Evolving Multi-cellular Artificial Life

Kurt Thearling
Thinking Machines Corporation
245 First Street
Cambridge, MA 02142
kurt@think.com

Thomas S. Ray
ATR Human Information Processing Lab
2-2 Hikaridai, Seika-cho, Soraku-gun
Kyoto 619-02 Japan
ray@hip.atr.co.jp

Abstract

In this paper we describe a number of experiments in which the ideas of multi-cellular evolution are applied to digital organisms in an artificial ecology. The individual organisms are parallel programs in a shared memory virtual computer where evolution by natural selection is shown to lead to increasing levels of parallelism.

1 Introduction

One of the greatest challenges in the development of naturally evolving artificial systems is crossing the threshold from single to multi-cellular forms. From a biological perspective, this transition is associated with the Cambrian explosion of diversity on Earth. During the Cambrian explosion, most of the complexity that we see in living organisms emerged rather abruptly some six hundred million years ago. The work presented in this paper is based on the Tierra system [2] in which the evolving entities are self-replicating machine code programs. Multi-cellular digital organisms are parallel processes. From the computational perspective, the objective is to use evolution to explore the as yet under-exploited possibilities inherent in parallel processing.

Transferring the concept of multi-cellularity from the organic to the digital domain could take many forms. To make the transfer we must first understand what the most basic, essential, and universal features of multi-cellularity are, and then determine the form that these features would take in the completely different physics of the computational system into which evolution is being introduced. The features that we will capture in the present model are: 1) that multi-cellular organisms originate as single cells, which develop into multi-celled forms through a process of binary cell division; 2) that each cell of a multi-celled individual has the same genetic material as the original cell from which the whole developed; and, 3) that the different cells of the fully developed form have the potential for differentiation, in the sense that they can express different parts of the genome (i.e., each cell can execute different parts of the program).

In the digital metaphor of multi-cellularity, the program is the genome, and the processor corresponds to the cell. In organic biology, there is at least one copy of the genome for each cell, because genetic information can not easily be shared across cell membranes. In most current parallel architectures, the same holds: since memory is not shared, there is an area of memory associated with each processor (cell), and there must be at least one copy of the program code in the memory of each processor. This provides a very simple model of multi-cellularity: each digital cell consists of a unique block of memory with its own copy of the program and its own processor.

However, if the parallel machine has a shared memory architecture, making copies of the genome for each cell needlessly wastes memory and processing time (to copy the genetic information). In this context evolution by natural selection would not likely find any advantage in such waste. Thus a more logical and efficient implementation in this evolutionary context is to share a single copy of the program in a single block of memory among multiple processors. Each cell in a single organism corresponds to a parallel processor. Multi-cellularity can develop from a single original processor through a process analogous to cell division. The initial cell (processor) can issue an instruction which would then create another cell (a parallel processor). They may exhibit cell differentiation by having different processors executing different parts of the shared program. Obviously all cells will contain the same genetic material, since there actually will be only one copy per multi-cellular individual. The work presented in this paper is based on this shared memory model of multi-cellularity.

2 Tierra

The Tierra system has already been described in detail elsewhere [2-5] so it will be described only briefly here. The software used in this study is available over the net or on disk[1]. A new set of computer architectures and associated machine code have been designed to withstand the genetic operations of mutation and recombination. This means that computer programs written in the machine code of these architectures often remain viable after being randomly altered by bit-flips which cause the swapping of individual instructions with others from within the instruction set, or by swapping segments of code between programs (through a spontaneous sexual process). These new computers have not been built in silicon, but exist only as software prototypes known as ‘‘virtual computers,’’ and have been called Tierra, Spanish for Earth.

Initially a self-replicating program was written in Intel machine language. This program was then implemented in

1. The complete source code and documentation (but not executables) is available via anonymous ftp from tierra.slhs.udel.edu and life.slhs.udel.edu in the file: tierra/tierra.tar.Z.

To appear in the *Proceedings of Artificial Life IV*, R. Brooks and P. Maes (eds.), MIT Press: Cambridge, July 1994.

the first Tierran language in the fall of 1989. The program functions by copying itself one byte at a time to another location in memory and dividing (i.e., giving the copy its own instruction pointer). Subsequently, both programs replicate, and the number of programs "living" in memory doubles in each generation.

These programs are referred to as "creatures" or "organisms." The creatures occupy a finite amount of memory called the "soup." The operating system of the virtual computer, Tierra, provides a "slicer" service to allocate CPU time to the growing population of self-replicating creatures. When the creatures fill the soup, the operating system invokes a "reaper" facility which kills some creatures to insure that some memory will remain free for occupation by newborn creatures. Thus a turnover of generations of individuals begins when the memory is full.

The operating system also generates a variety of errors which act as mutations. One kind of error is a bit-flip, in which a zero is converted to a one, or a one is converted to a zero. This occurs in the soup, which is the RAM memory where the "genetic" information that constitutes the programs of the creatures resides. The bit-flips are the analogs of mutations, and cause swapping among machine code instructions. Another kind of error imposed by the operating system is called a "flaw." A flaw causes possible errors in calculations taking place within the CPU of the virtual machine, slight alterations during the transfer of information, or error in the location of memory accesses.

The machine code that makes up the program of a creature is the analog of the genome, the DNA, of organic creatures. Mutations cause genetic change and are therefore heritable. Flaws do not directly cause genetic change, and so are not heritable. However, flaws may cause errors in the process of self-replication, resulting in offspring which are genetically different from their parents. Those differences are then heritable.

The self-replicating program (creature) running on the virtual computer (Tierra), with the errors imposed by the operating system (mutations) results in precisely the conditions described by Darwin as causing evolution by natural selection [1]. This is therefore an instantiation of Darwinian evolution in a digital medium.

2.1 Adding multi-cellularity to Tierra

To implement the ideas of basic multi-cellularity, two additional Tierra machine instructions had to be created to allow the digital organisms to carry out development. The most obvious candidate for inclusion was an instruction that would create an additional CPU for the creature. This process was modeled on binary cell division on purpose so that the execution of this instruction (which is called **split**) takes a single processor (CPU) and produces two CPUs upon completion of the **split**. When a creature is given a new time-slice by the Tierra simulator, each CPU in that creature is allocated its own copy of that time-slice.

The process of executing a **split** instruction follows. A new CPU is created for the current creature. The registers, stack, and IP for this new CPU are copied from the CPU that issued the **split** instruction. To differentiate between the CPUs after splitting, the DX register of each CPU is modified by shifting the value one bit to the left, and for the new CPU a value of 1 is added. As a result, each CPU's DX register has a different value after the **split**. If a sequence of **split** instructions is considered, the DX registers of each of the parallel CPUs within the creature will contain the address of their position in a binary tree of splits. For example, consider the following code fragment (assume that code is initially executed by a single CPU creature):

```
split    ; create 2 CPUs
split    ; split both CPUs, creating 4 CPUs
```

During the first split the DX registers for both CPUs are modified, first by shifting left by one bit (which has no effect since it is assumed that the DX register has previously been initialized to zero) and then by adding a value of 1 for the new CPU. Both CPUs then execute the second **split** instruction, creating two more new CPUs. The leftward shift and conditional addition to the DX register causes the four CPUs to end up with DX values of 00, 01, 10, and 11. This enumerates all four CPUs with different DX values from zero to three. In parallel processing terminology, the DX register contains the "self-address" of the parallel CPU.

In addition to the **split** instruction, an additional parallel construct was implemented: the **join** instruction. Once a CPU issues a **join** it waits until all other CPUs have also issued a **join**. Then all CPUs, other than the original single cell CPU, terminate. The **join** instruction was created to overcome Tierra's limitation of allowing only one CPU in a creature to issue a **divide** without causing an error. It was decided that an appropriate way to deal with this limitation was for each CPU to issue a **join** immediately before attempting to divide mother and daughter. There is no real organic biological analogy to the **join** instruction. It was introduced because it is a useful parallel programming tool. Another possible solution would have had a creature conditionally executing the **divide** instruction so that only one CPU actually performed the division of mother and daughter.

2.2 First steps

As with the original Tierra research, an "ancestor" was created to inoculate a new soup. The first parallel ancestor was designed to be very similar to the original serial ancestor described in [2]. Using the **split** and **join** instructions, it was possible to modify the original single celled ancestor and parallelize its functionality. The operation of the original single cell ancestor (as described in [2]) follows.

The ancestor first examines itself to determine where in memory it begins and ends. This is done by searching backward for the template that appears at its beginning and then searching forward for the template that matches its ending. To determine its size, the beginning address is subtracted from the end address. Space for the daughter is then allocated using this size information. The ancestor then calls the copy procedure which copies the entire genome into the daughter cell memory, one instruction at a time. Once the genome has been copied, it executes the **divide** instruction, which causes the creature to lose write privileges on the daughter cell memory, and gives an instruction pointer to the daughter cell (it also enters the daughter cell into the slicer and reaper queues). After this first replication, the mother cell does not examine itself again; it proceeds directly to the allocation of another daughter cell, then the copy procedure is followed by cell division, in an endless loop.

Only the copy loop was parallelized in the parallel ancestor, with one CPU copying half of the genome and another CPU copying the rest. Currently there is an arbitrary limit of sixteen CPUs per organism. The parallel ancestor uses only two CPUs. Any additional parallelism would have to evolve. The basic approach to parallelizing the single celled ancestor was to **split** immediately after allocating space for the daughter cell and to perform a **join** immediately before the **divide**. Within the copy loop the first CPU will copy the even numbered instructions and the second CPU will copy the odd numbered instructions.

Unfortunately the original instruction set was not very rich in its ability to manipulate registers. In fact, there was no way to operate using the DX register directly and therefore differentiating between parallel CPUs was very difficult. To operate on the DX register, it was necessary to push DX onto the stack and then pop it off into one of the registers that could be operated on. For example, consider the process of aligning one of the two parallel CPUs to copy the odd numbered instructions, conditional on the value in the DX register. The following code fragment performs this task:

```
pushC    ; save the CX register on the stack
pushD    ; push DX onto the stack
popC     ; now pop the value from DX into CX
ifz      ; if CX (aka DX) == 0
incA     ; inc AX (destination) to odd align
ifz      ; if CX (aka DX) == 0
incA     ; inc BX (source) to odd align
popC     ; return CX to its original value
```

Obviously it is difficult to achieve some fairly simple operations using the available instructions. The major cause of this difficulty results from an inability to act directly upon the register used differentiate between multiple CPUs in an organism (DX). As a result, any evolutionary activity that involves parallel CPUs must manipulate both the DX register as well as any other registers it uses actually to perform the desired operation. This greatly complicates the process and makes evolutionary improvements much more difficult to perform. Simple operations become somewhat "brittle" as a result of this limitation.

Figure 1 illustrates reproduction time[2] vs. time for a typical run using this instruction set. The new (parallel) ancestor starts out approximately twice as fast as the old ancestor, because it uses two CPUs rather than one. All of the evolutionary improvement is incremental, using serial processing improvements similar to the improvements in the original ancestor runs. No additional parallelism is added via evolution. Also note that the same parasitism and other natural phenomena that are described in [2] are observed in these new runs with multi-cellular digital organisms, as well.

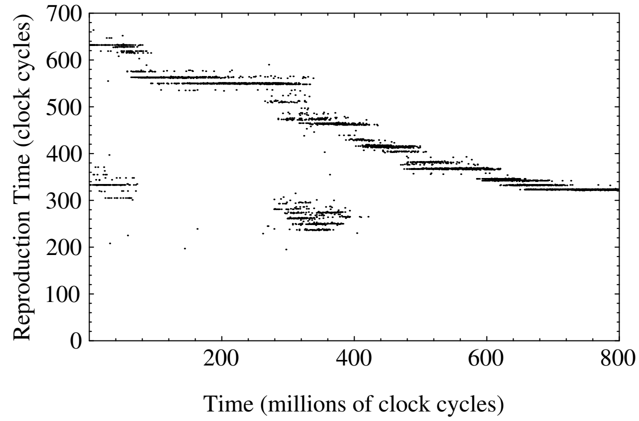

Figure 1: Evolution and the new (parallel) instruction set

2.3 Creating an evolvable parallel ancestor

This observed behavior was not unexpected. When the second author originally developed the Tierra instruction set, he first wrote the original ancestor and then included in the instruction set only those instructions which were used in that ancestor. Since multi-cellular digital organisms are naturally more complex than single cell digital organisms, an instruction set sufficient for a single cell organism would not necessarily suffice for a parallel organism.

To alleviate some of these problems, several additional instructions were added. These instructions perform operations that simplify the design of a multi-cellular ancestor. The first instruction added was **zeroD**, which explicitly zeros out the DX register. The second new instruction was **shr**, a shift right instruction for the CX register. Since the CX register typically contains the creature's size, a **shr** effectively divides the size by two. When performed in conjunction with a **split**, the CX register is modified to contain the portion of the genome to be copied by each parallel CPU.

The final two instructions that were added were based on known techniques for distributing work among parallel processes. An "offset" instruction takes the size (which is usually divided by the number of CPUs) and multiplies it by the CPU's self-address. This value specifies an offset into memory which evenly divides the memory among parallel CPUs. When added to a base address, the value specifies where in memory each parallel process should begin accessing data. Two versions of the offset instruction were created for the multi-cellular instruction set: **offAACD** and offBBCD. The **offAACD** instruction multiplies the CX register (size) times the DX register (self-address) and adds it to the AX register (the base address). The **offBBCD** instruction is similar except that BX is used instead of AX as the base register.

3 Evolution and Multi-cellularity

Using these new instructions, a new parallel ancestor was created. The multi-cellular ancestor is very similar to the ancestor described in [2] and is composed of 82 instructions.

2. Unlike the results presented in [2] which focused on creature size, reproduction time is now used to describe the progress of evolution of a digital organism. Since multiple CPUs are now possible within a single creature, the relationship between size and reproduction time is such that reproduction time is no longer directly inferrible from the size. A long creature with many parallel CPUs might very well reproduce faster than a short creature with few parallel CPUs.

The copy loop is parallelized for two CPUs, with each CPU copying half of the genome from mother to daughter (unlike the previous scheme, one CPU copies the first half of the genome while the other CPU copies the second half).

Once the new and improved multi-cellular ancestor had been created, any further improvements in its performance would be generated through evolution. Comparing the multi-cellular ancestor with its single celled cousin, the multi-cellular creature is clearly the more efficient reproducer. While the single celled ancestor requires approximately ten clock cycles per instruction copied, the multi-cellular ancestor requires only five (since two CPUs are operating in parallel, they are effectively doing the same amount of work in half the time). As a result, a multi-cellular ancestor will produce nearly twice as many offspring as a single celled ancestor in the same period of time. Obviously multi-cellular organisms will have an advantage and will dominate the population.

Experiments were run using the multi-cellular ancestor on a new version of Tierra[3] that runs on a Connection Machine CM-5 massively parallel supercomputer. By taking advantage of the size and speed of a supercomputer, much larger and faster evolutionary simulations have been achieved.

Figure 2 (top) shows a graph of reproduction time versus time for the new multi-cellular ancestor. For the first 200 million instructions, there is a gradual improvement in reproduction time due to optimizations such as template size reduction and taking advantage of the side effects of some instructions (upper band). In addition, there is also effective parasitism (lower band) until approximately 150 million clock cycles at which time most organisms become resistant to parasites. This type of behavior also manifested itself in simulations using single cell organisms [2].

A sharp discontinuity then appears at approximately 215 million clock cycles and represents a thirty percent improvement in reproduction time. This new optimization is added parallelism, and it corresponds to an increase from two to four CPUs per organism. In the genome length versus time graph (center), this change is even more noticeable since the dominating organisms have actually increased in size from 44 to 52. While the size 44 creatures have only two CPUs, the size 52 creatures have four. The larger but more parallel creatures are faster reproducers and as a result take over the population. This increase in parallelism is even more obvious when examining the graph of reproduction efficiency (the average number of clock cycles necessary to copy a single instruction from mother to daughter) versus time (bottom).

One noticeable characteristic the genome length versus time graph shows is that when there are two CPUs, reductions in size typically take place in multiples of two instructions. When there are four CPUs per organism, the multiple increases to four. Obviously the creatures are dividing the workload evenly and are not able to handle circumstances which do not provide even workloads. When the first size 52/four CPU creatures appear, fifteen out of the 52 instructions are meaningless (i.e., these instructions do not affect the execution of the creature's algorithm). In some sense these instructions are a form of computational intron, pieces of unnecessary code left over from some dead ancestor. The introns are used to pad out the size of a creature so that it is a multiple of four, simplifying the distribution of work among the parallel CPUs.

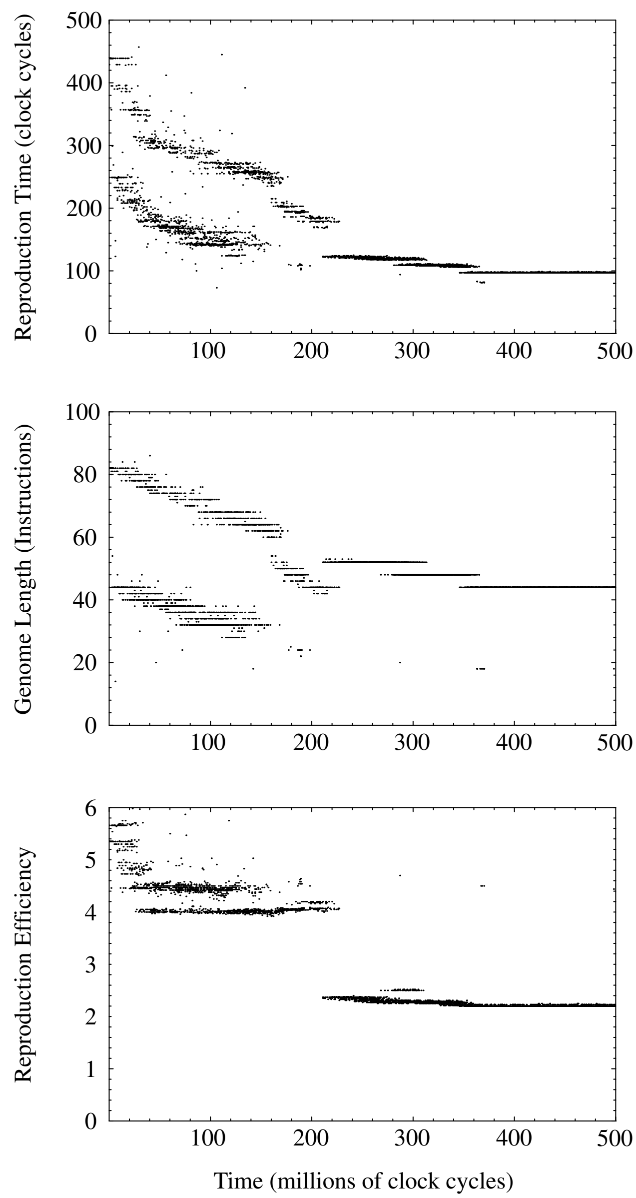

Figure 2: Various evolution characteristics vs. time

A large number of intron instructions are observed when the first four CPU creatures appear. Through evolution, the size of the creatures decreases to 48 (268 million clock cycles), then 44 (346 million clock cycles) and finally to 40 (1 billion clock cycles). In each of these improvements, evolution simply removes some intron instructions while keeping the length a multiple of four. If the intron instructions are removed from the size 40, 44, 48, and 52 creatures, we find that they are the *exact* same algorithm. Obviously at that point in

3. Although Tierra now runs on a parallel supercomputer, the parallelism in the digital organisms is unrelated to the parallelism in the computer that they are running on. Slower and smaller simulations of multi-cellular creatures have been run on workstations.

the process, evolution has optimized the basic algorithm as well as it can without major restructuring.

The size 40/four CPU creature that evolved from the size 82/two CPU ancestor is quite efficient in its use of parallelism. The genome of one such creature (0040aba) is:

```
nop0     ; beginning template
adrb     ; find beginning + template size
nop1     ;
subAC    ; sub template size from beginning
movAB    ; put beginning in BX
adrf     ; find end
nop0     ;
nop0     ;
subCAB   ; calculate size
mal      ; allocate space for daughter
incC     ; intron (since CX no longer used)
split    ; 2 CPUs
ifz      ; intron (since CX can't be zero)
movCD    ; intron (since ifz not true)
shr      ; size = size / 2
offAACD  ; split genome into 2 halves &
offBBCD  ; adjust AX and BX accordingly
zeroD    ; zero out DX before second split
pushB    ; save beginning on stack
shr      ; size = size / 2
split    ; 4 CPUs
offAACD  ; re-partition genome into 4 and
offBBCD  ; adjust AX and BX accordingly
nop1     ; copy loop starts here
nop0     ;
movii    ; copy instr from mother to daughter
decC     ; decrement number of instr to copy
ifz      ; if number of instr to copy == 0
jmp      ; jump forward to just before join
nop0     ;
incA     ; increment source address
incB     ; increment destination address
jmpb     ; jump back to start of copy loop
nop0     ;
nop1     ;
join     ; join up multiple CPUs
divide   ; divide mother and daughter
ret      ; return to beginning of creature
nop1     ; ending template
nop1     ; ending template
```

The first thing to notice is that there are still three intron instructions remaining. Simple removal of these introns is not possible since the workload distribution among the four CPUs requires the size to be a multiple of four. Unless the algorithm is radically changed, it would be difficult to evolve a more compact version of this creature. The first thing this creature does is examine itself (beginning and ending) and compute its size. It then allocates space for its daughter. This process is common to almost all viable creatures in the soup.

The next thing that happens is that the creature splits into two CPUs and then divides its size by two. The new size is then used to offset the source and destination registers for the move. Essentially one CPU copies the first half of the genome while the other CPU copies the second half. The creature then zeros out the DX register before splitting again, creating a total of four CPUs. Just before the **split**, the BX register (which contains the address of the beginning of the mother) is pushed onto the stack. This will be used later when the reproduction process has completed.

After the second **split**, the size once again divides by two, leaving the CX register with a value of size divided by four. Each CPU then offsets the source and destination registers again. Since the CPUs had previously partitioned the genome into halves, they are now hierarchically re-partitioning each of the halves into quarters.

Once each of the four CPUs is set up to copy its quarter of the genome, it enters a copy loop similar to the copy loop in the multi-cellular ancestor. After the copy loop is complete, each CPU waits for the others via a **join**, and once all four CPUs have joined, the creature issues a **divide** for its daughter. After the **divide** a return is performed, which pops the stack into the IP. Since the beginning address of the creature was previously pushed onto the stack, this return causes the creature to start all over again at the beginning of its genome.

3.1 Taking advantage of its creator

In one of the first simulations (after the addition of the new instructions), a very strange set of results was observed. Somehow the creatures had increased their reproductive efficiency so that it appeared to take only two clock cycles to copy the mother's entire genome to the daughter. Even if the creatures had managed to make use of the maximum number of CPUs allowed (sixteen), it would have been impossible to reproduce that quickly. Somehow the creatures must have taken advantage of a bug in Tierra enabling them to reproduce faster than should have been possible. When one of these creatures was examined, it appeared thus:

```
template marking beginning

split    ; 2 CPUs
split    ; 4 CPUs
split    ; 8 CPUs

find beginning, end, and size

divide   ;
mal      ; allocate space for daughter

copy loop
```

For some reason the first thing that happens is the creation of eight parallel CPUs. The creature finds its beginning and end and calculates its size (typical for all creatures). But then it attempts to **divide**, which won't work since the daughter's space has yet to be allocated. The next instruction allocates space for the daughter, which somehow seems to be in the wrong order. Finally a copy loop is entered to copy the mother's instructions to the daughter's space. According to the information saved about this genome, the mother and daughter were genetically identical. Somehow this process must be working correctly since it allows the mother to reproduce.

The key to understanding this process involves the way in which the **split** instructions adds additional CPUs to a creature. In the original version of the multi-cellular Tierra simulator, the execution of a **split** instruction would cause a CPU immediately to exit its time-slice. Any unused cycles left in the time-slice would have been added to the time-slice that both CPUs would receive the next time through the execution queue. As a result of performing three splits in a row, the organism created eight CPUs. These eight creatures each execute (on average) three time-slice's worth of instructions before moving onto the next parallel CPU. So, when the first of eight CPUs issues the **divide**, nothing happens (other than setting an error flag). That CPU then allocates space for a daughter and starts copying its instructions to the daughter.

It turns out that three times the average time-slice size is just enough time to copy all of the instructions to the daughter. So, by the time the slice is over the daughter is complete. After the slice ends, the next CPU performs a **divide**. Unlike the first **divide**, which generated an error because no daughter had yet been allocated, the second **divide** completes correctly. The daughter for the second **divide** is actually the daughter created by the first CPU. After the **divide**, the CPU then allocates another daughter and copies the genome to the daughter's space. This process continues for all eight CPUs, each of which (except for the last one) generates another daughter.

Unfortunately the aforementioned behavior was not desired. The creatures had serialized the parallel CPUs (since they are simulated in a serial fashion in Tierra) and used each CPU to generate its own daughter. They had taken a supposedly parallel process and "daisy-chained" its behavior together so that the beginning of one CPU's execution finishes up the execution of the previous CPU. On a real parallel computer, such behavior is inefficient. However, since Tierra emulates parallelism through time slicing, an algorithm which serializes the activity of its several CPUs can avoid the cost of calculating the offsets and coordinating their activity.

3.2 Fixing the bug(s)

After this bug was discovered, Tierra was modified so that cycles did not accumulate between time-slices (i.e., any cycles left in a time-slice upon execution of a **split** would be lost). This produced the desired behavior (daisy-chaining between parallel CPUs was prevented) and multi-cellular evolution (as described in section 3) was observed.

Unfortunately, this modification also produced a side-effect which was not considered at that time. By zeroing out the time-slice whenever a **split** was performed, Tierra implicitly imposed a large computational cost on additional parallelism. Although an increase in parallelism from two to four CPUs evolved, additional increases in parallelism were not observed. Consequently, Tierra was modified to remove the computational cost imposed on parallelism. Tierra now switches between a creature's CPUs after each instruction in a time-slice rather than after each CPU's time-slice completes. This corrects the original problem without imposing the unwanted cost on parallelism. After this final change was implemented, the evolution of additional parallelism (up to the specified limit of 16 CPUs) was quickly observed. Space does not permit us to discuss these new results here but they will be presented in detail in a forthcoming paper.

4 Conclusions and a Glimpse Into the Future

This first experiment with evolution of parallel processes has yielded fruitful results. Evolution has been able to spontaneously increase the level of parallelism, and effectively coordinate the activities of the additional processors without generating errors. However, differentiation between the processors has taken the form of manipulating different data, not executing different code. Essentially, this is a SIMD style of parallelism, rather than the more interesting MIMD parallelism that we hope to evolve in the future. Yet, given the nature of the problem at hand (copying a series of continuous bytes), a SIMD solution is the most appropriate. In order to evolve MIMD parallelism, where the different processors execute different code while coordinating their activities, evolution will have to be challenged with more complex problems.

From the biological perspective, the SIMD/MIMD distinction relates to the absence (SIMD) or presence (MIMD) of differentiation between cells. In differentiated organisms, different cell types express different suites of genes, which correspond to executing different parts of the same code. It is hoped that future digital organisms will evolve into complex forms exhibiting both SIMD and MIMD parallelism. In order to facilitate this evolution, protocols are being established to permit communication between cells and individuals within and between nodes of both real and virtual machines.

In order to challenge evolution with more complex problems, preparations are being made to create a large biodiversity reserve for digital organisms distributed across the global net [5]. Participating nodes will run a network version of Tierra as a low-priority background process, creating a virtual Tierran sub-net embedded within the real net. Digital organisms will be able to migrate freely within the virtual net. Given that the availability of energy (CPU time) at each node will reflect the activity patterns of the users, there will be selective pressures for organisms to migrate around the globe in a daily cycle, to keep on the dark side of the planet, and also to develop sensory capabilities for assessing deviations from the expected patterns of energy availability, and skills at navigating the net in response to the dynamically changing topology of the net and patterns of CPU-energy availability.

5 Acknowledgments

The authors would like to thank Danny Hillis, David Waltz, and the Santa Fe Institute for supporting this research.

The work of TSR was supported by grants CCR-9204339 and BIR-9300800 from the United States National Science Foundation, a grant from the Digital Equipment Corporation, and by the Santa Fe Institute, Thinking Machines Corp., IBM, and Hughes Aircraft. This work was conducted while at: Thinking Machines Corporation (KT), School of Life & Health Sciences, University of Delaware (TSR), the Santa Fe Institute (KT and TSR), and the ATR Human Information Processing Research Laboratories (KT and TSR).

6 Bibliography

[1] Darwin, Charles. 1859. *On the origin of species by means of natural selection or the preservation of favored races in the struggle for life*. Murray, London.

[2] Ray, T. S. 1991. An approach to the synthesis of life. In: Langton, C., C. Taylor, J. D. Farmer, & S. Rasmussen (eds), *Artificial Life II*, 371-408. Redwood City, CA: Addison-Wesley.

[3] ______. 1994. An Evolutionary Approach to Synthetic Biology: Zen and the Art of Creating Life. *Artificial Life* 1(1/2): 195-226.

[4] ______. In Press. Evolution, Complexity, Entropy, and Artificial Life. *Physica D*.

[5] ______. In Press. Evolution of parallel processes in organic and digital media. In: D. Waltz (ed.), *Natural and Artificial Parallel Computation*. Philadelphia: SIAM Press.