An SVP implementation of Game of Life

Thorben Krüger  Marten Lohstroh  Vasco Visser

Abstract

In this report we present our results of the practical assignment for the Concurrent Systems course at the University of Amsterdam. We were assigned the task to devise a parallel implementation of Conway’s well known “Game of Life”, specifically tailored to the Microgrid execution environment. This report first gives a brief introduction to SVP and the Microgrid simulator. A description of our implementation follows and some of its characteristics are discussed. Finally, the program was subjected to a number of benchmarking tests, the results of which are also presented and discussed.

1 Introduction

What started as a novel latency-tolerant processor architecture called DRISC [1], over time developed into a general concurrency model called Self-Adaptive Network Entity (SANE) Virtual Processor, or Self-Adaptive Virtual Processor (SVP) [2]. The SVP model is a parallel programming model for an abstract machine, which defines concurrency and communication in abstract terms. It provides dynamic control of concurrently executing programs, including the ability to manage the allocation of the required resources. The goal of SVP is to decouple the scheduling of programs in hardware from the program implementation i.e. implement once, compile once and run on different architectures.

The SVP model is implemented both as a parallel programming language called µTC and an architecture called the Microgrid. An execution environment called the Microgrid simulator is available to run SVP programs a software emulation of this architecture. A toolchain provides that through Simple Language (SL) statements, which during compilation it translates into µTC code, SVP semantics can be added to ISO C. Using these constructs, the programmer is able to expose concurrency by defining thread families that can be efficiently scheduled through arbitrary thread interleaving during execution.

In order to obtain some insight in the dynamics of the Microgrid, a computational problem carrying a sufficient amount of concurrency was chosen to implement, parallelize and deploy on the Microgrid simulator. Performance benchmarks were gathered from the simulation and used to measure speedup in terms of number of clock cycles used in different configurations.

The problem of choice was Game of Life (GOL), which is a cellular automaton devised by John Conway, that elegantly displays how a system with very simple initial state and ditto rules can result in intriguingly complex behavior producing patterns that evolve in surprising ways. As the name suggests, this process is in fact somewhat reminiscent of life.
2 Implementation

2.1 Problem

“The universe of the Game of Life is an infinite two-dimensional orthogonal grid of square cells, each of which is in one of two possible states, live or dead. Every cell interacts with its eight neighbors, which are the cells that are directly horizontally, vertically, or diagonally adjacent. At each step in time, the following transitions occur:

- Any live cell with fewer than two live neighbors dies, as if caused by underpopulation.
- Any live cell with more than three live neighbors dies, as if by overcrowding.
- Any live cell with two or three live neighbors lives on to the next generation.
- Any dead cell with exactly three live neighbors becomes a live cell.”

[3]

2.2 Details

The problem is broken down into square chunks of a constant size $n$, each containing $n^2$ cells. Each chunk is represented as a C struct containing pointers to its 8 adjacent neighboring chunks, an active/inactive flag, its own coordinates in the universe and two byte-arrays containing the state of all cells in the chunk. One of the arrays pointed to in the chunk struct contains the current state of the simulation, the other array is used to store calculated state for the next simulation step.

For each iteration, worker threads calculate new byte-array values for its particular chunk. To determine the cell states near the edge or corner of a chunk, the cell states at the corresponding edge or corner of a neighboring chunk have to be taken into account. Since a chunk carries references its neighbors, it can access them efficiently. A chunk’s current byte-array is replaced by the new, freshly determined one, if all worker threads have finished for the current iteration, i.e., all chunks have been processed.
2 Implementation

Fig. 2: Two levels of parallelism.

If the new byte-array for a chunk differs from its current one, the chunk is marked as active (i.e. the “active” flag is set). When a chunk has one or multiple state changes at an edge or corner, the “active” flag of the concerned neighboring chunks is set for the upcoming iteration. If the neighboring chunk is not present, the creation of a new chunk is requested by inserting the coordinates of the absent chunk in the request queue.

A central dispatcher is responsible for creating worker threads for each chunk for each iteration. The universe can grow virtually infinitely (i.e. until all memory is consumed) while the chunks that undergo no changes between iterations are declared inactive and forgo being pointlessly processed until a state change occurs at one or more of the edges of their neighboring chunks, which results in them being activated again. The dispatcher is also responsible for allocating and initializing new chunks that have been requested.

2.3 Remarks

The chunks in our simulation each represent a part of the two-dimensional GOL universe. Because of the two-dimensional nature of the universe it would be convenient to represent the part of the universe stored in a chunk as a two-dimensional array. However, in the C programming language it is impossible to allocate a two-dimensional with consecutive memory segments. A naive allocation of a two-dimensional array will actually result in multiple arrays, where each array represents a row in the two dimensional context. Having an ordered set of arrays to represent a two dimensional array has performance implications, for that reason it is preferable to store the data of an array in consecutive memory segments. Our program does not use a classic two dimensional array, but instead one array is allocated that, by the use of some “pointer magic”, is logically approached as if it were a two-dimensional array.

The current implementation still leaves room for several improvements. For example, the fact that each cell takes 1 byte to store no more than a single binary state is of course memory-inefficient. Also, the current fashion of processing starting conditions requires the initial condition to fit in one chunk, which imposes a possibly inconvenient lower limit on the chunk size. The exact size of the chunks would make for some inter-
Methodology

Testing parameter tuning. Setting it low might allow minimization of “wasted cycles” on chunks which have a low amount of activity but increase memory and scheduling overhead. Setting it high has fewer reference-following to neighboring chunks and dispatcher requests, but coarser granularity of active versus inactive regions.

A performance issue could arise when the universe drastically grows i.e., more chunks become alive and larger worker-thread families are dispatched per iteration. Although the worker-threads expose much concurrency, which would scale well on the Microgrid, the main-thread would become a bottleneck as for each iteration it sequentially processes requests, traverses the hash table and enqueues chunks for the next iteration. More concurrency could be exposed if the hash table traversal would be replaced by a thread family that operates on a simple array. Of course, replacing the hash table with an array negatively affects the look-up time which is suffered each time a chunk request is handled, but also this operation could be performed by a thread family using winner-takes-it-all synchronization in which each thread scans a part of the array. Nevertheless, when significantly more chunks enter the request queue then cores are available to perform partial array scans, the single-threaded hash table look-up would again out-perform the parallelized array traversal.

3 Methodology

The SL library offers performance counters for benchmarking code segments, which report the number of clock cycles per iteration. The worker code is of primary interest, because it has been properly parallelized. The performance implications of several individual parameters with regard to the worker code are investigated, namely:

- number of cores (restricts the number of cores per place);
- block size (restricts the number of threads per core per thread family);
- and locality (whether threads are bound to single core).

Speedup is measured between a sequential and two different parallelized versions of the code. Whereas the first version only features one level of parallelism (one thread per worker), the second version features an additional level of parallelism (one thread per cell).

Lastly, the hypothesized speedup discrepancy between the parallelized worker code and sequential main thread is investigated. Due to time constraints and the fact that the execution environment is very slow, we decided not to vary in problem size.

4 Results

Benchmarks where obtained using an initial condition referred to as “Rabbits” which is characterized as a Methuselah, which is a small “seed” pattern which takes a large number of generations in order to stabilize. Only random banked memory (RBM) configurations were found to work as expected, without exhibiting deadlock or reporting attempts to read or write from/to non-readable or non-writable memory. It was assumed that these problems emanate from a bug in the toolchain. Therefore we solely benchmarked this particular RBM hardware configuration.

Figure 3 shows the number of active chunks per iteration, which provides some background for the following benchmarks as they indicate the actual workload processed by the worker-thread family.
Figure 3 shows what are the performance critical aspects of the program. The line that shows the cycle count for the request queue processing has temporary spikes, these spikes occur when memory for one or more new blocks has to be allocated. The line depicting the hash table traversal seems to behave linear with respect to the number of blocks being calculated. By comparing the worker-thread lines for 2 and 32 cores, we can see that the worker part (i.e., calculating the new state for each iteration) is the most performance critical aspect. Also clearly visible is the benefit the worker threads have when more processors are available. In our program both the request queue processing and hash table traversal are sequential procedures, therefore their cycle count is not shown for different numbers of processors.

In Fig. 5, performances for various Microgrid parameters is compared to a sequential baseline. In Fig. 5a, with a relatively coarse grained level of parallelism, restricting it to concurrent worker threads, we compare the system’s performance using different numbers of cores to a sequential baseline. Of particular note in this plot is the region
beyond the 252\textsuperscript{nd} iteration. The clock cycle counts for the worker-parallelized version running on 32 cores henceforth become exactly equal to the sequential baseline. On the other hand, the clock cycle counts for all other numbers of cores (except 1) also become equal but on a slightly lower level. Finally, the single-cored graph resembles a copy of the sequential one, although with better performance numbers. \textit{(Note that this is not easily visible in the provided plot.)}

We observe similar performance drops in Fig. 5b, where measurements are plotted for a more fine-grained level of parallelism achieved by having each worker thread spawn a processing thread for each cell in its chunk. We observe performance radically dropping into the region of the baseline after a certain number of iterations which seems to depend on the number of cores a thread runs on.

Interestingly, when we tried \texttt{place\_local} creation of cell-processing threads on the worker level (i.e., spawning a thread for each cell), deadlock occurs exactly at the iteration where we observe these performance drops as in 5b.

An example of this can be observed in Fig 7, where the (green) graph for the place-local version of the code breaks off at about iteration 250. At the same point however, the performance for the place-default version of the code has a considerable performance drop. These two events might have a common cause, although we do not
have much insight into this issue.

Thread families can be restricted in the number of threads they spawn on each available core in the place allocated to them. It would make sense to argue that — apart from preventing resource deadlock — limiting the number of threads per core could reduce the likelihood of cache misses, but since the RBM architecture used in our testing only has a tiny cache of 1KB, cache misses are basically unavoidable. Apart from this, throttling the number of threads would only induce overhead. It is shown in figure 6a that indeed, choosing different block sizes in a two core configuration decreases performance, as long as the limit is lower than the actual thread count per core. Obviously, adding more cores undermines the limiting power of the block size. As shown in 6a, once more “free slots” are available then threads are active, the parameter does simply not affect performance.

5 Conclusions

We have devised a parallized implementation of Game of Life and were amazed by the ease at which we were able to compose it using the syntactic sugar SL adds to the C language. However easy it was to design and implement our program, all the more difficulty arose from using the simulation environment; inexplicable deadlocks or exceptions were encountered, with little hint of what and were a programming error was present. Our benchmarks also show some as of yet unexplained performance fluctuations.

Disregarding the erratic performance fluctuations, the benchmarks show some promising speedups. Our solution seems to scale well when increasing the number of cores.
References

