Literally Embedded Processors

William Butera and V. Michael Bove, Jr.
MIT Media Laboratory, Cambridge MA USA

ABSTRACT

We propose a robust programming model for dense ensembles of ultra miniaturized computing nodes which are deployed in bulk fashion, *e.g.* embedded into building materials. We sketch a hardware reference model as an initial guide to the application domain, then we describe a programming model based on mobile code fragments which self assemble into larger structures. We outline a simple representative application that we have developed and tested on a system simulator.

Keywords: Media processors, self-organizing systems, networks

1. BACKGROUND

In the next few years, process technology will arrive at the point where autonomous computing elements can be scaled to the size of large sand grains and sold at bulk prices. Coupled with a commensurate shrink in the footprint of sensors and actuators, the concept of "personal computing" will take on a radically new dimension. While the details of how people relate to this ultra-commoditized form of computing remain largely conjectural, a couple of points are already apparent:

- As the computing elements become resilient to environmental stress, they will migrate off the expensive, precision engineered motherboards, and into everyday objects such as furniture, clothing and random surfaces.
- People will find it more natural to deal with computation as a bulk item, preferring to manipulate it by the jar full, the bolt, the cord, or the shot glass.

One could loosely delineate commodity level computing as those instances where the price of the computing is so low that it is comparable to powdered detergent and where the form factor is so small that it seamlessly blends into the everyday environment. As a representative embodiment, we adopt the architecture advanced by Sussman, Abelson and Knight[1] – that of ultra-miniaturized computing nodes each fitted with an on board microprocessor, 50K of memory and a wireless transceiver, all shrunk down to the size of a pin head and powered parasitically.

In the domain of interest, these nodes would be embedded in a 2D surface (such as the plywood in a table top) with a density on the order of tens of thousands per square meter. Positioning would be pseudo-random, with no local restrictions on density or regularity. Once exposed to power, they should boot and self organize their local address space. External I/O would be via physical contact with an object fitted with a transceiver whose protocols are identical to the transceivers on the chips.

In the tradition of this work, we adopt as a starting point the assertion that the hardware is in sight, if not within reach, and that the most daunting uncertainties lie in the programming model and application domains.

In the following sections, we sketch a hardware reference model for a particle IC, discuss the ramifications of the programming model, define a programming model based on self-assembling code fragments, report work done on a simulator, and use the simulator to illustrate a representative application.

2. HARDWARE REFERENCE MODEL

As an aid to definition, we adopt a hardware reference model constructed around a single IC with dimensions 2 mm x 2 mm. Onboard subsystems include a block for power harvesting, a full featured microprocessor, a wireless transceiver for inter-particle communication, a 50 MHz internal clock, and approximately 50K RAM for program and data space.

The power harvesting subsection must couple to an external power source without requiring precision connections or placement. The chip should provide some latitude in orientation and complete freedom of position within the 2D surface. A ready example would be chips whose supply pins are "vampire" taps of unequal length. When the IC is deployed, the supply pins would pierce a layered substrate where Gnd and Vcc occupy different layers.

Designs for low-power embedded microprocessors are comparatively plentiful. The most suitable for this reference design would be those that occupy less than a 2 mm square, include a modest floating point unit, no caching circuitry, minimal pipelining, and draw less than 10 milliamperes at the target clock rate of 50 MHz.

The wireless subsystem must support inter-particle communication over a radius of several centimeters. Candidate techniques include monolithic near-field RF or electrostatic loading of a resistive medium. For the reference model, we assume a minimum bandwidth of at least 100 Kbit/s full duplex.

The sources and destinations for external I/O are devices (of arbitrary size) that are fitted with wireless devices executing protocols identical to the ones on the particles. When placed in proximity with the particle ensemble, these external devices mimic the behavior of the particles and are thus integrated into the local communication network.

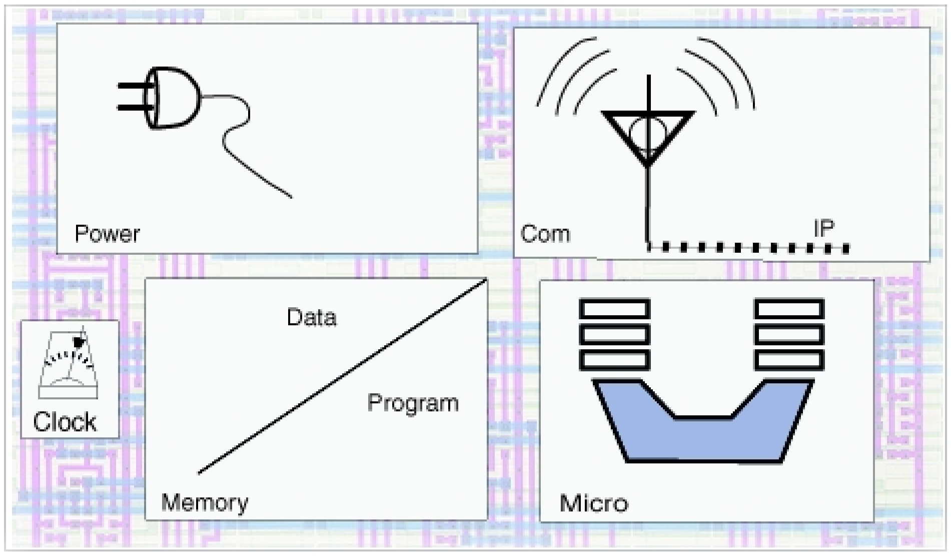

Figure 1: Reference model for a particle, including general purpose processor, power harvesting subsystem, wireless transceiver, and roughly 50K RAM.

While ill-suited as a blueprint for a product, this reference model captures the requisite characteristics for a particle IC:
1) no precision placement
2) no dedicated interconnects
3) no need to differentiate or sort particles by functionality – some particles in a mix can be pure processors, and some can contain various kinds of sensors or output devices (*e.g.* electronic ink)
4) asynchronous timing model
5) networking model based on spatial locality
6) vanishingly small unit cost ($0.002 / MIP)
7) node size on the order of a pin head

To expand on this last point, while the results reported here are applicable to systems which operate on a much larger spatial scale (for example, that of a factory floor) we specifically target the application domain where several thousand processing nodes fall within an arm's length of a human. This admittedly arbitrary niche reflects the authors' strong personal bias that this is where a large set of interesting new applications lie.

3. PROGRAMMING MODEL

An ensemble of thousands of miniature processing nodes, running asynchronously (and perhaps intermittently) and communicating locally via an *ad hoc* network places unusual demands on the programming model. Worst among these are:

- **Asynchrony:** Clock level synchrony is out of the question. Two neighboring particles cannot be guaranteed to have the same clock rate, let alone phase lock. Event level synchrony also seems beyond reach. In an unknown topology with sporadic unit failures, there is no way for a process on one particle to predict what processes will be running on a neighboring particle. Code running on one particle should never explicitly synchronize to events generated on another particle.
- **Extreme Fault Tolerance:** Allied with the inherent asynchrony is the propensity of individual particles to fail completely. A defining characteristic of the particle ensemble is that the user should be permitted certain tasks that will

remove or damage particles. For example, if the ensemble is layered into a wooden surface, the user should think nothing of driving a nail into that surface or machining it to an arbitrary shape.

- **Network Locality:** Particles can communicate directly only with other particles in the immediate spatial vicinity. While the size of the neighborhood can vary substantially, current experiments run on neighborhood sizes ranging from 8 to 20 particles.
- **Adaptive Topology:** Any ensemble of embedded particles will have final topology which is unknown at the time that much of the application code is written. While it will always be possible to recover an approximate coordinate system at run-time, no application code should rely on a particular spatial layout of the processors. As a consequence, no application code can explicitly address a processing node by location — neither as an absolute location nor as a relative location (*e.g.* two hops north).
- **Code Compactness:** On-particle memory is very limited, inter-particle bandwidth is slow compared to processor speed, and there is no external support for virtual memory. Functions running on a given particle should therefore be self-contained and sized to fit completely in a single particle.
- **Shared Data:** Nevertheless, the utility of a single particle's computation will often go up if it has access to results from local computations on neighboring particles. With the caveat that no process can predict what processes are running in the neighborhood, tagged data passed within the neighborhood should be available to processes running on a given particle.
- **Mobility:** Inter-particle migration of code segments will increase both the functionality of the individual particles and the adaptability of the overall system. The restriction here is that exact trajectory of the migrating code cannot be pre-ordained.

3.1 Self-Assembling Code

While this list reads like a compiler designer's epitaph, there is a mature body of appropriate techniques from work in distributed parallel processing[2]. However, at a density of thousands of particles per square foot, the aggregate system behavior moves into the gray region between traditional computer science and statistical mechanics. In this regime, traditional approaches to parallel computing either break down altogether or burden the hardware with an immense overhead for control.

As an alternative, we suggest *applying self-organization to construct a processing environment where application level software modules self-assemble from randomly distributed code fragments*. To contrast this approach with conventional programming practice, consider a simple function consisting of three interconnected tasks (Figure 2). Conventional programming techniques prescribe *a priori* a fixed set of predefined paths for the data flow among the tasks. Metaphorically, one could regard the tasks as blocks positioned in a static scaffolding.

Functions which self-assemble follow the alternative metaphor of organisms swimming around a medium, colliding randomly and interacting in response to chemical signatures embedded on their surface. In practice, we would approximate this by considering the machine's memory space as a fluid medium in which tagged data can be arbitrarily positioned (Figure 3). The individual subroutines would be encapsulated in an active wrapper that would support mobility, couple to the wrappers of other subroutines, and interact with the tagged data (Figure 4).

In the absence of input data, the dissipative mechanism dominates, suppressing the grouping of the code fragment, leaving them to diffuse randomly throughout the memory. The arrival of the relevant input changes the balance, naturally fostering certain groupings of the code fragments (Figure 5). This self assembly would continue until the feed-forward and dissipative mechanisms arrived at a new balance, supporting a stable grouping of the code fragments into a macro function.

To return from the metaphors back into specifics – a preliminary definition of the programming has been developed which captures much of this dynamic. This model is outlined below in three parts: the organization of the RAM space, a normative definition of the code fragments, and a description of how the two interact.

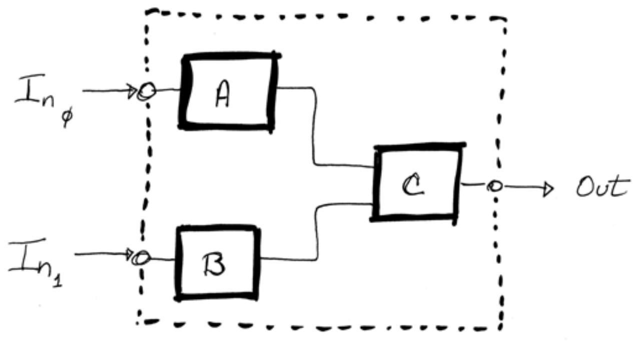

Figure 2: Traditional model: a function made up of tasks with fixed data flow.

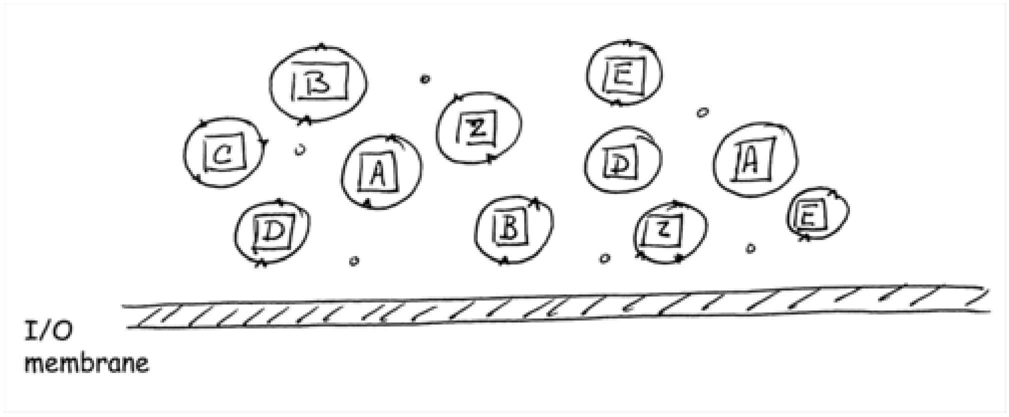

Figure 3: Mobile code fragments diffuse randomly throughout the system memory.

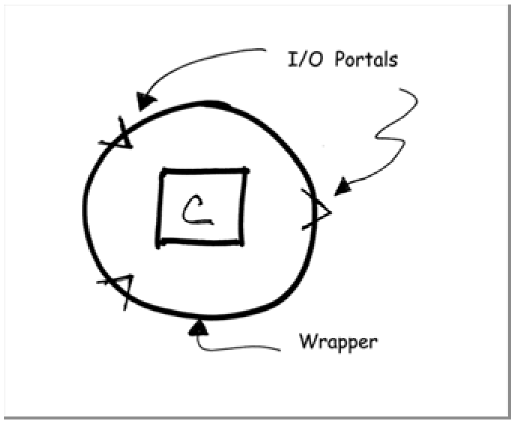

Figure 4: Individual subroutines are enveloped in a "wrapper" which supplies mobility and gates the I/O access.

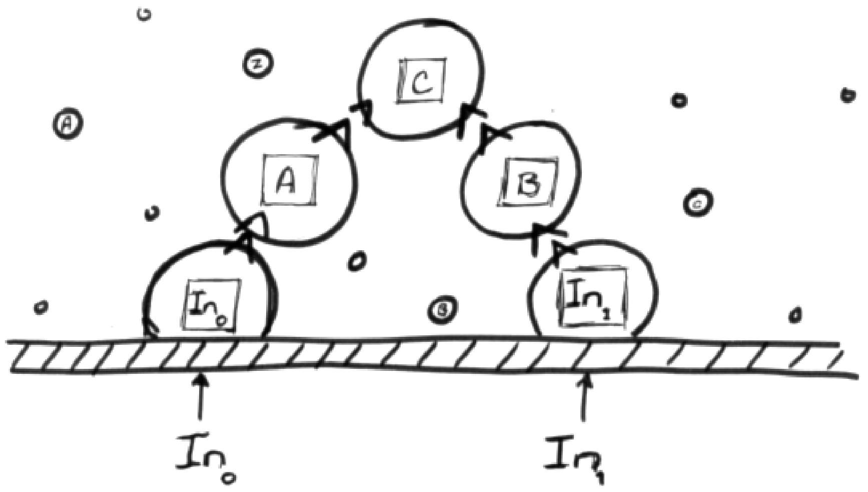

Figure 5: Input data appearing at the memory's "membrane" is enveloped in a "wrapper" and seeks to catalyze a self-organizing structure.

3.2 Memory Allocation

Programs running on a particle's microprocessor reside in the particle's RAM space. Most of the RAM is available for use as program, data, and scratch space for these programs. However, a section of the RAM is reserved what we refer to as the I/O space – an area which is at least readable by any program running on the particle's microprocessor. A subset of the I/O space is called the HomePage. The HomePage is an area where programs can both read and write tagged data. Any program local to the particle can post to the HomePage, and posts to the HomePage are readable by all local programs.

The remainder of the I/O space is subdivided into mirrored instances of the HomePages of neighboring particles. When a program on a given particle posts a piece of tagged data to the particle's HomePage, copies of that post appear at the mirror sites of all the neighboring particles. The caveat is that the latency in the mirroring operation is unconstrained.

Collectively, the I/O space functions as a public bulletin board, where the HomePage portion is writable and the entire I/O space is readable.

3.3 Code Segments

All software intended to run on a particle's microprocessor must be organized into autonomous modules – or code segments – which conform to three normative requirements:

- They are self-contained modules capable of fitting entirely in the RAM space of a single particle. Here, the phrase "self-contained" means that they do not explicitly depend on subroutines or functions which are external to the code segment.
- They gate their entire I/O through the I/O space in the particle's RAM..
- They define some support for 5 simple functions which the particle's OS can give them. These five functions are Install, De-Install, Update, Transfer-Refused, and Transfer-Granted.

A legal, if nearly useless, code segment would be one that answers a call to Install by posting a "Hello World" to the HomePage, answers the Update command by posting the text "I'm still here," answers the Transfer-Refused command by posting "Lonely Heart seeks Soulmate," responds to the Transfer-Granted command by erasing its posts and migrating to a neighboring particle, and responds to De-Install by erasing all of its posts and erasing itself.

RAM space permitting, a particle will accommodate multiple code segments simultaneously. But it is up to the particle's OS to enforce any pre-defined boundaries on RAM usage.

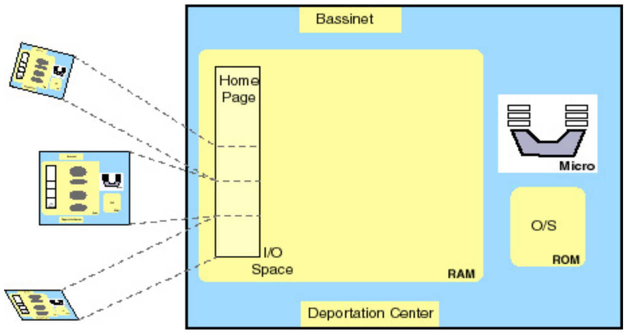

Figure 6: Organization of the RAM space of a particle.

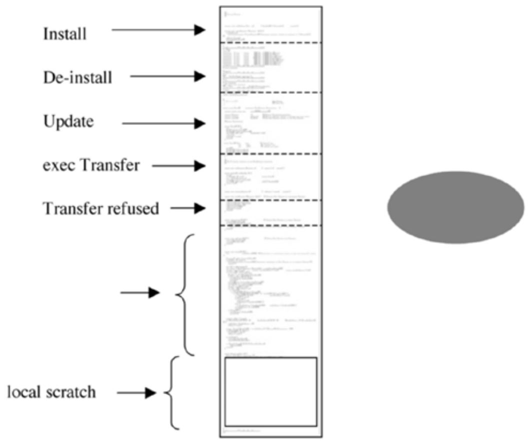

Figure 7: Code segments are self-contained functions which fit completely in the RAM space of a single particle, have handles for at least 5 commands, and gate all I/O through the particle's I/O space.

3.4 Run-Time Scheduling

At run-time, code segments migrate nomadically looking for particles on which to install themselves. In those particles where entry into the program RAM is successful, the code segments will set up shop and begin searching for relevant data in the I/O space. The side effect of the code segment's activities is additional posts to the HomePage. Often, the number of code segments seeking entry will exceed the particle's capacity. The allocation of program space is regulated by the OS in response to competition among the code segments. Each code segment must draw its competitive advantage from the I/O space and therefore, indirectly from the activity of other code segments. The competition is arbitrated by the particle's OS. And when a particular code segment loses out, it is de-installed and passed to the output port to migrate further via diffusion.

Metaphorically, the contents of the I/O space can be compared to soil with a particular nutritional profile. The code segments are in turn comparable with plant life trying to take root in the soil while concurrently contributing to the soil's nutritional capacity – albeit without depleting the existing storage.

4. SAMPLE APPLICATION

As a platform for application development, we have developed a simulator modeled after the Gunk simulator[3]. Code segments are written as Java objects with the five normative functions implemented as public methods. Each segment is archived individually as a serialized Java object. I/O ports, capable of mimicking the networking behavior of the particles, can be arbitrarily positioned to diffuse the code segment into the particle ensemble.

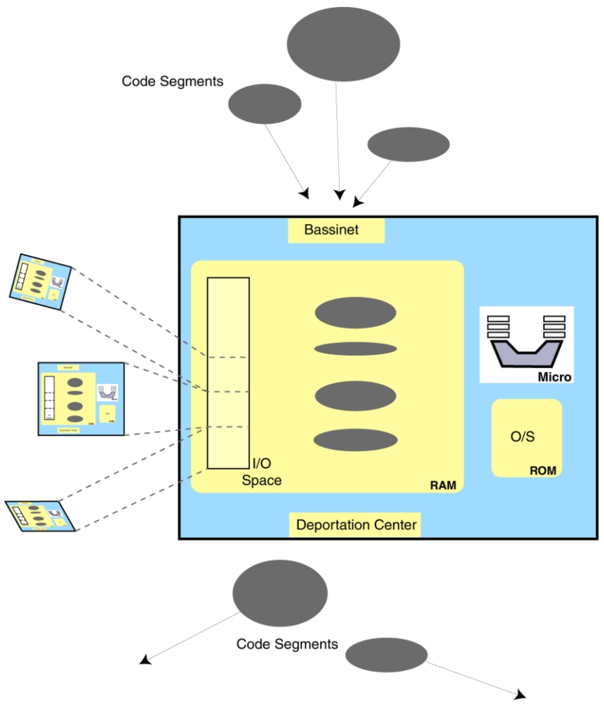

Figure 8: Behavioral description of programming model: code segments wander nomadically between particles, pass through the active FIFO (bassinet) and are installed in the RAM space, communicate via tagged posts to the I/O page, and compete for available space in the program RAM – with the losers queued for deportation to the neighboring particles.

A popular example of a code segment is one that spreads virally from a specific point of entry, producing a gradient field as a side effect. This "Scent" code segment executes a very simple strategy. On entry into a particle, it posts to the HomePage a message consisting of five numbers:

- an ID tag common to all Scent code segments
- two numbers which can be optionally used to identify the originating entity (typically, an IO port masquerading as a particle)
- an integer number, and
- a real number.

Once embedded in a particle, the "Scent" segment wakes up at irregular intervals and scans the neighboring HomePages for posts from other Scent code segments. If the code segment finds an uninfected particle, it replicates itself and infects the neighbor. If there are no uninfected particles in view, the code segment reads the integer number from all the neighboring posts, selects the smallest integer, increments that by one, and posts that number. The real number is then computed as the average of all the observed integers.

Using this simple strategy, the "Scent" code segment can build a gradient field. Figure 9 shows the simulator and a snapshot of the code segment's behavior. Particles (small icons) are pseudo-randomly distributed over a 2D surface. Two I/O devices (large circles) are in physical contact with the computing ensemble. The Scent code segment has entered the particle ensemble through the I/O port on the left and is spreading virally. The infected particles are color coded with an intensity that is inversely proportional to the real number in the Scent's post to the HomePage.

In Figure 10, the gradient field has blanketed the entire 2D surface. In response to the appearance of the gradient, the I/O device on the right diffuses two code segments into the particle ensemble: a "Marker" code segment and a "Halo" code segment. Any given marker segment will identify one of its neighboring particles which is closest to the gradient source and in turn infect the neighbor with a replica of the marker.

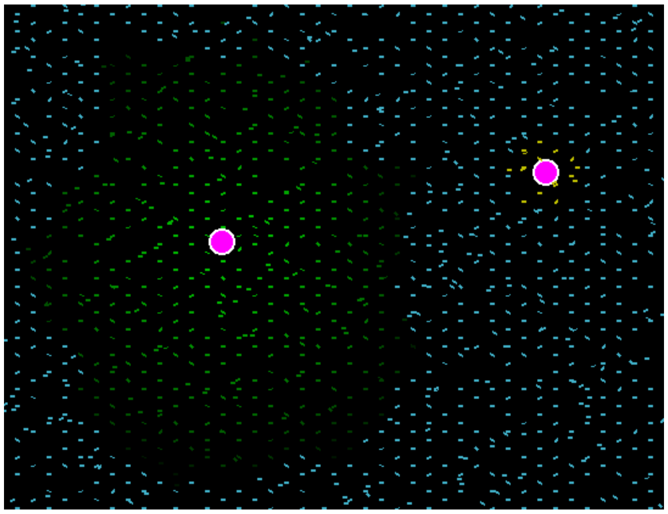

Figure 9: Snapshot of the simulator, showing code diffusing outwardly from the source on the left.

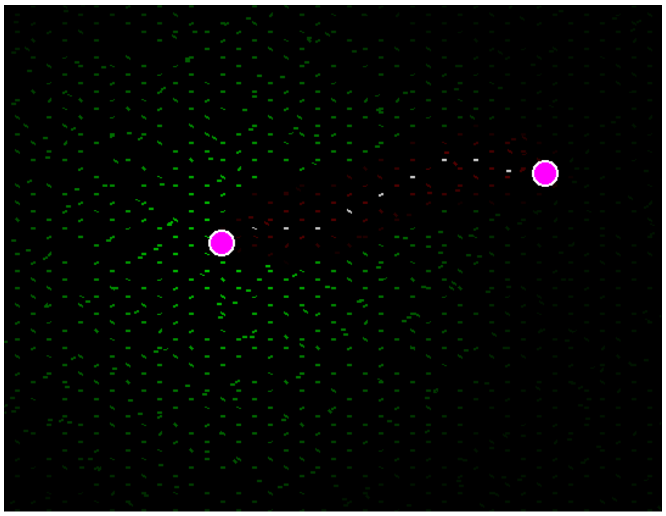

Figure 10: Creation of an "insulated" communication pathway between a source and a destination.

The halo code segments dynamically construct a mini-gradient field about the markers. In the case of multiple point to point connections, the halo acts to inhibit other marker trails from crossing each other.

While each of the three code segments is executing a simple strategy over a restricted locality, the global effect is a dynamically configurable communication network. This application continues in the tradition of those who use agent based techniques for load balancing in telecom networks[4]. At the scale of a table top, this global behavior has several attractive properties:

- The power requirements on the individual devices are extremely modest, with each device supporting a wireless link of several millimeters in length.
- The link is not opened until the external device is placed in contact with the computing surface – a modality which humans find very natural.

More promising than this simple example are applications which take advantage of the fact that each of the links in the communication chain is itself a computational device, and can perform useful tasks on the data passing through, pulling in additional computation and information from the surroundings as needed. Several are in development as of this writing, with a particular emphasis on media applications.

5. ACKNOWLEDGMENTS

The authors wish to thank numerous co-workers from the faculty, staff, and student body at the MIT Media Laboratory and the MIT Laboratory for Computer Science. Particular thanks to Joshua Lifton, Stefan Agamanolis and Chris Hanson. This work has been supported by the Digital Life Consortium and by a fellowship from Intel corporation.

REFERENCES

1. H. Abelson, D. Allen, D. Coore, C. Hanson, G. Homsy, T. Knight, R. Nagpal, E. Rauch, G. J. Sussman, and R. Weiss, "Embedding the Internet: amorphous computing," *Communications of the ACM,* **43**(5), 2000.
2. N. Lynch, *Distributed Algorithms*, Morgan Kauffman Publishers, San Francisco, 1994.
3. S. Adams, "A high level simulator for Gunk," Technical report, Massachusetts Institute of Technology AI Lab, Cambridge MA, November 1997.
4. R. Schoonderwoerd, O. Holland, J. Bruten, and L. Rothkrantz, "Ant-based load balancing in telecommunication networks," *Journal of Adaptive Behavior*, **5**(2), 1996