Continuous Space-Time Semantics Allow Adaptive Program Execution

Jonathan Bachrach, Jacob Beal, Takeshi Fujiwara
MIT Computer Science and Artificial Intelligence Laboratory
77 Massachusetts Ave, Cambridge, MA, USA
jrb@csail.mit.edu, jakebeal@mit.edu, fujiwara@sophie.q.t.u-tokyo.ac.jp

Abstract

A spatial computer is a collection of devices filling space whose ability to interact is strongly dependent on their proximity. Previously, we have showed that programming such a computer as a continuous space can allow self-scaling across computers with different device distributions and can increase robustness against device failure. We have extended these ideas to time, allowing self-scaling across computers with different communication and execution rates. We have used a network of 24 Mica2 Motes to demonstrate that a program exploiting these ideas shows minimal difference in behavior as the time between program steps ranges from 100 ms to 300 ms and on a configuration with mixed rates.[1]

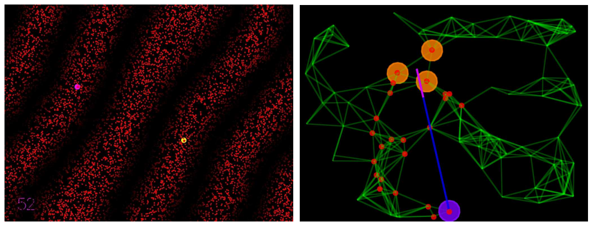
1 Introduction

Spatial computers are an increasingly prevalent class of systems, in which the computer is composed of a collection of devices that fill space and whose ability to interact is strongly dependent on their proximity. Spatial computers emerge across a wide variety of domains, including swarm robotics, biofilms, sensor networks, and reconfigurable computing.

In previous work, we have advocated abstracting a spatial computer as a continuous, space-filling computational material, which we call an *amorphous medium*. Our language, Proto[5], provides geometric primitives like neighborhood, density, and distance that make it simple to write programs that self-scale to spatial computers with different distributions of devices and that handle some device failures transparently. For example, a directable plane wave (Figure 1(a)) takes 31 lines of code and target tracking (Figure 1(b)) takes 28 lines.

We now take the same approach to time, viewing a computation as a process evolving over continuous time, and define continuous time semantics for Proto. An appropriate choice of primitives then makes it possible to write programs that self-scale to execute equivalently on spatial computers with different communication and execution characteristics.

[1]This work partially supported by NSF Grant #CCF-0621897



(a) Plane Wave (b) Target Tracking

Figure 1. Proto code allows compact descriptions of complicated behaviors.

The core problem is the duality between the continuous model of space and time and its imperfect simulation using discrete chunks of execution. The advantage of the continuous model is that it will allow us to talk about aggregates. Our challenge is to give the user tools for managing and adapting to the inaccuracies of execution.

2 Related Work

The Proto language is previously described in [5] and [2] using a discrete time semantics of global rounds of execution. Proto is built on the dual foundations of the amorphous medium abstraction for programming in continuous space[4] and the Gooze lightweight stream processing language[1].

The spatial computing languages closest to Proto are *LISP[8], which operates on field values similarly but assumes a regular grid of devices, and Regiment[11], which explicitly implements geometric operations but has a base-station centered semantics. More foreign approaches include TinyDB[10], which provides a database view of the devices making up the computer, and Kairos[7], which operates on the computer as an abstract graph.

Continuous time evolution has been a concern in other specialized domains of computing, such as multimedia processing (see, for example [6]), and is approximated in computational models of chemical and biological computing such as the Gamma calculus[3] and P-Systems[12].

The fact that computing devices evolve continuously is a major concern throughout the fields of networking and parallel and distributed computing. In these fields, however, the primary concern is often not to embrace the continuous evolution of time, but to banish it. This is difficult and, in many cases, impossible—see, for example, the many asynchronous impossibility results in [9].

3 Space-Time Programs

The key enabling idea for continuous time programming is the *configuration path*—a function that specifies the output value at each point in the amorphous medium as a function of time. We borrow this idea from variational mechanics, particularly the approach in [13]. Pushing the analogy further, we view the program as an invariant description of the dynamics of the system. To compute a configuration path, we need only know the initial state and evolve it forward in time according to the dynamics of the system.

Configuration paths allow us to use continuous time without giving our outputs continuous values[2] This is important because many familiar programming constructs use discrete values. For example, a finite state machine changes from state to state without passing through intermediate states in the transition. Configuration paths allow us to separate the quantization of state and time.

We can combine this with our continuous space abstraction, the *amorphous medium*, in which the computer is represented as a manifold where every point is a computational device. In this combined view, a space-time program specifies the evolution of a manifold function over time. This execution can then be approximated with discrete steps on individual devices.

4 Space and Time Operations

Using the configuration path model allows us to choose a few critical space-time primitives that make it simple to approximate a global continuous program using discrete steps on individual devices. By limiting space/time interaction to a few simple mechanisms, Proto allows the programmer to write succinct global programs that compile to an efficient implementation. Due to limited space, we will not explain Proto or how these operations are actually implemented.

[2]Alternatives like derivatives and continuous-time feedback control do not allow this.

Space Restriction Conditional code needs to be thought of differently when programming an aggregate rather than a single device, since in general different devices may need to take different branches. We handle this by providing a **restrict** operation, which limits the region of space where a piece of code is being evaluated. An ordinary branch is then implemented with a pair of **restrict** operations, one for the "true" branch and one for the "false" branch.

Incremental Evolution The evolution of program state over time is expressed incrementally, using a feedback loop construction **letfed**. The programmer controls evolution with two expressions: an initial state (used at the beginning or when a branch begins to run) and an incremental configuration path that takes the current state and a time difference **dt** and returns the state evolved forward by **dt**.

Field/Summary Operations The family of field/summary operations select data across continuous regions of space-time, compute with that data, then summarize into a single value (e.g. the minimum value in the region or the integral over the region). There are two sets of field/summary operations: one for neighborhood and one for histories.

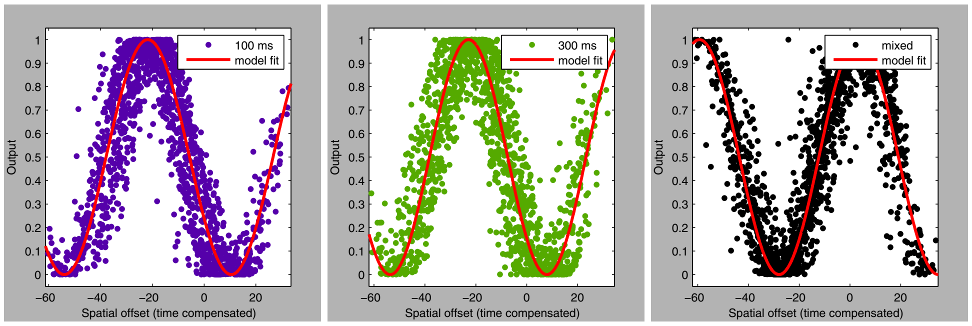
Neighborhood operations use values from the past lightcone of a point (implying message-passing in the discrete approximation) and give access to the computer's geometry through special operators. In the discrete approximation, the relationship between space and time may break down at short distances, so we provide independent **nbr-range** and **nbr-lag** operators that allow automatic compensation.

History operations are like neighborhood operations, but select values from the past of a single point rather than across a neighborhood. A history extends backward in time for as long as the value is defined; when a value becomes undefined due to the start of execution or space restriction, the history terminates. Because the past is fixed, the history operations are effectively just stereotyped feedback loops.

Evaluation Rate Although programs are specified in terms of continuous evolution, when they finally run they will be approximated with a sequence of discrete increments. If the size of each increment is too large, however, a program might get into trouble in any number of ways, from going unstable to breaking its specifications.

Consider, for example, running collision avoidance code on a swarm of robots. If the code is evaluated too infrequently, the robot may not have time enough to brake before slamming into an obstacle.

Rather than try to figure out a safe interval automatically, we give the programmer simple operations to limit increment size. The programmer can specify increment limits either in terms of frequency with **min-freq** or in terms of



(a) 100 ms (b) 300 ms (c) 100/200/300 ms

Figure 2. Spatial frequency of a plane wave is minimally affected by differences in execution rate.

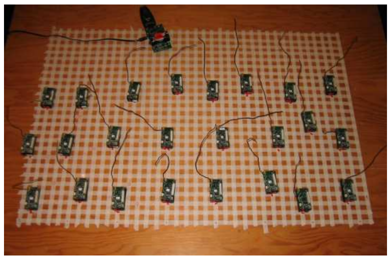


Figure 3. Experimental deployment of 24 Mica2 motes organized as three randomized rows of eight.

period with **max-period**. These point controls may then be composed to control the execution of the entire program.

5 Example and Verification

In order to verify our approach, we ran our plane wave application[5] with different processor update speeds and measured the impact on the application output. The plane wave application allows one to direct the angle of a planar wave by placing markers in the network.

We verified the code by executing it on 24 Mica2 motes arranged in three randomized rows of eight as shown in Figure 3. The leftmost mote in the middle row was set to be the destination, the rightmost in the middle set to be the source, the space period of the wave set to be 10 (20π cm) and the time period of the wave set to be 1 (2π seconds).

The motes were provided with coordinates and their radio range software limited to 9cm, allowing us to record data through a single logging device. The motes were then run with three different evolution increments: (a) all motes running at one update every 100 milliseconds, (b) all at 300 ms, and (c) a mixed population with five at 100 ms, five at 200 ms and the rest running at 300ms.

We gathered data by logging messages sent for approximately 12 wave cycles, ending up with approximately 1600 messages over approximately 80 seconds. Many messages apparently sent do not appear in the log, likely due to interference between messages. The motes nearest to the logging device are represented preferentially in the logs, with a maximum of 214 messages for a nearby mote, a minimum of 28 messages for a distant mote, and medians of 48.5 for run (a), 60 for (b) and 58.5 for (c).

Figure 4 shows that the motes synchronized to the same time frequency in all three runs. We plot time minus phase against output, which should re-align the outputs, then fit it against a sine function where phase and frequency are the free parameters. For run (a), the best fit frequency is 1.024, with RMSE 0.156, for run (b), the best fit is 1.026, with RMSE 0.161 and for run (c), the best fit is 1.024, with RMSE 0.135. This shows that the frequency of the wave is not affected by execution rate.[3]

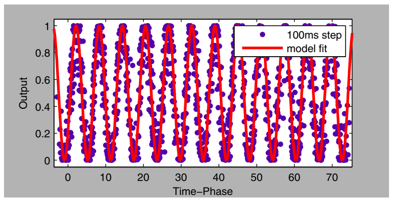
Figure 2 shows that the motes also synchronized to the same spatial frequency in all three runs. We plot output against position (compensated for time offset using our previous fit), then fit it as before. For run (a), the best fit frequency is 0.0974, with RMSE 0.1519, for run (b), the best fit is 0.1021, with RMSE 0.1606 and for run (c), the best fit is 0.1001, with RMSE 0.1338. Although not as precise as the time fit, the space fit shows that impact from differences in execution rate is minimal.

Finally, Figure 5 shows interpolated space time plots for the first 400 samples of the three experiments. Although the interpolation is noisy due to missing data, visual inspection shows that the individual waves at least appear decently regular.

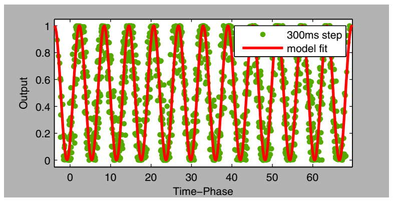
6 Conclusion

We advocate modelling computation on a spatial computer using continuous space and time. Using this abstrac-

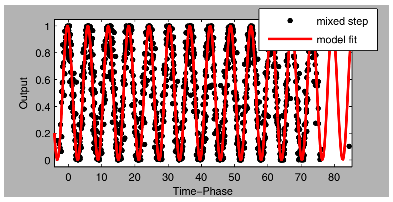
[3]The slight speed-up comes from a naive time synchronization method.



(a) 100 ms

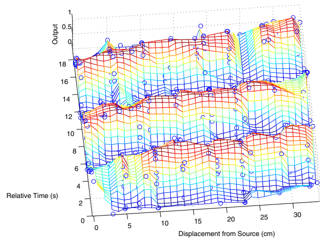


(b) 300 ms

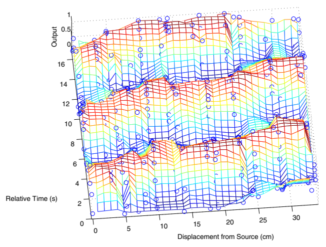


(c) 100/200/300 ms

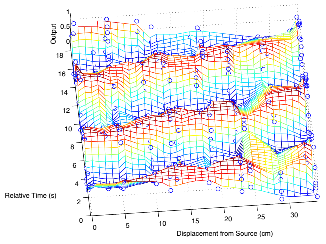
Figure 4. The time frequency of the plane wave is not affected by differences in execution rate.



(a) 100 ms



(b) 300 ms



(c) 100/200/300 ms

Figure 5. Interpolated space-time waves for the first 400 samples of each experiment show that individual waves are fairly regular (though the interpolation is noisy due to missing data).

tion with an appropriate choice of primitives allows programs to scale automatically to spatial computers with different communication and execution characteristics. We have modified the Proto spatial computing language to demonstrated scaling on a network of Mica2 Motes.

The work presented in this paper represents only the beginning of investigation into continuous space/time models of computation. Changing to a continuous model of time does not resolve problems so much as it exposes them. There are many open problems at every level: how best to describe and control aggregate behavior, how best to represent discrete issues in the continuous abstraction, and how to manage trade-offs between cost, efficiency and accuracy in discrete approximation.

Finally, the problems driving this model are not unique to our approach to spatial computers, but derive from basic issues of time and communication. Perhaps other languages for controlling aggregates could benefit from incorporating ideas of continuous state evolution, either in improving their expressiveness or decreasing the cost of implementation by relaxing constraints.

References

[1] J. Bachrach. Gooze: a stream processing language. In *Lightweight Languages 2004*, November 2004.

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

[3] J.-P. Banatre, P. Fradet, and D. L. Metayer. Gamma and the chemical reaction model: Fifteen years after. In *WMP*, pages 17–44, 2000.

[4] J. Beal. Programming an amorphous computational medium. In *Unconventional Programming Paradigms International Workshop*, 2004.

[5] J. Beal and J. Bachrach. Infrastructure for engineered emergence in sensor/actuator networks. *IEEE Intelligent Systems*, pages 10–19, March/April 2006.

[6] C. Elliott. Functional images. In *The Fun of Programming*, "Cornerstones of Computing" series. Palgrave, Mar. 2003.

[7] R. Gummadi, O. Gnawali, and R. Govindan. Macro-programming wireless sensor networks using *airos*. In *DCOSS*, pages 126–140, 2005.

[8] C. Lasser, J. Massar, J. Miney, and L. Dayton. *Starlisp Reference Manual*. Thinking Machines Corporation, 1988.

[9] N. Lynch. *Distributed Algorithms*. Morgan Kaufman, 1996.

[10] S. R. Madden, R. Szewczyk, M. J. Franklin, and D. Culler. Supporting aggregate queries over ad-hoc wireless sensor networks. In *Workshop on Mobile Computing and Systems Applications*, 2002.

[11] R. Newton and M. Welsh. Region streams: Functional macroprogramming for sensor networks. In *First International Workshop on Data Management for Sensor Networks (DMSN)*, Aug. 2004.

[12] G. Paun. *Membrane Computing: An Introduction*. Springer-Verlag, Berlin, 2002.

[13] G. J. Sussman and J. Wisdom. *Structure and interpretation of classical mechanics*. MIT Press, Cambridge, MA, USA, 2001.