Evolution, Entropy, and Parallel Computation

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

Abstract

In this paper the relationship between evolution and entropy is described for a model of self-reproducing parallel computation. As was recently shown [5], the performance of some types of parallel computation can be increased though a process analogous to evolution by natural selection. The work discussed in this paper explores the process by which evolution manipulates the entropy of instruction sequences in a population of parallel programs in an effort to discover more efficient uses of parallelism.

1 Introduction

The study of evolution has recently begun to be explored in a number of frameworks far afield from natural biology. In a number of recent papers [1-5], (artificial) evolution was applied to the optimization of a population of self-reproducing programs. By providing selection pressure for efficient self-reproduction, evolution is able to start with an inefficient (but working) program and over time produce "offspring" that are better able to perform this task.

The Cambrian explosion of diversity approximately 600 million years ago resulted in the evolution of single-celled organisms into much more complex multi-cellular life-forms. In this paper we hope to apply the metaphor of evolving multi-cellular living creatures to models of computation. The single-celled organisms correspond to scalar programs while multi-cellular organisms correspond to parallel programs. As it is commonly known in the parallel programming community, writing parallel programs is significantly more difficult that writing scalar programs. By using evolution to optimize the parallel computation, it is hoped that new and innovative techniques for parallel programming will be discovered.

During these experiments evolving parallel computation, it was discovered that detectable changes in the entropy of instruction sequences would correspond to changes in self-reproduction efficiency. This paper concludes with a discussion of changes in entropy during evolution and how these changes could be used to search for more efficient uses of parallelism.

2 Tierra

The experiments described in this paper were performed using the Tierra virtual computing environment. The Tierra system has already been described in detail elsewhere [1-4], so it will be described only briefly here. Tierra is a virtual computer that simulates the computational environment for a population of self-reproducing computer programs (aka "creatures"). These program are written in a machine language that has been designed to be robust to errors. Each Tierra program is assigned a separate processor responsible for the execution of the program. This processor contains items such as the instruction pointer, registers, and a stack.

The first Tierra creature was an unoptimized self-reproducing program created by Thomas Ray [1]. This program, also known as the "ancestor," self-reproduces by examining itself, allocating space for a daughter, and then copying its instructions (aka its "genome") to the daughter's space. Once the daughter has a complete copy of the mother's genome, the daughter is allocated its own processor and both mother and daughter then operate independently. This process continues to exponentially increase the number of creatures until the available memory fills up. At that point a process known as the "reaper" starts to kill off existing creatures to free up memory for new daughters. The decision as to which creature will be killed by the reaper is based on age and any errors that the creature might have generated in the past (errors include such things as allocating space for two daughters at the same time or jumping to an illegal location).

In addition to managing the population of programs, Tierra also introduces several different kinds of errors into the program execution process. These errors play the role of genetic operators such as mutation and produce the conditions that allow for evolution by natural selection to occur. In the results presented by Ray [1,2], evolution optimized the creatures by reducing their size (the number of instructions in the genome) thus increasing their ability to self-reproduce (smaller creatures can reproduce faster than larger creatures since there are fewer instructions to copy).

Figure 1 illustrates the evolutionary optimization of creature size (number of instructions) versus time for four different simulations. Each run was started with identical

To appear in the proceedings of PhysComp94, the 1994 Physics and Computation Workshop, November 1994.

parameter settings, with the exception of the random number seed. In each of the four runs it is quite obvious that the creatures evolve to be smaller as time progresses.

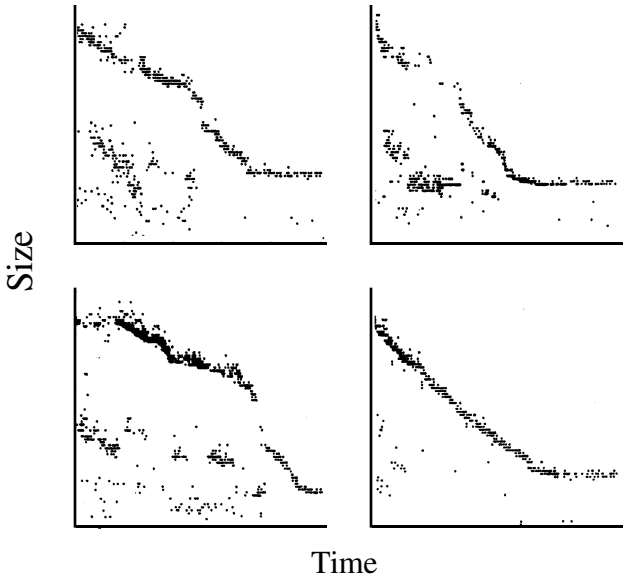

FIGURE 1. Evolution of size versus time in single-celled Tierra simulations.

The initial single-celled ancestor is 80 instructions long but a typical Tierra simulation usually evolves creatures containing between 22 and 30 instructions (the smallest self-reproducing creature ever discovered was composed of 22 instructions). In addition to the upper (monotonically decreasing) band of creatures, often there are significant numbers of creatures which have sizes approximately half the size of the current creatures in the upper band. These creatures, known as parasites, are unable to reproduce without the aid of other fully self-reproducing creatures. To survive, they use code found in another creature's program to cary out reproduction. Figure 2 illustrates the relationship between a parasite and a "host" ancestor in a single-celled Tierra simulation. This type of social interaction is found in nearly all Tierra simulation runs (other forms of complex social behavior have also been observed [1]).

2.1 Adding parallelism to Tierra

Initially the work on evolving self-reproducing computer programs in Tierra involved only non-parallel varieties. Each program was assigned one processor. More recently, the computational model in Tierra has been extended to include parallel computation [5]. In parallel Tierra programs, a program can be assigned multiple processors (aka CPUs). Each CPU has a separate set of registers and instruction pointer. The model is a shared-memory MIMD model of parallelism. Using a biological metaphor, the original Tierra programs correspond to single cell creatures while the new parallel CPU programs correspond to multi-cellular creatures.

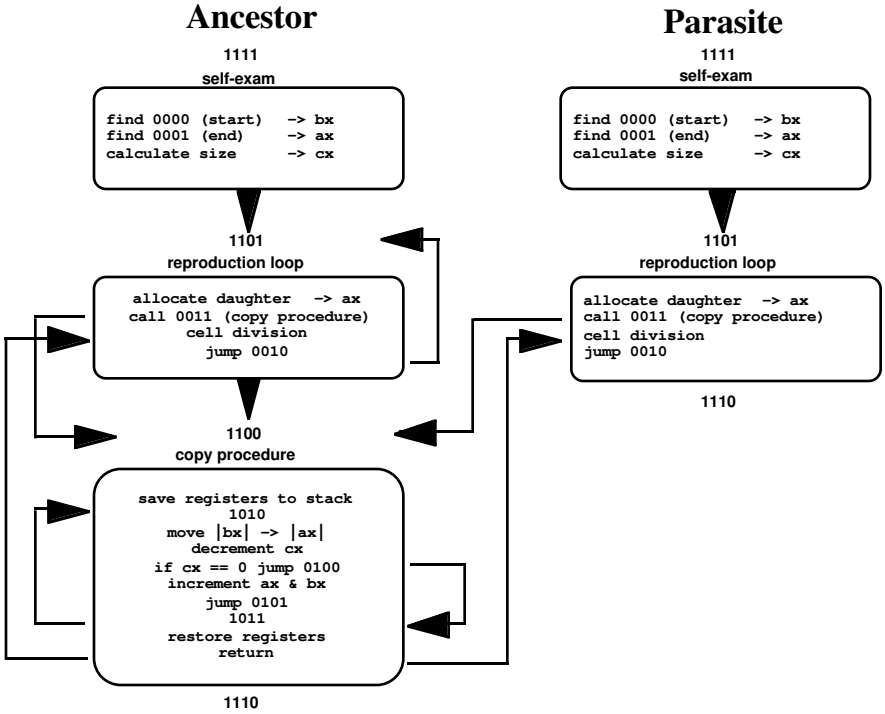

FIGURE 2. Ancestor and parasite execution relationships in a single-celled Tierra creature.

The development of parallelism in a Tierra program is modeled on binary cell division. When a CPU issues a new instruction called "split," an additional CPU is added to the processor structure for that program. Figure 3 illustrates splitting behavior for a Tierra creature that starts initially with a single CPU. The round objects in the picture represent CPUs while the square objects represent the programs that each CPU is executing. After the first CPU issues a split instruction, there are then two CPUs associated with that creature. Both of those CPUs then issue an additional split creating a total of four CPUs for that creature.

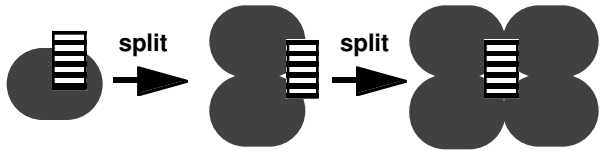

FIGURE 3. Splitting

Notice that while the number of CPUs in an individual creature can change (via the execution of split instructions), there is still only a single program associated with a creature. In the parallel processing model that we used, the CPUs have a shared program space and thus each CPU is executing the same program. Each CPU has its own instruction pointer (i.e., MIMD parallelism) and different CPUs can branch conditionally. So, although each CPU in a creature is executing the same program, different CPUs can be simultaneously executing different instructions.

Initially each program starts out with a single CPU and it is up to the program to allocate additional CPUs. After a split instruction is executed, a new CPU is created which has register values copied from the CPU that issued the split. The only register that is not copied unchanged is the D register. The D register in the original CPU is simply shifted to the left one bit. The new CPU has its D register shifted to the left one bit also but a one is then added to the LSB to differentiate between the two CPUs following the execution of the split. After the split both CPUs then begin executing the instruction following the split (see figure 4). By issuing a sequence of split instructions, an exponential number of processors can be allocated (up to a specified limit, which is sixteen in the simulations described in this paper). Depending on the effectiveness of the self-reproduction algorithm, the addition of multiple CPUs can often significantly improve the performance (or, in biologi-

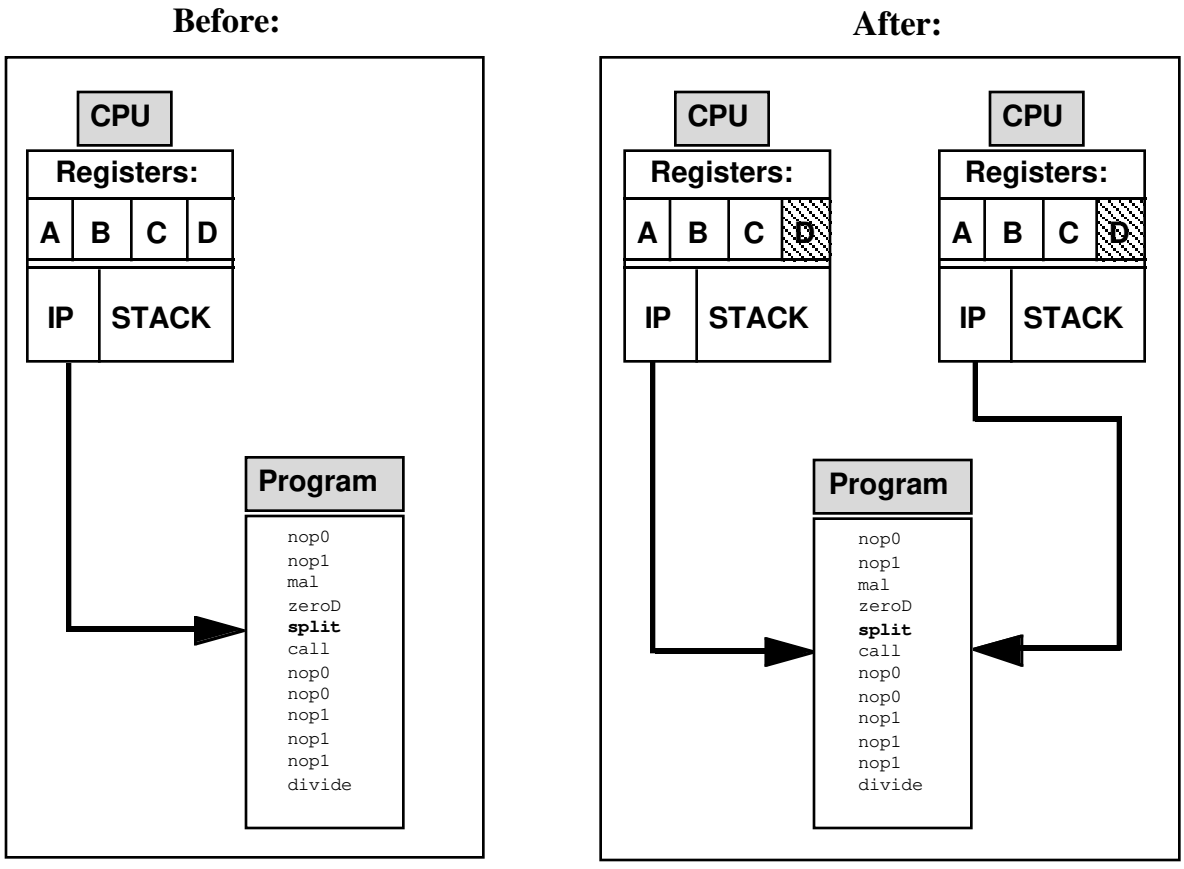

FIGURE 4. Multi-cellular processor structure before and after a split instruction is executed.

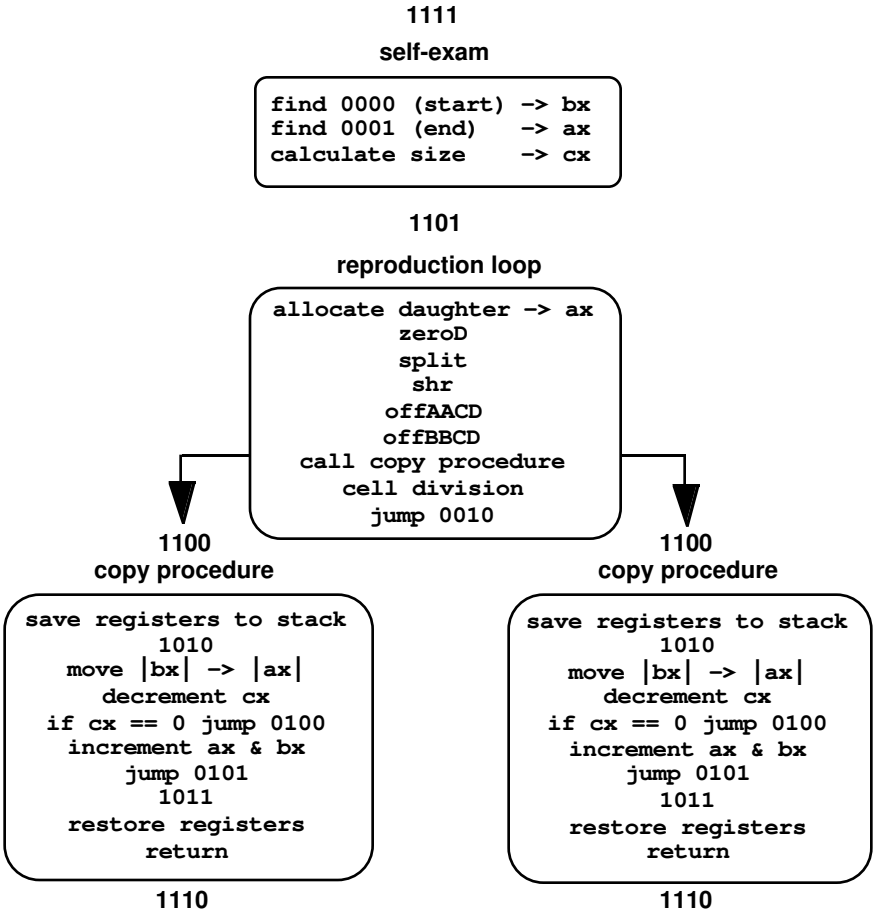

FIGURE 5. The multi-cellular ancestor.

cal terms, the fitness) of a program. Ideally, a program with two CPUs should be able to reproduce in approximately half the time as a similar program with only one CPU.

3 Evolution and Parallel Computation

As with the original Tierra research, a new (parallel) ancestor was created. This parallel ancestor was designed by the author to use two CPUs in parallel (in a completely *unoptimized* manner). Figure 5 illustrates the behavior of the parallel ancestor. This creature is very similar to the scalar ancestor in nearly all of its instructions. It differs from the scalar ancestor by containing a sequence of additional instructions just before the call to the copy procedure. Before copying begins, the creature splits into two CPUs and then divides the workload into two separate parts. When the each CPU calls the copy procedure, one CPU will end up copying the first half of the genome while the second CPU will copy the other half.

After this initial allocation of two CPUs, any further increases in the number of CPUs would have to occur as evolutionary optimizations. Comparing the multi-cellular ancestor with its single-celled cousin, the multi-cellular creature is clearly the more efficient reproducer. While the single cell 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 almost twice as many offspring as a single cell ancestor in the same period of time and will dominate the population.

Experiments were run using the multi-cellular ancestor on a new version of Tierra that runs on a Connection Machine CM-5 parallel supercomputer[1]. The initial results of experiments evolving parallel programs are described in [5].

The population was seeded with the two CPU parallel ancestor, which was composed of 82 instructions. During

1. Although Tierra now runs on a parallel supercomputer, the parallelism in the programs is unrelated to the parallelism in the computer that they are running on. By taking advantage the size and speed of a supercomputer, it was possible to run much larger and faster evolutionary simulations. Slower and smaller simulations of multi-cellular creatures have been run on workstations.

a Tierra simulation, whenever the total number of creatures of a particular genotype passed a user specified threshold, the complete genome (program instructions, reproduction time, and miscellaneous flags) was saved to an archive file. To observe the evolutionary process, the reproduction time[2] for each archived genome can be plotted versus the time that each genome was saved to the archive. Figure 6 (top) shows the the evolution of reproduction time versus time for a typical multi-cellular Tierra run.

Each point in this graph corresponds to the creation of a viable species of programs with the specified reproduction time. During the first 200 million clock cycles, two bands of evolution can be seen. At first the evolutionary process makes use of serial optimization techniques such as template size reduction and instruction side effects (upper band). There was also effective parasitism (programs that could not copy themselves but instead executed the code of other programs to carry out reproduction) during this first evolutionary phase (lower band). At approximately 320 million clock cycles there is a sharp improvement in reproduction time that corresponds to a thirty percent reduction in the time required to reproduce. This new optimization is added parallelism, and it corresponds to an increase from two to four CPUs per program. In the graph of genome length versus time (Figure 6, bottom) this change is even more noticeable since the dominating organisms have actually increased in size from 36 to 44. But the size 36 creatures have only two CPUs while the size 44 creatures have twice as many. In terms of fitness, the larger but more parallel creatures are better reproducers and as a result take over the population.

Once additional parallelism evolves it becomes the dominant form of improvement in reproduction efficiency. At approximately 500 million clock cycles creatures with eight CPUs can be seen. Approximately 100 million clock cycles later the final push to sixteen CPUs is achieved. When examining the genome size versus time graph it is easy to see that when additional parallelism is evolved, the size of the new creatures tends to overshoot the size that later evolves (with the same number of CPUs). Also notice that the size of the first creatures with an increased number of CPUs tends to be a multiple of the number of CPUs. Obviously the creatures that initially evolve the use of more CPUs are dividing the workload evenly and are not able to handle circumstances which do not provide even workloads. This produces an effect which we have referred to as the production of ``intron instructions.'' An intron instruction is a unnecessary instruction (to the proper execution of a program's self-reproduction algorithm) included in a program. They are often used to pad out a program so that its length is a multiple of the number of CPUs. For example, one common intron instruction involves incrementing an unused register. Since the execution of an intron instruction has no effect, it does not disrupt the process of self-reproduction. After each jump in additional parallelism succeeds, evolution attempts to remove the unnecessary intron instructions. In most cases further increases in parallelism take effect before many introns can be removed. In the final phase of evolutionary activity, the population is initially dominated by sixteen CPU creatures of size 64. Since there is no further increase in parallelism possible, evolution proceeds to remove introns and eventually produces creatures of size 60.

4 Evolution and Entropy

To examine the relationship between evolution and entropy, the entropy for each creature's genome was calculated. This was done by generating an instruction histogram for each creature's genome and computing the entropy for the distribution contained in the histogram. For example, consider the following tierra program:

```
genotype: 0060bgB parent genotype: 0060awa
1st_daughter: flags: 0 inst: 64 mov_daught: 72
2nd_daughter: flags: 0 inst: 65 mov_daught: 72

        nop0        ; 000 00 0
        adrb        ; 000 3c 1
        nop1        ; 000 01 2
        subAAC      ; 000 07 3
        movBA       ; 000 19 4
        adrf        ; 000 1d 5
        nop0        ; 000 00 6
        nop0        ; 000 00 7
        subCAB      ; 000 06 8
        ifz         ; 000 25 9
        ifz         ; 000 25 10
        mal         ; 000 3e 11
        split       ; 000 20 12
        ifz         ; 000 25 13
        adrb        ; 000 3c 14
        shr         ; 000 23 15
        offAACD     ; 000 26 16
        offBBCD     ; 000 27 17
        zeroD       ; 000 24 18
        ifz         ; 000 25 19
        adro        ; 000 1b 20
        ifz         ; 000 25 21
        split       ; 000 20 22
        split       ; 000 20 23
        shr         ; 000 23 24
        offAACD     ; 000 26 25
        pushB       ; 000 2d 26
        offBBCD     ; 000 27 27
        zeroD       ; 000 24 28
        split       ; 000 20 29
        movii       ; 000 3a 30
        shr         ; 000 23 31
        offAACD     ; 000 26 32
        offBBCD     ; 000 27 33
        incC        ; 000 2b 34
        zeroD       ; 000 24 35
```

2. Unlike the results presented by Ray [1] which focused on creature size, reproduction time is now used to describe the fitness of a program. 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 inferible from the size. A long creature with many parallel CPUs might very well be fitter than a short creature with few parallel CPUs.

```
split       ; 000 20 36
not0        ; 000 02 37
ifz         ; 000 05 38
ifz         ; 000 05 39
shr         ; 000 23 40
offAACD     ; 000 26 41
offBBCD     ; 000 27 42
nop1        ; 000 01 43
nop0        ; 000 00 44
movii       ; 000 1a 45
decC        ; 000 0a 46
ifz         ; 000 25 47
jmpo        ; 000 34 48
nop0        ; 000 00 49
incA        ; 000 28 50
incB        ; 000 29 51
jmpb        ; 000 15 52
nop0        ; 000 00 53
nop1        ; 000 01 54
join        ; 000 21 55
divide      ; 000 3f 56
ret         ; 000 37 57
nop1        ; 000 01 58
nop1        ; 000 01 59
```

This program (creature 0060bgB) was the first 16 CPU creature to evolve. To compute the entropy associated with this program, each instruction in the instruction set is enumerated and the total number of instances of each instruction in the program must be counted. For the creature 0060bgB, Table 1 illustrates the instruction count histogram.

The entropy for a creature can then be calculated using the distribution contained in the histogram. Shannon's well known equation for the entropy associated with a distribution ($-\sum p \cdot \log p$) is used to convert the instruction histogram to an entropy value[3]. For the distribution in Table 1 (creature 0060bgB), the entropy value is computed to be 4.387 bits.

For each point in time the total genotypic entropy for creatures created at that time was summed. This can be seen in Figure 7 (top), which graphs the total genotypic entropy versus time. At each of the three increases in evolved parallelism (two to four CPUs, four to eight CPUs, and eight to sixteen CPUs), noticeable increases in total genotypic entropy can be seen[4].

When the genotypic entropy is averaged over the number of creatures archived for each point in time, the relationship between genotypic entropy and evolutionary change can also be seen. In the average genotypic entropy versus time graph, sudden increases in entropy can be seen to precede the first two increases in parallelism. It can be argued that these increases in average genotypic entropy correspond to the exploration of the space of programs around the current population of programs. Increasing the average genotypic entropy indicates that new instructions are being tried out in the population of programs. Eventually this exploration produces an improvement in parallelism and the process slows down (decreasing the average genotypic entropy, at least for the immediate time being).

nop0	●●●●●
nop1	●●●●
not0	●
shl	
zero	
ifz	●●●●●●●
subCAB	●
subAAC	●
incA	●
incB	●
decC	●
incC	●
pushA	
pushB	●
pushC	
pushD	
popA	
popB	
popC	
popD	
jmpo	●
jmpb	●
call	
ret	●
movDC	
movBA	●
movii	●●
adro	●
adrb	●●
adrf	●
mal	●
divide	●
split	●●●●
join	●
offAACD	●●●●
offBBCD	●●●●
zeroD	●●●
shr	●●●●

TABLE 1. Instruction histogram for 0060bgB program

3. The probability p associated with each instruction in the instruction set is calculated by dividing the total number of instances of that instruction in a particular program by the program's length.

4. To simplify the comparison of the graphs in Figures 6 and 7, three pairs of lines have been drawn to indicate the time associated with the beginning and ending of each increase in parallelism. The three pairs of lines correspond to the transitions from two to four, four to eight, and eight to sixteen CPUs.

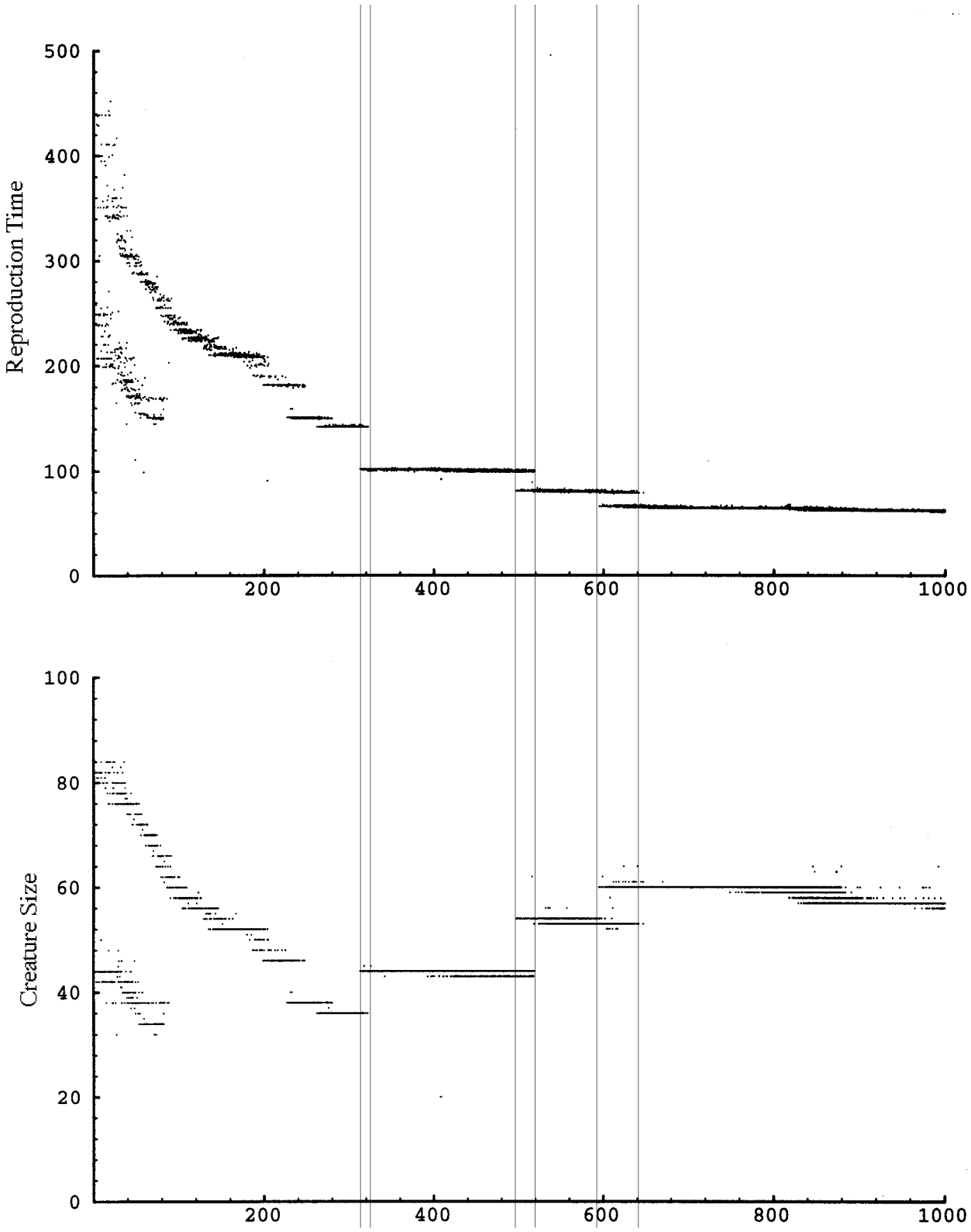

FIGURE 6. Evolution of reproduction time (top) and creature size (bottom) versus time.

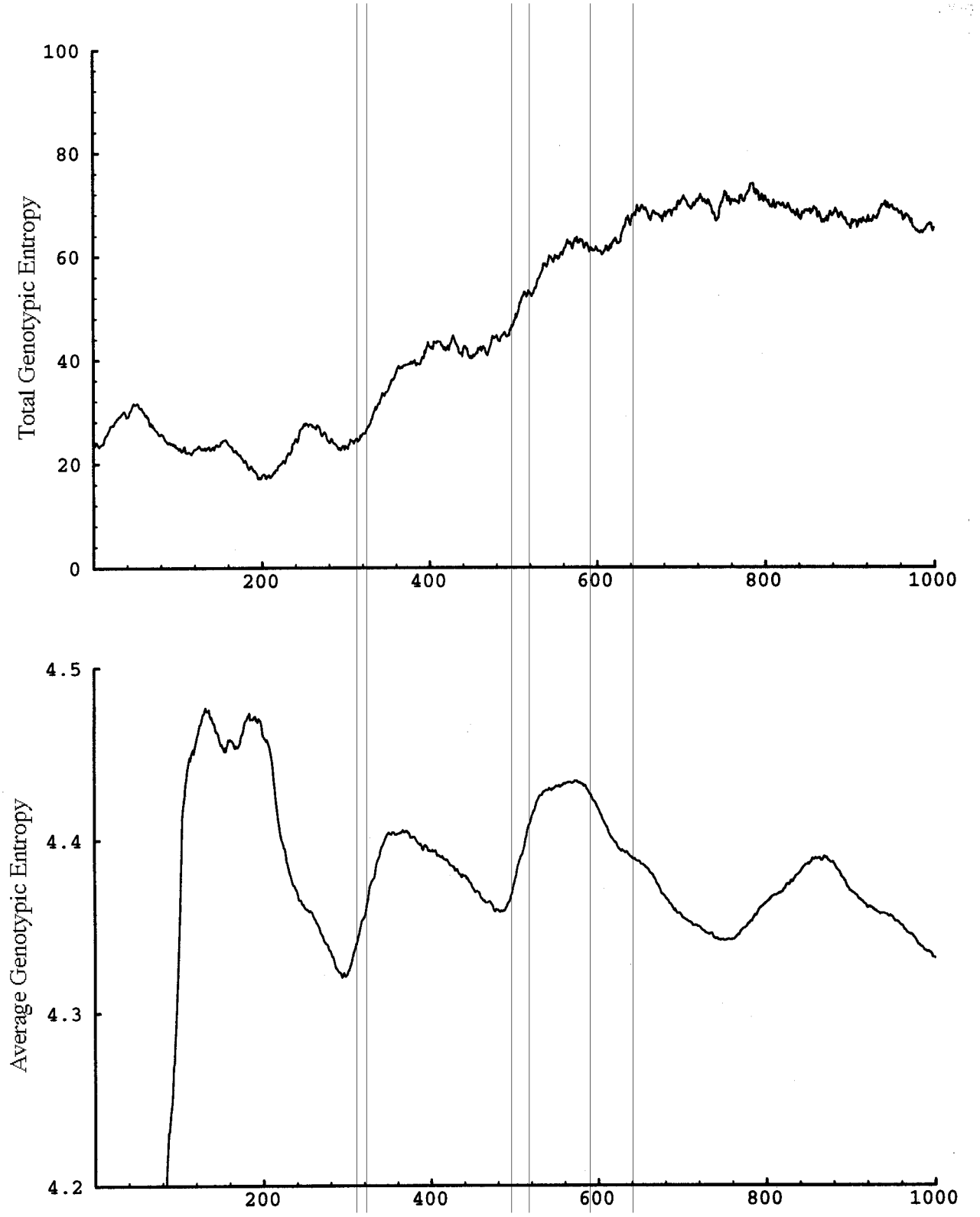

FIGURE 7. Evolution of total (top) and average (bottom) genotypic entropy versus time.

Two additional increases in average genotypic entropy, which correspond to phases of genotypic optimization, can also be seen in the graph. The first increase corresponds to the optimization of the two CPU population that started with the parallel ancestor. Since the two CPU ancestor was not very efficient, the optimization process is fairly long and involves significant increases in fitness. This is the first noticeable increase in average entropy, which can be seen to last longer than any of the other increases. The final increase in average entropy corresponds to the removal of intron instructions from the sixteen CPU creatures. Since this is the only time evolutionary optimization was able to remove significant numbers of intron instructions (all previous attempts to remove intron instructions were all interrupted by sudden increases in parallelism), it is not unreasonable to expect that it would produce a noticeable increase in average genotypic entropy while the state space of programs was explored. Finally, it should also be noted that the final increase in parallelism from eight to sixteen CPUs does not produce the expected increase in average genotypic entropy (although the total genotypic entropy graph does indicate evolutionary activity). There is a slight slowdown in the decrease in average entropy from the last increase in parallelism, but this effect is relatively minor. One possible explanation is that the building blocks for distributing self-reproduction algorithm among multiple CPUs have already been created during previous increases in parallelism. By the time that the final increase in parallelism in evolved, a smaller exploration of the program space around the current population of programs is required.

5 Conclusions

In this paper the evolution of parallel computation was described. Evolution was shown to optimize self-reproducing parallel programs to achieve significant improvements in performance. In addition, a relationship between changes in entropy in the program instructions and evolutionary change was shown to exist. This relationship corresponds to an exploration of the program space by the evolutionary process in order to increase the fitness of the parallel programs.

6 Acknowledgments

The author would like to thank Tom Ray, Katsu Shimohara, Danny Hillis, David Waltz, the Santa Fe Institute, and the HIP group at ATR (Kyoto) for supporting this research.

7 Bibliography

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

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

[3] _____. In Press. Evolution, Complexity, Entropy, and Artificial Life. *Physica D*.

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

[5] Thearling, K., and T. S. Ray. 1994. Evolving Multi-Cellular Artificial Life. In: R. Brooks and P. Maes (eds.), *Proceedings of Artificial Life IV*, Cambridge: MIT Press.