

Names: Aaron Robey, Matt Simpson E-Mail: ar17092@my.bristol.ac.uk ms16934@my.bristol.ac.uk

Course: COMS20001-Concurrent Computing

Report

Functionality and Design

Our implementation can indefinitely evolve board sizes up to and including 1040x1040. All user feedback has been successfully implemented, allowing the user to start data input with button SW1, begin output with SW2 and pause processing rounds by tilting the board. The correct LED patterns are displayed to represent these actions.

The Game of Life logic has been implemented using a server-client interface mixed with the limited use of channels. The distributor process functions as a server, and takes an array of server-client interfaces, one for each worker client. The worker clients pass the server an array, and the distributor copies a subsection of the board to said array. These sub arrays consist of a group of board rows. Each client then has access to several rows equal to the total board height divided by the number of worker clients active, plus two. These additional two rows are the rows bordering the client's subsection of board above and below.

The board itself has been packed down into characters, with 8 cells being stored in one uchar, with each bit of the uchar representing one cell. This solves the problem of larger board sizes being too big to work with given the memory constraints we are working under. The cells are not unpacked from their uchars within the worker clients- instead, we apply the game rules in a bitwise manner, treating each of the bits in the uchar as a cell, and allowing for the board to wrap around when an edge is reached.

We chose this method as it seemed it would not just be effective at reducing our memory usage but would also allow us to implement the Game of Life's rules in a relatively simple manner. When implementing our bit-packing we decided to separate the main steps: packing a bit into a uchar and retrieving a specific bit from a uchar into separate functions. This was to make testing the correctness of each step easier, and to make the process of packing the bits easier to follow. Also, as each cell could be converted directly into a bit and stored with no other changes, the rules for processing a board made up of characters converted almost directly into the rules for processing bits. The only new thing we needed to account for was checking a bit on a different uchar in edge cases.

In our initial implementation of this bit-packing scheme, to ensure that certain worker processes couldn't somehow load from the distributor, process a round and update the main stored board before another worker could even load their segment of the board, we included a second board that would be updated by all the workers. This board would then be made the board that all workers loaded from while another stored board was set-up to be updated. Obviously this resulted in us using far more memory than we had available, so in an alteration to our implementation we set up our distributor so that it would not allow workers to update the main board until all workers had loaded their data from it.

We have a maximum of eight worker clients running due to the constraints of the board- we require the other eight cores to run processes dealing with user input, data handling and timing.

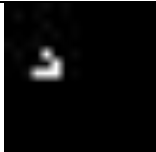



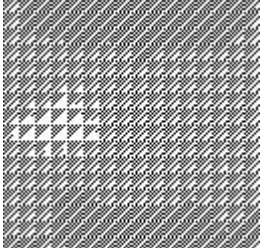

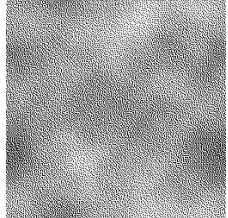
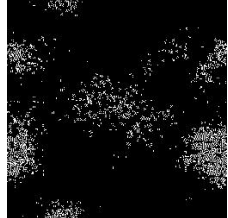
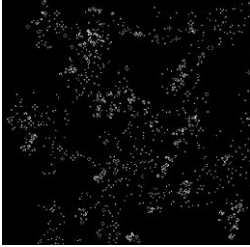

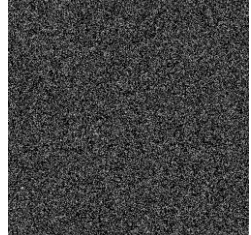
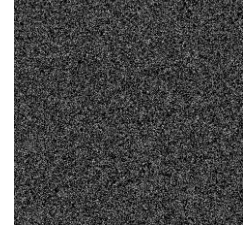
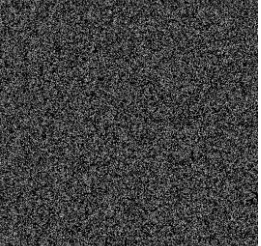
We opted to use a server-client implementation because it seemed to best fit the structure of the problem at hand- we needed a central process to distribute parts of an array to various workers and then have these workers output back to said process. This sounded like a server-client model and so we implemented it as such. This approach made it much easier to manage the threads than it would have been with a channel approach, as we could use server notifications to control when clients would process a round, have them only send output when requested, and so on. Using a server-client model also made development much more intuitive, though if we were to do this project again, we would probably use a fully channel-based solution for the sake of speed of round processing.

Our implementation also makes use of channels mixed with server-client interfaces. The server's load function is used only on the first round of a game. After that, the clients use asynchronous channels to communicate between themselves, sending the overlapping rows they need to process their outermost cells. This saves time by eliminating the need to receive redundant information from the server- the clients already have most of their subarrays updated to reflect the new round, as they processed them, so they need only receive updated versions of the outermost, overlapping rows which other workers are handling. However, this method does still require our worker threads to constantly be sending data to the server every round, which could be acting as a bottleneck in our system as every time the worker calls a server function, they send the entirety of the section of the board they are working on. But, there are still two main benefits to the implementation described above, the first being the greatly reduced memory usage over our initial implementation, as described earlier, and secondly using this method we can sync the I/O of our system with the processing of each round, allowing us to know that, for example, if a user request the exporting of the current round that the round will have finished processing before we begin to export the processed board.

Testing

Results after two rounds

The times are taken immediately after board processing is done, but before I/O checks, those checks add 0.02 secs.

Image Size	Initial Image	2 Rounds of Processing	Time per round			
			8 workers	4 workers	2 workers	1 worker
16x16			0.001 secs	0.002 secs	0.005 secs	0.009 secs
64x64			0.03 secs	0.04 secs	0.07 secs	0.15 secs
128x128			0.12 secs	0.16 secs	0.31 secs	0.62 secs
256x256			0.57 secs	0.74 secs	1.45 secs	2.50 secs
512x512			2.95 secs	3.94 secs	7.45 secs	11.01 secs
992x992	Generated in program, columns alternated between all live and all dead		n/a (Not tested, though would run)	n/a (Not tested, though would run)	40.5 secs	n/a (Unable to run on board of this size)
1000x1000	Generated in program, columns alternated between all live and all dead		n/a (Not tested, though would run)	22.28 secs	n/a (Unable to run on board of this size)	n/a (Unable to run on board of this size)
1040x1040	Generated in program, columns alternated between all live and all dead		18.41 secs	n/a (Unable to run on board of this size)	n/a (Unable to run on board of this size)	n/a (Unable to run on board of this size)

Testing Continued

As we can see above, our implementation can process images with sizes up to 1040x1040, with the main issue being that after reaching image size 512x512 the time taken to process an individual round grows substantially larger than what was needed prior. Another interesting observation we've made in the process of our testing is that as we shrink the number of worker processes, the maximum size image we can process also shrinks from 1040x1040 with 8 workers, to 1000x1000 with 4, to 992x992 with 2. This issue may be due to how we have distributed the worker processes across the tiles.

The table above shows the timings on our final implementation of the system. An earlier implementation, where we relied purely on server client communication with no channel communication had the following speeds:

Image Size	16x16	64x64	128x128	256x256	512x512
Time 8 workers	0.002 secs	0.04 secs	0.13 secs	0.63 secs	3.2 secs

When compared to the times observed in our final system, we can see that there is a slight improvement when using a mix of server-client and channel communication. The only difference between the two implementations is that channels are used for clients to send data between themselves rather than calling a server process, and we thus concluded that the server-client method communication is one of the biggest limiting factors on our speed. The channels we are using are streaming, or asynchronous, which we found to be faster than synchronous channels. This is both because streaming channels have the fastest data transfer rates in XC and because synchronous channels required additional control logic to run each round to schedule when workers would send and receive data.

Number of worker clients is also clearly a limiting factor on speed, as the test data shows. However, within the constraints of the board's hardware it is difficult to free up cores to run more workers, as they are needed to run various I/O and data handling processes.

We think one of the main advantages our system has is the ability to process large boards, with this mainly being achieved due to our bit-packing strategy, and how we split resources between the two tiles. Packing the cells down into characters, with one uchar representing 8 cells, allows for far bigger boards to be held in memory.

We also found our system to be quite easy to test, understand, and follow the processing of due to the nature of the server-client relationship between the distributor and the worker threads. This is mainly because in our system the distributor handles most of the required synchronizing between the worker threads when necessary and makes it easy to control and halt the processing of the entire system after any given round.

However, the server-client system does have one major drawback which can be clearly seen in the test results above, and that is that when processing large boards, our system takes a considerable amount of time to process each round. This could possibly be rectified by redesigning our system without a server client relationship between the distributor and the workers, and instead simply having a pipeline where one of the workers can send the processed data to the distributor when processing is finished, or having each worker use a channel to send its processed cells back to a controlling process.

Critical Analysis

As the results above show, while our implementation can process smaller images, up to 256x256, relatively quickly. When it is used to process larger images, e.g. 512x512 and 1040x1040 it slows down quite dramatically. As stated in Functionality and Design, we believe this is due to a bottleneck occurring in processing due to our current implementation requiring each worker to update the servers board each round. Due to this we feel that the biggest improvement we could make to our system would be removing the use of a server-client relationship and instead designing our system entirely around the communication between worker processes using channels. The two main reasons we could see this improving performance is that in a system designed entirely around channel communication between workers, we wouldn't need to spend additional time during each round of processing managing the order in which workers can load from and to the server. Also, there is the fact that client-server communication is slower than channel communication in general. Finally, in our current implementation, we do still require each client to store their segment of the array twice, the segment they are reading from for processing, and the segment they update. In a future implementation, we would design our system to avoid this issue which would mean that instead of storing essentially three instances of the board, we would only need to store two.

While we do feel our system could be improved in terms of its processing speed on larger images, we feel that our method of implementing the system does have benefits. The main benefit of this implementation, and the reason we ended up using it, was that in the development of the system, this method made sense when considering how the logic of the system was to work. As understanding how the system is distributing work to each of the worker threads, collecting the outputs of these threads, and handling I/O made much more sense when thinking of the process as a server managing the information sent to a series of worker clients, with the server also handling any user input. The other benefit of this implementation is that we can know that not only are all the workers only updating the server when every other has loaded from the server, but also due to the nature of the implementation, that whenever a user interacts with the system, either by pausing processing, or exporting the current state of the board, that all processing for the on-going round has to be completed before the users request is acted upon, removing any potential errors due to requests made interfering with processing.

While bit-packing has allowed us to process images larger than we could have if we had stored the board with a character representing each cell, we are still quite limited in board sizes we can store and process. We believe that by implementing a different way of compressing the boards data we could process boards of a substantially larger size. One method of compression could consider that on a typical game of life board, the majority of cells are dead. So instead of storing each cell individually, regardless of if they are living or dead, we could simply store the first cell in a sequence of repeated cells, and store how many times that cell is repeated before there is a change. While on boards with certain patterns, for example boards with columns of alternating live and dead bits, this could prove worse than simple bit packing, in general this compression method would require far less space than individually storing each live and dead bit on the board. However, this method would again slow down processing time, as either we would need each worker thread to decompress the board, which would also reduce the benefit gained from this compression method, or we would need to perform processing on these compressed versions of the board which could prove both difficult to implement, and slow.

Were we to try and improve upon our existing implementation rather than start from scratch, we could try to expand the mixed use of channels with server-client interfaces. Where currently each client checks in with the server at the end of each round, to ensure that the server always has an accurate count of round number, live bits and so on, we could eliminate this bottleneck by shifting some of the control logic into the client processes themselves. This would greatly increase the processing speed of each round.

Additionally, consolidating our I/O and data handling processes into a smaller number of processes would free up valuable cores on the board. This would allow us to run more workers, which would increase the speed with which rounds could be processed. The best way to do this would likely be to combine DataInputStream and DataOutputStream, as the program will never need to import and export simultaneously.

In summary, our implementation using 8 worker processes can evolve the Game of Life on a board of size 1040x1040, with each worker process processing 130 rows, and each round taking 18.41 seconds to process. As we reduce the number of processes the maximum board size decreases. This is due to a memory issue. While we are unsure exactly why this occurs, we believe it could be due to the way we have had to manually distribute the processes across both tiles to maximize the board size we can process with 8 processes.