Scaling Conway's Game of Life on Parallella

Richard Gagnon
School of Electrical Engineering and Computer Science
University of Ottawa,
Ottawa, Canada
7736989
rgagn094@uottawa.ca

Eric Tessier
School of Electrical Engineering and Computer Science
University of Ottawa
Ottawa, Canada
8172292
etess097@uottawa.ca

Abstract - In this paper we explore two parallel implementations of Conway's Game of Life and examine whether or not these parallel implementations are worth the time and effort on the Parallella platform. We will test performance on versions of the game running solely on the host processor as well as two implementations taking advantage of the Epiphany 16-core mesh available on the Parallella board. We will discuss the advantages and disadvantages of each implementation and present a recommendation based on the findings.

I. INTRODUCTION

A. Parallelizing Sequential Code

One of the many advantages to writing sequential code is the fact that it is quite simple relative to parallel code. In addition, programmers do not always require a deep underlying knowledge of the hardware they are working on. It is also worth mentioning that the debugging process is often simpler as debuggers can move through the code in logical manner. As a result when parallelizing code or writing parallel code from the start, much more attention is required from developers and one must ask themselves if the additional time and effort required will yield adequate boosts in performance.

B. Differences in Architecture

While sequential code will execute on the host processor, parallel code will be executed mainly on the Epiphany eCore mesh with initialization and I/O operations performed by the host. The difference in platforms where the majority of execution is taking place will certainly have an effect on the performance of both implementations. Introducing parallel code will also introduce additional overhead that the sequential code will not require. Communication and initialization between the Epiphany core mesh and the host will require additional time as well as memory access time will differ between the two. These are some of the issues that must be examined to determine the value of parallelizing Conway's Game of Life on the parallella platform.

II. BACKGROUND KNOWLEDGE

A. Conway's Game of Life

Invented by British mathematician John Horton Conway in 1970, Game of Life is a zero player game, thus the state of the game evolves without any input from players [1]. The game is composed of an N x M board where N and M can take any whole positive value. This board is populated N * M cells each possessing two states: Alive or Dead. The game is meant to be an analogy to the rise, fall and alternations of living organisms [1]. The game is provided an initial state and will evolve as cells come to life or die based off the states of their neighbouring cells and 4 simple rules. Games will often evolve into self sustaining shapes or interesting patterns that players can observe. A small parallel version of the game of life is provided as an example on the board and will be used as a foundation for this paper.

B. *The Parallella Board*

The implementations and tests outlined in this paper are performed on an 18 core version of the Parallella board. This board is in some ways similar to other small single board computers such as the Raspberry Pi. The Parallella board is a credit card size board measuring in at 55mm by 90 mm. It boasts a Xilinx Zynq Dual Core ARM A9 host processor and an Epiphany 16-core coprocessor. Some interesting features of this board are the addition of an FPGA within the Zynq system on a chip (SoC) and an impressive power efficiency of 5 Watts [2]. As efficient as the board is however, it suffers from substantial overheating issues as the passive heatsink is simply incapable of cooling the processors even at idle causing recurring system crashes.

C. *The Epiphany Coprocessor*

Loaded on the parallella board is an Epiphany many-core processor manufactured by semiconductor company Adapteva. The Epiphany processor is composed of 16 RISC cores organized in a 2-dimensional mesh on-chip network, each clocked at 1GHz. There is no inter-core bus, cache or speculation hardware on the chip. Instead each core is mated to a router on the network and possesses its own local memory banks consisting of 32KB [3]. All core memory banks are addressable by all the cores on the mesh allowing for inter-core communication. In addition to a core's local memory, a portion of the board's SDRAM is shared between the host and Epiphany processors. Adapteva provides an SDK to allow developers to implement applications with this architecture.

III. IMPLEMENTATION 1 - ARM ONLY

A. *Motivation*

Although the topic of this paper is to study the value of parallelizing Conway's Game of Life, a sequential version of the code was needed to compare performance against parallel implementations and measure any potential speed ups. It's important that this implementation is as similar as possible to the parallel implementations and performs the same operations.

B. *Design*

This implementation of the game will execute solely on the ARM host processor. It is to be used a baseline for measuring the speed ups of the parallel implementations. The dimensions of the board are configurable in both dimensions by the user so that this particular implementation can be tested on various board sizes. The cells of the board are stored in an N * M character array where N and M are the dimensions of the board. Each iteration, every single cell is checked and its state is updated depending on the state of it's 8 neighbours. The ARM SoC possesses independent 4-way set-associative L1 caches for each core. This means that this implementation is expected to be very fast as the character array containing the cells will likely be cached, resulting in very fast read and write operations.

IV. IMPLEMENTATION 2 - PARALLEL USING SDRAM

A. *Motivation*

The first parallel implementation of Conway's Game of Life was, like the ARM only version, meant to have configurable board sizes in both dimensions while of course utilizing the ecores to parallelize and hopefully improve upon the performance of the ARM only program. It also had to give the option to choose the number of cores being utilized in any possible configuration of board size in order to permit testing of the same program while using different amounts of cores.

B. *Design*

In this implementation, the host processor is given values for all the configurations of the program through input parameters and passes it to the ecores through shared memory. Once this is done, each core is now able to determine which cells it's responsible for based on the number of cores in use and the number of cells in play. As demonstrated in fig. 1, this is implemented by giving the first core (C0) in the workgroup the first cell in the game. Then, going from left to right, top to bottom, the next core is assigned the next cell until there are no more cores where it all restarts from the first core. The state of each cell is at all times kept in shared memory accessible by all of the ecores and the host. This allows easy access to the state of each cell whenever needed, however mandates that the SDRAM is accessed many times by every core at every iteration of the game. These numerous accesses to the shared memory will have a drastic effect on the efficiency of the program.

	1	2	3	4	5
1	C0	C1	C2	C3	C0
2	C1	C2	C3	C0	C1
3	C2	C3	C0	C1	C2
4	C3	C0	C1	C2	C3
5	C0	C1	C2	C3	C0

Figure 1: Implementation 2 cell distribution for each core on an 5x5 board.

V. *IMPLEMENTATION 3 - PARALLEL USING LOCAL MEMORY*

A. *Motivation*

This second implementation aimed to improve on the first, mainly by reducing the number of SDRAM accesses required. To do this, the local memory of each eCore was utilized. By utilizing local core memory to check the states of neighboring cells, we can greatly reduce the number of lengthy SDRAM accesses and improve execution time leading to greater speed ups.

B. *Design*

Like implementation 2, a copy of the board of cells resides in the shared SDRAM and is used by the host processor to print the state of the board to the console for users to see. What differentiates this implementation to the first is that the state of the cells in each row are also stored in the local memory of a core. An example of this using an 8x8 board is portrayed in fig. 2. It can be seen that the cells of row 0 are store in the local memory of core 0, cells of row 1 are store in the local memory of core 1, and so on. Each core will check the neighbours of each of its cells by accessing either its own local memory or neighbouring cores' local memory. Unfortunately, this means that the number of rows is locked to the number of cores being used. The number of columns is however still configurable.

	1	2	3	4	5	6	7	8
1	CORE 0							
2	CORE 1							
3	CORE 2							
4	CORE 3							
5	CORE 4							
6	CORE 5							
7	CORE 6							
8	CORE 7							

Figure 2: Implementation 3 cell distribution for each core on an 8x8 board.

C. *Expected Performance*

Performance of this implementation is expected to be superior to that of both previous implementations. Reads to local memory of a core are expected to be similar to that of a cache read on ARM processor. Reads to local memory of neighbouring cores won't be as fast however. The inter-connecting mesh has a maximum throughput of 1 read transaction every 8 clock cycles [3]. In addition to this, during execution, each core is expected to read from neighboring core memory between 0 to 6 times per cell and as a result the mesh is expected to be very crowded. A likely faster implementation that could perhaps be used in a future project would be to favor writes over reads. Write transactions on the eMesh are approximately 16x more efficient than read transaction with a maximum throughput of 0.5 terabit/sec [3]. Instead of reading from neighbouring cores, neighbouring cores could instead write the states of their cells to their respective neighbours removing the need for read transactions on the mesh.

VI. RESULTS

A. *Considerations*

Using the ARM only program as the non-parallelized version of the game of life, comparing its performance with that of the parallel implementations will demonstrate which is more efficient. It must be noted that every result shown below is an averaged result of five tests implementing 1000 iterations of the game, and that every test was timed both with the initialization of the epiphany platform included and without it.

B. *First parallel implementation*

Starting with an 8x8 cell board as shown in table 1, every test completed by the program that uses the SDRAM was outperformed by the non-parallelized program. Using 4 cores, the program took 5 times longer while with 16 cores, that was cut down to taking twice as long as the ARM only version. Even after initializing the epiphany platform, the rest of the program was never able to surpass the efficiency of the single core program making it only to 80% its efficiency with all 16 cores. These results remain fixed as the size of the board increases as shown in tables 2 and 3 (board sizes being 16x16 and 16x32 respectively), however it is interesting to note that using 4 cores always yields a performance at about 20% efficiency, no matter the board size nor where the timer starts. Using 8 or 16 cores on the other hand seems to yield better results as the board grows in size and when the timer begins after the initialization of the epiphany platform.

C. *Second parallel implementation*

In the program using the local memory instead of the SDRAM, fewer tests are possible since the set up is not as configurable, however the performance is much greater. In the 8x8 board using 8 cores, the performance did not quite overtake the ARM only program at 65% efficiency, though it did better than the other parallel version. Removing the initialization of the epiphany platform from the timing though demonstrates that it is able to compete with the timing of the ARM only program being 98% as efficient. At a 16x16 and 16x32 board using 16 cores, the newer program is able to outperform even with the initialization of epiphany included in the timing. As the board doubles in size from 16x16 to 16x32, the efficiency increases with it from 33% to 49% more efficient.

ARM only: 19.20				
# Cores	Parallel - SDRAM		Parallel - Local	
	w/ init	w/o init	w/ init	w/o init
4	96.31 (0.20)	90.31 (0.21)		
8	54.25 (0.35)	45.54 (0.42)	29.56 (0.65)	19.68 (0.98)
16	39.46 (0.49)	23.62 (0.81)		

Table 1: Execution times (in ms) of implementations on 8x8 board, speed up in parentheses

ARM only: 77.54				
# Cores	Parallel - SDRAM		Parallel - Local	
	w/ init	w/o init	w/ init	w/o init
4	391.12(0.20)	385.19(0.20)		
8	202.01(0.38)	193.30(0.40)		
16	116.55(0.66)	100.89(0.77)	58.43 (1.33)	41.92 (1.85)

Table 2: Execution times (in ms) of implementations on 16x16 board, speed up in parentheses

ARM only: 154.62				
# Cores	Parallel - SDRAM		Parallel - Local	
	w/ init	w/o init	w/ init	w/o init
4	777.43(0.20)	770.99(0.20)		
8	397.17(0.40)	387.50(0.40)		
16	218.92(0.71)	202.31(0.76)	103.19(1.49)	86.28 (1.79)

Table 3: Execution times (in ms) of implementations on 16x32 board, speed up in parentheses

VII. CONCLUSION

Writing code in parallel is a much more difficult task than writing sequential code. As seen in the results, making that code more efficient than the sequential version is yet another difficult task. The first of the 2 parallel programs sacrificed efficiency for configurability while the second program did the opposite. While it is possible to create one both efficient and configurable, the programming becomes much more complex than the simple sequential program. That being said, scaling up such a program to a massive size would render it exponentially more efficient than a sequential program. For certain situations, parallelizing code can be worth the extra work for a more efficient program, however for general use, it seems to cause more problems than it solves.

VIII. REFERENCES

[1] M. Gardner, "Mathematical Games - The fantastic combinations of John Conway's new solitaire game "life"", *Web.archive.org*, 1970. [Online]. Available: https://web.archive.org/web/20090603015231/http://ddi.cs.uni-potsdam.de/HyFISCH/Produzieren/lis_projekt/proj_gamelife/ConwayScientificAmerican.htm. [Accessed: 07- Dec- 2018].

[2] "Parallella-1.x Reference Manual", Parallella.org, 2009. [Online]. Available: http://www.parallella.org/docs/parallella_manual.pdf. [Accessed: 07- Dec- 2018].

[3] "Epiphany Architecture Reference", Adapteva.com, 2011. [Online]. Available: http://adapteva.com/docs/epiphany_arch_ref.pdf. [Accessed: 07- Dec- 2018].