Year 2 Concurrent Computing GOL Report - TB1 2019

Arun Steward & Junoh Park - University of Bristol

Functionality and Design

Both of the two submitted final non-memory sharing versions of Conway's Game Of Life, versions 1.6 and 1.6.5, are fully operational. They fit the Stage 4 specification, being compliant implementations that take two different approaches to solving various problems throughout the development process at Stage 3. Versions 1.6 and 1.6.5 mainly diverge at Stage 3 where it was necessary to make the program handle any even number of threads.

Firstly, version 1.6 is based on version 1.4 and solves this problem by using "grey cells" of value 128 that are skipped when applying the GOL logic; that being to add a matrix of appropriate dimensions to the top of the input in order to allow the number of threads to divide this augmented image evenly. The reason for doing this is that this approach exceeds the specification for the program at this stage of development, meaning that it is able to handle any number of threads being used rather than only an even number as specified. Another benefit to this system is that, unlike the conventional approach of having uneven sized "workers" and then dividing the work as evenly as possible between them, this system allows for the use of more threads than there are rows of the input. However, by padding the input using these "grey cells", a performance penalty is incurred. The effects of this will be explored later because, by adding extra height to the input image, there could be "workers" present whose work is not useful in computing the desired result. For example, in the 16x16x10 input. Hence, in these situations the performance scaling of the implementation is non-existent at some points.

Version 1.6.5 is a more conventional approach based on version 1.4.5 which uses the most even distribution of work that is possible without padding. This implementation also allows for odd numbers of threads, something we determined would be useful at Stage 3 for both versions in order to allow a logical processor to be left free for the "distributor" thread which at this stage still performs constant work of receiving and re-sending the world from and to the "worker" threads. Initially this approach had a flaw wherein if the number of threads exceeded the height of the input, the program would fail at run-time. This problem was overcome by having "dead workers" if this situation was happening, which perform no computation at all, leaving the number of "alive workers" equal to the number of rows of the input. This solution is not particularly efficient, although is more optimal given a larger matrix and a sensible number of threads.

At Stage 4, both solutions employ a halo-exchange system that eliminates the need for the "distributor" thread to send the world to the "workers" each turn. Instead, the "workers" directly share their halos through dedicated channels to each-other which significantly cuts the amount of data sent at the start and end of each turn. In version 1.6.5 this was a fairly straightforward logic problem. However, in version 1.6, this caused a substantial complication due to the padding of "grey cells" now potentially present on the top of the image. This presented a far more complex problem which through careful pen on paper algorithmic design was overcome using logic to jump over the "grey cells", forming two cycles of halo exchanges within the "workers".

For Stage 5, we decided to use the memory sharing model by making both previous implementations share the world and the sections of it in slices over channels rather than arrays over channels. This way, the communication of a section is done by pointer instead of by value. Throughout the project, since Stage 1b, the approach to transmitting bits of the world via array was to pass sections of size sixteen bytes over a channel. The reason for doing this was that, as the passing of these sections over the channel is synchronous, we wanted to minimize the number of these events as we feared that they would impact the program's performance. The reason for using sixteen bytes as the size of each array and slice was because we knew that it was a number that would divide all of the image widths that we needed to be able to input. If we wanted to take in an image which was not of a width of a multiple of sixteen bytes, then this optimization would not work. These extension versions, 1.7 and 1.7.5 will be analysed later on.

Tests, Experiments and Critical Analysis

Version 1.0

Version 1.0 requires little introduction; it is the first naïve implementation of the program wherein the Game of Life logic is computed and enacted on a sequential, single threaded approach inside the "distributor" function. This is the slowest implementation of the program. This implementation is fairly consistent as can be seen in the appendix.

The data distributions here are closely modeled by a normal distribution. The largest input produced a set of results that are closely represented by a normal distribution with mean and standard deviation shown above. This data shows a baseline average time for future reference as well as demonstrating the consistency of this program too.

Hypothesis (1): *As this version only uses one thread, the distributor, to compute the new state of the input data; provided that the testing computer is idling when the test is performed, and it was, it will almost always be able to allocate an uninterrupted logical processor to the distributor thread.*

Hypothesis (1) could explain the consistency and hence the task is seldom interrupted in its execution. This behavior is not only observed in the Linux resource monitor during computation and therein supports the Hypothesis (1). But also, when testing on Windows 10, where the core clock speed of the processor (in our case an Intel Core i7 7700HQ @ 2.81GHz) is seen to increase to the maximum boost clock possible on one single core (in our case 4.2GHz), which itself indicates a single thread taking a large amount of processing power.

Version 1.1

Version 1.1 is the first major leap in complexity and under certain circumstances, also reduces the run-time. This is because there are both upsides and downsides to the new methodology at play. This new version aims to send the world in as most an efficient way as is possible without employing the use of some simplified ZIP compression. The results of Version 1.1 are displayed in the appendix.

The distributions here in comparison to the results of version 1.0 are very interesting. Primarily and predictably, this new version is faster on the larger input images than version 1.0.

Hypothesis (2): *The new version is able to use multiple threads and consequentially multiple logical processors in parallel rather than just one as before.*

Hypothesis (2) is confirmed by looking at the resource monitor again while the process is in execution. On the 512x512x8 test, eight cores can be seen to be doing work. Not only is this observed in the resource monitor but evidenced by the data above for version 1.1, showing a 1.35x improvement over version 1.0 in the benchmark.

More interestingly however is the fact that, upon closer inspection, contrary to our expectations, version 1.1 is actually slower on small input matrices.

Hypothesis (3): *The inefficiencies incurred by the sending of the world to and receiving the world from "worker" threads every turn, is only outweighed by the increase in performance granted by leveraging the use of multiple threads only after a certain threshold has been reached.*

This Hypothesis (3) makes intuitive sense as, for instance, the 16x16x8 test spends a large proportion of time sending the world over channels to the "worker" threads and back. In addition, because this particular test is so small, computation wise, the single threaded version 1.0 can outperform the new version as it is not restricted by this added time complexity. In our testing, we found that test 64x64x4 was the threshold where the new version was able to outperform the old version.

Versions 1.2 & 1.3

Version 1.2 changes nothing to do with the performance of the program, simply adding on the ability of the program to: print an intermediate PGM image during execution, pause the execution of the current game mid-execution and to abort the current task and print any progress made up to that point.

In theory, version 1.3 also changes nothing to do with the performance of the program. This version adds the "ticker" function to the gol.go file that simply prints out the current number of alive cells after every two seconds. In practice, on the lab computers, this has very little to no real effect on the overall run-time greater than what could be considered noise. However, when testing an eight thread benchmark on a lower core count CPU (Intel i7 7700HQ @ 2.81GHz, a 4 core, 8 logical processor CPU), we observed a slight impact on performance from this added function.

Hypothesis (4): *The ticker thread every two seconds will need access to a logical processor of the CPU and will achieve this by blocking one of the "worker" threads, interrupting that "worker" from it's computation and therefore being detrimental the performance of the program.*

This Hypothesis (4) is seemingly the only possible explanation to the symptoms that can be seen through the task manager (Windows 10) where one logical processor can be seen to briefly dip from 100% use for what, presumably, is the ticker thread being woken by the scheduler. Be aware that this impact is not seen on a lab computer because they have 12 logical processors so have no requirement to suspend a "worker" in order to facilitate the "ticker". However it is worth noting that this performance impact is non consistent, fairly small and non reproducible on a lab computer so is in reality both insignificant and also the statistics which show the decreased performance are incomparable to the other versions as they were taken using a different computer so are omitted from this report.

Versions 1.4 & 1.4.5

Versions 1.4 and 1.4.5 both solve the Stage 4 task of implementing the ability to use a number of threads not equal to a power of two, but for all even numbers. it is pleased to report at this stage that we have not only one working solution to this problem but two, both radically different in their approach to the issue. In addition to this, both versions 1.4 and 1.4.5 exceed the specification and can utilize any positive natural number of threads, not just even positive natural numbers.

Version 1.4 achieves these outcomes by utilizing the padding paradigm as mentioned in the first section. The performance results for version 1.4 are as in the appendix.

Version 1.4.5 achieves the outcomes more conventionally, by having variable sized workers. The performance data for version 1.4.5 is in the appendix.

It can be seen that version 1.4.5 is substantially faster on the large matrices than version 1.4, although version 1.4 is far faster than version 1.4.5 on the smaller benchmarks, such as 16x16x8. These effects are both able to balance each other out. Both versions are approximately 17.5x faster than their predecessors due to splitting large functions into smaller ones, allowing for much more compiler optimization.

Hypothesis (5): *Version 1.4 has a more efficient "distributor" function implementation than version 1.4.5, however version 1.4.5 is faster at computing a larger input's turn because it has the inherent advantage of not having the added complexity of grey cells which slow down the individual "worker" threads in version 1.4.*

Hypothesis (5) is supported by the data mentioned above as previously explained and can be validated by considering the extra steps put into the "worker" threads by version 1.4 whereby the worker must first check each cell for being a "grey cell" before computing any progress. Then version 1.4.5 adds extra complexity into the distributor function to decide how best to split the task between the workers. These two additions support the Hypothesis (5).

It is further worth noting at this stage that: Firstly, at this time, version 1.4.5 cannot handle the case where the number of threads exceeds the height of the input, this is resolved later in version 1.6.5. Secondly, the need to have two versions at this stage was dictated by fears that the padding concept used in version 1.4, our first approach to the problem, would cause issues when attempting to implement the halo-exchange system in Stage 4.

Versions 1.5 and 1.5.5

These versions were used as a development platform for creating versions 1.6 and 1.6.5. As such they are not functional but version 1.5.5 has been included in the final submission to show the progression from stage 1.4.5. At this stage, version 1.4.5 continues to suffer from the same issue as version 1.4.5.

Versions 1.6 and 1.6.5

These versions represented a major leap in performance, at this stage both introducing the long anticipated halo-exchange systems.

Version 1.6 is at continuation of version 1.5 and 1.4 before it. The performance increase is substantial from version 1.4 as can be seen in the appendix.

Version 1.6.5 is a continuation of version 1.5.5 and 1.4.5 before it. Again, there is a huge increase in performance demonstrated in the appendix.

Both versions at this stage increase in performance significantly from their predecessors. This is exhibited primarily in the mean benchmark completion times of these implementations.

Hypothesis (6): *The halo exchange system is a marked improvement in the amount of data being transmitted between threads, representing a decrease in data transferred from the entire board plus the halos of each worker going to and from the distributor each turn to simply the halos going directly between workers.*

Hypothesis (6) explains the reason for the 1.5x multiplier for performance for version 1.6 compared to version 1.4 and the 1.7x multiplier for version 1.6.5 compared to version 1.4.5. In addition to this, there is a hidden extra reason in the Hypothesis (6) for the increase in performance, not only does the "distributor" not need to be woken by the scheduler during computation, removing the possibility of disruption to the workers, but the data is no longer all sent via one thread. This allows for less time to be wasted as the one thread redirects all inbound sections. The culmination of these improvements are the giant leaps we see above. Both versions can also be seen above to use a lot less memory on average than their predecessors, most probably caused by not needing to transfer the entire board and the halos any more.

At this stage we can also see that the performance difference between versions 1.4 and 1.4.5 have carried over and have been amplified by the new improvements too. This is because just as in those versions, the same flaws still apply to both of their descendants, the one extremely important exception being that the issue with version 1.4.5 and 1.5.5 crashing when the number of threads exceeds the height of the input has been fixed at this stage.

Versions 1.7 and 1.7.5

These versions are where Stage 5 is implemented onto the foundations provided by versions 1.6 and 1.6.5 respectively. Stage 5 involves the conversion into using a memory sharing methodology.

The results for versions 1.7 and 1.7.5 are outlined in the appendix.

At this stage there is no meaningful change in performance characteristics between these two versions and their immediate predecessors. Although the differences between version 1.7 and 1.7.5 are just as before.

Hypothesis (7): *The speed of sharing information using shared memory is approximately equal to the speed of doing the same over value sharing channels if and only if there is no change in performance.*

As there is no data to contradict the Hypothesis (7), we can assume with high probability that there is very little difference in speed between sharing pointers to shared memory over channels and sharing the data itself.