Concurrent Computing Coursework 1: Game of Life

Andre Adel Alphonse Ghattas[1] and Fernando Fortes Granado[2]
[1] ag16145@bristol.ac.uk, *Candidate 97020 (Erasmus study abroad – Computer Science)*
[2] yt18331@bristol.ac.uk, *Candidate 71820 (Study abroad – Electrical Engineering)*

December 6, 2018

1 Functionality and Design

1.1 Data input and output

DataInputStream and *DataOutputStream* did not see any drastic changes in our implementation compared to the initial code. In the former, we code the values read from bytes to bits so that we can use less memory and reduce communication between threads. We do this and send the coded values to the distributor as we compute them. In *DataOutputStream*, we decode the values we receive before sending them out. We therefore decode the bits to bytes so that we could get the correct output image. We also interact with the LED's in both threads. In the former, we light up the green LED when we start reading the image and in the later, we turn off all LED's to indicate that writing is done.
To change the size of input, one could easily redefine the name of the input and output (*INFRM*, *OUTFRM*) and the width (*IMWD*) and the height (*IMHT*).

1.2 The Distributor

The distributor starts out by detecting the tilt of the board. When it's detected, it starts processing the image by sending each part to the corresponding worker thread and flashing the additional green light to indicate ongoing processing. It then turns off the LED's to indicate the end of the processing of the image. After that, it executes an infinite while loop. At each iteration, it sends a message to each worker telling them to keep working. This helped us in solving a bug we had where each worker stopped computing values after a specific number of cycles. In the while loop, the distributor also detects whether the thread handling the buttons has sent it any triggers. In the case where the button SW2 is pressed, a message is sent to all workers telling them to send their part of the computed image. The distributor then receives the image from the workers and forwards it to *DataOutputStream* which outputs it to the user. Finally, the distributor re-executes the while loop and the process restarts.

1.3 The Workers

A worker starts out by receiving the part of the image it has to work on from the distributor. On the first iteration, it writes the time to a variable (this is used to compute the time it takes the program to execute 100 cycles). A counter is then initialized to 0 (it's used to count 100 cycles). The thread then enters an infinite while loop. It starts out by sending its upper and lower rows to the neighbouring workers. This is done using their *id*'s. The workers with even *id*'s send their rows first and then receive the rows they need and vice versa. The reason for this is that updating the upper and lower rows requires the neighbouring pixels including those in the part of the image computed by other workers. The worker then proceeds to compute the new image which is stored in a new array. It first counts the number of live neighbours and then applies the Game of Life rules. With the bitset optimization added (where each pixel correspond to one bit rather than one byte), this was not trivia as we had to manipulate bits rather than just directly manipulate uchar's. After the Game of Life algorithm is executed, we change the image state so that it could be updated for upcoming rounds. We then test if we reached the 100^{th} cycle in which case we output the time elapsed. Finally, the worker sees what message the distributor has sent. If it received a 0, it executes a new cycle. If the value received is a 1, it sends the computed image state to the distributor as this means an output request was generated (SW2 was pressed).
We have modularized the code so that it is easy to change the number of worker threads. To do so, we only need to change the globally defined variable *nb_workers* and add/remove workers in the main function.

1.4 I/O handling threads

The *button_listener* and the *leds_handler* threads were created to avoid having multiple threads write to the same memory resources (buttons and LED's respectively). They are just simply a layer to tell the appropriate threads when a button is pressed and to light up a LED when it is requested by a thread.

1.5 Timer thread

A timer thread was initially sketched and the function was left in the code. However, we decided that it was not the most efficient way to measure 100 cycles as using a timer inside of the appropriate thread seemed easier and more accurate. We thus measure the time inside each worker thread.

2 Tests and Experiments

2.1 Image Outputs

The first image we used was the 16x16 test image provided. After 2 rounds, we get the following output image.

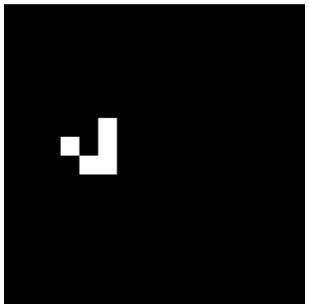

Image 1: output after 2 rounds on test.png

We ran 2 rounds on each image provided and got the following results.

Image 2: output after 2 rounds on 64x64.png

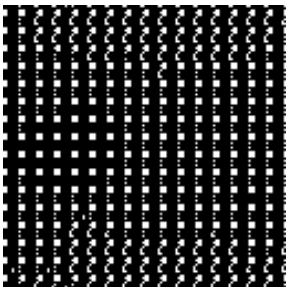

Image 3: output after 2 rounds on 128x128.png

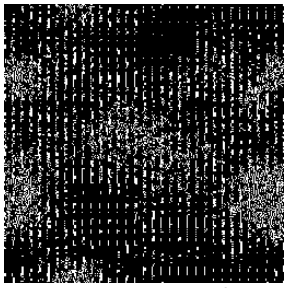

Image 4: output after 2 rounds on 256x256.png

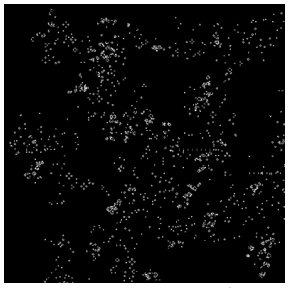

Image 5: output after 2 rounds on 512x512.png

We also ran 15 rounds of the game of life on our own input and got the following result:

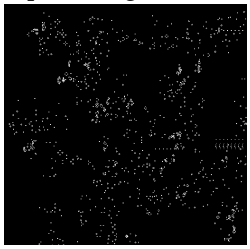

Image 6: input for our chosen image

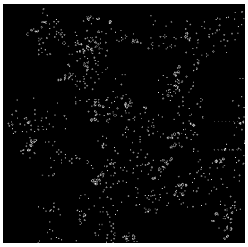

Image 7: output for our chosen image

2.2 Virtues and Limitations of the System

The system that we implemented has some virtues and some limitations.
First, we implemented a lot of optimizations in our system.
The bitset optimization is a big virtue of our system as it enables the different components of the system to communicate less. It also uses a lot less memory to store the images in the workers which helps in processing large images.
The *data_handler* function is also an optimization as it helps us make use of more cores during the computation of the images. It lets us shut down *DataInputStream* as soon as we are done using it.
We also do not store the state of the image in the distributor thread. Sending each worker the part of the image it needs to work on directly enables us to eliminate the array that used to store the values of the pixels in the distributor. The way we send around rows amongst workers also helps us do that as we don't need the distributor to send the relevant upper and lower rows itself. This means we use less memory and less distributor-worker communication.
As much as we optimized our system, given the time frame, we could not implement a perfect one. Therefore, our system has its limitations.
First, the bitset implementation has a shortcoming in that it compresses each line rather than the image as a whole. Therefore, if the algorithm gets an image input with a small width and a big height, the optimization would not be very effective.
The need to store two arrays in each worker is also another limitation of our system. It seemed like the most convenient way of implementing the workers but it is not the most efficient as it uses additional memory and time to copy the new array into the old one at each iteration.
Having an infinite while loop in the distributor also wastes a lot of computing power as it sends a "work" message to the workers at each iteration. It does not seem useful but it actually is as the workers do not do the required computations otherwise.

2.3 Timing the Process

We carried out tests on our system to see how well it works for different image sizes and with different configurations. We measured the time it took workers to finish 100 cycles. To do that, we placed a timer in each worker thread which calculated the time it took for its part of the image to get received and for it to execute 100 cycles on the image it received. We timed this for 3 numbers of workers in 3 different configurations with 5 image sizes (16x16, 64x64, 128x128, 256x256, and 512x512). We picked either 2, 4 or 8 workers. Workers were either placed all on one tile which did not have the distributor, placed on

one tile with the distributor or split into two groups and each placed on a different tile. We got the following measurements (in ms):

Table 1: times measured for the 16x16 image

16x16	1 tile	2 tiles	1 tile with distributor
2 workers	31	32	26
4 workers	31	30	24
8 workers	30	29	N/A

Table 2: times measured for the 64x64 image

64x64	1 tile	2 tiles	1 tile with distributor
2 workers	432	433	432
4 workers	411	407	405
8 workers	349	404	N/A

Table 3: times measured for the 128x128 image

128x128	1 tile	2 tiles	1 tile with distributor
2 workers	1716	1715	1705
4 workers	1595	1590	1589
8 workers	1399	1531	N/A

Table 4: times measured for the 256x256 image

256x256	1 tile	2 tiles	1 tile with distributor
2 workers	8111	8113	8108
4 workers	7664	7649	7656
8 workers	8325	8168	N/A

Table 5: times measured for the 512x512 image

512x512	1 tile	2 tiles	1 tile with distributor
2 workers	11126	11124	11065
4 workers	7672	9266	9216
8 workers	8334	8408	N/A

We can see that the general rule is that as the number of workers increases, the time it takes the program to do the necessary computations decreases. This makes sense because as we add more workers, computations are more parallelized and are therefore made faster. However, increasing the concurrency of the system implies increasing the communication between different components of the system. This can be quite costly in time. We can see that it becomes noticeable for bigger images. In fact, the quickest way to compute the 256x256 and the 512x512 images with our architecture is by using 4 workers and not 8. Therefore, in these cases, the concurrency does not compensate the communication.

We can see that in general the tile configuration does not have a big impact on the speed gain. Working on 2 tiles instead of 1 for the 256x256 image gives us only a gain of 15ms which is relatively not huge as it is overall a 0.196% gain in speed. There are some exceptions though as we can see a 17% decrease in the time it takes to compute the image using 1 tile instead of 2 for the 512x512 image and a 13.614% decrease in the time it takes to compute the 64x64 image using 1 tile as opposed to using 2. We can see that there is no real general pattern. For example, putting 4 workers and a distributor on one tile seems to be the optimal choice for the 16x16 image but isolating the workers with the distributor is not the optimal choice for any of the other image sizes.

We also thought about the performance of our system with streaming channels. There would be a slight speed gain in general. That is because streaming channels are not synchronized. A thread that is sending a message to another does not have to wait for the other to receive its message in order to execute the rest of its code. The messages are instead stored in a buffer and the thread can continue to execute the rest of its code without being blocked by the receiving end. Thus, using asynchronous channels gives our program a slight speed gain.

By examining which parts of the process takes longer, we have noticed that it is the reading and the processing of the image that takes the most amount of time. In fact, it takes longer than actually computing the images in each worker and exchanging relevant data among workers. This is because it requires the communication of a big amount of data between the distributor and the workers. The distributor sends out the data as it receives it which is better than storing it and then sending it. But, this does not get rid of the problem completely as there is still a huge number of bytes to be communicated. Thus, in general, communication takes more time than image computation in the system we have designed.

The same rule holds for communications and computations in workers. For example, with a 64x64 image and 8 workers on two tiles, communication takes 338ms while computation takes 27ms. Therefore, we can deduce that the communication takes more time than the computation as this is a trend that recurs for all configurations tested (which are the same as the ones given above). We can also notice the trend that the computation phase for each worker takes longer when the workers are all on one tile as opposed to when they are not. We could in fact explain this by the fact that they make use of more computing power when they are split between both tiles while if they are placed on the same tile, each uses one core only. We can also observe that communications take longer when the workers are split on different tiles. This result makes sense as well, as they are further apart when they are not on the same tile and thus channels require more time to do communication among them.

Thus, we have measured time for different configurations of the system and we have also conducted time measurements for different components of the system and provided possible explanations for the measurements observed.

3 Critical Analysis

3.1 Performance of the system

Our system has been designed so that it is as efficient as possible within the time frame given. We can see in the tables above that it is able to compute images of sizes up to 512x512. It also does it in an acceptable time. Analysis of the system let us see that there is always an optimal configuration of computing the images for each image size. Therefore, it performs overall well. It uses bitsets to work with more compact images in the components. It also shuts down *DataInputStream* thread when it is no longer needed. It sends the data from the distributor to the workers only once in an efficient and elegant way. Therefore, the overall architecture of the system has been implemented well.

Given the time frame in which we had to design our system, there are many limitations which come with our implementation. As mentioned above, storing two arrays in each worker is quite costly in terms of memory. It is one of the reasons that made our image size upper bound 512x512 (it does not work with 1024x1024 images which use too much memory). Despite the fact that we think we implemented communication between workers as efficiently as we could, we can see that in the results above that it is quite costly in terms of time. This might be because communication between threads takes time in general. The reading and processing of the image is costly as well as mentioned above which affects the performance of the overall system.

3.2 Ways to improve the system and further extensions

Given the time frame in which we had to develop our system as well as the hardware we had to work with, there are many extensions and optimizations that could have been added.

Regarding the code itself, there were some optimizations that could have added and some optimizations that were added could have been made better.

For instance, storing two arrays per worker is quite costly in terms of memory. This could have been avoided by instead using an array to store the state of the part of the image of the worker and another to store three rows of the image only. That way we could iterate over part of the image to compute the new values of the bits and store them in the second array. After reaching the 4th row we would start copying the second array into the first while modifying the values in the second to store the next three rows. This would go on until we have computed the whole part of the image we need to compute and stored it properly in the first array for further computations in subsequent rounds of the game. This would have implied more moving around of data between arrays but it would have used less memory. It is a trade-off that we would have considered making if we had more time to add this optimization in our system.

Another possible addition to the code was to merge the distributor and *DataInputStream*. One main purpose of their separation is the modularization of the code. But, having them as 2 separate threads forces us to have more communication and to thus use more channels. This slows down the system and limits the number of channels we could use for communication between other components of the system between which communication is mandatory.

One optimization we could have added but which slightly goes beyond the scope of the others is the addition of more compact ways to store sparse images. One could imagine storing the indices of black pixels in a predominantly white image or vice versa. This would require more complex operations and data structures though. Therefore, it was just an idea we had to be able to use less memory but we could imagine that it is hard to implement in practice.

An improvement to the bitset optimization we implemented would be to treat the two dimensional image as a one dimensional array. As mentioned before, our optimization does not help with images which have a small width and a big height. This improvement would have enables us to compress the image as a whole and not just each row by itself making the optimization more flexible which would help us get better (mostly speedier) performance for all image sizes.

A further optimization could be parallel distribution of the data when it is received from the distributor to the workers but that would imply parallel reading of the image which could require tweaks to the hardware as parallel memory access would be needed. We could further elaborate on this and imagine a system where each worker reads its own part of the image in parallel without the intervention of a distributor. Shared memory could be a solution.

Regarding the hardware, one important limitation is the number of channels that we could use. Having more channels would have made direct communication between workers and *DataOutputStream* possible. This would have reduced the overall communication in the system as the workers would not have been obligated to send their data to the distributor first so that it could send it to *DataOutputStream*. We have already seen that communication is one of the most costly operations done in our system. Therefore, this would have resulted in speedier system.

Finally, one could image a system where each worker computes its part of the image in place without having store arrays locally. However, that would require more advanced technology.