Physical Evolutionary Computation

Eric Schulte
University of New Mexico
eschulte@cs.unm.edu

David H. Ackley
University of New Mexico
ackley@cs.unm.edu

April 17, 2011

Abstract

Although evolutionary computation (EC) is an "embarrassingly parallel" process, it is often deployed on essentially serial machines, and even its parallel implementations typically retain the globally synchronized and regimented style typical of serial computation. We explore a radical 'physical evolutionary computation' (PEC) hardware/software framework that is based on real time and real space rather than an abstract sequence of events—for example, the mutation rate is specified in Hertz. PEC supports massive and reconfigurable parallelism, using a prototype hardware 'tile' that begins evolutionary computation within three seconds of applying power. Although each tile is a simple computer by today's standards, tiles can be plugged together into a wide variety of size and shape computing grids—even while the computation is running.

We present our initial explorations with this framework, defining mechanisms for representing and sharing problems, and mapping between computational spaces and physical ones. We discuss advantages of PEC—such as extremely robust operation—as well as its challenges, and touch on some unexpected, and potentially useful, level-crossing interactions that can be explored when the embedding of the computational within the physical is made real.

1 Scalable abstractions

Abstract computational models simplify the natural world to help us understand it, and also serve as outlines for the engineering of—preferably ever-more-powerful—computing machines. Researchers have explored computing models inspired by everything from mathematics and philosophy to the natural and social sciences, but abstractions based *algorithmic software on reliable serial hardware* have dominated computer science and engineering, accruing volumes of widely-understood tools and techniques. The basic *von Neumann machine* 'CPU+memory' model abstracts physics into deterministic boolean algebra, and abstracts time into a total ordering of discrete steps from a beginning to an end. Even more aggressively, it eliminates the distances and dimensions of physical space entirely, declaring all memory equally cheap to access by the CPU. Where that assumption is invalid, the von Neumann machine is simply a poor descriptive model. On the other hand, when used for machine design, that assumption has held simply because engineering was commanded to make it so, which it has done spectacularly for decades, scaling clock speed and memory capacity over many orders of magnitude.

Of course, many different abstractions can be imposed on any situation, and the continued dominance of serial algorithmic computing is now uncertain as its scalability dwindles and it yields ground to alternate models such as cache-coherent multicores [Sutter, 2005], which offer a degree of parallelism while preserving many of the computational and spatiotemporal abstractions of the classic framework.

This paper explores the *physical evolutionary computation* (PEC) framework (Figure 1), which ab-

stracts reality quite differently than the classical approach. It retains explicit units of space and time, and it marginalizes brittle, finite, serial algorithms in favor of robust, open-ended, parallel computational processes, capable of filling whatever volume of computational space-time one's hardware, energy, and patience can supply.

By anchoring evolutionary computation firmly in space and time—with many 'evolution tiles' spatially interconnected in some geometry, and genetic operations occurring not in a global algorithmic lockstep, but independently triggered by hardware timers and arriving local stimuli—we obtain a naturally distributed system that is extremely scalable, without any *a priori* edges or 'privileged points', and so inherently robust that, basically, the system can't *not* evolve.

This paper presents our initial experiences with a prototype PEC system. Section 2 motivates and presents the design we have developed. Six sample tasks illustrate the system behavior in Section 3. Section 3.2 provides results and analysis of the performance of the system, and a few anecdotes of some ways in which this system surprised our assumptions drawn from traditional computation. Finally we discuss the application of our findings to alternate hardware systems and problem domains, and review the implications of this model of computation in section 4.

2 Physical Evolutionary Computation

2.1 Parallel Evolutionary Computation

Evolutionary Computation (EC) is a domain independent stochastic search technique modeled after the biological process of evolution. EC is *scalable* in the population size and number of generations, and famously *embarrassingly parallel* [Andre and Koza, 1998] in that multiple individuals can generally be evaluated simultaneously on multiple processors, and many researchers have explored such approaches [Eklund, 2002, Sterling, 1998, Stender, 1993, e.g.].

However, perhaps due to their derivations from serial designs, even parallel EC's are typically organized so that their aggregate computation is fully equivalent to a single serial execution path. Such rigid global control can provide the cardinal benefit of repeatability, but it is of course uncharacteristic of real biological systems (and non-trivial real-world systems generally), and it limits the ultimate system scalability by requiring global synchronization points, typically at least once per generation.

In the PEC framework, we prioritize the goal of *robust indefinite scalability* over the goal of maintaining a serial-equivalent computational path, so we adopt a design goal of eliminating as far as possible all 'privileged points' in space (e.g., root nodes, dependence on edges or corners) and time (e.g., sequential phases, synchronization points, global clocks). Here we describe our prototype PEC system including hardware evolutionary tiles and the PEC process they support.

2.2 Evolutionary Tiles

The hardware layer of a PEC system is composed of a fungible collection of interconnected *evolutionary tiles*, whose quantity and configuration may change during the course of a computation. A group of connected tiles is shown in Figure 2.

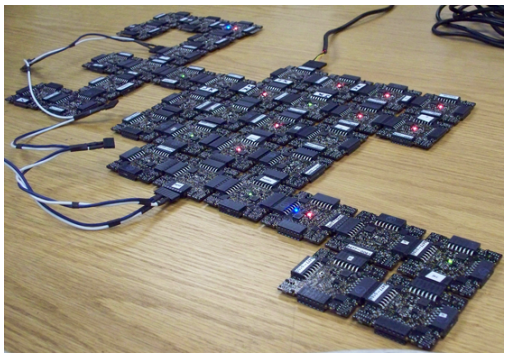


Figure 2: 37 Evolutionary Tiles. With external power connections *left* and an external data connection *top*.

As part of a research effort in Robust Physical

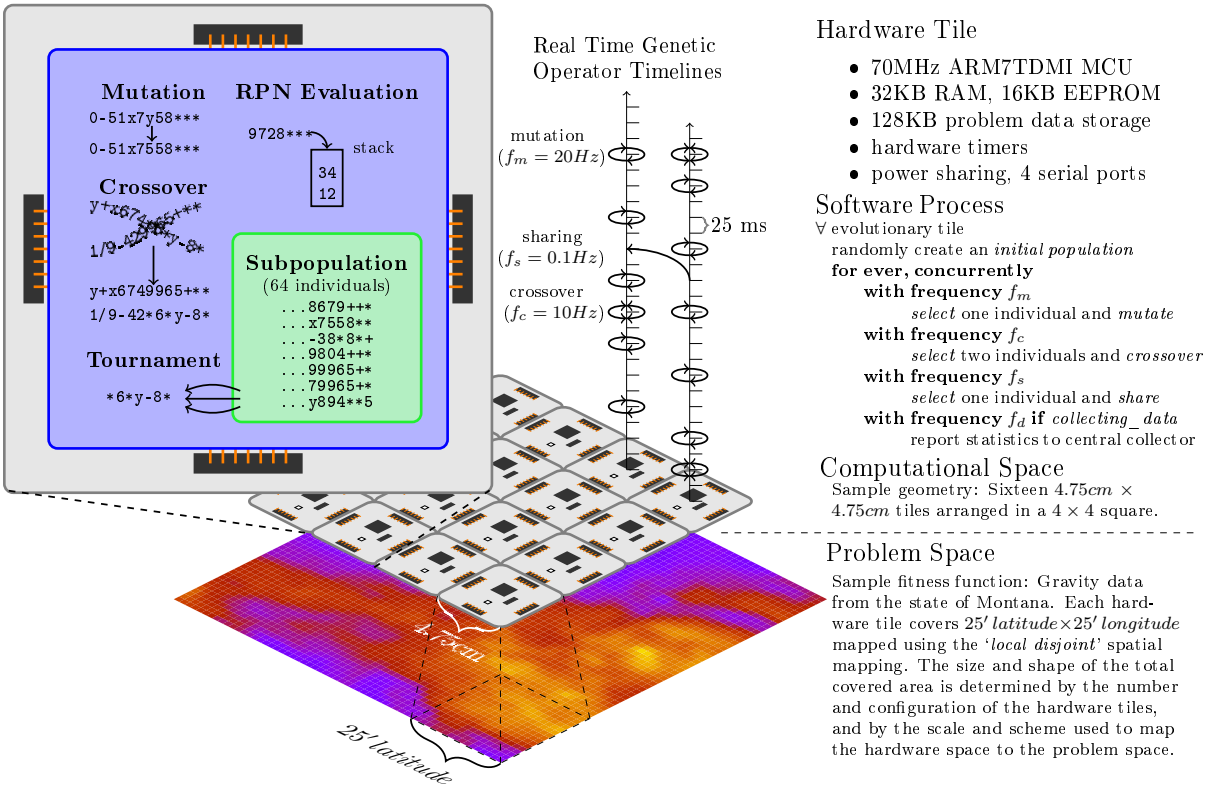


Figure 1: Physical Evolutionary Computation. Prototype overview: Hardware and software in physical space, mapped to sample fitness function in problem space. See text for details.

Computation, hardware and support software has been developed for a prototype tilable computer that has been brought to market under the name *Illuminato X Machina* (IXM) [liquidware.com, 2010]. The IXM is a self-contained traditional Von Neumann machine with a single processor, a modicum of RAM and persistent memory, and additional hardware such as timers that can be used to trigger computation at specific intervals (details in Figure 1). Each of the tile's four edges holds a connector capable of sharing both power and/or information with either a traditional computer or power supply, or more frequently, with a neighboring tile. The support software defines several basic abstractions including a simple but powerful packet format, as well as managing the hardware to perform interrupt-based asynchronous communications via all four tile edges simultaneously. Although the PEC process is certainly not limited to specifically this hardware, all the demonstrations shown in this paper were run on groups of IXM tiles.

2.3 The PEC Processing Mechanism

We present the PEC process in three sections: the processing local to a single hardware tiles and its immediate neighbors, the mechanisms for handling 'global' events such as changing the fitness function, and finally housekeeping details. In many cases it will be obvious that other evolutionary computation mechanisms could be substituted for the ones we used without significantly distorting the overall processing structure, but here we discuss only the specific mech-

anisms we have implemented and tested.

2.3.1 Local Genetic Operators

Each tile in a PEC system houses a single subpopulation of candidate solutions, so an overall group of tiles evolves using the 'island' approach to genetic algorithm parallelism [Collins, 1992]. On *power up* the tile subpopulation is initialized with 64 randomly generated individuals, and this subpopulation size is then held constant by all further evolutionary actions.

The population serves as both a source and repository for candidate solutions which are accessed and stored through *tournaments* and *incorporation* respectively.

Tournaments perform selection in the PEC system. In a tournament three individuals are chosen from the population at random, these individuals are then evaluated, their resulting finesses are compared, and the individual with the best fitness is selected. Individuals are only evaluated at the time of selection and their fitness is not stored, so it is possible that a single individual may be evaluated multiple times or not at all. In the presence of a changing problem space the individual's fitness is expected to be valid only at the time of selection.

When new individuals are generated through tournaments and local reproduction, or are received from neighboring tiles, they are *incorporated* into the local population. To maintain a constant population size during such an incorporation, a randomly chosen existing individual is evicted from the population. Incorporation is the only means by which individuals are added to or removed from the population.

Subpopulations evolve locally through *mutation* *crossover* and *sharing*, each of which is triggered at a fixed frequency by the hardware clock—although processing pile-ups can cause small delays.

The *mutation* operation, occurring twenty times a second ($f_m = 20Hz$) in our experiments, begins by tournament-selecting an individual to copy from the population. Random points along the individual's genotype are chosen—one point per 16 symbols of genotype length, but no less than one point—and then each point is either: Replaced with a randomly chosen value from the underlying symbol set, with a 50% chance; deleted from the individual, with a 25% chance; or has a randomly chosen symbol inserted into the individual immediately preceding the point, with a 25% chance. The resulting individual is then incorporated into the population.

In a *crossover* operation, which occurs ten times a second ($f_m = 10Hz$), two individuals are selected via independent tournaments and copied from the population. A random point is chosen independently along each genotype, and single point crossover is used to create a new individual, which is then incorporated into the population.

Finally, individuals are also *shared* between tiles, at a rate of once every ten seconds ($f_s = 0.1Hz$). When sharing, an individual is tournament-selected and copied into messages sent to every adjacent tile. Upon arrival at a neighboring tile, a sharing message triggers a reflex action that incorporates the shared individual into the recipient's subpopulation. Sharing propagates individual solutions between tiles (see section 3.2.3, below, for more); note that each successful act of sharing also amounts to up to a fourfold reproduction within the aggregate global population—and that individuals and subpopulations housed in more densely-connected tiles have an advantage in that regard.

2.3.2 Global Coordination

Some aspects of a PEC computation, inevitably, need to be coordinated globally across an entire group of tiles, and various *control messages* are defined for this purpose, containing information such as the problem specification (Section 3.1.1), the scheme for mapping between problem space and computational space (Section 3.1.3), and the frequencies of the PEC genetic operators discussed above. Parameter-setting messages may be global in intent—in which case they are propagated throughout the group—or may be sent from one tile to an immediate neighbor in response to a direct request, which occurs when a tile has just rebooted or been added to the group.

For purposes of problem space mapping and data collection, each tile in the group also maintains a 2D coordinate specifying its position in the two dimensional spatial configuration of the group, with the

origin specified—in our experiments so far—by the location of the external data connection. These coordinates are calculated and maintained using control messages. A reflex action triggered by coordinate message arrival updates the local state to reflect the new coordinates, and forwards appropriately-transformed coordinates to each of its neighbors, using a message serial number to avoid looping.

2.3.3 Housekeeping and Maintenance

In addition to the evolutionary operations described previously, there are several maintenance tasks proceeding concurrently, which we touch on briefly here for completeness and intuition.

Serial port management The low-level software frequently sends "Are you (still) there?" messages out all serial ports, to allow prompt detection of tile connections and disconnections.

Problem data update If a (partially or completely) new problem data matrix is injected into the group, it propagates from tile to tile incrementally and automatically as a background process. Evolution continues during this process except during erasure of flash sectors which requires a tile reboot.

System software update If a new version of the PEC software itself is made available to the group, it will propagate tile to tile until the whole group is updated, again while evolution continues on the group as a whole.

This completes the high-level description of the PEC process, though there are a number of details yet to present, such as the mechanism whereby subpopulations determine what portion of a function space they will optimize; these mechanisms are explained in the next section where their application is demonstrated.

3 PEC Demonstrations

Here we illustrate the PEC process on a small variety of fitness functions, using with three different tile group geometries and five problem space to computational space mapping schemes.

Though PEC scales to any number of tiles, all demonstrations here are performed on a group of 16 tiles connected to a external power supply, and to an external computer (connection shown in Figure 4). The external machine collects data from the group, and sends periodic control messages (section 2.3.2) to assert the desired problem and settings of the PEC parameters.

3.1 Setup

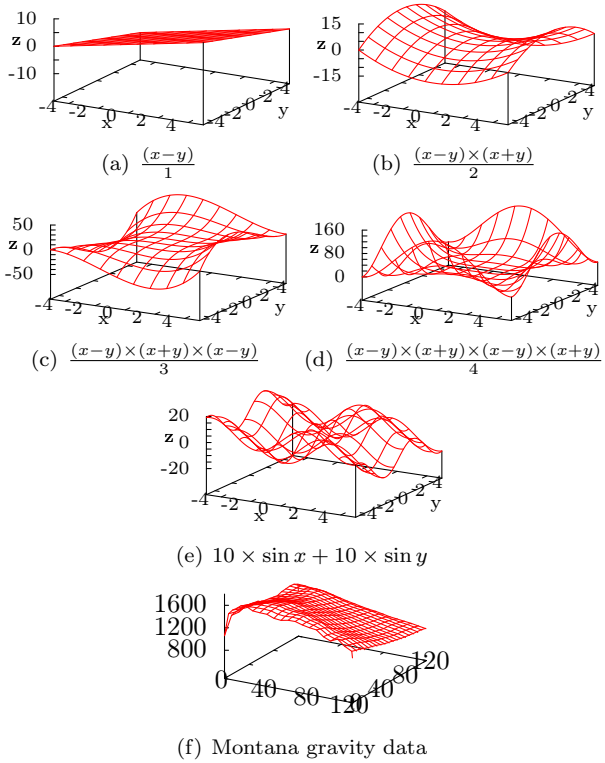
3.1.1 Fitness Functions

Experiments are run against six fitness functions, consisting of matching a set of points in three dimensions. Evolved individual candidate solutions take the form of algebraic expressions over two variables represented as character strings in a reverse polish notation (RPN) that provides access to only the conventional arithmetic operators (with safe division), single digit constants, and the variables. Individuals are assigned a fitness based on their ability to match the surface of the fitness function. Two of the dimensions of the fitness function are aligned with the two spatial dimensions in which the group of tiles are configured.

The six targets—five analytic functions and one data set—are shown in Figure 3. The first four are polynomial equations of increasing degree and complexity which can be exactly matched by RPN individuals. The fifth is a trigonometric equation which cannot be exactly matched given the operators available to the RPN individuals, and the last consists of gravity anomaly data collected from the state of Montana [Arthur A. Bookstrom and Johnson, 1995], encoded as a 128KB data matrix that the PEC support software incrementally propagates and burns into each tile's flash memory.

3.1.2 Geometric Configurations

Tiles are assembled into groups using the geometric configurations shown in Figure 4. The geometry affects the communication patterns between



(a) $\frac{(x-y)}{1}$ (b) $\frac{(x-y)\times(x+y)}{2}$

(c) $\frac{(x-y)\times(x+y)\times(x-y)}{3}$ (d) $\frac{(x-y)\times(x+y)\times(x-y)\times(x+y)}{4}$

(e) $10 \times \sin x + 10 \times \sin y$

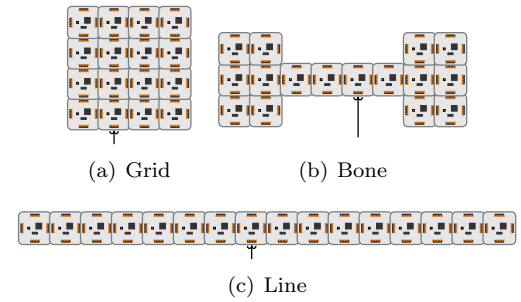
(f) Montana gravity data

Figure 3: Problem sets

the group's tiles—which has been shown to impact the performance of distributed genetic algorithms [Stender, 1993]. In addition, the geometry may affect the portion of the problem space that is sampled by a group. For example, due to its aspect ratio the line configuration (Figure 4(c)) can sample more extreme values of the problem space (under most mappings, see section 3.1.3 below), and, because each tile has at most two neighbors, it will have more restricted intertile communication—which has robustness consequences, but may also may be beneficial in some circumstances.

3.1.3 Mapping Schemes

The PEC practitioner decides how to align the dimensions of the hardware and problem spaces. This work demonstrates the mapping schemes shown in

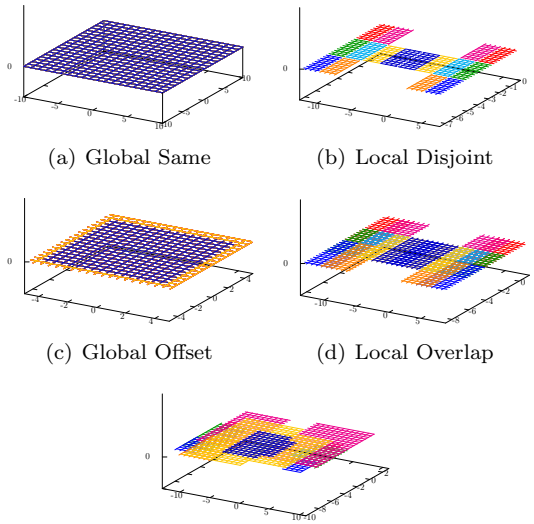


(a) Grid (b) Bone

(c) Line

Figure 4: Group Configurations

Figure 5.



(a) Global Same (b) Local Disjoint

(c) Global Offset (d) Local Overlap

(e) Adaptive

Figure 5: Schemes mapping hardware to problem space. Shown assuming the 'Bone' configuration from figure 4.

Global Same (Figure 5(a)) maps each tile to the same portion of the problem space. This scheme is what many non-spatialized parallel ECs use; it is also one natural choice when there is no obvious match between problem and hardware dimensionality. *Local Disjoint* (Figure 5(b)) maps each tile into a non-overlapping equal-sized portion of the problem space, preserving the neighborhood relationships of

the computational space. *Global Offset* (Figure 5(c)) is similar to global same; however the domain of each tile is slightly offset based upon the tile's position in the computational space. *Local Overlap* (Figure 5(d)) is like local disjoint, except the domain of each tile is slightly increased causing overlap in the problem space. Finally, the *Adaptive* (Figure 5(e)) scheme begins with the same mapping as local disjoint. During the course of an adaptive run, tiles share their mean fitness, and based on their comparative performance, tiles increase or decrease the size of their domain in the problem space, such that the better-performing tiles take on larger portions of the problem space.

3.2 Empirical Results

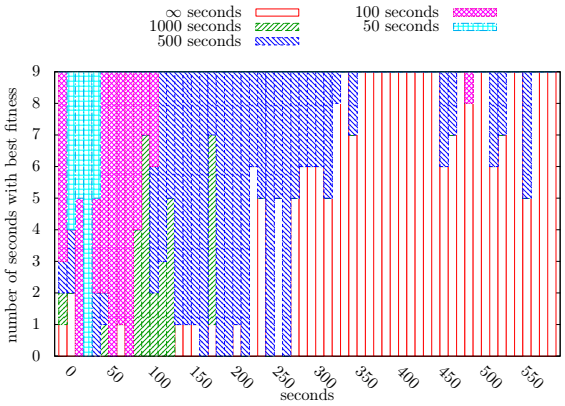
3.2.1 Group-wide error by scheme

Using the Grid configuration, 10 runs were performed for each combination of the first five problem sets, and the five mapping schemes. The results are shown in Figure 6.

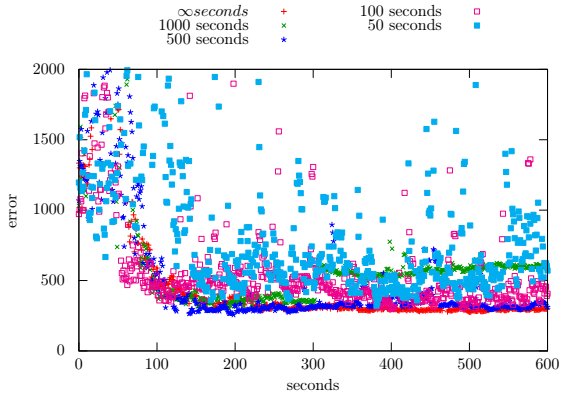
Performance by scheme is consistent across all five problem sets with the three 'local' schemes (*local disjoint*, *local overlap* and *adaptive*) resulting in less error than the two 'global' schemes (*global same* and *global offset*). Against all but the simplest problem set (Figure 6(a)) the *local overlap* scheme outperforms all others, showing consistent improvement over the course of the run even against the most difficult problem set (Figure 6(f)), against which several other schemes cease improving after about a thousand seconds.

3.2.2 Group-wide error by robustness

To explore the robustness of PEC in the face of unreliable hardware, we simulated hardware errors by programming the tiles to reboot probabilistically with an experimenter-specified mean time to failure per tile ($MTTF_{tile}$). We evaluated several levels of hardware instability ranging from $MTTF_{tile} = 1000$ sec to $MTTF_{tile} = 50$ sec, which—using 16 tiles overall—typically resulted in a failure somewhere in the grid about once per a minute at the slow end, to about once every three seconds at the fast end. We found that all trials with $MTTF_{tile} < 5$ sec failed utterly, in the sense that the group lost its programmed PEC parameters and reverted to default settings (as described in Section 4.1.2). The data for survivable $MTTF_{tile}$'s are shown in Figure 7.



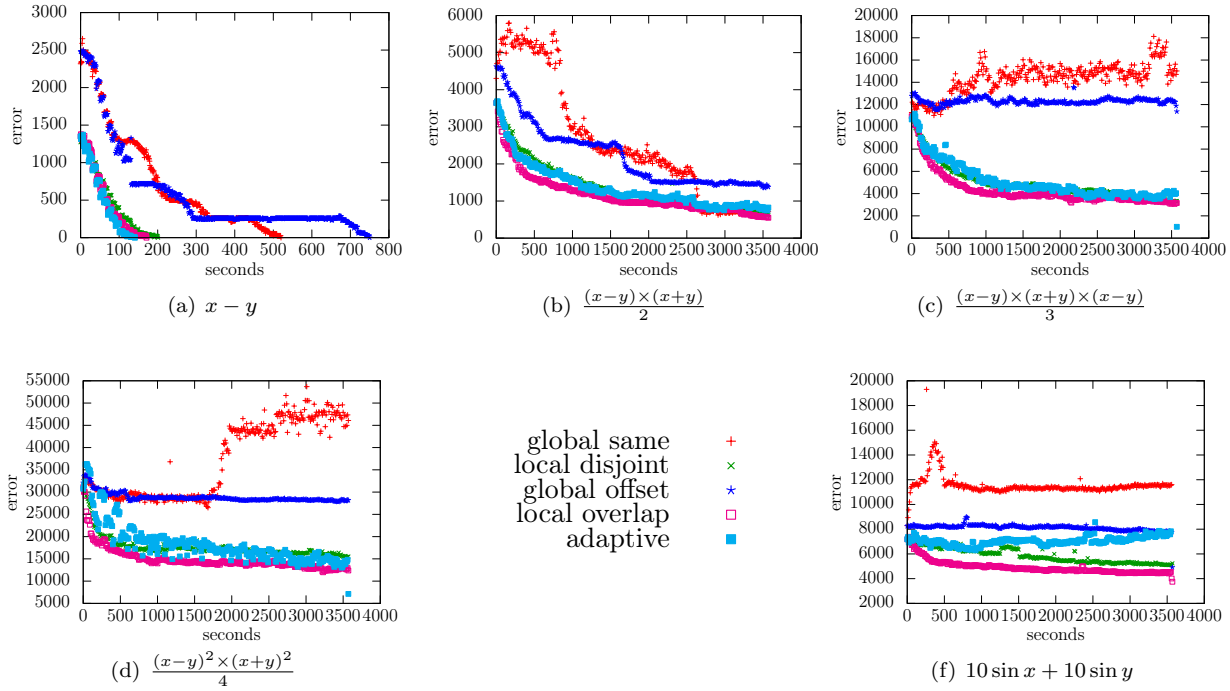
(a) Best Performers by Robustness



(b) Error by Robustness

Figure 7: Error by robustness over gravity data

In this data, perhaps surprisingly, the more reliable hardware does not always do better than the more fallible, particularly early on. In the first 50 seconds, $MTTF_{tile} = 50$ sec performs the best—though all are doing poorly—while from 50 seconds to 100 seconds the $MTTF_{tile} = 100$ sec tends to win, from 100 to 250 seconds $MTTF_{tile} = 500$ sec is preferable, and only in the final 300 seconds is the perfectly stable



(a) $x - y$ (b) $\frac{(x-y)\times(x+y)}{2}$ (c) $\frac{(x-y)\times(x+y)\times(x-y)}{3}$

(d) $\frac{(x-y)^2\times(x+y)^2}{4}$ (f) $10\sin x + 10\sin y$

Figure 6: Error by mapping scheme over analytic fitness functions in the Grid geometry

hardware ($MTTF_{tile} = \infty$ sec) the most consistently successful performer. Though the effect here is relatively weak, it prompts us to hypothesize that unreliable hardware may stand to *improve* performance in some cases by impacting the explore/exploit tradeoffs of a PEC process—because on each reboot the local subpopulation is re-populated with randomly generated individuals.

3.2.3 Spread of solutions through the group

As discussed above (section 2.3.1), the incorporation process evicts a randomly-chosen population member; we have so far made no use of elitism, and consequently it is possible for a good or perfect solution to be discovered only to be lost again.

Figure 8 shows six snapshots of a successful run against the second problem 3(b) using the *local disjoint* scheme 5(b). The exact solution first appears roughly 1900 seconds into the run 8(c), this solution is then immediately lost 8(d), and then quickly recovered 8(e) after which it overtakes the entire group 8(f).

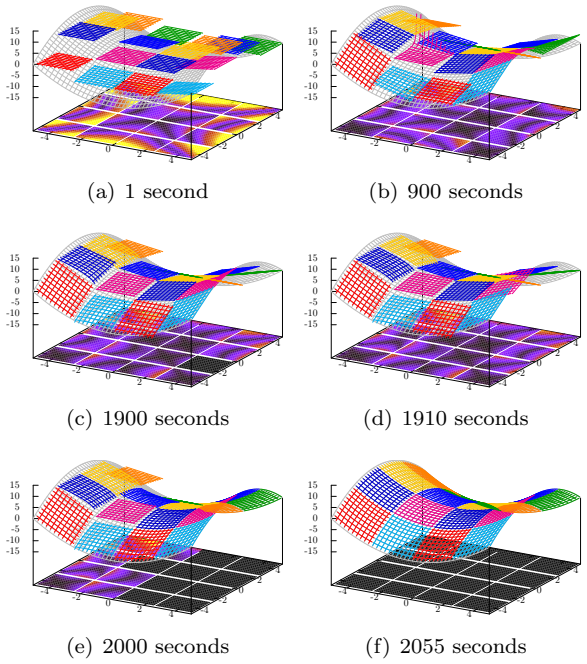
4 Discussion and Conclusion

4.1 Life with PEC

With their go-go operation and wild-and-woolly parallelism, PEC systems are much like natural systems, and for just that reason they violate typical assumptions of how a computer "should work." Here we offer two brief tales of mistakes we made during development, illustrating some of the perils of carrying serial and synchronous assumptions uncritically into this new context.

4.1.1 Instant Winners

All the data we have presented is gathered over repeated runs in each experimental condition, and per-



(a) 1 second (b) 900 seconds

(c) 1900 seconds (d) 1910 seconds

(e) 2000 seconds (f) 2055 seconds

Figure 8: Spread of the exact solution over the course of a run. A video of this run is available at the following url. `http://cs.unm.edu/~eschulte/research/pec/runs/grid.1.1.4.mp4`

forming those separate runs was accomplished using a controlling script running on the external computer, which sent periodic control messages to the connected tile group to begin and end runs and to change PEC parameters.

Since it is not possible to communicate with all tiles instantaneously, messages intended for global distribution pass through a group of tiles somewhat like ripples spreading on a pond, and during that process such messages race against all other messages moving in the system. For example, when a 'global reset message wave' reaches some tile and causes it to reset, there might still be an in-flight sharing message inbound to that tile from a 'downwave' neighbor, which can recreate an evolved individual and the PEC parameters that the global reset was attempting to erase.

During early data collection we found that sometimes fit individuals were surviving the 'official' end of one experiment and then immediately dominating the next experiment, looking like *extremely rapid* evolution! Now, however, the experiment driver script on the external machine is designed to recheck the state of tiles after a parameter change and to repeatedly "flood" the group until all tiles are on board—but without global privileged points in computational space-time there is simply no 100% reliable hook by which a PEC practitioner may take hold of computation.

A robust solution more in the PEC spirit, if one cared more about results than statistical independence, would be to abandon any pretense of dividing time into discrete 'experiments' and instead just periodically redrive the currently desired problem and parameters into the grid, whether it has changed or not. Or on the other hand, if independence is *really* critical, then falling back to the 'physically privileged point' in PEC—i.e., power-cycling the entire grid, with all the delays and inconvenience that implies—is the way to go.

4.1.2 Lord of the Flies

When a tile joins a working group, either due to it rebooting or being a hardware addition, it must learn the goal and parameters of the group. Immediately upon powering up a tile reads its own "inherent parameters" from persistent memory and begins evolving using those default settings. It then attempts to acquire the "cultural knowledge" of the goal and scheme in use by its neighbors in the group

Early versions of our PEC software failed to distinguish 'read from persistent memory' vs 'learned from neighbors' parameter sets. As a result, it could happen that a pair of adjacent tiles might both reboot in a narrow window of time, then share their inherent parameters with each other—each convincing the other that their inherent 'state of nature' settings were actually the 'culturally sanctioned' settings of the group as a whole. Small subgroups formed in this way proved persistent, and, in some cases, grew to overtake an entire group.

We addressed this by adding a *provenance bit* that tracks the source of a parameter set and allows a tile

to ensure it does not falsely advertise its inherent parameters as cultural. Still, however, we observed that in sparsely connected groups (e.g. configuration 4(c)), when combined with a high frequency of hardware reboots, it is possible for isolated subgroups to lose the sanctioned parameters of the group and to revert to evolving under their inherent parameters. Similarly, as tiles reboot, for whatever reason, they temporarily "drop out" of the group and stop passing messages, which can cause the network communication graph to become temporarily disconnected, exacerbating the global coordination issues mentioned earlier.

4.2 Scalability vs control

PEC's inherent scalability stands in stark contrast to the ultimate unscalability of serial EC strategies. On the other hand, however, tasks like pausing, restarting, or exactly repeating a computation—trivial in serial models—are decidedly non-trivial, or even impossible, with PEC. PEC tiles generate new solutions spontaneously and steadily, a process that is necessarily destructive as well as creative; lacking a synchronous global clock it isn't even easy to *define*, let alone capture, a consistent global snapshot of an entire group. Any communications must propagate through the group, at some finite velocity, while the group as a whole continues to evolve. Similarly it is impossible to write a complete state to a group of tiles as the state will change as it is being written.

On the other hand, a potential benefit of the focus on robust indefinite scalability is that, lacking any requirements for rigid global control, PEC carries far fewer design commitments than many parallel EC approaches, making it easier for PEC to 'play nice' within a larger computational system. Give a PEC grid power and a task, and it gets right to work without a lot of fuss; if the solutions coming back aren't good enough fast enough, it is easy to try plugging in more grid: robust indefinite scalability means the new hardware will automatically be recruited by the group and increase its computational power.

Indeed, so far our sense has been that those items which proved most difficult with PEC were often tasks of coordination and measurement desirable for researching *PEC itself*, while PEC's comparative strengths—such as non-stop and real-time operation and extreme robustness—are more likely to benefit 'embedded PEC practitioners' deploying evolutionary computation within a larger system design.

4.3 Computational Space-Time

The prototype hardware layer used in this effort continues a long history of implementing GP on distributed hardware systems [Sterling, 1998, Poli et al., 1999, Langdon, 2005, Andre and Koza, 1995, e.g.]. Our particular hardware system is unusual in the composability of its hardware tiles, with each physical connection simultaneously defining both a communications link and a relative spatial position.

The discrete two-dimensional space created by a group of tiles, augmented with the temporal dimension defined by tile's hardware timers, results in a discretized—if not globally synchronized—three-dimensional grid of computational space-time. Each event in this space can be seen as filling a narrow rectangle oriented to have either temporal or spatial extent. The event may extend in time while confined to a single spatial tile, or extend between adjacent tiles over a brief period of time. Composing such events is the job of the software designer writ anew, once we move beyond the CPU and admit that events shall be—in time and space—wherever we can desire to put them.

As computer engineers pack hardware into successively smaller and denser pockets of space and time it may become increasingly necessary or advantageous for computational models to acknowledge the basic realities of spatiotemporal locality, local causality, and imperfect reliability. As we hope this work suggests, evolutionary computations—perhaps slightly reconceived—are well-suited to such constraints and may even benefit from the imposed structure.

4.4 Reification

The rigid mapping between hardware elements and points in computational space allow for novel computational metrics. Distances and volumes in computa-

tional space-time may be measured in direct physical units. Each IXM tile, for example, measures 4.75 centimeters on a side, and hardware timers are scheduled with millisecond granularity.

Due to this reification of computational metrics, real ratios exist between measurements in the computational and problem spaces—like the legend on a map, each centimeter of computational space will represent a real distance in the problem space. Consequently, real-valued computational metrics such as the maximum rate of spread of a fit individual through a group (its '*innovation velocity*', perhaps) may be translated directly into real velocities in e.g., kilometers per hour in the problem space.

4.5 Abstract different

In its treatment of space and time, PEC is most aggressively concrete just where traditional models of computation are most aggressively abstract. But for all that, PEC still *is* an abstraction, smoothing over everything from the details of the fitness function and the specific genome representation and its interpretation, to the size and geometry of the tile group performing the computation, to the hardware details of the tile itself.

It is just a *different* abstraction, and, by representing space-as-space and time-as-time, as suggested above, it is an abstraction that allows studying couplings between the computational and the physical in a very direct, and we think rather refreshing, way. For example, it can become well-defined, potentially even useful, to consider questions that could hitherto have only seemed either frivolous or incoherent, such as: Is the idea of a breadbox bigger than a breadbox?

With the technical knowledge and manufacturing prowess gained from decades of von Neumann machine development, combined with the relentless cost reductions of multimillion unit blockbusters such as the cellphone, an opportunity is emerging for researchers to investigate substantially novel computational architectures—reslicing across the traditional lines of hardware and software, using masses of cheap and small von Neumann machines as a flexible computational 'clay', rather than continuing to push the serial computer as a scalable design in itself.

In the evolutionary computing community, as PEC illustrates, we can push our methods down closer to the hardware, and look towards full systems with embedded adaptive components, rather than continuing to strand evolutionary computing in the isolated environments of the standalone research tool.

References

[Andre and Koza, 1995] Andre, D. and Koza, J. R. (1995). Parallel genetic programming on a network of transputers. In Rosca, J. P., editor, *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, pages 111–120, Tahoe City, California, USA.

[Andre and Koza, 1998] Andre, D. and Koza, J. R. (1998). A parallel implementation of genetic programming that achieves super-linear performance. *Inf. Sci.*, 106(3-4):201–218.

[Arthur A. Bookstrom and Johnson, 1995] Arthur A. Bookstrom, G. L. R. and Johnson, B. R. (1995). *Montana state geologic map: a contribution to the Interior Columbia River Basin Ecosystem Management Project.* U.S. Dept. of the Interior, U.S. Geological Survey. http://pubs.usgs.gov/of/1995/of95-681/.

[Collins, 1992] Collins, R. J. (1992). *Studies in Artificial Evolution.* PhD thesis, UCLA, Artificial Life Laboratory, Department of Computer Science, University of California, Los Angeles.

[Eklund, 2002] Eklund, S. (2002). A massively parallel gp engine in vlsi. *Computational Intelligence, Proceedings of the World on Congress on*, 1:629–633.

[Langdon, 2005] Langdon, W. B. (2005). Pfeiffer – A distributed open-ended evolutionary system. In Edmonds, B., Gilbert, N., Gustafson, S., Hales, D., and Krasnogor, N., editors, *AISB'05: Proceedings of the Joint Symposium on Socially Inspired Computing (METAS 2005)*, pages 7–13, University of Hertfordshire, Hatfield, UK. SSAISB 2005 Convention.

[liquidware.com, 2010] liquidware.com (2010). The liquidware.com shop. http://liquidware.com/shop.

[Poli et al., 1999] Poli, R., Page, J., and Langdon, W. B. (1999). Smooth uniform crossover, sub-machine code GP and demes: A recipe for solving high-order boolean parity problems. In Banzhaf, W., Daida, J., Eiben, A. E., Garzon, M. H., Honavar, V., Jakiela, M., and Smith, R. E., editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1162–1169, Orlando, Florida, USA. Morgan Kaufmann.

[Stender, 1993] Stender, J. (1993). *Parallel Genetic Algorithms: Theory and Applications*. IOS Press, Amsterdam, The Netherlands, The Netherlands, 1st edition.

[Sterling, 1998] Sterling, T. (1998). Beowulf-class clustered computing: Harnessing the power of parallelism in a pile of PCs. In Koza, J. R., Banzhaf, W., Chellapilla, K., Deb, K., Dorigo, M., Fogel, D. B., Garzon, M. H., Goldberg, D. E., Iba, H., and Riolo, R., editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, page 883, University of Wisconsin, Madison, Wisconsin, USA. Morgan Kaufmann. Invited talk.

[Sutter, 2005] Sutter, H. (2005). The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3). http://www.gotw.ca/publications/concurrency-ddj.htm.