participating particles. For example, in Nagpal's origami system a fold that is specified will be satisfactory if it is approximately in the right place and if most of the particles on the specified fold line agree that they are part of the fold line: dissenters will be overruled by the majority. In Proto a programmer can address only regions of space, assumed to be populated by many particles. The programmer may not address individual particles, so failures of individual particles are unlikely to make major perturbations to the behavior of the system.

An amorphous computation can be quite immune to details of the macroscopic geometry as well as to the interconnectedness of the particles. Since amorphous computations make their own local coordinate systems, they are relatively independent of coordinate distortions. In an amorphous computation we accept a wide range of outcomes that arise from variations of the local geometry. Tolerance of local variation can lead to surprising flexibility: the mechanisms which allow Nagpal's origami language to tolerate local distortions allow programs to distort globally as well, and Nagpal shows how such variations can account for the variations in the head shapes of related species of *Drosophila* [41]. In Coore's language one specifies the topology of the pattern to be constructed, but only limited information about the geometry. The topology will be obtained, regardless of the local geometry, so long as there is sufficient density of particles to support the topology. Amorphous computations based on a continuous model of space (as in Proto) are naturally scale independent.

Since an amorphous computer is composed of un-synchronized particles, a program may not depend upon *a priori* timing of events. The sequencing of phases of a process must be determined by either explicit termination signals or with times measured dynamically. So amorphous computations are time-scale independent by construction.

A program for an amorphous computer may not depend on the reliability of the particles or the communication paths. As a consequence it is necessary to construct the program so as to dynamically compensate for failures. One way to do this is to specify the result as the satisfaction of a set of constraints, and to build the program as a homeostatic mechanism that continually drives the system toward satisfaction of those constraints. For example, an active gradient continually maintains each particle's estimate of the distance to the source of the gradient. This can be used to establish and maintain connections in the face of failures of particles or relocation of the source. If a system is specified in this way, repair after injury is a continuation of the development process: an injury causes some constraints to become unsatisfied, and the development process builds new structure to heal the injury.

By restricting the assumptions that a programmer can rely upon we increase the flexibility and reliability of the programs that are constructed. However, it is not yet clear how this limits the range of possible applications of amorphous computing.

X Future Directions

Computer hardware is almost free, and in the future it will continue to decrease in price and size. Sensors and actuators are improving as well. Future systems will have vast numbers of computing mechansisms with integrated sensors and actuators, to a degree that outstrips our current approaches to system design. When the numbers become large enough, the appropriate programming technology will be amorphous computing. This transition has already begun to appear in several fields:

- **Sensor Networks:** The success of sensor network research has encouraged the planning and deployment of ever-larger numbers of devices.

The ad-hoc, time-varying nature of sensor networks has encouraged amorphous approaches, such as communication through directed diffusion [28] and Newton's Regiment language [45].

- **Robotics:** Multi-agent robotics is much like sensor networks, except that the devices are mobile and have actuators. *Swarm robotics* considers independently mobile robots working together as a team like ants or bees, while *modular robotics* consider robots that physically attach to one another in order to make shapes or perform actions, working together like the cells of an organism. Gradients are being used to create "flocking" behaviors in swarm robotics [47, 38]. In modular robotics, Stoy uses gradients to create shapes [54] while De Rosa *et al.* form shapes through stochastic growth and decay [52].

- **Pervasive Computing:** Pervasive computing seeks to exploit the rapid proliferation of wireless computing devices throughout our everyday environment. Mamei and Zambonelli's TOTA system [35] is an amorphous computing implementation supporting a model of programming using fields and gradients [36]. Servat and Drogoul have suggested combining amorphous computing and reactive agent-based systems to produce something they call "pervasive intelligence" [53].

- **Multicore Processors:** As it becomes more difficult to increase processor speed, chip manufacturers are looking for performance gains through increasing the number of processing cores per chip. Butera's work [13] looks toward a future in which there are thousands of cores per chip and it is no longer reasonable to assume they are all working or have them communicate all-to-all.

While much of amorphous computing research is inspired by biological observations, it is also likely that insights and lessons learned from programming amorphous computers will help elucidate some biological problems [46]. Some of this will be stimulated by the emerging engineering of biological systems. Current work in synthetic biology [57, 51, 27] is centered on controlling the molecular biology of cells. Soon synthetic biologists will begin to engineer biofilms and perhaps direct the construction of multicellular organs, where amorphous computing will become an essential technological tool.

References

[1] H. Abelson, D. Allen, D. Coore, C. Hanson, G. Homsy, T. Knight, R. Nagpal, E. Rauch, G. Sussman, and R. Weiss. Amorphous computing. Technical Report AIM-1665, MIT, 1999.

[2] Harold Abelson and Gerald Jay Sussman with Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 2nd edition, 1996.

[3] H. L. Ashe and J. Briscoe. The interpretation of morphogen gradients. *Development*, 133:385–94, 2006.

[4] Jonathan Bachrach and Jacob Beal. Programming a sensor network as an amorphous medium. In *DCOSS 2006 Posters*, June 2006.

[5] Jonathan Bachrach, Jacob Beal, and Takeshi Fujiwara. Continuous space-time semantics allow adaptive program execution. In *SASO 2007*, 2007. (to appear).

[6] Jonathan Bachrach, Radhika Nagpal, Michael Salib, and Howard Shrobe. Experimental results and theoretical analysis of a self-organizing global coordinate system for ad hoc sensor networks. *Telecommunications Systems Journal, Special Issue on Wireless System Networks*, 2003.

[7] Jacob Beal. A robust amorphous hierarchy from persistent nodes. In *CSN*, 2003.

[8] Jacob Beal. Programming an amorphous computational medium. In *Unconventional Programming Paradigms International Workshop*, September 2004.

[9] Jacob Beal. Amorphous medium language. In *Large-Scale Multi-Agent Systems Workshop (LSMAS)*. Held in Conjunction with AAMAS-05, 2005.

[10] Jacob Beal and Jonathan Bachrach. Infrastructure for engineered emergence on sensosr/actuator networks. *IEEE Intelligent Systems*, 2006.

[11] Jacob Beal and Gerald Sussman. Biologically-inspired robust spatial programming. Technical Report AI Memo 2005-001, MIT, January 2005.

[12] Wes Beebee. M68hc11 gunk api book. `http://www.swiss.ai.mit.edu/projects/amorphous/HC11/api.html` (visited 5/31/2007).

[13] William Butera. *Programming a Paintable Computer*. PhD thesis, MIT, 2002.

[14] Jason Campbell, Padmanabhan Pillai, and Seth Copen Goldstein. The robot is the tether: Active, adaptive power routing for modular robots with unary inter-robot connectors. In *IROS 2005*, 2005.

[15] L. Clement and R. Nagpal. Self-assembly and self-repairing topologies. In *Workshop on Adaptability in Multi-Agent Systems, RoboCup Australian Open*, January 2003.

[16] E.F. Codd. *Cellular Automata*. Academic Press, 1968.

[17] Daniel Coore. Establishing a coordinate system on an amorphous computer. In *MIT Student Workshop on High Performance Computing*, 1998.

[18] Daniel Coore. *Botanical Computing: A Developmental Approach to Generating Interconnect Topologies on an Amorphous Computer*. PhD thesis, MIT, 1999.

[19] Daniel Coore, Radhika Nagpal, and Ron Weiss. Paradigms for structure in an amorphous computer. Technical Report AI Memo 1614, MIT, 1997.

[20] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Stuygis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *7th ACM Symposium on Operating Systems Principles*, 1987.

[21] Ellie D'Hondt and Theo D'Hondt. Amorphous geometry. In *ECAL 2001*, 2001.

[22] Ellie D'Hondt and Theo D'Hondt. Experiments in amorphous geometry. In *2001 International Conference on Artificial Intelligence*, 2001.

[23] D. Ganesan, B. Krishnamachari, A. Woo, D. Culler, D. Estrin, and S. Wicker. An empirical study of epidemic algorithms in large scale multihop wireless networks. Technical Report IRB-TR-02-003, Intel Research Berkeley, 2002.

[24] Orrett Gayle and Daniel Coore. Self-organizing text in an amorphous environment. In *ICCS 2006*, 2006.

[25] J. Hill, R. Szewcyk, A. Woo, D. Culler, S. Hollar, and K. Pister. System architecture directions for networked sensors. In *ASPLOS*, November 2000.

[26] H. Huzita and B. Scimemi. The algebra of paper-folding. In *First International Meeting of Origami Science and Technology*, 1989.

[27] igem 2006: international genetically engineered machine competition, 2006. `http://www.igem2006.com` (visited 5/31/2007).

[28] Chalermek Intanagonwiwat, Ramesh Govindan, and Deborah Estrin. Directed diffusion: a scalable and robust communication paradigm for sensor networks. In *Mobile Computing and Networking*, pages 56–67, 2000.

[29] J. M. Kahn, R. H. Katz, and K. S. J. Pister. Mobile networking for smart dust. In *ACM/IEEE Intl. Conf. on Mobile Computing and Networking (MobiCom 99)*, August 1999.

[30] Jacob Katzenelson. Notes on amorphous computing. (Draft Paper), 1999.

[31] L. Kleinrock and J. Sylvester. Optimum transmission radii for packet radio networks or why six is a magic number. In *IEEE Nat'l Telecomm. Conf*, pages 4.3.1–4.3.5, December 1978.

[32] Thomas F. Knight and Gerald Jay Sussman. Cellular gate technology. In *First International Conference on Unconventional Models of Computation (UMC98)*, 1998.

[33] Attila Kondacs. Biologically-inspired self-assembly of 2d shapes, using global-to-local compilation. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 2003.

[34] Marco Mamei and Franco Zambonelli. Spray computers: Frontiers of self-organization for pervasive computing. In *WOA 2003*, 2003.

[35] Marco Mamei and Franco Zambonelli. Spatial computing: the tota approach. In *WOA 2004*, pages 126–142, 2004.

[36] Marco Mamei and Franco Zambonelli. Physical deployment of digital pheromones through rfid technology. In *AAMAS 2005*, pages 1353–1354, 2005.

[37] Norman Margolus. *Physics and Computation*. PhD thesis, MIT, 1988.

[38] James McLurkin. Stupid robot tricks: A behavior-based distributed algorithm library for programming swarms of robots. Master's thesis, MIT, 2004.

[39] Vineet Mittal, Murat Demirbas, and Anish Arora. Loci: Local clustering service for large scale wireless sensor networks. Technical Report OSU-CISRC-2/03-TR07, Ohio State University, 2003.

[40] Radhika Nagpal. Organizing a global coordinate system from local information on an amorphous computer. Technical Report AI Memo 1666, MIT, 1999.

[41] Radhika Nagpal. *Programmable Self-Assembly: Constructing Global Shape using Biologically-inspired Local Interactions and Origami Mathematics*. PhD thesis, MIT, 2001.

[42] Radhika Nagpal and Marco Mamei. Engineering amorphous computing systems. In F. Bergenti, M. P. Gleizes, and F. Zambonelli, editors, *Methodologies and Software Engineering for Agent Systems, The Agent-Oriented Software Engineering Handbook*, pages 303–320. Kluwer Academic Publishing, 2004.

[43] Dust Networks. `http://www.dust-inc.com` (visited 5/31/2007).

[44] Ryan Newton, Greg Morrisett, and Matt Welsh. The regiment macroprogramming system. In *International Conference on Information Processing in Sensor Networks (IPSN'07)*, April 2007.

[45] Ryan Newton and Matt Welsh. Region streams: Functional macroprogramming for sensor networks. In *First International Workshop on Data Management for Sensor Networks (DMSN)*, August 2004.

[46] Ankit Patel, Radhika Nagpal, Matthew Gibson, and Norbert Perrimon. The emergence of geometric order in proliferating metazoan epithelia. *Nature*, August 2006.

[47] D. Payton, M. Daily, R. Estowski, M. Howard, and C. Lee. Pheromone robotics. *Autonomous Robotics*, 11:319–324, 2001.

[48] N. Priyantha, A. Chakraborty, and H. Balakrishnan. The cricket location-support system. In *ACM International Conference on Mobile Computing and Networking (ACM MOBICOM)*, August 2000.

[49] H. Raffle, A. Parkes, and H. Ishii. Topobo: A constructive assembly system with kinetic memory. *CHI*, 2004.

[50] Erik Rauch. Discrete, amorphous physical models. Master's thesis, MIT, 1999.

[51] Registry of standard biological parts. `http://parts.mit.edu` (visited 5/31/2007).

[52] Michael De Rosa, Seth Copen Goldstein, Peter Lee, Jason Campbell, and Padmanabhan Pillai. Scalable shape sculpting via hole motion: Motion planning in lattice-constrained module robots. In *Proceedings of the 2006 IEEE International Conference on Robotics and Automation (ICRA '06)*, May 2006.

[53] D. Servat and A. Drogoul. Combining amorphous computing and reactive agent-based systems: a paradigm for pervasive intelligence? In *AAMAS 2002*, 2002.

[54] Kasper Stoy. *Emergent Control of Self-Reconfigurable Robots*. PhD thesis, University of Southern Denmark, 2003.

[55] Andrew Sutherland. Towards rseam: Resilient serial execution on amorphous machines. Master's thesis, MIT, 2003.

[56] John von Neumann. The general and logical theory of automata. In Lloyd Jeffress, editor, *Cerebral Mechanisms for Behavior*, page 16. John Wiley, 1951.

[57] R. Weiss, S. Basu, S. Hooshangi, A. Kalmbach, D. Karig, R. Mehreja, and I. Netravali. Genetic circuit building blocks for cellular computation, communications, and signal processing. *Natural Computing*, 2(1):47–84, 2003.

[58] R. Weiss and T. Knight. Engineered communications for microbial robotics. In *Sixth International Meeting on DNA Based Computers (DNA6)*, 2000.

[59] Ron Weiss. *Cellular Computation and Communications using Engineered Genetic Regular Networks*. PhD thesis, MIT, 2001.

[60] Matt Welsh and Geoff Mainland. Programming sensor networks using abstract regions. In *Proceedings of the First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI '04)*, March 2004.

[61] Justin Werfel, Yaneer Bar-Yam, and Radhika Nagpal. Building patterned structures with robot swarms. In *IJCAI*, 2005.

[62] Kamin Whitehouse, Cory Sharp, Eric Brewer, and David Culler. Hood: a neighborhood abstraction for sensor networks. In *Proceedings of the 2nd international conference on Mobile systems, applications, and services*. ACM Press, 2004.