Manifest Destiny: A Simulation of American Westward Expansion

Matthew Pisano
pisanm2@rpi.edu
Rensselaer Polytechnic Institute
Troy, New York, USA

Gunnar Eastman
eastmg2@rpi.edu
Rensselaer Polytechnic Institute
Troy, New York, USA

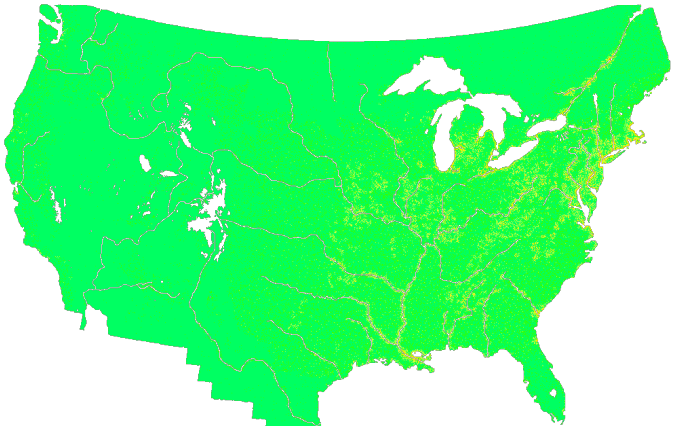

Figure 1: A simulation result ending in the year 1863.

ABSTRACT

In this work we create a tool that simulates the Westward expansion of the United States over the one hundred year period from 1763 to 1863, using geo-spatial attributes to identify which locations on a map are fit for settlement and then modeling the growth of all settlements. Our tool, Manifest Destiny, works in parallel, using MPI I/O for reading in large amounts of geo-spatial and environmental data and writing out the population distribution over time. Representing the American continent as a 2D grid, each MPI rank processes one column, with every CUDA thread calculating one or more cells. The growth of a cell depends on factors including elevation, precipitation, temperature, and local resources and water features. This simulation both produces accurate population estimates for the time period and scales easily to more processors and higher simulation resolutions. Additionally, we perform both strong and weak scaling tests on our implementation to measure its effectiveness at utilizing high-performance computing resources.

Parallel Computing '24, April 23, 2024, Troy, NY

KEYWORDS

Population, Simulation, Cellular Automata, Parallel I/O, MPI, CUDA

ACM Reference Format:
Matthew Pisano and Gunnar Eastman. . Manifest Destiny: A Simulation of American Westward Expansion. In *Proceedings of Parallel Computing '24.* ACM, New York, NY, USA, 14 pages.

1 INTRODUCTION

The settlement of the American west was one of the longest and more impactful population shifts in the history of the United States. Beginning soon after independence in 1783, American settlers quickly pushed beyond the boundaries set by the British crown in 1763 and into the Appalachians. Another important factor in western expansion was the extensive network of French trading posts along the Mississippi river and into the Great Lakes region. While the Spanish did have de-facto control of the West coast, areas outside of the Mexican heartland remained underdeveloped. Using this historical knowledge, along with geo-spatial and environmental data, we have created an in-depth, fast, and accurate simulation of this expansion.

We model this expansion after the year 1763, using a combination of census data and population estimates from the time, along with elevation, surface gradient, maritime trade,

precipitation, temperature, biome, and natural resource data. In order to map these resources onto a space that we can simulate, we split the North American continent [1] into discrete squares of about nine square miles each; although this resolution can be increased up to only one square mile for performance testing. We then mapped our gathered datasets onto these over 619,000 individual cells. In our simulation, each cell is stored as an 8D vector that contains all of the population, elevation, etc. data for that cell. To reduce overhead from using multiple arrays, we flatten our grid of cells into a 1D array, using the column and cell width to remap each index back to its original 2D coordinates. In an effort to save space and reduce calculation times, we ensure that each cell dimension can be stored within one *unsigned short integer*.

After the data has been fully processed, it can be fit into a file of between 9MB and 81MB, depending on the resolution of the data. In order to efficiently process these relatively large files, we utilize MPI parallel I/O to share the task of loading in and writing out data among each process. The usage of MPI also allows us to split the calculations between each process, or rank. In particular, each process gets r/n columns of data, where r is the number of cells per row of data and n is the number of MPI processes. If the number of MPI ranks does not evenly divide the row size of the data, the last row gets the remainder columns. Finally, we use CUDA to process each cell in parallel. For lower resolutions, there if ample room for each thread to only calculate once cell per iteration. This allows us to significantly speed up our simulation, reducing execution tome by an order of magnitude.

2 RELATED WORK

The field of population simulations has become more effective and insightful in the past decades as more powerful computers and methods of parallelization have become available. These examples of population simulations may be focused on human population distribution, the growth of non-human animal populations, or more simple cellular automata simulations. Here, we have selected three of the most relevant to our work.

2.1 Population simulations

Life-Stage Simulation Analysis [Wisdom et al. 2000] uses a matrix model approach to simulate population rate for growth for the purposes of species conservation planning. Differing "vital rates" are randomly sampled from a distribution, such as species fertility, survival rates, growth rates, etc. These different attributes are then used to calculate their effects on population growth rates through the use of matrix algebra. This paper, similarly to our own, uses relevant attributes to calculate approximate growth rates. While their matrix-based approach yields results that are more predictable from the distribution of attributes, it is complex and computationally expensive. While we initially experimented with a more complex method of processing a cell's attributes, we opted for a more straightforward strategy. For our approach, we limit our "cell value" calculations to a short series of additions, based on the properties of each cell. Even though our method is more simplistic, it remains accurate while being much cheaper computationally.

Similarly to Manifest Destiny, other human-centric methods of population simulation rely more on cellular automata models. One such example is the study of Italian urban growth by Cosentino et al. [Cosentino et al. 2018]. This models how urban areas may grow in Italy up to the year 2035. Unlike our approach, this paper uses an existing cellular automata framework, *FUTURES*. This framework helps to allow for a more higher-level interface when interacting with the variables of population growth. Specifically, this framework is comprised of three sub-models: the future potential of an area, the demand for moving to that area, and its internal growth. In contrast, our simulation uses a wide variety of different attributes for each cell and each are controlled individually within our source code. This has the drawback of small changes to each value, say the overall growth factor, can have wildly differing results. However, this has the advantage of giving us much more granular control of how our simulation evolves over time. By fine-tuning these values, we are able to create very accurate results for our simulation's low complexity.

Another automata-based model is presented within the paper *Numerical Simulation of Population Distribution in China* [Uue et al. 2003]. This uses a very simular approach to ours. Here, the authors split the Chinese mainland into a grid of equal cells. The value of each cell is determined by the population in the year 2000 within that area. Other data is also included at each cell. This includes the elevation, the presence of a rail network, biome, and precipitation. The primary difference between this simulation and ours is the attributes that each cell uses to calculate its growth. While many of these attributes are shared, this paper concentrates more on modern infrastructure, while ours concentrates more on geography and industrial minerals.

2.2 Factors of Population Growth

While some works within the realm of population simulation concentrate on the simulations themselves, others concentrate more on the attributes that best determine population growth. A review by Sibly and Hone [Sibly and Hone 2002] exemplifies this trend. The paper primarily concentrates on what natural factors most impact the population growth of

[1] Excluding most of Mexico and the Northern regions of Canada

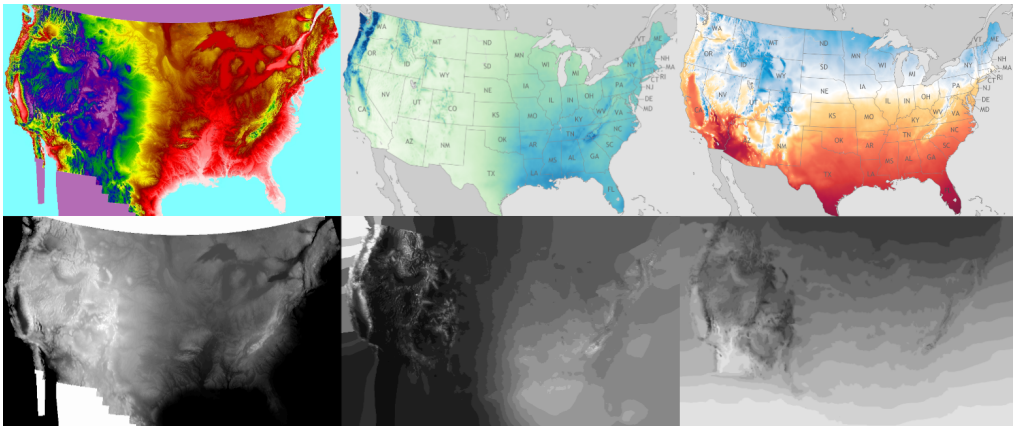

Figure 2: Images corresponding to our three pre-processed and discretized attributes. (Top) Our raw samples for elevation, precipitation, and temperature (left to right). (Bottom) Our pre-processed and discretized data for elevation, precipitation, and temperature (left to right). Darker colors represent lower values of elevation, precipitation, and temperature and lighter colors represent higher values.

different species. In addition to common factors, such as temperature and the presence of natural resources, the authors also stress the importance of a species' natural growth rate. This could be the average seasonal ring size in Birch trees or the number of offspring per season in geese or elk. We model this critical factor with the variable growth rates of our cities. Once a cell is settled, it begins to grow on its own. This proportional growth rate is set for all small towns. As the town grows into a city, the proportional growth rate shrinks to model the maturing of the settlement. Indeed, our results appear to align closely with this study, as these growth factors are the single most impactful value in our simulation, and the ones we spent the most time fine-tuning.

Another study from Lutz and Qiang [Lutz and Qiang 2002] has a much more specific scope: the growth of the human population over the 20^{th} century. According to this study, one of the most important determinants of growth in industrial societies is the population density of the area. This effects a wide variety of factors, including the birth rate and death rate. They show that areas with very high population density have a depressing effect on the growth rate of a population, while relatively unpopulated areas have an uplifting effect. The principle presented in this paper is central to our method of calculating population growth. As previously mentioned, the growth of a settlement shrinks in discrete steps along with its population. Since each cell has a fixed area, the population and population density are always proportional. In addition to the growth rate, our simulation also takes population density into account when modeling inter-cell migration. By taking both of these factors into account, we are able to create realistic results.

2.3 History of the American West

Finally, information on how the young United states expanded between the late 1700s and the mid 1800s is critical for ensuring our simulation is as grounded in reality as possible. A paper by Vandenbroucke [Vandenbroucke 2008] helps to provide some of the valuable information. The first portion of this paper introduces important information, such as the area of "improved" or settled land between 1770 and 1910. This trend starts slowly in the beginning, but experiences sharp jumps in rates as the 19^{th} century progresses, from less than one million acres of land to nearly 350 million. We model this increasing rate of Westward expansion in our simulation by growing the probability that a cell will become explored by the square of the current iteration. This produces rates of expansion similar to those found within this paper's data and within its own simulations. Another paper, [Kinnahan 2008] acts as an important grounding source for our expansion data. We reference its maps on territorial expansion and county population data to cross reference our results with itself and the numerous other population sources that we utilize. Even though we do not directly model the contributions of Native American tribes to our overall population model, analyzing their movements over time can be very useful for tracking those of American settlers. [Bowes 2016] helps to supply this knowledge. Using the information and map within chapter 5, we can observe when and where these tribes were, often forcibly, migrated from their original homeland to reservations. The time at which these removals happen coincides closely to the times in which these areas became more popular with settlers and homesteaders on their push Westward.

3 IMPLEMENTATION

We divided the implementation of Manifest Destiny into several phases. The first phase was to collect and pre-process all of the data that is relevant to our population simulation into a form that is easily readable by our *C* code. Next, we designed, tested, and fine-tuned our population growth algorithm. This is the core of our project and has the biggest impact on the overall program output. Concurrently with the work on our algorithm, we built up an MPI framework that could execute our simulation in parallel, along with efficiently loading and saving data using parallel I/O. The final portion of our implementation was to include CUDA. This was not built into the MPI phase as the program would be easily to work with and debug if it could run on our own hardware while making small tweaks to the population growth algorithm.

3.1 Data Pre-Processing

As large amounts of geo-tagged JSON or binary data is both difficult to work with and difficult to visualize, we have chosen to represent the attributes of each cell as a series of images. Although all of the attributes that we measure are available in image form, not all have the same resolution, range, and map-projection. To accommodate this diversity, we have chosen a target map projection, *Universal Transverse Mercator* (UTM), and a resolution, 994x623. This gives us an image where each pixel is almost exactly 9 square miles. UTM is a resolution commonly used by the United States Geological Survey (USGS) and our resolution allows us to have maximal detail while retaining a compatible level of detail with other image-based data sources. Some of our images, namely elevation [Joe Grim [n. d.]] and biomes [MapResources.com [n. d.]], along with precipitation and temperature [Climate.gov 2021], already had a UTM projection (see figure 2). For these, we just had to shrink and crop the image to match our dimensions. For others, like rivers and lakes [United States Geological Survey 2023], we used a combination of USGS data and reference images. We compiled our resource data for coal [Mineral Information Institute 1993], iron [James A. Bowen 1901], and gold [Wentz et al. 2014; Western Mining History [n. d.]] from a variety of different sources (see figure 3) [2].

The final component of our data pre-processing was gathering data about our starting date, 1763. We have compiled starting population data from holistic estimates of colonial populations, census data from individual cities, and estimates of territorial expansion of the time (see figure 4). Our goal when creating this starting distribution was to keep its total population as accurate as possible to the ground truth value [Lumen Learning [n. d.]]. In addition to this, we also utilized a map of the approximate extent of colonial settlement in 1763 [United States Office of the Historian [n. d.]]. To avoid harsh lines and deterministic growth patterns, we have added noise to our data that blurs the lines of territorial extent. In concrete terms, we have randomly selected cells with very low population, removed that population, and distributed that lost population to nearby unpopulated cell. We have observed that this produces much more natural population distributions as our initial state would behave just as future settled portions of the map do. In addition to this noise, we also plotted the most populated French forts and trading posts that existed at the time [Hall et al. 1920]. This allows our population to grow in other regions that are not just centered on the American East coast, as these forts historically contributed to many modern cities along the Mississippi and around the Great Lakes. Finally, we used several cities as markers to both cross-reference our other data and to monitor the progress of specific city growth over the course of our simulation. The included New York City [Demographia 2001; Joseph A. Gornail, Steven D. Garcia [n. d.]], Boston [Lawrence W. Kennedy 1997], the French-Canadian province of Quebec [Statistics Canada 2010], and Baltimore [Sidney Redner 1998].

Finally, we further discretized our data that was presented as continuous, such as temperature and precipitation (see figure 2). This resulted in a set of images that were either greyscaled or colored using a small number of colors. This process allowed us to compile all of our data into images, visually fix any parsing errors, and to easily make any updates to our population estimates.

Elevation is measured in feet above sea level, as one of 46 values ranging from 0 to 14,200. Average yearly temperature

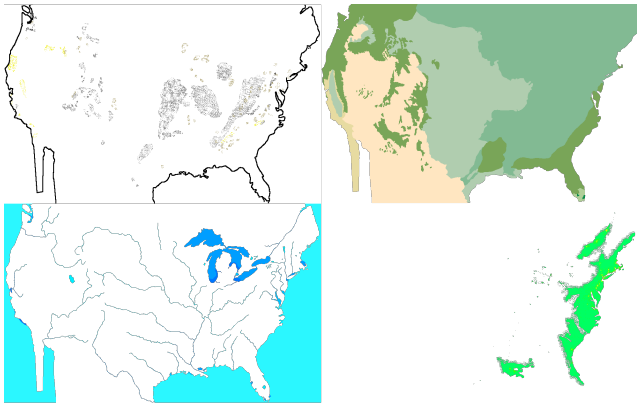

Figure 3: Images corresponding to our four discrete attributes. (Top) The resource and biome data images (left to right). (Bottom) The water/maritime trade data image and the full 1763 population data image (left to right).

[2]The black outline around the resource data has been added for visualization purposes only and is not a part of our input data.

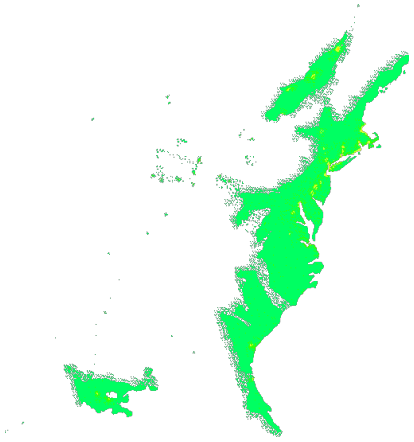

Figure 4: The East coast of the starting population distribution for 1763. More yellow colors indicate a higher cell population and uncolored areas represent unexplored cells.

is measured in Fahrenheit, stored as one of 16 values from 20 to 80, rounded to the nearest multiple of four. Precipitation is measured in inches, and is simply stored as the nearest multiple of five from 0 to 80, for a total of 16 values. Starting population, as represented by the map, is stored as the population density of that cell, measured in people per square mile, which is between 0 and 3,000, but the maps for 1763 mostly stays below a population density of 400.

Water features were initially stored as a binary: a pixel on the map was either deemed as underwater or not, but we decided to expand it to make the distinction between rivers or lakes and the ocean. Rivers and lakes are further separated into "Low", "Medium", "High", or "Best", to ensure that massive trade hubs such as the Mississippi River is registered as a better location for settlement than The Great Salt Lake, in Utah. Each pixel in the mineral resource map is either marked as not having a resource, being a source of iron, being a source of gold, or having a low, medium, or high amount of coal.

We identified seven biomes that comprise the majority of the United States: "Coniferous Forest", which covers most of the northeast; "Deciduous Forest", which appears in the southern parts of the East Coast and the northern parts of the West Coast; "Grassland", which covers the Great Plains; "Desert", which represents most of the Rocky Mountains area; "Shrubland", which only appears in California and western Arizona ; and "Marshland", and "Tropical Forest", which are only used in the southernmost parts of Florida. As such, each pixel in the biome map has one of eight values: one of the biomes listed above, or "Ocean".

These seven, pre-processed images [3] were then turned into a binary object using the *NumPy* library. This $x \times y \times 8$ array is then saved as a *NumPy* file, which can be loaded by Manifest Destiny. Importantly, since the original array has been flattened, there is no intrinsic way to determine how our program should step through the series of bytes that it loads. This must be known in order to properly split it by cell boundaries and determine where a cell's neighbors are located within the flattened array. We solved this issue by appending six bytes onto the end of each file. These bytes encode the length of one row, the length of one column, and the length or one cell, with each taking up two bytes. This data can easily be read in at program start up and used to populate a *dimension struct* that is passed into the simulation code.

3.2 Simulation Algorithm

Our approach to solving the problem of simulating Westward expansion can best be described as a cellular automata approach. This closely mirrors the approach of [Uue et al. 2003] in their population simulation of China. As mentioned in previous sections, the growth of a cell's population is determined by the eight attributes of the cell in question, along with its neighboring cells. Similarly to the general population growth described in [Sibly and Hone 2002], we take into account the temperature, precipitation, and biome attributes that each cell has. We have tuned the effect of these values to very closely mirror the actual population growth of the period. Another important component of our simulation is grounding it with real timescales. To ensure that our simulation has a high temporal resolution, we model one year's worth of growth with ten iterations. Therefore, 100 years of simulation starting in 1763, to the date 1863, would take 1000 iterations. We also make use of random noise within our simulation to model the complex effects of variables that our program does not account for. We apply this random "jitter" to a variety of factors, including the overall value of a cell or the growth factor of a cell at a given iteration.

To avoid race conditions within our parallel program, we make a copy of our data array. While this does double our program's memory footprint, it has the significant benefit of allowing us to separate the result of our last iteration from the result currently being computed at each iteration. Pointers to these arrays are swapped after each iteration. This ensures that the freshly computed values will now serve as the input to the next iteration and the result from two iterations in the past can be safely overwritten with new data. As this swap happens at the end of each iteration, the data pointed to by our final array always has the result from the final iteration.

[3] We do not have a map dedicated to gradient as it can easily be pre-calculated from the differences between neighboring pixels from the elevation map.

3.2.1 Cell Values. The first step of our simulation is to calculate the "value" that each cell holds on a scale from 0 to 100 points (with the most promising cells being allowed to overflow up to 200). This value influences when a cell is explored, when it is settled, and how fast it grows. To model the dampening effects that harsh climates or terrains pose to population growth, our simulation never settles cells that are underwater, above an elevation of 10,000 feet, or have a gradient of over 60%. To model the effects of nearby water resources on growth, we gather information from nearby cells within a radius of 2 or more from the target cell. The more, and better water surrounding a cell, the more favorable it is for settlement and population growth. We also take into account the abundance of mineral resources at a particular cell, with coal being the least valuable, iron being more valuable, and gold being the most valuable resource. To model the effects of terrain on population growth, we scale the value of a cell down linearly as the elevation and gradient of that cell decreases. This results in areas that are low or flat (preferably both), being settled first and growing faster. Temperature, precipitation, and biome all give a flat bonus or penalty to each cell, depending on their values. For temperature, a flat bonus of 15 points is given to cells with a yearly average temperature between $45°F$ and $70°F$. for precipitation, a flat bonus of 10 is given to cells with a yearly average rainfall of over 30 inches per year. A flat penalty of -10 points is given to cells with desert-like conditions: a rainfall of less than 10 inches per year. If the cell is a swamp or a desert, a flat penalty is applied: -10 for swamps and -15 for deserts. All bonuses or penalties are added to, or subtracted from, the cell value.

Algorithm 1 Explore Chance Computation and Population Update

Require: $pop = 0$
1: $explore\_chance \leftarrow 0.002 + \frac{iteration^2}{8000000.0}$ ▷ Scale by year
2: $scaled\_value \leftarrow \left(\frac{cell\_value}{100.0} + 3\right)$ ▷ Ensure min value of 3
3: $scaled\_chance \leftarrow explore\_chance \times scaled\_value \times MAX\_JITTER$
4: **if** $nearby\_population.count > 0$ **then**
5: **if** $jitter < scaled\_chance$ **then**
6: **return** MIN_POP
7: **else**
8: **return** 0
9: **end if**
10: **else**
11: **return** 0
12: **end if**

3.2.2 Cell Population. The changes in a cell's population are divided into three phases: *exploration*, *settlement*, and *Growth*. A cell is first explored, when its current population is zero. This happens randomly as long as the cell is near other cells that have some permanent population. Once explored, we give that cell a population density of two people per square mile. This minimum population density is turned into a concrete population for the cell by multiplying it by the number of square miles within the cell's area (see algorithm 1).

An explored cell becomes settled once a city spawns onto it. This is more complex than the exploration algorithm. If a cell is not near to any existing cities, the population of that cell suddenly grows to tens of people per square mile. If a cell is nearby existing cities, its population may randomly grow to a small fraction of its neighboring city. This second selection helps to model city sprawl. Finally, a cell's population may jump by a smaller amount to model the settlement of smaller cities and suburbs (see algorithm 2).

Algorithm 2 Population Update and City Settlement

Require: $pop = MIN\_POP$
1: $RSQ = RES\_SCALE\_SQ$ ▷ Scale with higher resolutions
2: $near\_avg \leftarrow near\_avg$ ▷ Average neighbors pop
3: **if** $near\_avg \leq \frac{MIN\_POP}{1.5}$ **then**
4: ▷ If the cell is explored with no nearby cities
5: **if** $jitter < 8$ **then**
6: **return** $8 \times MIN\_POP \times RES\_SCALE\_SQ$ ▷ Settle city
7: **end if**
8: **else if** $jitter < \frac{cell\_value}{7}$ **then** ▷ If the cell is settled with nearby cities, expand the city
9: **if** $near\_avg < 150$ **then**
10: **return** $MIN\_POP + \frac{near\_avg}{3} \times RSQ$
11: **else if** $near\_avg < 300$ **then**
12: **return** $MIN\_POP + \frac{near\_avg}{4} \times RSQ$
13: **else if** $near\_avg < 500$ **then**
14: **return** $MIN\_POP + \frac{near\_avg}{5} \times RSQ$
15: **else**
16: **return** $MIN\_POP + \frac{near\_avg}{8} \times RSQ$
17: **end if**
18: **else if** $jitter > MAX\_JITTER - \frac{cell\_value}{20}$ **then**
19: **return** $1.5 \times MIN\_POP \times RSQ$
20: **end if**

A settled cell may also grow its own population every iteration. A cell may grow based on a combination of its own population and the population of its neighboring cells. Initially, a cell grows by 1.1% per iteration and receives 2% of its average neighboring population as migration. Cells with a higher than locally average population are considered to be city centers. The growth of these cells is doubled initially. All of these growth factors scale downward in steps as a cell's

population, and the population of its neighbors, grows. This is to simulate the rapid expansion of small towns, in comparison to their initial population, and their slow maturing into cities. For example, it is reasonable for a boom town to double its population within a year, but it is much more rare for a large, established city to do the same (see algorithm 3).

Algorithm 3 Population Growth

$near\_avg \leftarrow nearby\_population.avg$ ▷ Average neighbors pop
$neighbor\_bonus \leftarrow \left(\frac{cell\_value}{100.0}\right) \times near\_avg$ ▷ Add neighbor bonus to cell value
$near\_factor \leftarrow scale(neighbor\_bonus, near\_avg, iteration)$
if $pop \times (1 + jitter\_range \times 0.5) > near\_avg$ **then**
    $city\_center\_bonus \leftarrow 2.1$
else
    $city\_center\_bonus \leftarrow 1$
end if
$growth\_factor \leftarrow 0.011 \times city\_center\_bonus$
$growth\_factor \leftarrow scale(growth\_factor, pop, iteration)$
$near\_growth\_factor \leftarrow near\_growth\_factor \times RES\_SCALE$
$growth\_factor \leftarrow \frac{growth\_factor}{RES\_SCALE\_SQ}$
$pop\_growth \leftarrow \left(1 + \left(\frac{cell\_value}{100.0} \times growth\_factor\right)\right) \times pop$
$near\_growth \leftarrow near\_factor \times neighbor\_bonus$
return $pop\_growth + near\_growth$

3.2.3 Caveats. To improve both our simulation's accuracy and its simplicity, we have chosen to only simulate the expansion of (formerly) British and French possessions. This is reflected in our simulation's initial state, as only British colonies and French holdings and forts are modeled. Spanish settlements, the other main European power in the area, and the many different Native American tribes are omitted due to their low direct contribution to American populations. Due to a combination of disease, colonial expansion, and conflict, the native population of the Americas suffered an extreme decline from pre-Colombian populations. By 1800, their numbers had shrunk to only 600,000 [Thornton 1987] and continued to shrink throughout the 19^{th} century. Even though interactions with native tribes had a significant expansion on both American politics and population, their population did not significantly contribute to the overall population of the United States, due to their small populace. The Spanish population of the west coast was primarily confined to California. While many colonies and *pueblos* had been founded in the area, their total population remained low, when compared to the young United States [Chapman 1921].

The environmental data that we use for precipitation and temperature [Climate.gov 2021] show average yearly data from 1991 t0 2020. Due to the effects of climate change and other anthropogenic changes to the environment, the values from our target time period (1763 to 1863) were slightly different. However, the difference does not have a meaningful impact on our simulation. As mentioned in section 3.2.1, the temperature and precipitation have a significant effect on cell values, but this effect is only applied as a flat bonus for simplicity. Since the yearly average temperature for North America has only changed by about $2.9°F$ from 1910 to 2024 [National Oceanic and Atmospheric Administration 2024] and the global average has risen by about $2°F$ since 1850 [Lindsey and Dahlman 2024]. While this temperature is enough to cause significant environmental harm. It has not enough to significantly change the calculated values of our cells. Due to this, the utilization of modern temperature data for our simulation should not cause drastic inaccuracies in our results. We observe this more formally in section 4.

3.3 MPI

The first parallel component that we implemented was the introduction of MPI to our serial code. In our program, MPI serves two purposes. Primarily, it allows us to run our simulation in parallel across multiple ranks. To achieve this, we split our world by multiples of the height of the original image. This becomes the column length of the 2D grid, before it is flattened into the 1D array. If the total number of columns within the original array is not evenly divided by the number of columns, the last rank receives any remaining columns. We determine the length of each column and their total numbers by reading in the row, column, and cell dimensions from the last six bytes of each file (see section 3.1). To ensure that every rank is able to share information with it neighboring ranks, each rank exchanges one buffer with the rank to its West and the rank to its East. These "ghost rows" ensure that all cell calculations within a rank can get relevant data about its neighbors, even if those neighbors lie beyond a rank's boundaries.

The secondary purpose of MPI is to allow for the quick reading of input files and the quick writing of output files. Similarly to processing, each rank is responsible for reading from, and writing to, disjoint sections of files. The distribution of these sections is determined by reading in the last six bytes of the file first and using the dimensional information to recover the original array dimensions. We had encountered an issue with our MPI parallel I/O code that caused our last six bytes of metadata to be duplicated randomly. This was especially common in simulations with a high number of MPI ranks. As this primarily occurred when there were a high number of MPI ranks working at once, we suspected a

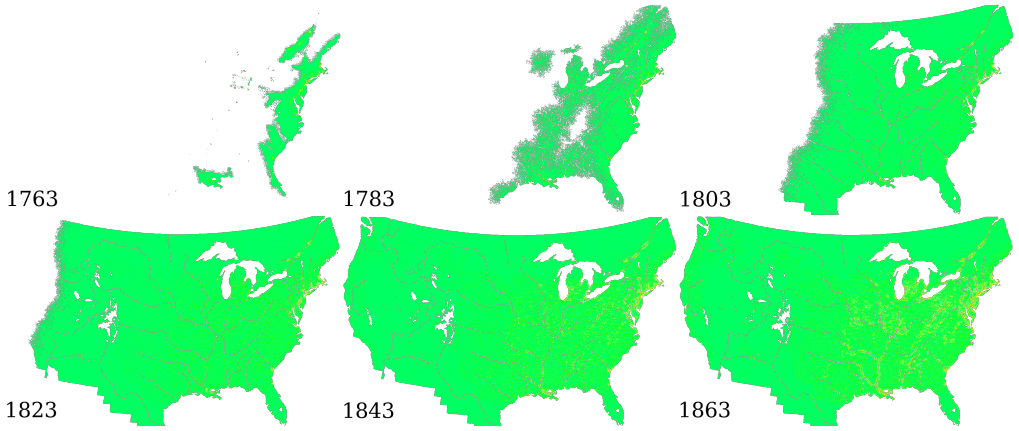

Figure 5: A series of world states at $t = 0$ (Top left) to $t = 1000$ (Bottom right) in steps of 200 iterations each. More yellow colors indicate a higher cell population and uncolored areas represent unexplored cells.

race condition may have been at fault. This was confirmed when we limited our metadata saving to just *rank 0*. Once this restriction was imposed, our race condition disappeared.

3.4 CUDA

In order to serve as an interface between the C-compiled MPI code and the CUDA-compiled code, we use a host function to serve as a launcher for the kernel. This launcher is compiled along with the rest of the CUDA code, but only runs on the host architecture. This results in a function that can be called directly from anywhere within the MPI portion of the program, without having to mix MPI-specific code and CUDA-specific code. Within this launcher function, we call the kernel itself. While the number of threads per block can but up to 1024, we have set it to 256 for our unrestricted evaluations. For our main evaluations we further lower this value to 32 threads to better demonstrate the performance of MPI, without reaching peak performance with only one rank due to the abundance of threads. Each rank utilizes one of the available GPUs. If there are more MPI ranks than GPUs, multiple ranks may share access to each device. Importantly, after each kernel launch, the program waits until all kernels have finished and are ready to synchronize with the rest of the program.

Within each kernel itself, we loop through each cell until all have has their updated populations calculated. Starting at each thread's absolute index, $blockId \times blockDim + threadId$, we take strides of $gridDim \times blockDim$ across the 1D array until the end is reached. This ensures that each cell is processed by a thread, even if there are more cells than total threads available within the program. In this case, each thread processes multiple cells in sequence until it has strode to the end of the array. The index computed by each thread is them multiplied by the dimension of each cell to get the absolute index within the array. This method ensures that each thread begins its computation exactly on a cell boundary and all calculations can use the index as a reference point to access the cell's attributes at their particular offsets. After the new population has been updated (see section 3.2.1), it is saved to the result array at the specific offset that the population attribute is placed at, relative to the cell's beginning.

3.5 Compilation

As this project involves both MPI and CUDA components, it cannot be easily compiled within a single command. To ensure that both the syntax and extra instructions do not clash, we compile our program in several stages. The first stage involves compiling all non-CUDA files into object files using the *mpicc* compiler. Next, the CUDA file is compiled into its own object file with *nvcc*. Finally, all object files are linked together using *mpicc*. The MPI objects are linked together first with the CUDA object being linked in last. Additionally, we also include any required CUDA libraries in our linking, such as *cudadevrt* and *cudart*. Both the debug and release versions of our project use the same compilation scheme, with the release version having higher levels of optimization.

3.6 Resolution Scaling

In order to implement a weak scaling evaluation of our program, we must determine the best way to increase the world size along with the number of processors. Since the base scale of our world already includes the entire continental United States, increasing the geographical world area is not an option. To change the world size without changing the area of the continent that the world covers, we have chosen to increase the resolution of the world instead. The base world size has a scale of 1 pixel for about every 3 miles in

length. This results in each pixel having an area of almost exactly 9 square miles.

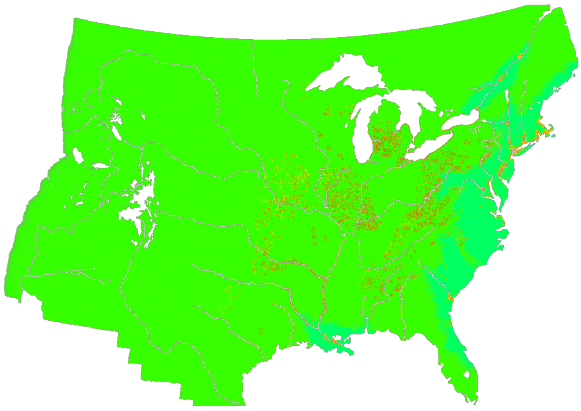

Figure 6: A world state at $t = 1000$ from a simulation with 4 MPI ranks, 4 CUDA devices, and a resolution scale of 3 (max).

Figure 7: The population distribution of the United States in the year 1870. Darker colors indicate a higher population density. Reproduced from [Kinnahan 2008].

For higher resolutions (see figure 6), we shrink the size of each pixel by 1 mile in length at a time. As our starting population numbers are based on people per square mile, our simulation does not support worlds with resolutions of less than one square mile per pixel. Therefore, we evaluate our program on three world sizes: 9 sq. mi. / pixel (resolution scale 1), 2.25 sq. mi. / pixel (resolution scale 2), and 1 sq. mi. / pixel (resolution scale 3). Here, "resolution scale" refers the level of detail (LOD) on the world. Higher values indicate a higher LOD. Our method for losslessly increasing the world's LOD is simple: duplicate each pixel by 4 for a resolution scale of 2 and by 9 for a resolution scale of 3. All attributes are duplicated here, such as elevation or temperature, except for population. To avoid an increased starting population, we scale each pixel's population down so that the number of people per square mile in that pixel is identical to the unscaled data.

4 RESULTS

For our results, we have decided on a standard iteration count of 1000. This brings our simulation from its starting point in 1763 to the year 1863. The output from our simulation is saved as a checkpoint every 200 iterations to better showcase how our population distributions evolve over time (see figure 5). Using the *OpenCV* library from Python, we are able to transform our raw array output back into a viewable image. The colors in our output images are the same as those from the input distribution to allow for a seamless visual comparison.

One of the major issues we faced in our work was with the distribution of our random *jitter* (see section 3.2). Our initial models that utilized MPI only made use of traditional C pseudorandom number generation using the *rand*() standard library function. This produced a very uniform distribution which allowed our simulation to be both accurate and natural. As the STL *rand*() function is not available within CUDA code, however, we were forced to replace it with compatible code. Due to issues with CUDA's *curand*() function (see section 6.1), we chose to implement our own pseudorandom generator. We chose the *Linear Congruential Generator (LCG)* [Thomson 1958] algorithm due to its low computational complexity and uniform distribution. Due to differences in the random distribution of our generator and that of the STL, our simulation results from CUDA show slightly more dense urban populations and slightly more sparse rural populations.

Figure 5 shows the final result, along with the checkpoints, for a simulation using 4 MPI ranks and no CUDA devices. This represents the most accurate implementation of our simulation. As time progresses within our simulation, expansion Westward begins slowly, but quickly picks up after the year 1800. The same trend is true for our population centers. As industrialization begins accelerate growth in 1840, we can observe the populations of cities growing significantly from their previous values. These population centers are concentrated on the coasts, along the Great Lakes, and in areas of high natural resource density. This mirrors the real population distribution on the time period (see figure 7).

When we do not restrict the number of CUDA blocks and utilize 256 threads per block, our simulation with 4 MPI ranks and 4 GPUs can compute all 1000 iterations in *under 2 seconds*, when using a resolution scale of 1. Thanks to the

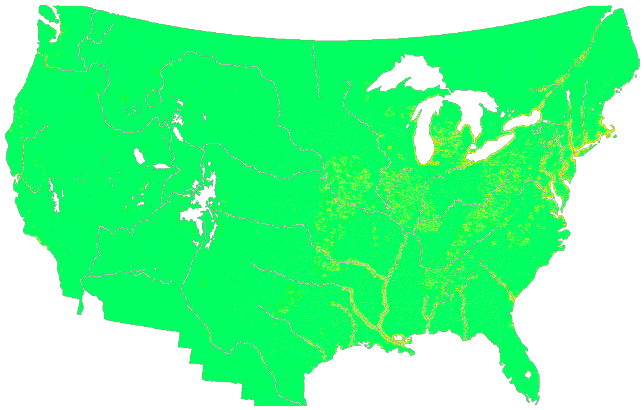

Figure 8: A world state at $t = 1000$ from a simulation with 4 MPI ranks and 4 CUDA devices.

high capacity for parallelization within each CUDA device, increasing the scale to 3 only increases the execution time by 0.4 seconds; this is despite the world size being 9 times larger. However, in order to better showcase the scaling capacity of of MPI, we limit the number of CUDA device blocks to 32 and the number of threads per block to 32. Figure 8 shows the final result from a similar simulation over 1000 iterations and using 4 MPI ranks and 4 CUDA devices. It has a similar overall population to the final image in figure 5, with the CUDA simulation having a final population of 35,373,965 and the MPI-only simulation having a population of 33,778,227. The primary difference between the two outputs is how that population is distributed. In comparison, the ground-truth population from 1860 was 31,443,321; our simulations have an error of less than 10% in this case.

4.1 Comparison to Census Data

One of the metrics by which we wanted to measure the accuracy of our simulation is comparison to historical population statistics. Table 1 contains historical census data of several American cities, as well as the United States as a whole, compared with the results that our simulation produced for those areas, in its most accurate case, seen above in Figure 5.

Looking at all of these stats, we find that our simulation only barely overestimates the total population of the United States judging by the 1860 Census, simulating a population that is only 7.4% larger than the true population. However, when trying to compare individual cities, we find, using the produced map as a metric, that this simulated population is more spread out than the population at the time.

According to table 1, our simulation is accurate within 10% of the total U.S. population of the time and is only off by tens of percentage points for several major cities. New Orleans has the lowest error of only -3.25%, wile coastal cities such as Boston, New York, and San Francisco [4] have an average error of -54%. The reason for this difference between the Oceanic cities and New Orleans is likely immigration. For example, our estimate for NYC matches more closely with the 1850 census that measured 590,000 people. Anomalies such as the Irish potato famine from 1845 to 1852, which saw over 600,000 Irish immigrants come through the port of New York [National Archives and Records Administration 2024], are not accounted for in our simulation. Additionally, the San Francisco bay area saw over 25,000 Asian immigrants between 1848 and 1852 [Ward 1997]. As our simulation does not account for these sudden spikes in immigration during these time periods, this factor is likely a large contributor to our errors.

Location	Census	Simulated	Difference
U.S. Total	31,443,321	33,778,227	+7.426%
Boston	177,840	103,500	-41.8%
New Orleans	168,000	162,537	-3.25%
New York	1,080,000	510,165	-52.76%
San Francisco Bay	68,714	21,735	-68.37%

Table 1: Comparing our simulated 1863 populations to historical 1860 census data.

4.1.1 Calculating Population. The calculation of the U.S. total population is performed quickly and automatically. We have implemented an addition to our image processing Python script that can sum to total population within a simulation output. This particular metric is always accurate to the output of our simulation.

Estimating the values of cities is less precise, however. In order to avoid creating a complex script that would sum the population within a specified curve or set of pixels, we have opted to manually estimate the population of cities. We perform this be selecting which pixels correspond to a city's greater metropolitan area and calculating the population in each pixel based on its color. While the colors of the pixels often do not exactly match with the population values of that pixel, the difference between colors is small enough that even maximum errors remain small. For the most dense cities, our pixels change color every 250 people per square mile. Since our image creation algorithm always takes the closest pixel value, our maximum error here would only be 125 people / sq. mi. for any given pixel.

5 RUNTIME ANALYSIS

To further evaluate the performance of our simulation at scale, we perform both strong and weak scaling tests. All

[4] For the San Francisco Bay, we compared our calculations to the combined population of San Francisco and Santa Clara counties.

results were calculated by using an optimized release build of our simulation. This is compiled with maximum optimizations *-O3* for the MPI code and *-Xptxas -O3* for the CUDA code. These optimizations have a dramatic impact, reducing unoptimized run times of over thirteen minutes down to less than one, making it take, on average, only seven percent of the original run times.

5.1 Strong Scaling

One of the more interesting results of our testing is that our initial parallelization process allowed for so many CUDA threads to be active at once that adding additional MPI ranks slowed down the program due to excessive overhead costs. Using one MPI rank with this many CUDA blocks would cause the program to take one second, but using sixteen would cause it to take just over five seconds. This was in the case that each process was allowed up to 65,535 blocks. Table 2 and figure 9 both show the case of only allowing 32 blocks, with 32 threads each. This allows for a far better indicator of the power of parallelization on these tasks, but this too sees a plateau after 8 MPI ranks, with 8,192 total CUDA threads.

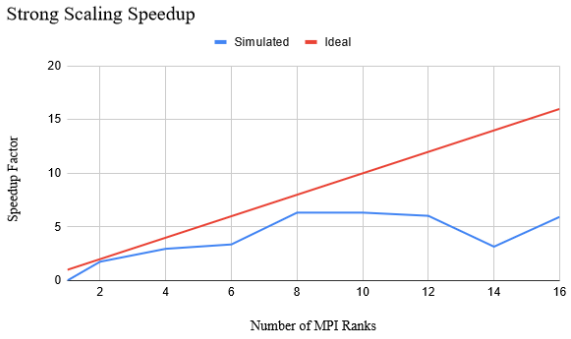

Figure 9: A *strong scaling* study of Manifest Destiny on AiMOS, limited to 32 blocks per MPI rank and a block size of 32 threads.

Ranks	Runtime (s)	Speedup	Load Time	Save Time
1	138.009	None	8.20	0.00485
2	78.415	1.76x	0.280	0.00545
4	21.039	6.56x	0.465	0.0110
6	40.96	3.369x	0.656	.0153
8	21.789	6.33x	0.857	.0128
12	22.863	6.04x	1.12	.0128
16	23.241	5.94x	1.65	0.0208

Table 2: Program speedup from adding MPI ranks, including time saved while loading input data and saving output data with MPI parallel I/O.

Despite the plateau, we do find an ultimate speedup factor of roughly 6x when using eight or more ranks, with a peak speedup of 6.56x when using four ranks. MPI I/O is also quite fast, with the writing of output never taking more than two hundredths of a second, and the time spent reading the input only taking more than two seconds in the case of a single rank. As the number of GPU resources grows, the time taken to save and load becomes a greater proportion of overall execution time. This is especially true when writing out multiple checkpoints in one run. These I/O timings were measured by taking the difference between the number of CPU clock cycles between the beginning of a parallel I/O operation and it's end.

It is because of this plateau that our simulated results in figure 9 diverge from an ideal, linear speedup, though we do note that they align well when using eight or fewer ranks, following a roughly linear trend with two ranks providing a 1.76x speedup and eight providing a 6.33x speedup.

5.2 Weak Scaling

As mentioned in section 3.6, we are able to increase the world size by duplicating each original cell, achieving a higher LOD. Table 3 shows our timing data for these increasing levels of detail. Figure 10 showcases the changes in efficiency across these three resolutions. While using four ranks actually shows an increase in efficiency, the overhead of using nine ranks and over nine thousand total CUDA threads manages to slow down the process significantly, dipping down to 1.29 x 10^8 cell updates per second, a slowdown of roughly 36%.

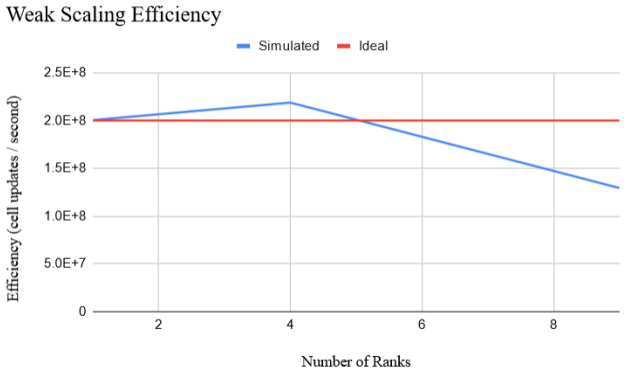

Figure 10: A *weak scaling* study of Manifest Destiny of AiMOS, limited to 32 blocks per MPI rank and a block size of 32 threads.

Ranks	Res. Scale	Cells	Time (s)	Time (cycles)
1	1	619,262	3.809	1,950,383,997
4	2	2,477,048	11.324	5,797,688,406
9	3	5,573,358	43.084	22,058,971,766

Table 3: A *weak scaling* study of Manifest Destiny on AiMOS, limited to 32 blocks per MPI rank and a block size of 32 threads. Each rank is given a constant area of the world, thus the number of ranks must square along with the total world area.

As shown by table 3, the effects on runtime of increasing the size of the world are notable, with the lowest resolution having a runtime of under four seconds and the highest resolution having a runtime just over forty seconds. However, the scaling does have a significant effect on the quality of data produced. While our simulation scale with its default resolution has accurate to within tens of percentage points of the ground truth, our results with higher resolutions are off by hundreds of percentage points. While we do employ measures to help make these resolutions more accurate, such as accounting for a the same real area of neighbors (about 216 sq. mi.) for all resolutions or limiting the cell growth rate for higher resolutions, the result is still inaccurate. For this reason, we do not directly compare the results of these scaled simulations to the grounds truth. Instead, we focus on measuring their runtimes for our weak scaling tests.

6 DISCUSSION

6.1 Limitations

The most significant limitation that our current implementation faces is its handling of artificially increased resolutions (see section 3.6). As the resolution scale increases further from the default of 1, our simulation becomes less accurate. Higher resolutions still produce results with much higher global populations, along with much denser cities, as seen in figure 6. This is in spite of our efforts to modify our algorithm to fit all scales without disturbing the default resolution, in particular by decreasing the overall growth rate of cities and increasing the change that cities will widen and lower their density. A naive solution would be duplicate our algorithm with significantly changed growth and migration rates for each resolution scale, but this approach is not salable and would significantly increase the size and complexity of our algorithm.

Another limitation with the current state of our work is how we handle the generation of pseudorandom numbers. As mentioned in section 4, we attempted to simply replace our calls to STL *rand*() with calls to CUDA's *curand*(). However, this always caused the CUDA kernel to crash whenever the CUDA random function was called. This crash also occurred silently, so no outward errors would be produced. The simulation would never change from the starting state as the kernel would always crash before updating a cell's population. Our solution to this issue was to implement our own *Linear Congruential Generator (LCG)* algorithm for generating pseudorandom numbers from the product of a cell's index and the current iteration. This strategy works very well, but the difference in random distribution between our function and that of the STL leads to slightly different values.

A more minor limitation lies within our visual representation of our output data. In order to ensure a balance between precision and contrast, the images that we derive from our simulation output files are composed of only 13 individual pixel colors (see section 4.1.1). The colors, ranging from 0 people per square mile to 3000 people per square mile, serve as an approximate visual indication of a pixel's population value. This has the benefit of creating images with a high contrast between dense cities and sparse countrysides, but has the downside of making the exact estimate of population difficult. A solution would be to implement a script that could sum the population values of pixels, when given a list of coordinates, however we opted to exclude this improvement as it would take a significant amount of time to implement and to select areas for calculations.

6.2 Group Contributions

The group was small, but the work was divided as evenly as possible. While Gunnar found much of the data, Matthew was instrumental in implementing the python scripts that would read that data and turn it into the NumPy pre-processed file. As far as parallelization, Matthew wrote much of the code involving MPI, while Gunnar wrote the code involving CUDA, and ran the majority of the tests on AiMos. The work was also split within this report: Gunnar wrote a first draft of the Implementation section, as well as writing the Results, Analysis, and Discussion sections, while Matthew revised the Implementation section and wrote the remainder of the report.

6.3 Future Work

An obvious avenue for future work would be to incorporate a more holistic method of calculating a city's population. This would include specific events that would modify a city's population at a certain time within the simulation. For example, influential events that we currently omit include the massive influxes of immigrants to the East and West coasts during the 1850s (see section 4.1), or the beginning of the American Civil War, which had already significantly impacted urban populations by our simulation's end date in 1863. Additionally, we could dynamically change the values of certain cells

based on the current iteration. For example, cells with gold could become depleted within a certain number of decades after being first settled, or the relative value of iron and coal could increase as America began to feel the effects of industrialization. Certain harbors and waterways could also be good candidates for change. As populations grew on the West coast, they began to draw more significant amount of trade, which further increased the population. This type of feedback look is not currently modeled in our simulation and would serve as a way to increase our simulation's accuracy.

Another area that deserves future improvement are the issues that we describe in section 6.1. This includes the difference in distribution between our random algorithm and that of the C STL. A straightforward solution to this issue would be to continue attempting to implement CUDA's inbuilt random generator, however the lack of information in the silent crash combined with the lack of alternatives makes this especially difficult at present. Another avenue for further development would be how our simulation handles differing resolutions. As our limited efforts to better match the results of higher resolutions with those of lower ones has been successful thus far, continuing to modify our single algorithm seems like the most promising approach. The fixes that we have already implemented primarily involve scaling growth multipliers and a cell's area of effect by the resolution scale. Since this value is 1 for our default resolution, it is always unaffected while the other resolutions are modified. Continuing to fine-tune these values may be the key for a working, unified algorithm.

7 CONCLUSION

In this paper, we have introduced Manifest Destiny, a tool for simulating the Westward expansion of the United States over during its "golden age" of expansion. We have implemented this simulation using a combination of MPI parallel processing, parallel I/O, and CUDA. Our program's cellular automata approach mirrors many similar works within the field of population modeling and is able to produce results that are closely accurate to historical values while remaining as simple as possible. The cells within our simulation grow as a consequence of several commonly cited factors of population growth including geographical, environmental, and population data. Our strong scaling tests indicate that our implementation is able to successfully take advantage of highly parallelized hardware and our weak scaling tests indicate that each rank is able to handle a large amount of data when under a constant load. Overall, Manifest Destiny is able to simulate accurate values while taking full advantage of a high-performance computing environment.

REFERENCES

John P Bowes. 2016. US Expansion and its Consequences, 1815–1890. *The Oxford Handbook of American Indian History* (2016), 93.

Charles Edward Chapman. 1921. *A history of California: the Spanish period*. Macmillan Company.

Climate.gov. 2021. U.S. annual temperature and precipitation. https://www.climate.gov/media/13728 [Online; accessed March 25, 2024].

Claudia Cosentino, Federico Amato, and Beniamino Murgante. 2018. Population-Based Simulation of Urban Growth: The Italian Case Study. *Sustainability* 10, 12 (2018). https://doi.org/10.3390/su10124838

Demographia. 2001. City of New York Boroughs: Population Population Density from 1790. http://www.demographia.com/dm-nyc.htm [Online; accessed March 20, 2024].

Robert Hall, Harriet Smither, and Clarence Ousley. 1920. French Posts and Forts in Louisiana and New France. https://etc.usf.edu/maps/pages/5200/5276/5276.htm [Online; accessed April 13, 2024].

James A. Bowen. 1901. Production of Iron in the United States. https://etc.usf.edu/maps/pages/3000/3062/3062.htm [Online; accessed March 25, 2024].

Joe Grim. [n. d.]. Topographic Map of United States. http://www.joeandfrede.com/usa/usa_topo_med_res.png [Online; accessed March 21, 2024].

Joseph A. Gornail, Steven D. Garcia. [n. d.]. 1700-1775 NYC: Metropolitan Progress: Setting the Stage for American Independence. https://www.history101.nyc/new-york-city-in-the-1700s [Online; accessed March 20, 2024].

Thomas P Kinnahan. 2008. Charting Progress: Francis Amasa Walker's" Statistical Atlas of the United States" and Narratives of Western Expansion. *American Quarterly* 60, 2 (2008), 399–423.

Lawrence W. Kennedy. 1997. Population Trends in Boston 1640 - 1990. http://www.iboston.org/mcp.php?pid=popFig [Online; accessed March 21, 2024].

Rebecca Lindsey and LuAnn Dahlman. 2024. Climate Change: Global Temperature. https://www.climate.gov/news-features/understanding-climate/climate-change-global-temperature [Online; accessed April 22, 2024].

Lumen Learning. [n. d.]. United States Population Chart. https://courses.lumenlearning.com/suny-ushistory2os2xmaster/chapter/united-states-population-chart [Online; accessed March 20, 2024].

Wolfgang Lutz and Ren Qiang. 2002. Determinants of human population growth. *Philosophical Transactions of the Royal Society of London. Series B: Biological Sciences* 357, 1425 (2002), 1197–1210.

MapResources.com. [n. d.]. USA Map with Biomes. https://www.mapresources.com/products/usa-illustrator-vector-biome-map-usa-xx-782584 [Online; accessed March 26, 2024].

Mineral Information Institute. 1993. Coal Areas in the United States. http://coaleducation.org/lessons/mii/doc3.htm [Online; accessed March 25, 2024].

National Archives and Records Administration. 2024. Irish Ship Arrivals at the Port of New York During the Potato Famine, 1846–1851. https://www.archives.gov/files/research/immigration/port/nyc-1846-1851.pdf [Online; accessed April 23, 2024].

National Oceanic and Atmospheric Administration. 2024. Global Time Series. https://www.ncei.noaa.gov/access/monitoring/climate-at-a-glance/global/time-series/northAmerica/land/ann/12/1910-2022 [Online; accessed April 22, 2024].

Richard M Sibly and Jim Hone. 2002. Population growth rate and its determinants: an overview. *Philosophical Transactions of the Royal Society of London. Series B: Biological Sciences* 357, 1425 (2002), 1153–1170.

Sidney Redner. 1998. Population history of Baltimore from 1790 - 1990. http://physics.bu.edu/~redner/projects/population/cities/baltimore.html [Online; accessed March 20, 2024].

Statistics Canada. 2010. Progress of population, 1700 to 1825. https://www65.statcan.gc.ca/acyb02/1867/acyb02_1867001803-eng.htm [Online; accessed March 20, 2024].

W. E. Thomson. 1958. A Modified Congruence Method of Generating Pseudo-random Numbers. *Comput. J.* 1, 2 (01 1958), 83–83. https://doi.org/10.1093/comjnl/1.2.83 arXiv:https://academic.oup.com/comjnl/article-pdf/1/2/83/1175362/010083.pdf

Russell Thornton. 1987. *American Indian holocaust and survival: A population history since 1492*. Vol. 186. University of Oklahoma Press.

United States Geological Survey. 2023. National Hydrography Dataset. https://www.usgs.gov/national-hydrography/national-hydrography-dataset [Online; accessed March 21, 2024].

United States Office of the Historian. [n. d.]. Proclamation Line of 1763, Quebec Act of 1774 and Westward Expansion. https://history.state.gov/milestones/1750-1775/proclamation-line-1763 [Online; accessed March 23, 2024].

TX Uue, YA Wang, SP Chen, JY Liu, DS Qiu, XZ Deng, ML Liu, and YZ Tian. 2003. Numerical simulation of population distribution in China. *Population and Environment* 25 (2003), 141–163.

Guillaume Vandenbroucke. 2008. The US westward expansion. *International economic review* 49, 1 (2008), 81–110.

Geoffrey Ward. 1997. *The West: An Illustrated History*. Little, Brown Co.

D.A. Wentz, M.E. Brigham, L.C. Chasar, M.A. Lutz, and D.P. Krabbenhoft. 2014. Mercury in the Nation's streams—Levels, trends, and implications. *Geological Survey* Circular 1395 (2014). https://doi.org/10.3133/cir1395

Western Mining History. [n. d.]. The Top Ten Gold Producing States. https://westernmininghistory.com/2099/the-top-ten-gold-producing-states [Online; accessed March 25, 2024].

Michael J. Wisdom, L. Scott Mills, and Daniel F. Doak. 2000. LIFE STAGE SIMULATION ANALYSIS: ESTIMATING VITAL-RATE EFFECTS ON POPULATION GROWTH FOR CONSERVATION. *Ecology* 81, 3 (2000), 628–641. https://doi.org/10.1890/0012-9658(2000)081[0628:LSSAEV]2.0.CO;2