Game of Life

George Herbert
cj19328@bristol.ac.uk

James O'Sullivan
gx19928@bristol.ac.uk

December 10, 2020

1 Parallel Implementation

This implementation is produced for a single machine. It uses multiple worker goroutines to compute each iteration of the Game of Life board.

1.1 Goroutines

Each worker runs on a goroutine called 'worker'. The worker goroutines are passed two channels: one to receive each section of the board they are required to operate on, and one used to send events on. Additionally, two integers are passed: an initial y coordinate to start calculating the next state from and the number of turns. The number of worker goroutines is defined in the command-line flags as the number of threads. Every round, each worker is sent a section of the board to calculate the next state for their section. It calls the 'calcNextState' function to calculate the next state of the board for that section; at the end of the round, these sections are sent back to the distributor, which consecutively appends them to produce the state of the next game board.

Our implementation also utilises a ticker goroutine which sends an 'AliveCellsCount' event every two seconds; a mutex lock is used to prevent the 'completedTurns' and 'world' variables being written to while the event is being sent. The number of alive cells is calculated using the 'calcNumAliveCells' function, which iterates over the entire board, counting any cell with a value of 255.

Another goroutine called 'handleKeyPresses' is used to keep track of keys pressed throughout the game's progression. To do this the goroutine loops, waiting for a value from the 'keyPress' channel to process. The following keypresses have been implemented: pressing 's' produces a PGM file of the current board state; pressing 'q' also produces a PGM file but additionally terminates the program; 'p' toggles the pausing and resuming of the game board at the end of the turn that is currently being computed.

As an extension to the requirements, our implementation enables the keyboard controls to work when the game is paused.

1.2 Benchmarking and Scalability

Benchmark results were collected using the 512×512 image for 100 turns of the board. We benchmarked our implementation from one worker goroutine to 128. The benchmarks were run using a Linux lab machine, which has an Intel Core i7 processor, with six cores, each with two threads. The benchmark program was run five times, and a mean was calculated for each worker; five was deemed a suitable number of benchmarks to take an average from, as variance was minimal for benchmark results with more than one worker.

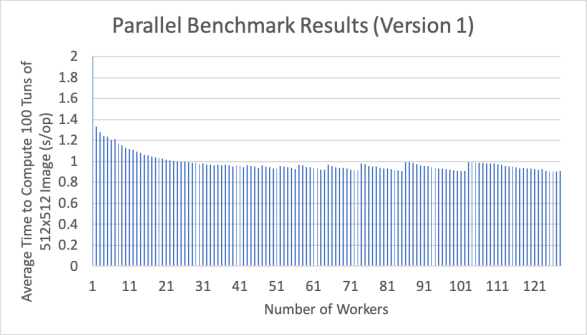

Figure 1: Benchmark results for our initial parallel implementation

We displayed the benchmark results for the first iteration of our program in a bar chart (Figure 1). At higher numbers of workers, there are sudden but significant jumps in the time taken to perform the 100 turns of the board, after which the average time begins to decrease and level out again. These jumps start occurring at around 35 workers but begin to become particularly noticeable at around 60 workers. Analysing our code revealed this was due to the method used to allocate rows of the board to each worker. The row-allocation formula we were using was as follows:

$$n_{abl} = \left\lfloor \frac{r}{w} \right\rfloor$$

$$n_l = r - (n_{abl} \times (w - 1))$$

where n_{abl} is the number of rows allocated to all but the last worker, n_l is the number of rows allocated to the last worker, r is the total number of rows, and w is the number of workers

For lower numbers of workers, the number of rows allocated to each worker was relatively equal, due to these numbers dividing nicely into 512. However, for higher numbers, the row-allocation system began to divide the rows up inefficiently. A good example of this difference is between 102 and 103 workers. For 102 workers, the rows were allocated so that 101 workers had five rows, and one worker had seven.

$$n_{abl} = \left\lfloor \frac{512}{102} \right\rfloor = 5$$

$$n_l = 512 - 5 \times (102 - 1)) = 7$$

This distribution is roughly equal, and so the program runs relatively fast. However, for 103 workers, 102 workers had four rows, and one worker had 104 rows—this is a very uneven split, which in turn caused the jump in benchmark time.

$$n_{abl} = \left\lfloor \frac{512}{103} \right\rfloor = 4$$

$$n_l = 512 - 4 \times (103 - 1)) = 104$$

To rectify these discrepancies, we improved the row-allocation algorithm by creating a new function called 'calSectionHeights'; this function returns a slice containing the height of every section that each

worker will process. The slice is initialised with a length of the number of workers. The function then iterates over the height of the image, allocating each row one by one to the workers. Consequently, the number of rows allocated to each worker only varies by at most one between workers. As shown in the bar chart below (Figure 2), this has levelled out the sudden jumps that occurred before. This algorithm was also used in the distributed implementation.

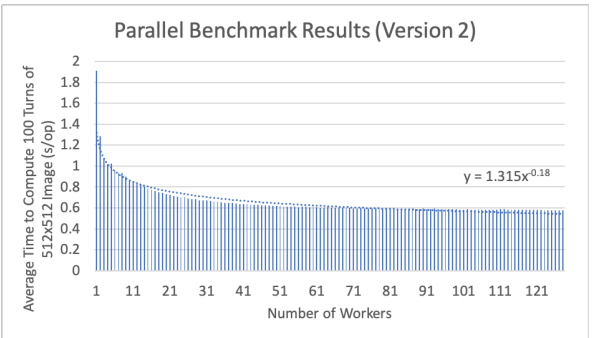

Figure 2: Benchmark results for our improved parallel implementation

The graph follows a negative exponential trend, decreasing at a slower rate as the number of workers increases. This trend may continue until more goroutines can no longer be added, due to the computer running out of memory (providing the image is sufficiently large enough). However, it is likely there reaches a point whereby the performance benefits from additional goroutines is matched (or possibly even outweighed) by the performance overhead of managing them.

The primary limitation of our implementation is that the maximum number of worker goroutines is the height of the input image. This limitation exists because our implementation does not allocate sections of the board to the workers on a sub-row level. Instead, our algorithm divides the board up into rows. Each worker is sent a collection of rows, comprising a section of the board to calculate the next state of, and the two vertically adjacent rows. For an image of r height and w workers, each worker will compute the next state for either $\lfloor \frac{r}{w} \rfloor$ or $\lceil \frac{r}{w} \rceil$ rows. If the number of workers is increased to the point that it is equal to the number of rows, each worker will be computing the next state for precisely one row. Therefore, any additional workers will be computing the next state of zero rows, and there will be no performance benefit. The reason we decided not to allocate sections on a sub-row basis was that at higher numbers of workers, performance benefits are negligible, and it is likely increasing the complexity of the algorithm for allocating sections of the board to workers would decrease performance.

2 Distributed Implementation

This implementation is designed for a distributed system, with workers operating on multiple AWS nodes. It consists of three major parts: the controller, the engine, and the workers (Figure 3).

2.1 System Design

We opted to use the net package to communicate data between the components of our distributed system. We decided to do this as it gave us both flexibility and full

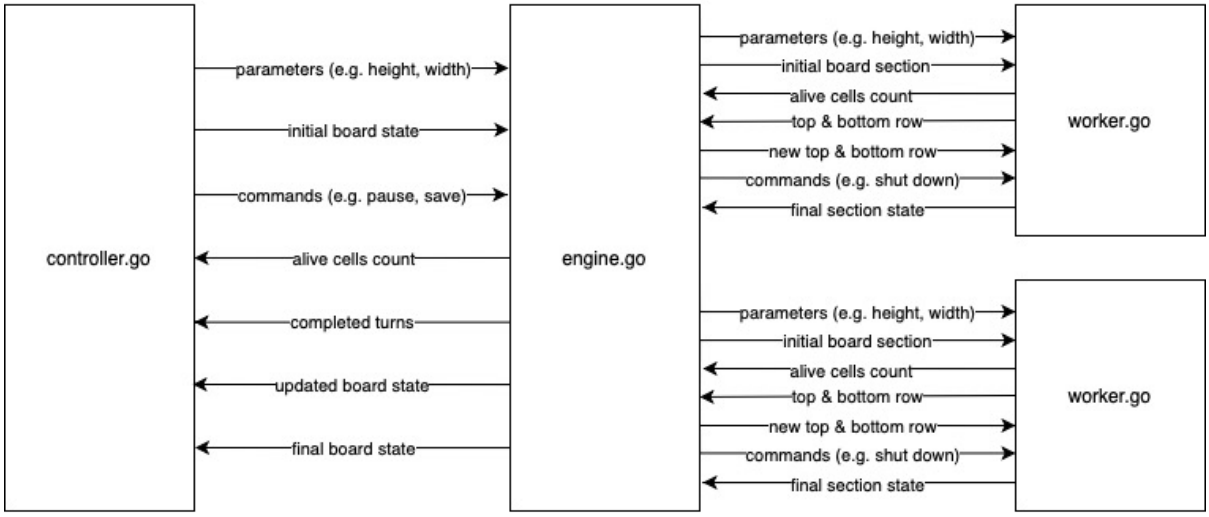

Figure 3: Structure of our distributed implementation with two worker nodes

control over precisely what data is communicated over the network. Our engine is configured as the server, whereas both the controller and workers are clients. Data is communicated via TCP packets as opposed to UDP—the retransmission of lost packets and error checking properties TCP offered a more robust, less error-prone system, albeit at some loss of speed.

The controller has four primary responsibilities. It sends the initial Game of Life board to the engine, as well as its parameters. Additionally, it is responsible for capturing keypresses from the user and forwarding the appropriate commands to the engine. It receives an update on the number of alive cells from the engine every two seconds. Lastly, it saves a PGM image of the board state once all turns have been computed, or when the 's' key is pressed.

Similar to the parallel implementation, the 'manageKeyPresses' goroutine in the controller loops, waiting for a value from the 'keyPresses' channel. The value received from the channel determines the command that is sent to the engine: either 'SAVE', 'PAUSE', or 'SHUT_DOWN'.

The engine handles the logic of the distributed system. After receiving the initial board and parameters from the controller, the role of the engine is to communicate and coordinate with the AWS workers to produce further iterations of the state of the board. To accomplish this, the engine initially uses the 'sendPartToWorker' function to send each worker a section of the board to compute iterations on; additionally, the engine sends parameters about each section.

The engine uses a halo exchange scheme to receive edge rows from each worker and send them on to the workers that require them. We decided our distributed implementation should use a halo exchange scheme, as opposed to transferring the entirety of each section at the end of each round for efficiency purposes. Transferring

data over TCP is not a quick process, so the fewer bytes that need to be transferred, the faster our implementation is. Additionally, at the end of each round, a command is sent to each worker telling it what to do next: either 'SEND_WORLD', 'SHUT_DOWN', 'CONTINUE', or 'DONE'.

The final part of our distributed implementation is the worker nodes. These act as data processors, handling the portion of the game board they have received and computing the next iteration appropriately. At the end of every round, each worker sends its two edge rows to the engine, to send to other workers that require them. Additionally, the number of alive cells remaining in its section of the board is sent back to the engine. Sending the number of alive cells back to the engine is necessary so that our engine can relay this information back to the controller every two seconds.

We designed our system to be relatively robust, but there are some limitations. Between boards being computed, worker nodes can disconnect, and new worker nodes can connect. However, while a given board is being computed, if a worker node disconnects, our system will crash. Additionally, the game engine is a central point of failure; if the engine disconnects from the workers at any stage, the whole system will crash.

As an extension to the requirements of the system, our implementation enables multiple controllers to be connected simultaneously. Each of these controllers can communicate with the engine and control the game currently being processed.

2.2 Benchmarking and Scalability

Benchmark results were collected using the large 5120×5120 image for 100 turns of the board. We benchmarked our implementation from one worker node to six. Each node in the distributed system was running on an AWS c4.xlarge EC2 instance, which has: an Intel Xeon E5-2666 processor, which has ten cores, each with two threads; as well as 750 megabits per second of dedicated EBS bandwidth. As with the parallel implementation, the benchmark program was run five times, and a mean was calculated for each worker. Five was deemed a suitable number of benchmarks to take an average from, as variance was minimal for benchmark results for all workers. We benchmarked three different versions of our distributed implementation, as shown in Figure 4.

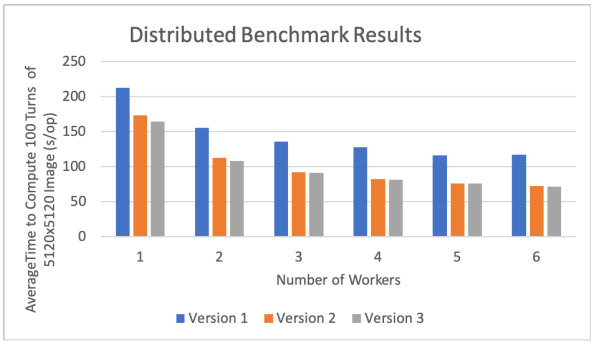

Figure 4: Benchmark results for our parallel implementation

For all versions of our implementation, there is a clear trend. As the number of AWS worker nodes increases, the average time to compute 100 turns of the 5120×5120 image decreases. This pattern

is because each node can compute the next iteration of their section at the same time. So, by increasing the number of nodes, the size of the section each node has to compute the next iteration of decreases. Consequently, the overall time decreases.

Although we could not benchmark our implementation with more nodes due to there being a 32 vCPU limit on our AWS accounts, this trend would likely not continue for much longer. The overhead introduced from managing each node, and transferring rows between nodes and the engine would begin to outweigh the performance increase received from the larger number of nodes.

After benchmarking the first version of our implementation and profiling it with pprof, we recognised that a large proportion of our program was spent sending and receiving sections of the board. Having analysed our code, we discovered that a newline byte was being sent after every cell. The repercussion of this was every time a section of the board was communicated around the network, an equal number of newline bytes were communicated to the number of bytes representing cells. As a result of this, communications between our distributed components were taking almost twice as long as necessary. For example, when sending the 512×512 image from the controller to the engine, we were sending a total of $(512\ bytes \times 512\ bytes) \times 2 = 524288\ bytes$, twice as many as necessary. This repeated inefficiency throughout our program was having a detrimental impact on performance.

To rectify the problem for the second version, we altered the algorithms so that a newline byte was only necessary once the entirety of a section had been sent. As a result, there was an almost 50% reduction in the number of bytes being sent across the distributed system. In turn, this almost halved transmission times and made the program 29% faster on average across the benchmarks, as shown in Figure 4.

To further optimise our implementation, for the third version, we provided each worker node with multiple worker goroutines. Similar to the parallel implementation, upon receiving a section of the board, each worker node splits their section into several subsections. Each worker goroutine computes the next iteration of their subsection. At the end of each turn, the subsections are appended back together, and the implementation continues as before. As shown in Figure 4, as a result of this optimisation the third version of our implementation is 3% faster than the second on average across the benchmarks, with more significant speed increases in the lower numbers of worker nodes.