JV14984 RS14369

Concurrent Computing

Functionality & Design

Our program accepts images up to 946,729 pixels which is approximately a 973x973 pgm image. These are loaded in with a modified image loading system to save memory on the explorer tiles. To begin loading in the image the user clicks Button SW1. While loading of the image the main LED lights green. Once loading is finished the smaller separate LED flashes once per processing round and processing can be paused by tilting the board on its x-axis. Doing so will produce a status report listing the current round, number of live cells and total processing time thus far as well as the average processing time per round. Processing can be resumed by tilting the board back to a horizontal orientation.

Pressing Button SW2 immediately begins exporting the most recently completed round. The blue LED will be lit while exporting is taking place.

The image is internally double buffered which allows us to immediately begin exporting a complete, correct image whenever required. Rows of the image are distributed to worker processes to complete by passing references to pointers. The workers will always be working on either one of the buffers at any time, but never both at the same time. At the end of each round pointers tracking the two buffers are swapped and the next round of processing commences. By using pointers to describe data for workers rather than segments of memory we greatly reduce communication costs and can process at great speed. The downside of the double buffered approach is memory overhead that holding two of the images introduces.

To counteract this we optimized other parts of the program for memory such as the loading and export stages. By replacing the sscanf function provided for reading in the image with our own input function we were able to save 30kb of memory on one of the tiles. We also internally store the pgm image as bits of an array of uchars rather than the uchars initially read in.

In order to improve our processing efficiency we implemented a bounding box which determined the section of the image that contained live cells and only processed that section. This reduced the amount of time spent processing part of the image that wont change in a given round.

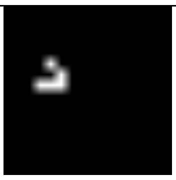
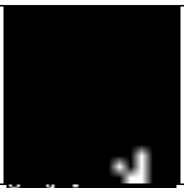
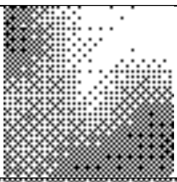
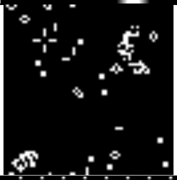
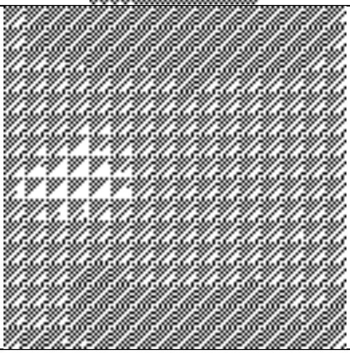
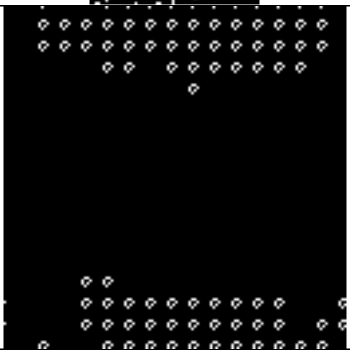
We are able to visualize the evolution of the Game of Life on a screen by taking advantage of the board’s high speed USB port.

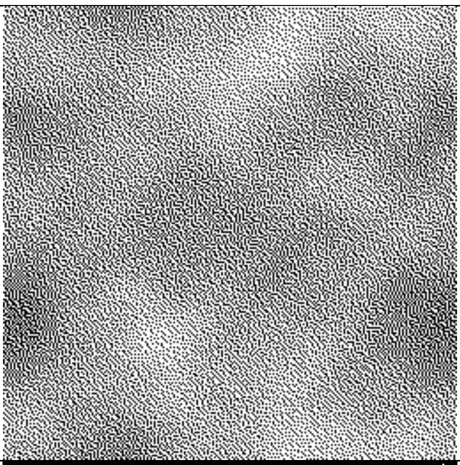
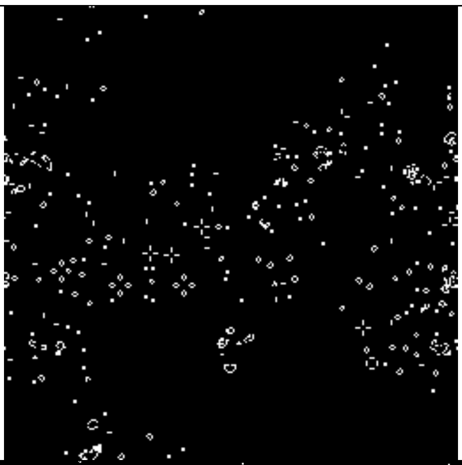
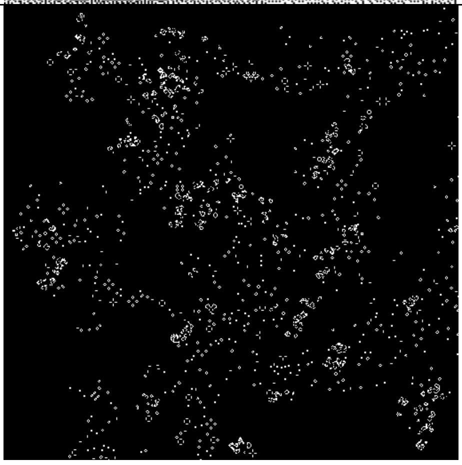
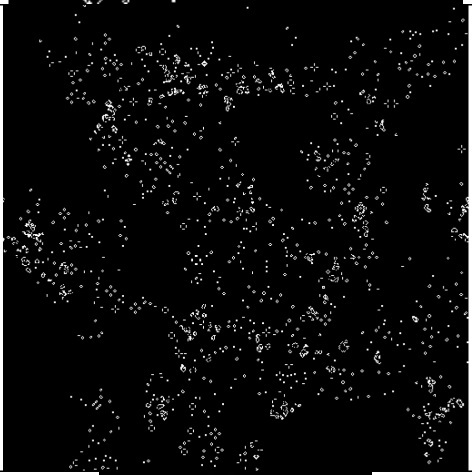
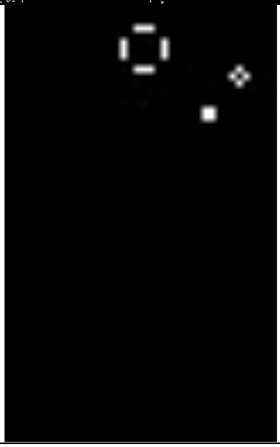
Tests & Experiments

Table 1: Game of Life Test Results

Image Size	# of Workers	10000 round completion time (ms)	Average processing time per Round (ms)
16 x 16	1	9400	0.94
16 x 16	2	4700	0.47
16 x 16	3	3300	0.33
64 x 64	1	191600	19.16
64 x 64	2	95500	9.55
64 x 64	3	65900	6.59
128 x 128	1	750100	75.01
128 x 128	2	376500	37.65
128 x 128	3	250900	25.09

Table 2: Test Images and Results

Input Image	Output Image
	
	
	

Critical Analysis

The results shown in Table 1 above show that the number of workers directly impacts the throughput per round. The throughput per round with x workers can be approximated by dividing the throughput per round when the program is ran sequentially (with 1 worker) by x. This means that one way we could improve the performance of our implementation is to add more worker threads.

Our current implementation utilizes 4 worker threads. We initially had 8 worker threads but had to reduce the amount of threads in order to utilize the USB port (which restricted our core usage) to visualize data on the screen. We felt that adding this extra functionality was worth the tradeoff of having our processing speed. In order to improve the utilization of cores in our implementation we made parts of our code distributable so that tasks shared cores with their caller.