Progress Report: Final Project
CS 442/542, Fall 2023

Kelsey Knowlson
Victoria Lien
Bryce Palmer
Hank Wikle

November 17, 2023

1 Overview

The project code can be found on Lobogit.

2 Python Prototype

We began by creating a simple serial Python prototype. The code represents the entire domain as a boolean vector, with `True` values corresponding to alive cells, and `False` values corresponding to dead cells. The core of the update functionality is a matrix-vector multiplication, in which the state vector is multiplied by a stencil matrix, that computes the number of neighbors for each cell. This approach is inspired by 2-dimensional finite differencing methods discussed in Dr. Jacob Schroder's Scientific Computing course [1], and this initial prototype uses a modified version of code provided by Dr. Schroder, originally developed for the PyAMG project [2], to generate the stencil matrix.

3 Stencil Matrix Generation

We found that the algorithm used to compute the stencil matrix was more complex than was necessary for our use case, so our next task was simplifying the logic and translating it into C.

4 Sparse Matrix and Vector Implementation

We implemented three different sparse data structrues in C for use in the project:

4.1 Sparse Boolean Matrix

This data structure implements the COO sparse matrix format, with the modification that no values are stored; if a row and column exist in the structure, its value is assumed to be `True`, otherwise it is assumed to be `False`. The structure begins with a set capacity and uses table doubling [3] to increase capacity as necessary when new entries are added. The basic

structure appears below:

```
struct COOBooleanMatrix {
    unsigned* rows;
    unsigned* cols;
    unsigned num_nonzero;
    unsigned capacity;
};
```

4.2 Sparse Boolean Vector

The second data structure, which represents the state vector, is a vector of boolean values. This sparse boolean vector was implemented in much the same way as described above, with the exception that instead of storing separate indices for rows and columns, it stores a single set of indices. The structure is reproduced below:

```
struct SparseBooleanVector {
    unsigned* indices;
    unsigned num_nonzero;
    unsigned capacity;
};
```

4.3 Sparse `char` Vector

Finally, we implemented a sparse vector of `char`, which is intended to be used for the intermediate step of counting neighbors. The decision to use `char` rather than another integral type is motivated by the desire to conserve space in memory, together with the observation that since a cell can have at most 8 neighbors, there is no need to represent values larger than 8. This data structure is implemented similarly to the sparse boolean vector, with an additional member that stores the value of each element. The structure appears below:

```
struct SparseCharVector {
    unsigned* indices;
    unsigned char* values;
```

```
    unsigned num_nonzero;
    unsigned capacity;
};
```

5 Preliminary CARC Testing

Initial testing of the serial implementation revealed memory allocation was the fundamental obstacle for large scale experimentation. The naive implementation without any sparse encoding could allocate at least 10,000,000,000 bytes while an order beyond that incurred a segmentation fault. Performance declined immensely with large allocations. But fortunately, all serial implementations are functional.

6 Significant Changes from Initial Proposal

The project remains largely unchanged from the initial proposal, with one exception: due in part to time constraints, and due in part to the challenges of one-to-one translation between C and Python code, we have decided to focus exlusively on the C implementation of the project.

In addition, we have narrowed our initial pool of experiments down to three:

1. Performance analysis of matrix-vector multiplication vs. nested for loop for updating state
2. Analysis of communication time performance as matrix partition size changes
3. Performance analysis of threaded vs. MPI implementations

7 Next Steps

• Integrate already written data structures and methods related to sparse matrix multiplication into the existing C code.
• Determine best partitioning for each matrix for communication.
• Start the process of implementing parallelism within the serial code for the

naive and sparse matrix implementations.
• Implement threading in the existing code as an additional comparison between threading and MPI.
• Add timings to both the serial and parallel versions of the code to determine calculation and communication differences between the implementations.

The current C code for our Game of Life implementation is written serially but does not include the method of using sparse matrix communication, which is the goal. We plan to integrate that code into the current code then partition the matrices and utilize parallelism.

Additionally, we need to add methods of timing the outcomes and measuring the speed of each of our implementations for comparison, and potentially add threading to the code if we have enough time.

8 Open Questions

• Should we simplify our plans and focus more on a single topic rather than multiple experiments?
• If so, which of the topics seems to be the most fruitful for the project?
• We are encountering a malloc issue for large large blocks of memory and would like input on how to address this problem.
• When we are working with domains with large memory demands that exceed the capacity of a single node, how would we set up a problem such that each node has the appropriate section of domain?

References

[1] J. Schroder, *Lecture 15: Two-dimensional finite differencing*, University lecture, MATH/CS 471: Introduction to Scientific Computing, University of New Mexico, Oct. 10, 2023.

[2] N. Bell, L. N. Olson, J. Schroder, and B. Southworth, "PyAMG: Algebraic multigrid solvers in python," *Journal of Open Source Software*, vol. 8, no. 87, p. 5495, 2023. DOI: 10.21105/joss.05495. [Online]. Available: https://doi.org/10.21105/joss.05495.

[3] E. Demaine, S. Devadas, and N. Lynch, *Lecture 5: Amortization*, University lecture notes, 6.046J: Design and Analysis of Algorithms, Massachusetts Institute of Technology, 2015. [Online]. Available: https://ocw.mit.edu/courses/6-046j-design-and-analysis-of-algorithms-spring-2015/13e2c7d165259712327af0af312a068e_MIT6_046JS15_lec05.pdf (visited on 11/17/2023).