PARALLELIZING CONWAY'S GAME OF LIFE

Eric Gentry

1. INTRODUCTION

For this final project, we were given the task of programming a parallel version of Conway's *Game of Life*. The rules to the Game of Life are simple and relatively well known and will not be discussed here (for details, see wikipedia). Our primary challenge was to write and parallelize a Game of Life simulator. Ideally, it should scale well to an arbitrary number of cores.

The code written and used for this project is available at `https://github.com/egentry/Game_of_Life`. Section 2 will go into details about the algorithmic choices behind this project. Section 3 will evaluate the algorithmic design choices in terms of theoretical models and empirical performance data.

2. PCAM

This section seeks to explain the design choices under the "Partitioning – Communication – Agglomeration – Mapping" framework (*PCAM*).

2.1. *Partitioning*

At the atomic level, every location on a 2D grid needs to be evolved to a new state, using information from the old states of the surrounding 8 neighboring cells. Within a timestep, any cell can be updated in any order. This update step, which is done for each cell location, is the finest level of partitioning appropriate for this problem.

2.2. *Communication*

Each of our tasks requires knowledge of the old states of the 8 surrounding neighbors; local channels to each of those neighbors are required for this information. Within a timestep, this communication can take place in any order. Extra care needs to be taken in adhering to periodic boundary conditions — no boundary conditions need to be enforce, so long as channels are created between appropriate tasks on the boundaries of the overall computational domain. The only global communication required is for Input/Output tasks.

2.3. *Agglomeration*

egentry@ucsc.edu

A typical node can handle updating far more than a single grid point. By agglomerating we can increase the amount of cells we can evolve for a fixed number of processors, and by agglomerating well we can minimize the communication required per computation.

The communication in this problem is entirely local, which means only the boundaries of a computational domain need to be communicated. In order to minimize the communication (related to the size of the boundary) per computation (related to the area of the domain), we effectively have a "surface area-to-volume" problem. Since this task-channel structure is analogous to fixed grid hydrodynamical solvers, we know the best way to minimize "surface area-to-volume" is to decompose in the dimensionality of the problem. We have a 2-dimensional problem, so we want to decompose and agglomerate tasks in 2-dimensions. For ideal performance, grid points should be agglomerated into rectangles within the computational domain.

Since this problem is a fixed grid, with fixed computation per grid point, we also expect the loads on each processor to be relatively static (with the exception being I/O related processors). Given this static load, we do not expect load re-balancing to necessary.

2.4. *Mapping*

In short, much of the mapping will be left to MPI and the batch system on Grape. Using the batch system we will requires a number of nodes, and will use all the cores on each node. This will minimize the inter-node communication, which will be slower than intra-node communication. Alternatively, we could have used a hybrid model, using `MPI` between nodes, and `OpenMP` within nodes, but for simplicity we have kept a single programming model.

Once allocated nodes and cores, the only further mapping might happen when we create a Cartesian virtual communication topology within `MPI`. We will allow `MPI_Cart_create` to reorder processor ranks with the knowledge of how we plan to communicate, but this only allows for the opportunity of re-mapping cores. Re-mapping is not guaranteed.

3. PERFORMANCE MODEL

In this section, we will construct a *wall-clock* runtime model. We'll then make and test predictions regarding the weak scaling of our model.

3.1. *Theory*

As discussed in in Section 2, N tasks are agglomerated in rectangular sub-domains of size N_x by N_y, with n_{gc} guard cells of padding in each dimension. For simplicity, we'll enforce $N_x = N_y = \sqrt{N}$. The number of "true" cells in each sub-domain is then:

$$N_{\text{true}} = (N_x - n_{gc})(N_y - n_{gc}) = (\sqrt{N} - n_{gc})^2 \quad (1)$$

but in order to make full use of the guard cells (when $n_{gc} > 1$), we'll need to do a variable number of calculations each timestep. If all of our guard cells are accurate, then we can do $(\sqrt{N} - 1)^2$ updates; the next time step we can do $(\sqrt{N} - 2)^2$ time steps, until we are left with just the "true cells" where we do $(\sqrt{N} - n_{gc})^2$ updates. At that point we will need to communicate with nearby cells to refresh our guard cell information. But before thinking about communication, let's find the average time spent updating cells. The average number of cells updated per cycle is:

$$\langle N \rangle_{\text{update}} = \frac{1}{n_{gc}} \sum_{i=1}^{n_{gc}} (N_x - i)(N_y - i) \quad (2)$$

$$\langle N \rangle_{\text{update}} = \frac{1 - 6\sqrt{N} + 6N + 3n_{gc} - 6\sqrt{N}n_{gc} + 2n_{gc}^2}{6} \quad (3)$$

This will result in an average computational time:

$$T_{\text{comp}} = t_{\text{update}} \langle N \rangle_{\text{update}} \quad (4)$$

(We will use the convention T_i referring to the *total* wall-clock time spent doing type of action i within a single cycle. Similarly, t_i will be the time per task within the overall action.)

Every n_{gc} time steps, we will need to communicate between adjacent sub-domains to update guard cells. For communication we expect both start-up costs t_{start} and length-dependent costs t_{size}. Each sub-domain will need to communicate n_{gc} rows or columns in 4 directions. This will incur an average $4t_{\text{start}}$ per time step, as well as the length-dependent costs $2(N_x + N_y)t_{\text{size}}$, where t_{size} is the cost per variable communicated, and $2(N_x + N_y)$ is the perimeter (total size of the messages communicated). (Time per communication will be expected to be n_{gc} times longer, but will only need to run once every n_{gc} time steps.) For simplicity, we will assume the constants t_{start} and t_{size} are the same between all processors, even though this is not true for the physical network topology.

This brings our average total time cost to:

$$T = T_{\text{comm}} + T_{\text{comp}} \quad (5)$$

$$= 4t_{\text{start}} + 4\sqrt{N}t_{\text{size}} + t_{\text{update}}\langle N \rangle_{\text{update}} \quad (6)$$

$$= 4t_{\text{start}} + 4\sqrt{N}t_{\text{size}} \quad (7)$$

$$+ t_{\text{update}} \frac{1 - 6\sqrt{N} + 6N + 3n_{gc} - 6\sqrt{N}n_{gc} + 2n_{gc}^2}{6}$$

This is ugly, but it's worth pointing out a few key features. At no point does our time model depend on the number of processors, P. That is because we have formulated our model in a way that makes a weak scaling test natural. (It also simplifies the design, as agglomeration need not depend on the number of processors.) Secondly, for fixed N, we expect T to drop with increasing n_{gc}, and we expect it to keep dropping for all $n_{gc} < \sqrt{N}$. This indicates that our model is insufficient for yielding useful information about how to optimize the choice of n_{gc}. If we set $n_{gc} = \sqrt{N}$ we get the minimal T, but we will just be running identical copies of the same domain on every processor — we will have the serial case. For a more meaningful result, we should consider the total *wall-clock* time per "true" cell on each processor, which we will term τ

$$\tau \equiv \frac{T}{N_{\text{true}}} \quad (8)$$

$$= \frac{1}{(\sqrt{N} - n_{gc})^2} \left(4t_{\text{start}} + 4\sqrt{N}t_{\text{size}} \right. \quad (9)$$

$$\left. + t_{\text{update}} \frac{1 - 6\sqrt{N} + 6N + 3n_{gc} - 6\sqrt{N}n_{gc} + 2n_{gc}^2}{6} \right)$$

Now we have our theoretical model, and if given constants for $t_{\text{start}}, t_{\text{size}}, t_{\text{update}}$ we can either run a weak scaling test, and if that is successful, use our model to choose an optimal number for n_{gc}.

3.2. *Verification*

Using my code, I ran tests on GRAPE to determine the constants:

$$t_{\text{start}} = 3.73 \cdot 10^{-5} \text{ s} \quad (10)$$

$$t_{\text{size}} = 3.56 \cdot 10^{-9} \text{ s / short} \quad (11)$$

$$t_{\text{update}} = 6.60 \cdot 10^{-9} \text{ s / timestep / true grid point} \quad (12)$$

(These constants were all determined using multiple processors on one node. Only using one node will introduce errors which we will discuss later.)

3.2.1. *Weak Scaling*

On GRAPE, I ran weak scaling test ranging from 1 processor to 64 processors (8 nodes, of 8 processors per node), running for 100 timesteps, with
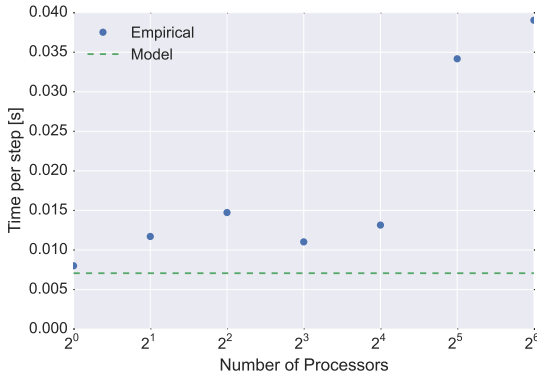
Figure 1. Weak scaling tests, ranging from 1 processor to 64 processors. Equation 9 was used as our performance model, with empirically determined constants.

a 1024×1024 grid on each process with 1 guard cell of padding on each edge (i.e. $(1024 - 2)^2$ true grid points). The results can be seen in Figure 1. For a small number of processors (≤ 8), the results are relatively close to the prediction made by our performance model. Unfortunately, once we use greater than 8 processors (requiring multiple nodes) we find that performance begins to suffer significantly. This is likely due to the additional cost of communicating between physical nodes, which is more costly than communicating between processors on the same node.

3.2.2. *Guard Cell Scaling*

As noted above, I was interested in how the number of guard cells affected performance. Using 64 processors (8 nodes of 8 processors each), I ran tests with 1 through 64 guard cells, running 100 evolution steps. Results can be seen in Figure 2. The data are significantly worse than the performance model we constructed in Equation 9. This worse performance appears to be entirely due to the effect of using multiple node, while benchmarking on only one node (compare the benchmarked time for 64 processors in Figure 1 to the times shown in Figure 2).

It should be noted that our performance models, once the appropriate constants were determined, suggested very little change in runtime as a function of number of guard cells. This is unsurprisingly, because our communication times are still dominated

by the startup, even for a 1024^2 grid. Furthermore, I did not optimally package the sending of guard rows / columns into a single communication. Instead, I made a new send/receive call for each guard row / column. In the future, proper buffering of multiple guard rows / columns would be more appropriate.

4. DISCUSSION
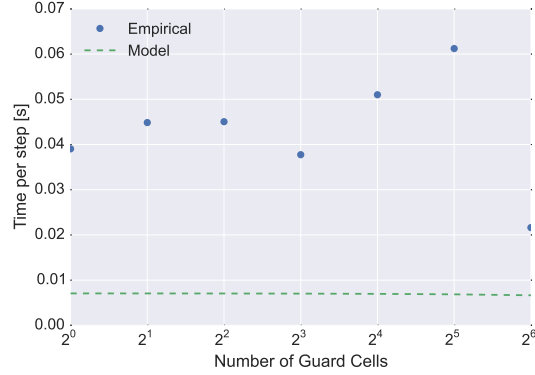
Figure 2. Weak scaling tests, ranging from 1 processor to 64 processors.

I was successful in constructing a parallel solver for Conway's Game of Life, with a 2D domain decomposition. For a small number of processors, the runtime is dominated by processing time and is accurately predicted by our performance model. As we use use more nodes, we find that communication costs begin to rise, which was not predicted by my model. A better performance model might seperately include *intra-node* communication costs and *inter-node* communication costs.

The inter-node communication costs lead to an interest possibility with regards to our guard cell scaling. Our performance model would predict that the number of guard cells shouldn't affect the runtime significantly, since the model predicted that computational costs would dominate the runtime. But, since communication costs dominate the runtime for large numbers of processors, a greater number of guard cells might be a way to minimize these costs. Unfortunately, my guard cells were not sufficiently well implemented to test this possibility; it remains just a hypothesis.