Continuous Computation: Taking Massive Parallelism Seriously[1]

B. J. MacLennan
Department of Computer Science
University of Tennessee
Knoxville, TN 37996-1301
mclennan@utkcs2.cs.utk.edu

1 Discrete vs. Continuous Computation

Modern computation is rooted in the discrete. This is most obvious in the digital (especially binary) representation of data. At a higher level, data is usually organized into structures with a finite number of discrete-valued features. The same characteristics also appear in the process of computation itself, which proceeds by discrete steps and must be completed in a finite number of steps. The discrete and the finite are the hallmarks of traditional computer science.

Recently there has been a rebirth of interest in analog computation, since for many applications it's convenient for each component of the data representation to take on one of a continuum of values, rather than one of a finite set of discrete values. However, this is just a part of what we mean by continuous computation.

The subject of this conference is emergent computation. It's based on the observation that when very large numbers of elements interact new and unexpected phenomena may emerge from their collective behavior. We hypothesize that many of these phenomena result from the fact that a *large* number of elements is an approximation to an *infinite* number of elements. To test this hypothesis we've been investigating *continuously parallel analog computation* [3, 4, 6, 7]. By 'continuously parallel' we mean the limit of 'massively parallel' as the number of processing elements becomes infinite (and in fact a continuum). In this computational paradigm representations do not have a finite number of discrete-valued features. Rather, they have a continuum of continuous-valued (i.e. analog) features. The processing elements also form a continuum — thus there is an uncountable infinity of

[1]Poster presentation at Los Alamos National Laboratory Center for Nonlinear Studies 9th Annual International Conference, *Emergent Computation,* Los Alamos, NM, May 22-26, 1989.

processors. Our motto has been, "If you can count them, you don't have enough."

2 Reasoning About Continuous Computation

Of course our postulation of a continuum of processors operating on continuous data structures is an idealization (just as there is a mathematically idealized theory of discrete computation). Actual implementations will often have only a (large) finite number of processors, and (large) finite numbers of elements in the data structures. Even the continuous values of the elements may be represented digitally. However, these are all considered discrete approximations to the idealized continuous computer. The requirement is that the number of elements be sufficiently large that continuous mathematics can be reliably applied. It has often been observed that the mathematics of the infinite and the infinitesmal (i.e. analysis) is simpler than the mathematics of the finite (i.e. combinatorics).

The traditional theory of computation is sequential, and parallel computation is often understood in terms of sequential computation (e.g., we imagine the operations of the processing units interleaved so that they take place sequentially). It is our hypothesis that this style of reasoning, which we call *covert sequentialism,* will not scale up. It may be adequate when there are a few hundreds or perhaps even a few thousands of processors, but it will be incapable of helping us to understand machines with millions of processors. Continuous parallelism is one way to fight covert sequentialism; since the points of a continuum cannot be processed sequentially, at discrete points in time, continuous parallelism forces us to adopt a new theory of computation.[2] Therefore, even though the number of processors may in facts be finite, we expect that it will be useful to think of them as forming a continuum.

Part of this wholesale adoption of the continuous is the assumption of continuous rather than discrete state transitions. In practice, of course, continuous state transitions may be approximated by discrete transitions in sufficiently small steps. The implications of assuming continuous transformation in the temporal domain are not clear. Although some differential equations may defy analytic solution, difference equations seem to be at

[2]Of course, a linear continuum can be sequentially processed providing the elements are processed at a finite *rate,* that is, provided the infinitesmal elements are processed in infinitesmal time intervals.

least as bad. Our guess is that, as usual, continuous mathematics will be more tractable than discrete.

In many cases we may be able to reason *qualitatively* about continuous computation; an example would be reasoning in terms of inequalities. This may provide a capability analogous to that which logic provides for discrete systems.[3] A tool for qualitative reasoning is desirable, since quantitative reasoning will prove too difficult in many cases. In other words, in many cases, the best source of a quantitative answer will be to run the computation. Qualitative reasoning will help by showing us how to design the program, and, for example, convince ourselves of convergence. The goal is to increase the rigor of the back-of-the-envelope reasoning that we currently do about energy surfaces and the like.

3 An Overview of Continuous Computation

What would a continuously parallel computer be like? For several years now we've been investigating a class of continuously parallel computers which we call *field computers* [3, 4, 6, 7]; this name is explained below.

We've said that in continuous computation, data are represented as continua of continuous-valued elements. Thus, if ϕ is such a data structure, it will have a value ϕ_t for each t in some continuum Ω. These continua are typically closed and bounded subspaces of a Euclidean space, since these are easiest to represent physically. (Other topologies are possible; e.g., we could have a continuum with the topology of a Möbius strip.) We further require that the values ϕ_t of the data structure be drawn from some continuum I, which is typically a closed interval of the reals, although finite dimensional vector spaces are also useful. Finally we require that data structures be continuous, and in fact that they be uniformly continuous (since physical media cannot sustain an infinite gradient). Thus they belong to a space $\Phi(\Omega)$ of continuous functions $\Omega \to I$. In practice we require that $\Phi(\Omega)$ be a subspace of $L_2(\Omega)$, that is, functions with finite L_2 (Euclidean) norms. Physically realizable structures have additional useful properties, such as bounded domain and range.

It will be apparent that the data structure described above is very much like the *fields* that are employed in science and engineering. Thus we may have scalar fields (such as potential fields) and vector fields (such as gradient

[3] For an example of a qualitative logic of the continuous, see [5].

fields). It is on this basis that we've chosen the name *field computer* for our class of computers.

Field computer may have a number of field storage units capable of holding fields of compatible type for as long as is required by the computation. These are analogous to the registers and memory cells of a digital computer. For concrete examples, think of a charge distribution on a plate, a light intensity distribution across an optical device, or the volume distribution of a chemical's concentration.

Computation on a field computer is required to be continuously parallel: an operation is applied to an entire field to yield another entire field. (Scalars are treated as zero-dimensional fields.) Thus computation proceeds by the application of various linear and nonlinear operators. We call an operator T a *field transformation* if

$$T: \ \Phi(\Omega) \to \Phi(\Omega').$$

Thus field transformations are operators between function spaces.

4 General Purpose Field Computers

An interesting question is whether there's a set of field transformations that's "universal" in the sense of allowing the computation of any field transformation in a large class. Such a universal set of operations would provide a theoretical basis for designing a general purpose (i.e. programmable) field computer (in much the same way that the universal Turing machine provides a theoretical basis for designing general purpose digital computers). We have shown [3, 7] that there is such a set of operations, namely these three:

- local addition: $(\phi + \psi)_t = \phi_t + \psi_t$.
- outer product: $(\phi \wedge \psi)_{st} = \phi_s \psi_t$.
- general product: $(\phi\psi)_{su} = \int_\Omega \phi_{st}\psi_{tu} \mathrm{d}\mu(t)$.

As is shown in the next section, these operations can be combined to approximate, to a desired degree of accuracy, most any field transformation (linear or nonlinear).[4]

[4]There are no doubt other sets of universal operations, but for theoretical purposes this set is sufficient.

It is of course to be expected that practical general-purpose field computers will have additional built in operations (e.g., convolution and correlation, integration, differentiation, gradient, scaling, thresholding, Fourier transformation). Further, we expect that the routing of fields through transformational units and to and from storage units will be under the control of something like a program. The principal difference is that the primitive operations will be field transformations rather than the usual simple digital operations on words and bytes.

Although many general purpose field computers will execute field transformations at discrete time intervals, there is no reason why we cannot design a field computer to operate with continuous state transitions (like the old analog computers). Unlike the old analog computers, the program for such a computer will be a set of differential equations that define the instantaneous transformation of *fields* (rather than scalars).

5 Polynomial Approximation of Field Transformations

The problem of approximating a desired transformation is that of finding a formula (in terms of field sums and products) that minimizes the error in some appropriate sense. The theory is much like the familiar theory of polynomial approximation, except that instead of scalar multiplication (and powers) we have field products (outer and general). For example, the Taylor theorem from functional analysis may be used to find locally good approximating series [3, 7]. To see this, note that the derivatives[5] of a field transformation, evaluated on a fixed field, are multilinear operators; therefore they can be computed by field product with a fixed field (see [7] for details).[6] We call these fields the *gradients* of the transformation, and write $\nabla^k T(\phi)$ for the kth gradient of T evaluated at ϕ. If the first n gradients of T are defined, then its Taylor expansion is:

$$T(\phi+\alpha) = T(\phi) + \sum_{k=1}^{n-1} \frac{\nabla^k T(\phi)\alpha^{(k)}}{k!} + R_n(\phi,\alpha)$$

where by $\alpha^{(k)}$ we mean the k-fold outer product $\alpha\wedge\alpha\wedge\cdots\wedge\alpha$ and $R_n(\phi,\alpha)$ is an appropriate error term.

[5] Given our realizability constraints, the Fréchet and Gâteaux derivatives are the same.

[6] Some of these fields may be "generalized functions" (e.g., Dirac deltas). Physical realizability requires that they be approximated.

The Taylor series expansion gives "polynomial" approximations which are locally good around the expansion point. This suggests that the more general problem is finding polynomials

$$T(\phi) \approx K_0 + K_1\phi + K_2\phi^{(2)} + \cdots + K_n\phi^{(n)}$$

that satisfy other, perhaps global, definitions of goodness. Our research is addressing this issue.

The compound outer products $\alpha^{(k)}$ that occur in these "polynomials" can be eliminated by writing the polynomial in "Horner's Rule" form:

$$T(\phi) \approx ((\cdots(K_n\phi + K_{n-1})\phi + \cdots + K_2)\phi + K_1)\phi + K_0$$

This form is especially useful for implementation by neural networks with conjunctive synapses (sigma-pi units), since each term of the polynomial is computed by a layer of the network.

6 Learning and Training

The problem of learning is that of finding a "polynomial" that either "passes through" the sample fields, or that minimizes an appropriate error measure. Although there is much to be done in the theory of learning in field computers, it appears so far that many of the neural network learning algorithms (e.g., outer product) generalize in the obvious way to the continuous dimensional case.

For a simple example, suppose that we have a finite set of sample input-output pairs (ϕ_k, ψ_k) and that these fields are band limited. Specifically, assume that only the first M generalized Fourier coefficients of the ϕ_k are nonzero, and only the first N of the coefficients of the ψ_k. Let $\overline{\phi}$ and $\overline{\psi}$ be the sample averages of the ϕ_k and ψ_k. Now suppose we want to approximate (in a least-squares sense) the mapping $\phi_k \mapsto \psi_k$ by a first-degree polynomial

$$\psi = K + L(\phi - \overline{\phi})$$

The fields K and L can be determined by linear regression. Therefore let $K = \overline{\psi}$; it remains to determine L.

Since the fields are band limited, we can write

$$L = F_N^{-1}\Lambda F_M$$

where the field vectors F_M and F_N^{-1} perform direct and inverse discrete generalized Fourier transforms:

$$F_M = \begin{pmatrix} e_1 \\ \vdots \\ e_M \end{pmatrix}, \quad F_N^{-1} = (f_1, \ldots, f_N)$$

(The e_k and f_k are orthonormal basis fields for the domain and range.)

Since Λ is an $N \times M$ matrix, the problem is now finite dimensional. Λ is the solution to the equation

$$P = \Lambda R$$

where R_{mn} is the correlation coefficient between the m^{th} and n^{th} Fourier coefficients of the $\phi_k - \overline{\phi}$, and P_{nm} is the correlation coefficient between the n^{th} Fourier coefficients of the $\psi_k - \overline{\psi}$ and the m^{th} Fourier coefficients of the $\phi_k - \overline{\phi}$. The matrix equation can be solved by any of the usual methods.

7 The Problem of Fields of High Dimension

If you look carefully at the "polynomials" in Section 5, you will see that if $T: \Phi(\Omega) \to \Phi(\Omega')$, then K_k is a field in

$$\Phi(\Omega' \times \Omega^k).$$

For example, if $\Omega = \Omega' = [0, 1]$ then K_k is a $k+1$ dimensional unit hypercube. Although this is not a problem in the mathematics, it does create an implementation problem. Space is three dimensional, so fields of greater than three dimensions cannot be represented by spatial distributions. There are several solutions to this problem. First, we can often "buy" an extra dimension by representing it through temporal extension (thus the field is represented by a space-time distribution). This however sequentializes some of the computation, and in any case represents only one additional dimension. Second, in many practical cases we find that the fields K_k are *sparse*; that is they are approximately zero except over a lower-dimensional subspace $\Upsilon \subseteq \Omega$. In this case we can approximately represent K_k using fields in $\Phi(\Upsilon)$. Third, we may discretize the field, that is, replace it by a field over a finite set. This has two problems. The obvious problem is that it introduces sampling or averaging error. The less obvious problem is that unless the K_k become smoother with increasing k, we will find that the number of points

increases exponentially (N^k). Finally, we can perform a partial discretization, by dividing one or more of the dimensions up into a finite number of regions, and hence reducing the field to one of lower dimension. For example, we can reduce a three dimensional field to a two dimensional field by slicing it into a finite number of layers and averaging across the thickness of each layer. This has two problems: it deviates from continuous parallelism, and it introduces error since the pieces must be joined continuously. All of these solutions are probably workable in different situations, but the problem of dimension has not been eliminated. We plan empirical studies to determine the practical significance of the problem.

8 Representation of Constituent Structure

In most cases it is clear (at least in principle) how field computers can be made or programmed to perform many sensorimotor and perceptual tasks. Associative memory and pattern recognition in these domains are also understandable. It is much less clear how field computers can be made to process higher-level, apparently more structured information, such as linguistic structures. In particular, the constituent structure of natural language seems to allow — syntactically — the nesting of structures to any depth. Of course, there are pragmatic limitations to the depth that actually occurs, no doubt due to neuropsychological limitations. Representations of constituent structure proposed to date, for example, in neural networks, have been ad hoc and unconvincing. We discuss briefly one way that field computers can represent constituent structure.

Call a space *self-similar* if it's homeomorphic to a proper subspace of itself. Furthermore, call it *twice self-similar* if there are two proper closed subspaces of the space that can be separated by open sets (this will be the case if the space is T_4, i.e. satisfies Tietze's first axiom), and that are each homeomorphic to the superspace. N-times self-similar spaces are defined in the obvious way.

If a space Ω is twice self-similar, then it can be embedded in itself in two different ways. This allows the representation of binary trees of any size in these spaces. To see how, note that if Λ and P are disjoint closed subsets in a T_4 space then there is a continuous function f into [0, 1] that's identically 1 on Λ and identically 0 on P. Suppose that Λ and P are homeomorphic to Ω and that λ is the homeomorphism of Ω into Λ, and ρ that of Ω into P. Let $l : \Omega \to \Omega$ and $r : \Omega \to \Omega$ be any continuous extensions of λ^{-1} and

ρ^{-1}. Define the "construction" (as in LISP 'cons') of the fields ϕ and ψ as follows:

$$[C(\phi,\psi)]_t = f_t \phi_{l_t} + (1 - f_t)\psi_{r_t}$$

If $\chi = C(\phi,\psi)$ then we can recover the left and right components by $\chi \circ \lambda$ and $\chi \circ \rho$. Since C is a homeomorphism $C : \Omega^2 \to \Omega$, we can construct trees arbitrarily deep, for example,

$$C[C(\alpha,\beta), C(\gamma,\delta)]$$

and later recover their components.

We must point out that the foregoing analysis depends on idealized fields (no noise, etc.). Physically realizable fields, which are subject to noise and unable to sustain infinite gradients, will not permit unlimited nesting. The most deeply nested subtrees will become degraded to the point that eventually they cannot be recovered at all. This is very attractive, however, since the pragmatic limitation on nesting emerges in a natural way from the properties of the representing medium. The result has an obvious application to neural networks.

9 Field Computation of Simulated Annealing

In simulated annealing we are trying to find the best field according to some evaluation operator [2]. This is done by making random perturbations in the field. If a perturbation improves our evaluation of the field, then it's accepted. If it doesn't improve it, then with some probability it may be accepted anyway. This probability is an increasing function of a parameter called the "computational temperature" of the system. During the optimization process this parameter is gradually decreased according to an "annealing schedule." The necessity of processing the entire field makes simulated annealing an ideal application of field computation. To find the field ϕ that minimizes the evaluation functional E the following steps are iterated:

1. Evaluate the current field:

$$e = E(\phi)$$

By field computation E operates on the entire field in parallel.

2. Randomly perturb the field to get a new guess:

$$\phi' = P(\phi)$$

The perturbation is typically small so that we don't "overshoot" good solutions. There are many ways of computing the perturbation, but perhaps the easiest is to let $P(\phi) = \phi + \rho$ where ρ is random and small ($||\rho|| \leq \epsilon$). There are several possible sources for such a random field. One is to obtain it from a generator of true random fields. Another is to generate N pseudorandom numbers and to use them as the first N Fourier coefficients of a field. Thus,

$$\begin{aligned} \rho \quad = \quad & \epsilon \text{ normalize}(\sigma) \\ & \text{where } \sigma = r_0 e_0 + \cdots + r_N e_N \\ & \text{and normalize } \sigma = ||\sigma||^{-1}\sigma \end{aligned}$$

Here the e_k are the basis fields and the r_k are random numbers.

3. Evaluate the perturbed field. If the perturbed field decreases E, then it's accepted:

$$\begin{aligned} & e' = E(\phi') \\ & \Delta e = e' - e \\ & \text{if } \Delta e < 0 \text{ then } \phi := \phi' \end{aligned}$$

4. If the perturbation increases E, it is still accepted with a probability depending on the "temperature" T:

$$\text{if } \Delta e \geq 0 \text{ and ran} \geq \exp(-\Delta e/kT) \text{ then } \phi := \phi'$$

where ran is a random number and k adjusts the overall algorithm.

5. Finally, the temperature T may or may not be decreased, as determined by the annealing schedule.

Notice that in this field computation algorithm the perturbation operator may alter any or all of the field ϕ. This differs from typical implementations of simulated annealing, which update one "atom" at a time. Such serialization may be approximated in field computation, but parallel perturbation agrees more closely with real annealing.

10 Field Computation of Genetic Algorithm

To further illustrate field computation, we show how a field computer might be used to implement a variant of the genetic algorithm [1]. We let Ω be the "genetic space"; it has the form:

$$\Omega = \Omega_1 \times \Omega_2 \times \cdots \times \Omega_N$$

where the Ω_n are the sets of alleles for each gene (note that the space of alleles may be a continuum). The state of the simulation is represented by a population density field $\pi : \Omega \to [0, 1]$. We simulate the evolution of π by iterating the following steps:

1. Compute the fitness of the population:

   $$\phi = F(\pi)$$

   Notice that the fitness may depend on the population density; this is more general than the most common genetic algorithms, in which the fitness is a constant field over the genetic space.

2. Compute the population-weighted fitness and normalize it to get the probability of breeding:

   $$\beta = \text{normalize}(\phi \times \pi) \quad (\times \text{ denotes local, i.e. pointwise, product})$$

3. Compute genotype population density after crossover:

   $$\gamma = \mathrm{X}(\beta \wedge \beta)$$

   Since the crossover operation is bilinear there is a field X that permits the crossover to be computed as a general product.

4. Compute the new population density after random mutation:

   $$\pi' = \mu \otimes \gamma$$

   where $\otimes$ represents convolution. Here μ is a probability density field representing the probability of mutation; μ_v is the probability of a mutation vector v, and μ_0 is the probability of no mutation.

There is an important difference between this algorithm and the usual genetic algorithms. The fields involved (π, ϕ etc.) have as their domains the entire genetic space Ω. Thus the algorithm is evaluating in parallel *all possible* genotypes, not just those represented in the population. It thus is *explicitly* parallel where traditional genetic algorithms are *implicitly* parallel [1, p. 20]. The trouble with such explicit parallelism is that the genetic space may be very "big," that is, of high cardinality if the Ω_n are discrete, or of high dimension if the Ω_n are continuous. Therefore, we expect this algorithm to work best where we are trying to optimize in a space defined by a modest number of continuous-valued parameters.

11 Conclusions

Emergent computation depends on massive parallelism. We have argued that truly massive parallelism occurs when the processing and data elements are sufficiently numerous that they can be treated as a continuum, a situation we call *continuous parallelism.* We have sketched a theory of continuously parallel analog computation, called *field computation,* and have shown a theoretical basis for a universal field computer. We have shown how constituent structure, simulated annealing, and a variant of genetic algorithms can be implemented on field computers. The theoretical basis for field computation is now mostly complete. The next step is to investigate practical issues in the architecture of general-purpose field computers, and in their implementation in electronic, optical and molecular technologies.

12 Bibliography

1. Goldberg, D. E., *Genetic Algorithms in Search, Optimization, and Machine Learning,* Addison-Wesley, Reading, 1988.

2. Kirkpatrick, S., Gelatt, C. D., Jr., and Vecchi, M. P., "Optimization by Simulated Annealing," *Science,* Vol. 220, No. 4598 (13 May 1983), pp. 671-679.

3. MacLennan, B. J., "Technology-Independent Design of Neurocomputers: The Universal Field Computer," *Proceedings, IEEE First International Conference on Neural Networks* (June 21-24, 1987), Vol. III, pp. 39-49.

4. MacLennan, "Field Computation and Nonpropositional Knowledge," Naval Postgraduate School Technical Report *NPS52-87-040,* September 1987.

5. MacLennan, "Logic for the New AI," *Aspects of Artificial Intelligence,* J. H. Fetzer (ed.), Kluwer, Dordrecht, 1988, pp. 163-192.

6. MacLennan, "Field Computation: A Model of Massively Parallel Computation in Electronic, Optical, Molecular and Biological Systems," extended abstract in Proceedings of AAAI Spring Symposium, *Parallel Models of Intelligence: How Can Slow Components Think So Fast?,* Stanford, March 22-24, 1988, pp. 180-183.

7. MacLennan, "Outline of a Theory of Massively Parallel Analog Computation," *International Joint Conference on Neural Networks* (June 18-22, 1989), accepted for poster presentation, abstract to appear in proceedings.