Assignment 3 - Scientific Computing SC9505

Jared Wogan

2022-12-14

All code was compiled inside an Ubuntu WSL environment on an AMD 3900X with 32GB of RAM. All timing and performance data was run with randomly generated initial data, using a version of the program that does not render anything visually. All plots were generated for a fixed number of iterations taken to be 10000. A repository containing the entire project can be found here: https://github.com/JaredWogan/SC9505[1].

1 Conway's Game of Life

Conway's game of life, the brain child of John Conway in 1970, is an example of a cellular automaton. The game is played on a two dimensional grid of square cells (originally taken to be infinite), with each cell taking one of two possible states: alive or dead. Each cell is aware of it's eight neighbors, and it is these eight neighbors that fully determines the evolution of each cell. The game is instantiated by supplying an initial state, often called a seed, which is then evolved according to a handful of very basic rules. The rules are given by the following:

1. Any cell that is alive with two or three alive neighbors survives.
2. Any dead cell with exactly three neighbors that are alive becomes alive.
3. All other cells that are alive die, and all remaining dead cells stay dead.

With these rules, a wide variety of interesting and often beautiful patterns can be generated. It is even more interesting to know that the game of life is Turing complete; it is able to simulate any Turing machine. We will now see the game in action, and explore two different methods that can be used to parallelize the serial code.

[1] I have decided not to include the code within this document as this would become unnecessarily long.

2 Serial Code

The serial code has been written in C++ using the object oriented programming paradigm. There are two two classes, one which contains the grid of cells, and another which orchestrates the visuals using the SDL2 graphics library. The grid class is responsible for initializing the grid and updating the cells during each iteration. The camera class takes an instance of the grid class and renders it on screen, and allows the user to pan around and toggle the state of each cell.

We can be confident that the serial code is working by using the visual output to reproduce known patterns. One such example of a periodic pattern can be seen in Figure 1. When performance data was collected, an alternative version of the program that does not have a visual output was instead used to remove any impact rendering may have on the execution speed.

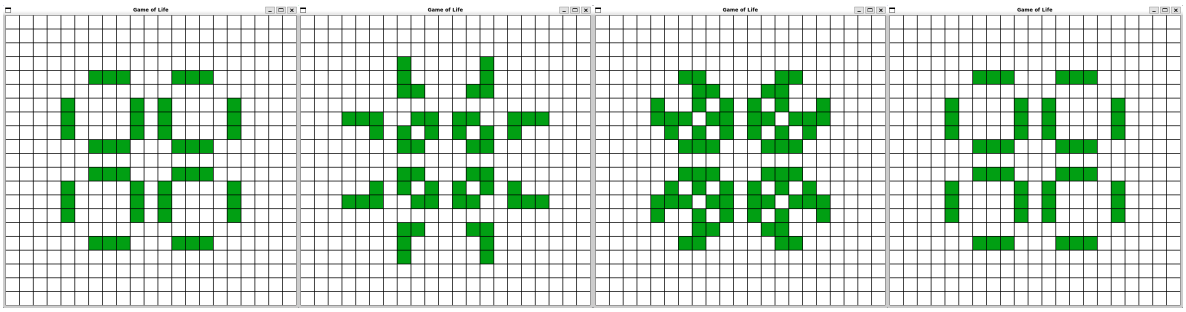


Figure 1: Example of a periodic pattern repeating itself.

For the serial version of the code, we can expect the total runtime of the program to scale as $\left(8n^2 - 4n\right) * T_{check}$ where n is the number of cells along either axis of the grid, and T_{check} is the time it takes for the program to check the status of an adjacent cell in order to update any given cell.

3 Sequentially Parallelized Code

Parallelizing the code in a sequential manner requires us to divide the grid into blocks along one of the two possible directions. Here, we take each block to be equally sized and we perform the split along the rows (vertically) of the grid. A schematic diagram of the split can be seen in Figure 2.

Updating the code required changes within the grid class which is responsible for updating the cells during each iteration. As can be seen in Figure 2, we were required to implement a new method in the grid class that is responsible for distributing the boundary data to each process, and another for collecting all the data onto the main process so that it can be rendered.

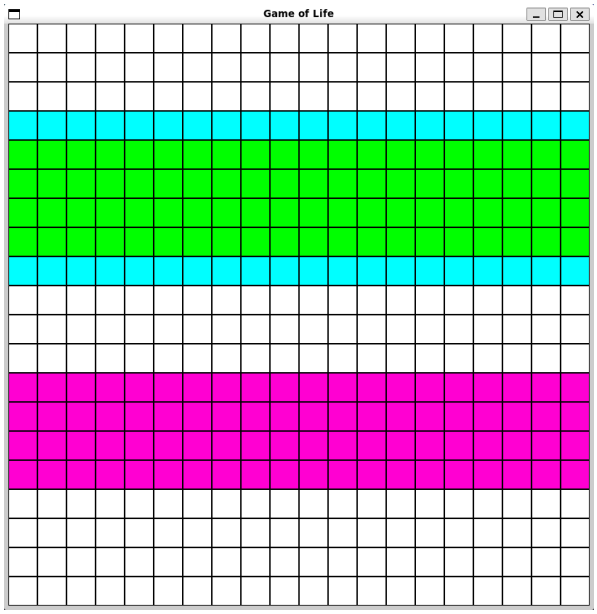


Figure 2: A schematic representation of the data a single thread processes. Here, the green cells belong to a single thread. The cells in cyan are boundary cells that must be received by the green process each iteration in order to update the cells in green.

The camera class also needed to be updated so that it only renders the grid on a single process, otherwise we would have a graphics window for each process running. Any events processed by the camera class (including mouse input, key strokes, or exit commands) must also be sent among the threads so that each process is aware of the command.

In Figure 3, we compare the runtime of the parallelized code as a function of the total number of cells in the grid. Immediately we see that the parallel code does run faster than the serial code, but the scaling is nearly identical. We would expect the program to scale roughly as $\mathcal{O}\left(n^2\right)$, where n is the number of cells along each axis, but in the figure we can see we actually get slightly better scaling (a slope of 1 in the figure corresponds to $\mathcal{O}\left(n^2\right)$ scaling since we are effectively plotting n^2 on the x-axis).

The plot suggests that the parallel code is always faster than the serial code, but this is not actually the case. For small system sizes, such as $n = 4$, the parallel code when run on 4 threads is actually slower than the serial code. This is a result of the overhead required for communication. At such small system sizes, is it not worth using or writing parallel code.

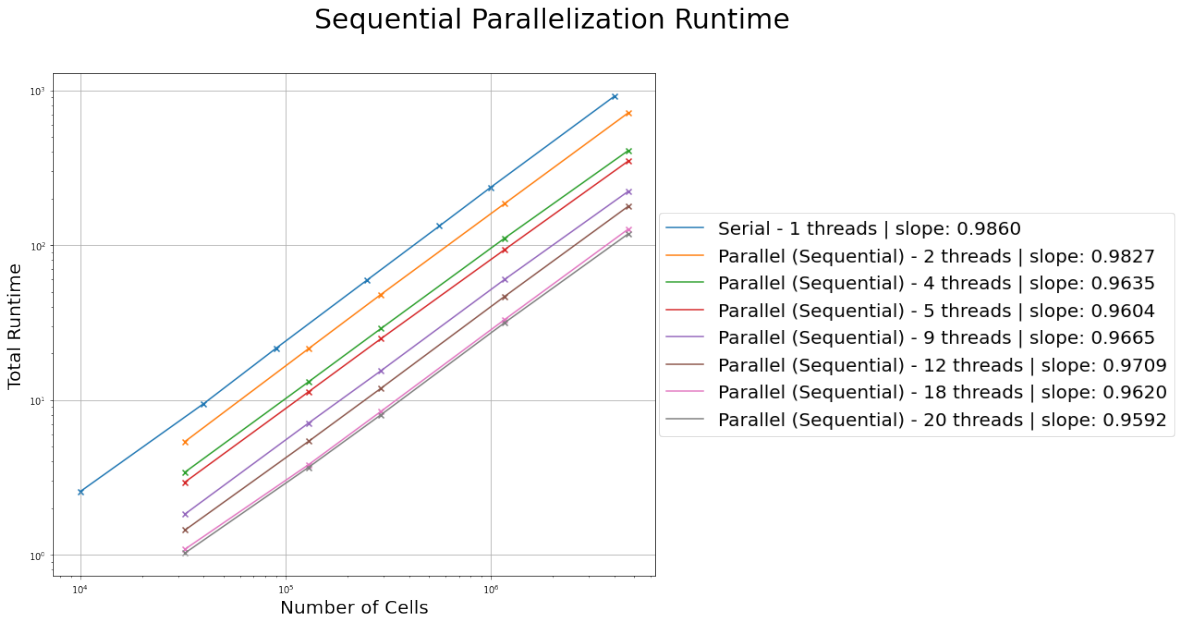


Figure 3: A plot showing the runtime of the program for varying grid sizes and thread counts.

We can look at the speedup and efficiency of the parallel code, which is displayed in Figure 4. We see that increasing the number of threads consistently increases the speedup of the program, but as we increase the number of threads our efficiency continues to decrease. We can also see that our speedup and efficiency increases slightly as the number of cells increases.

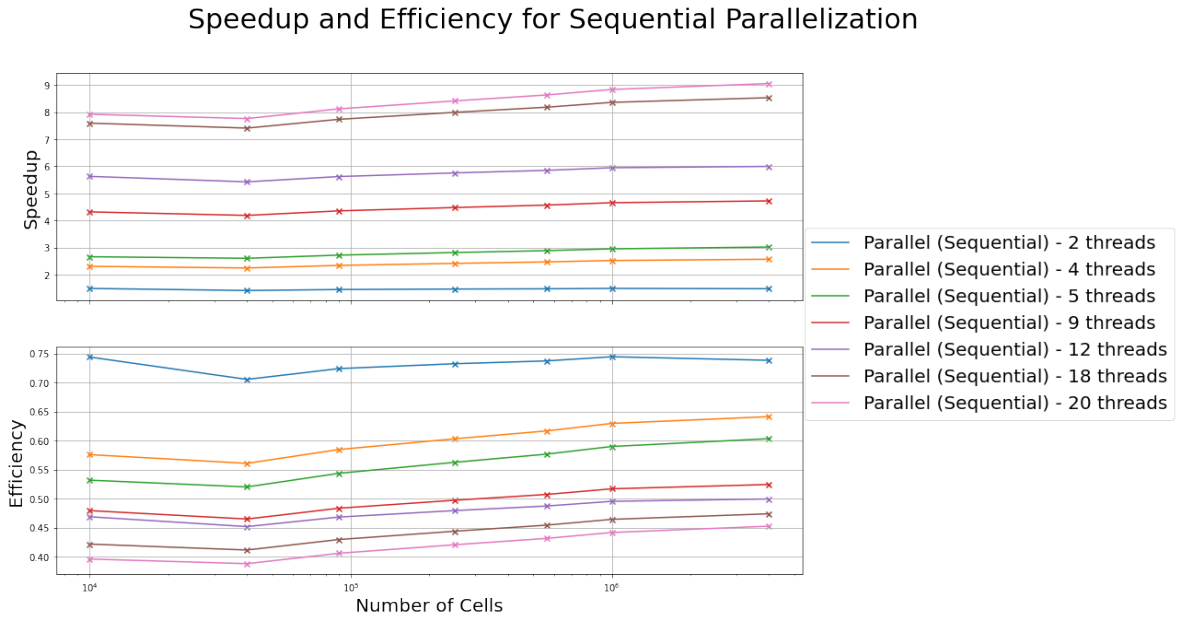


Figure 4: A plot showing the speedup and efficiency of sequential parallelization.

If we instead look at the ratio of the total run time to the communication time as in Figure 5, we see that as the number of cells in the grid increases, we spend more time computing and less time communicating. This again is expected behavior and suggests that as we increase the number of cells in the grid, our program will become more efficient.

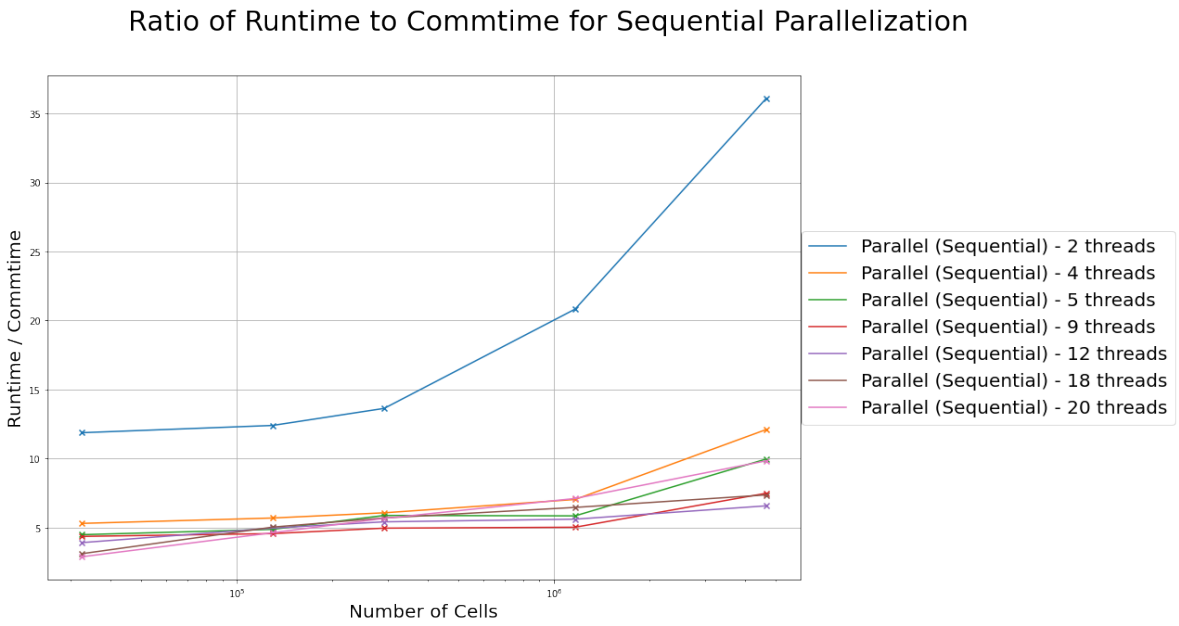


Figure 5: A plot showing the ratio of the total runtime to the communication time for sequential parallelization.

4 Two Dimensional Parallelized Code

If we instead decide to parallelize the program using a two dimensional grid of threads, we will need to update the methods used to update any given cell, and the methods used to transfer boundary data between threads. As can be seen in Figure 6, the two dimensional parallelization requires boundary data to be received from (up to) eight adjacent threads. As we will see in the analysis of the programs performance, this adds additional communication time and ultimately slows the program compared to the serially parallelized version.

Looking at the total runtime of the two dimensionally parallelized code, we still see that the parallel code outperforms the serial code as is expected, as can be seen from Figure 7. We also see that the scaling is nearly the same as the serial code, just like in the sequentially parallelized code. Again, the scaling is roughly $\mathcal{O}\left(n^2\right)$, where n is the number of cells along either axis of the grid.

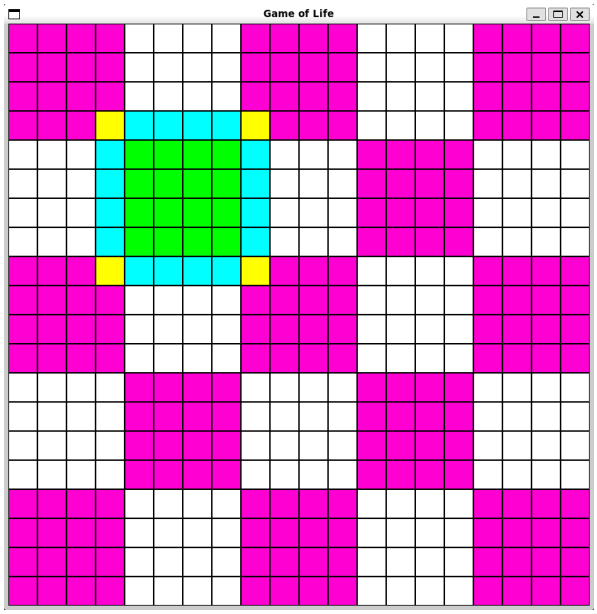


Figure 6: A schematic representation of the data a single thread processes. Here, the green cells belong to a single thread. The cells in cyan and yellow are boundary cells that must be received by the green process each iteration in order to update the cells in green.

Just as for the sequentially parallelized code, the two dimensional parallelization is not always faster than the serial code as the plot may suggest. The case here is actually worse than it is for the sequentially parallelized code. Here, the parallel code only faster after $n = 40$ when tested on 4 threads. In this case we must go to even larger systems for the use of the parallel code to be worthwhile, and this is once again due to the overhead required for the communication between the threads.

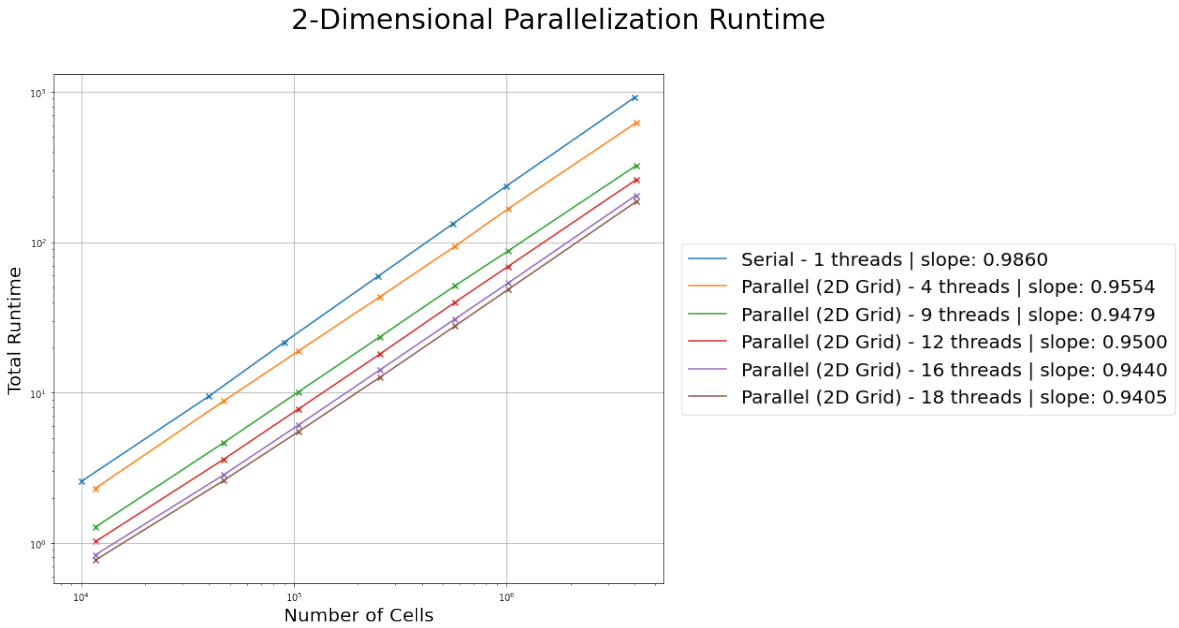


Figure 7: A plot showing the runtime of the program for varying grid sizes and thread counts.

If we instead look at the speedup and efficiency

presented in Figure 8, we actually find that the two dimensional parallelization is slower and less efficient than the sequentially parallelized code, as can be seen in Figure 8. As was hinted to earlier, I suspect this is due to the larger amount of boundary data that must be sent between the threads on each iteration.

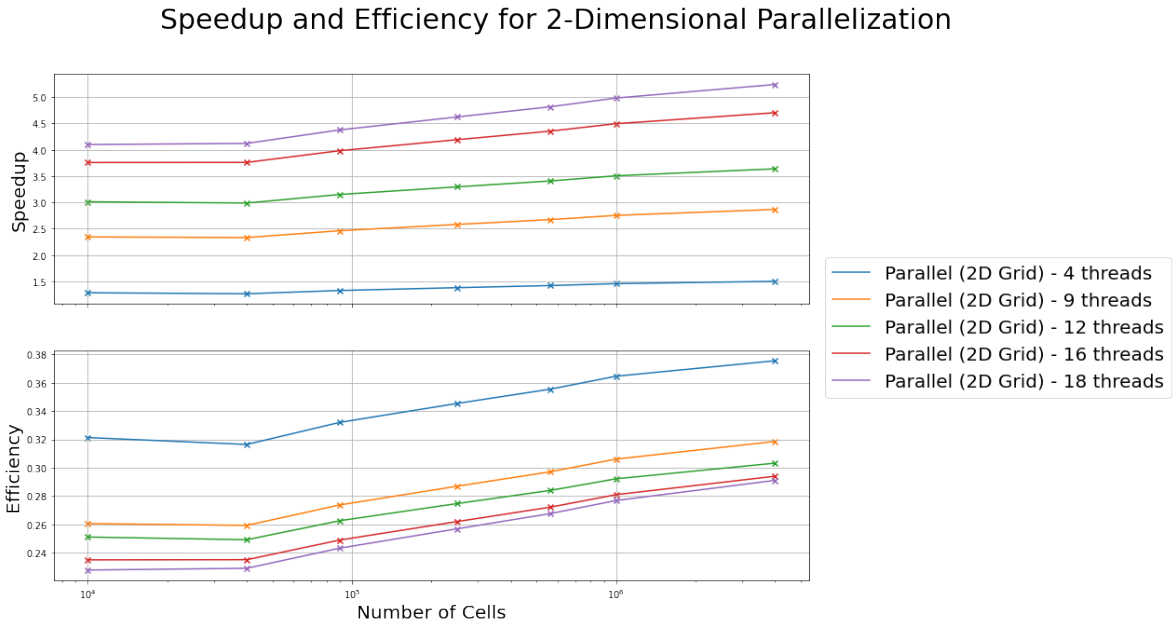


Figure 8: A plot showing the speedup and efficiency of two dimensional parallelization.

The idea that a more substantial amount of time is spent communicating in the two dimensional case is further justified when we look at the ratio of the total runtime to the communication time, which can be seen in Figure 9. Here, we see that this ratio is smaller when compared to the sequentially parallelized code. It is still worth noting that as the number of cells increases, we still see that this ratio tends to increase, indicating that the program still becomes more efficient as the number of cells grows.

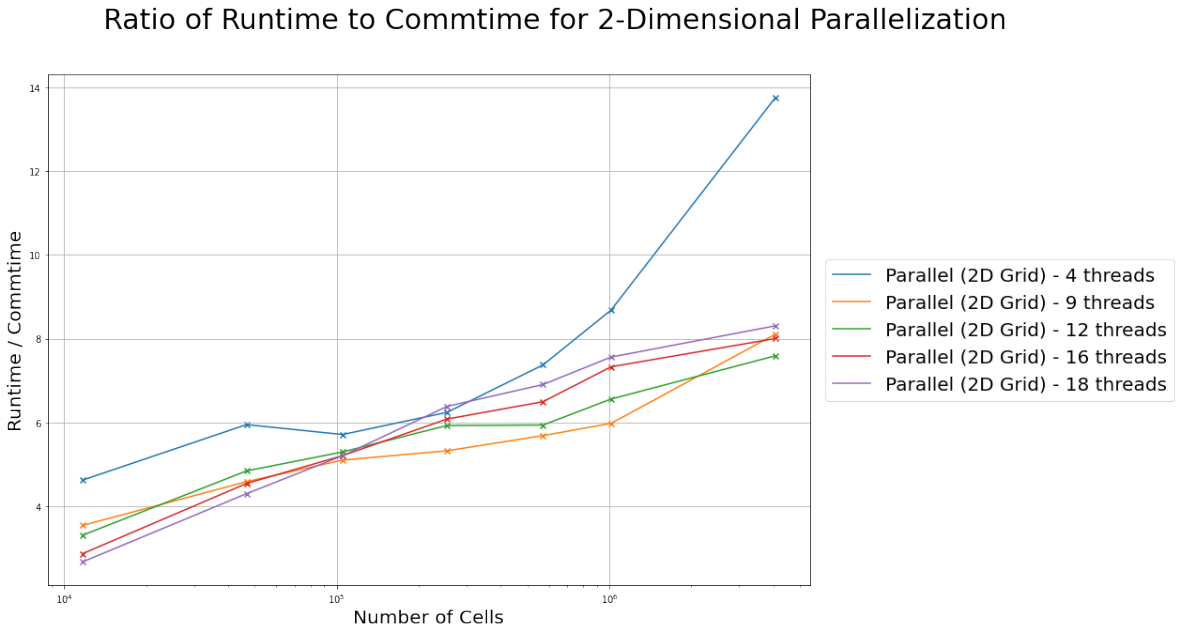


Figure 9: A plot showing the ratio of the total runtime to the communication time for two dimensional parallelization.

5 Concluding Remarks

We have explored the game of life and seen my implementation of it in C++ using the SDL2 library. We have also seen how MPI can be used to parallelize the code in two different ways. The first way involved dividing our array of cells along one axis and allowing each thread to work on just one of the equally sized blocks. The second method split the array along both axes, and using a grid of threads instead.

We saw that for our problem, the sequentially parallelized code was faster and more efficient than the two dimensional decomposition, which is a result of the extra communications that must be done in the two dimensional case. In both parallelization schemes, we never saw superlinear scaling, but it was clear from the plots that both schemes become more efficient as the number of cells in the grid increases.

It is worth mentioning that in both parallelization schemes, the code has built in restrictions that require the number of cells along the axis that gets divided to be divisible by the number of threads. This requirement forces each thread to be given an equally sized block of the whole grid to work with. In the future, it would be nice to extend the code to work for arbitrarily sized grids, but much of the code currently depends on the fact that each grid is equally sized.

A Python Performance Analysis

Attached is Python Jupyter Notebook which was used to generate the plots seen throughout the document.

Analysis

December 21, 2022

[ ]:
```
import numpy as np
from sklearn.linear_model import LinearRegression
import matplotlib.pyplot as plt
import pandas as pd
import subprocess as sp
import itertools as it
```

[ ]:
```
serial_dir = "../Serial/data.csv"
parallel_seq_dir = "../ParallelSeq/data.csv"
parallel_grid_dir = "../ParallelGrid/data.csv"
```

[ ]:
```
serial_data = pd.read_csv(serial_dir, sep=", ", engine="python")
parallel_seq_data = pd.read_csv(parallel_seq_dir, sep=", ", engine="python")
parallel_grid_data = pd.read_csv(parallel_grid_dir, sep=", ", engine="python")
```

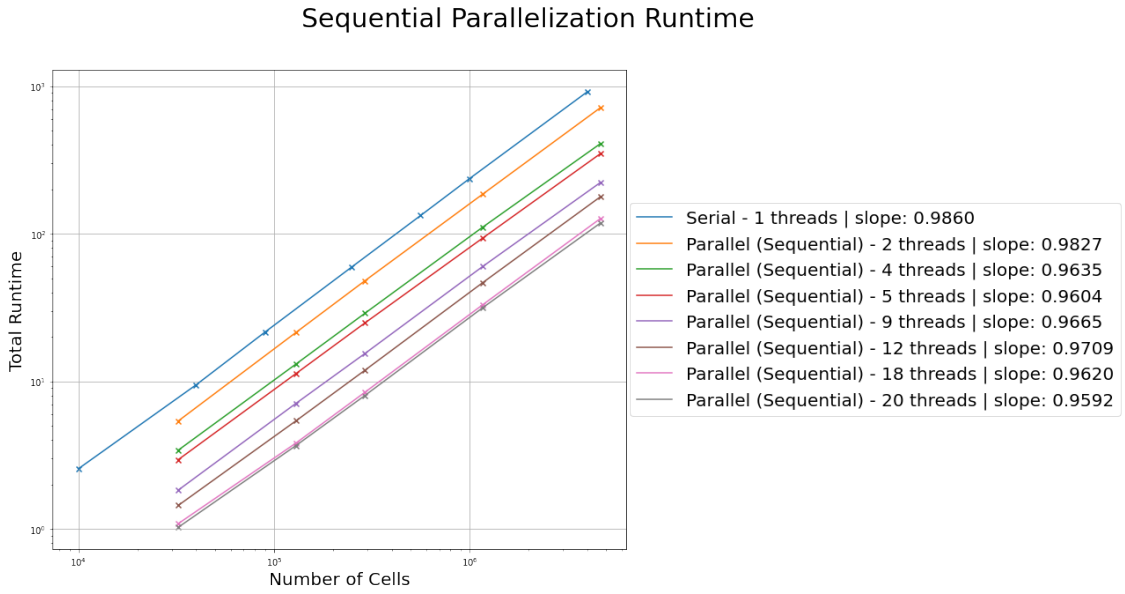
[ ]:
```
num_iters = max(serial_data["iterations"])
print(num_iters)

serial_data = serial_data[serial_data["iterations"] == num_iters]
serial_data["threads"] = 1
serial_data["size"] = serial_data["size_x"] * serial_data["size_y"]

parallel_seq_data = parallel_seq_data[parallel_seq_data["iterations"] ==␣
 ↪num_iters]
parallel_seq_data["size"] = parallel_seq_data["size_x"] *␣
 ↪parallel_seq_data["size_y"]

parallel_grid_data = parallel_grid_data[parallel_grid_data["iterations"] ==␣
 ↪num_iters]
parallel_grid_data["size"] = parallel_grid_data["size_x"] *␣
 ↪parallel_grid_data["size_y"]
```

```
10000
```

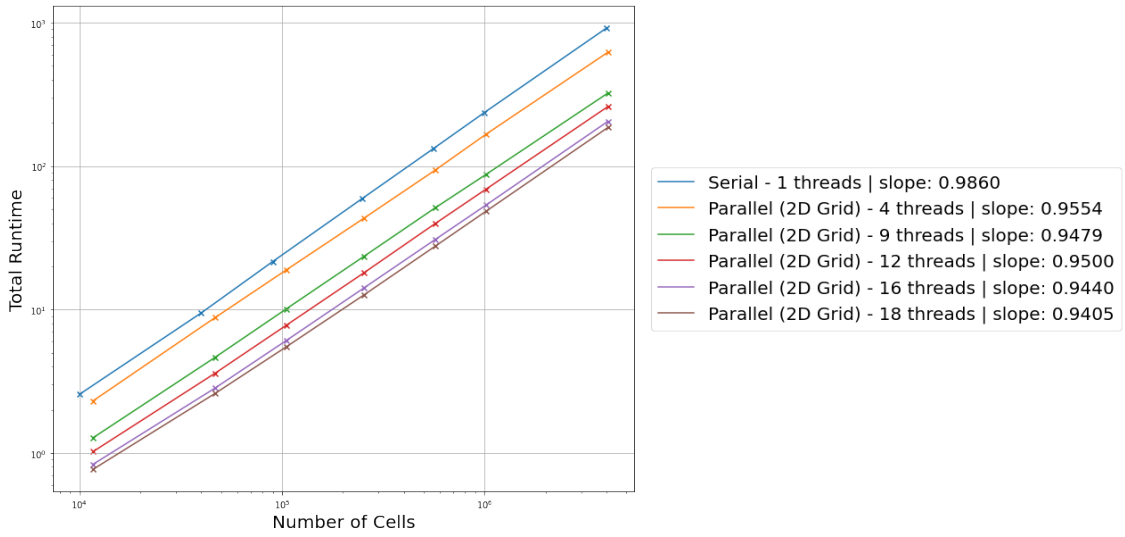
[ ]:
```
def plotCellsRuntime(dataFrame: pd.DataFrame, ax: plt.Axes, label: str):
    threads = dataFrame["threads"].unique()
```

```
    for t in threads:
        # Plot the data
        data = dataFrame[dataFrame["threads"] == t]
        sizes = data["size"].unique()
        times = np.array([
            data[data["size"] == s]["total_runtime"].mean()
            for s in sizes
        ])
        # Fit a linear model to the log-log data
        x = np.log10(sizes).reshape(-1, 1)
        y = np.log10(times).reshape(-1, 1)
        model = LinearRegression()
        model.fit(x, y)

        # Annotate the plot with the slope
        ax.scatter(sizes, times, marker="x")
        ax.plot(sizes, times, label="{} - {} threads | slope: {:.4f}".
↪format(label, t, model.coef_[0][0]))
```

[ ]:
```
fig, ax = plt.subplots(
    nrows=1, ncols=2,
    figsize=(20, 10),
    gridspec_kw={"width_ratios": [1.8, 1]}
)
fig.suptitle("Sequential Parallelization Runtime", size=30)

# Plot runtime vs. number of cells
plotCellsRuntime(serial_data, ax[0], "Serial")
plotCellsRuntime(parallel_seq_data, ax[0], "Parallel (Sequential)")

# Get the legend handles and labels
h, l = ax[0].get_legend_handles_labels()

# Set labels
ax[0].grid()
ax[0].set_xlabel("Number of Cells", size=20)
ax[0].set_xscale("log")
ax[0].set_ylabel("Total Runtime", size=20)
ax[0].set_yscale("log")

# Add the legend
ax[1].legend(h, l, fontsize=20, loc="center")
ax[1].axis("off")

# Show the plot
plt.show()
```



[ ]:
```
fig, ax = plt.subplots(
    nrows=1, ncols=2,
    figsize=(20, 10),
    gridspec_kw={"width_ratios": [1.8, 1]}
)
fig.suptitle("2-Dimensional Parallelization Runtime", size=30)

# Plot runtime vs. number of cells
plotCellsRuntime(serial_data, ax[0], "Serial")
plotCellsRuntime(parallel_grid_data, ax[0], "Parallel (2D Grid)")

# Get the legend handles and labels
h, l = ax[0].get_legend_handles_labels()

# Set labels
ax[0].grid()
ax[0].set_xlabel("Number of Cells", size=20)
ax[0].set_xscale("log")
ax[0].set_ylabel("Total Runtime", size=20)
ax[0].set_yscale("log")

# Add the legend
ax[1].legend(h, l, fontsize=20, loc="center")
ax[1].axis("off")

# Show the plot
plt.show()
```

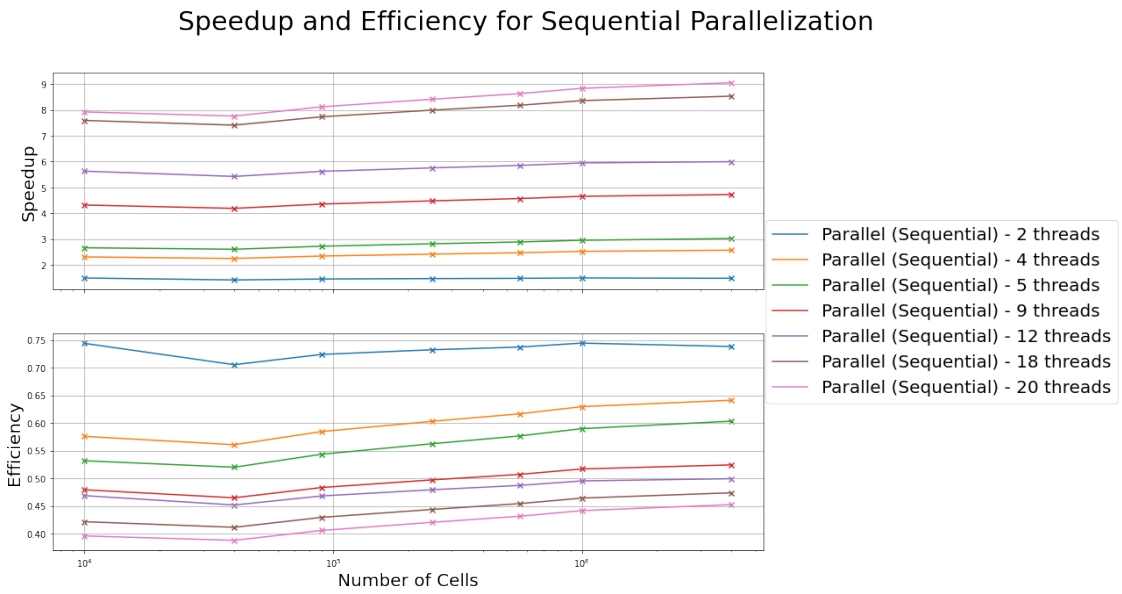


[ ]:
```python
def plotSpeedupEfficiency(serialDataFrame: pd.DataFrame, dataFrame: pd.
↪DataFrame, ax_s: plt.Axes, ax_e:plt.Axes, label: str):
    threads = dataFrame["threads"].unique()
    serial_sizes = serialDataFrame["size"].unique()
    serial_times = np.array([
            serialDataFrame[serialDataFrame["size"] == s]["total_runtime"].
↪mean()
            for s in serial_sizes
        ])

    for t in threads:
        # Get the parallel data
        data = dataFrame[dataFrame["threads"] == t]
        sizes = data["size"].unique()
        times = np.array([
            data[data["size"] == s]["total_runtime"].mean()
            for s in sizes
        ])
        # Fit a linear model to the log-log data
        x = np.log10(sizes).reshape(-1, 1)
        y = np.log10(times).reshape(-1, 1)
        model = LinearRegression()
        model.fit(x, y)

        # Predict parallel times for serial sizes
        serial_x = np.log10(serial_sizes).reshape(-1, 1)
        prediction = np.array([
```

```
            10**val[0] for val in
            model.predict(serial_x)
        ])

        # Calculate speedup
        speedup = serial_times / prediction

        # Plot the speedup
        ax_s.scatter(serial_sizes, speedup, marker="x")
        ax_s.plot(serial_sizes, speedup, label="{} - {} threads".format(label,␣
 ↪t))

        # Plot the effieciency
        ax_e.scatter(serial_sizes, speedup / t, marker="x")
        ax_e.plot(serial_sizes, speedup / t, label="{} - {} threads".
 ↪format(label, t))
```

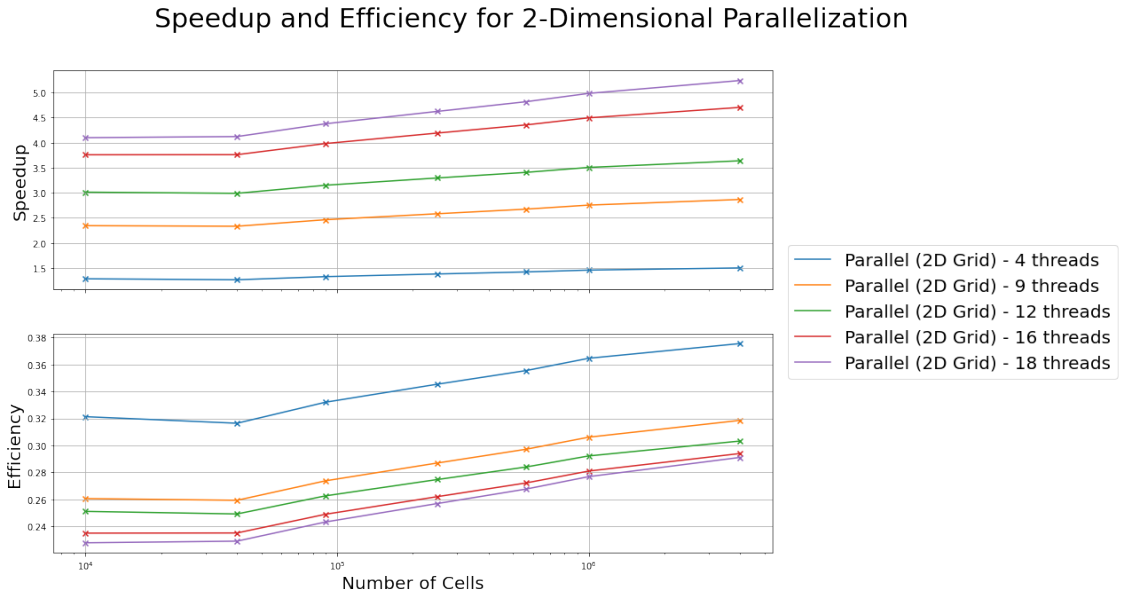
[ ]:
```
fig, ax = plt.subplots(
    nrows=2, ncols=2,
    figsize=(20, 10),
    sharex=True,
    gridspec_kw={"width_ratios": [4, 1]}
)
fig.suptitle("Speedup and Efficiency for Sequential Parallelization", size=30)

# Create grid spec for the legend
gs = ax[0, 1].get_gridspec()
for a in ax[0:, -1]:
    a.remove()
legend_ax = fig.add_subplot(gs[0:, -1])

# Plot the speedup and efficiency
plotSpeedupEfficiency(serial_data, parallel_seq_data, ax[0, 0], ax[1,0],␣
 ↪"Parallel (Sequential)")

# Get the legend handles and labels
h, l = ax[0, 0].get_legend_handles_labels()

# Set labels
ax[0, 0].grid()
ax[1, 0].grid()
ax[0, 0].set_xscale("log")
ax[0, 0].set_ylabel("Speedup", size=20)
ax[1, 0].set_xscale("log")
ax[1, 0].set_xlabel("Number of Cells", size=20)
ax[1, 0].set_ylabel("Efficiency", size=20)
```

```
# Add the legend
legend_ax.legend(h, l, fontsize=20, loc="center")
legend_ax.axis("off")

# Show the plot
plt.show()
```



[ ]:
```
fig, ax = plt.subplots(
    nrows=2, ncols=2,
    figsize=(20, 10),
    sharex=True,
    gridspec_kw={"width_ratios": [4, 1]}
)
fig.suptitle("Speedup and Efficiency for 2-Dimensional Parallelization",␣
 ↪size=30)

# Create grid spec for the legend
ax[0, 0].grid()
ax[1, 0].grid()
gs = ax[0, 1].get_gridspec()
for a in ax[0:, -1]:
    a.remove()
legend_ax = fig.add_subplot(gs[0:, -1])

# Plot the speedup and efficiency
plotSpeedupEfficiency(serial_data, parallel_grid_data, ax[0, 0], ax[1,0],␣
 ↪"Parallel (2D Grid)")
```

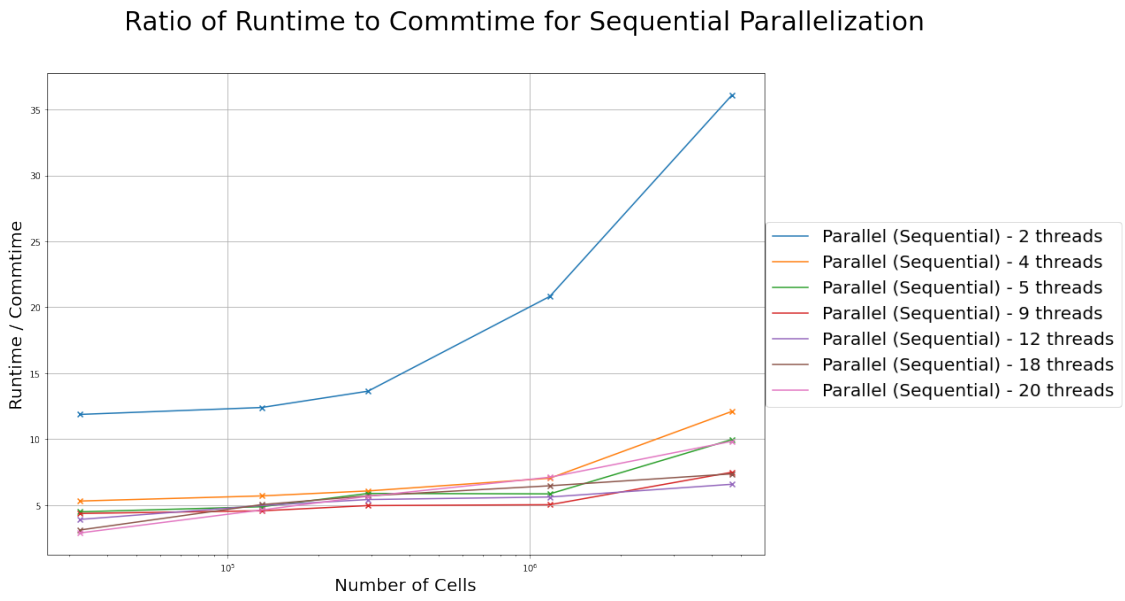
```
# Get the legend handles and labels
h, l = ax[0, 0].get_legend_handles_labels()

# Set labels
ax[0, 0].set_xscale("log")
ax[0, 0].set_ylabel("Speedup", size=20)
ax[1, 0].set_xscale("log")
ax[1, 0].set_xlabel("Number of Cells", size=20)
ax[1, 0].set_ylabel("Efficiency", size=20)

# Add the legend
legend_ax.legend(h, l, fontsize=20, loc="center")
legend_ax.axis("off")

# Show the plot
plt.show()
```



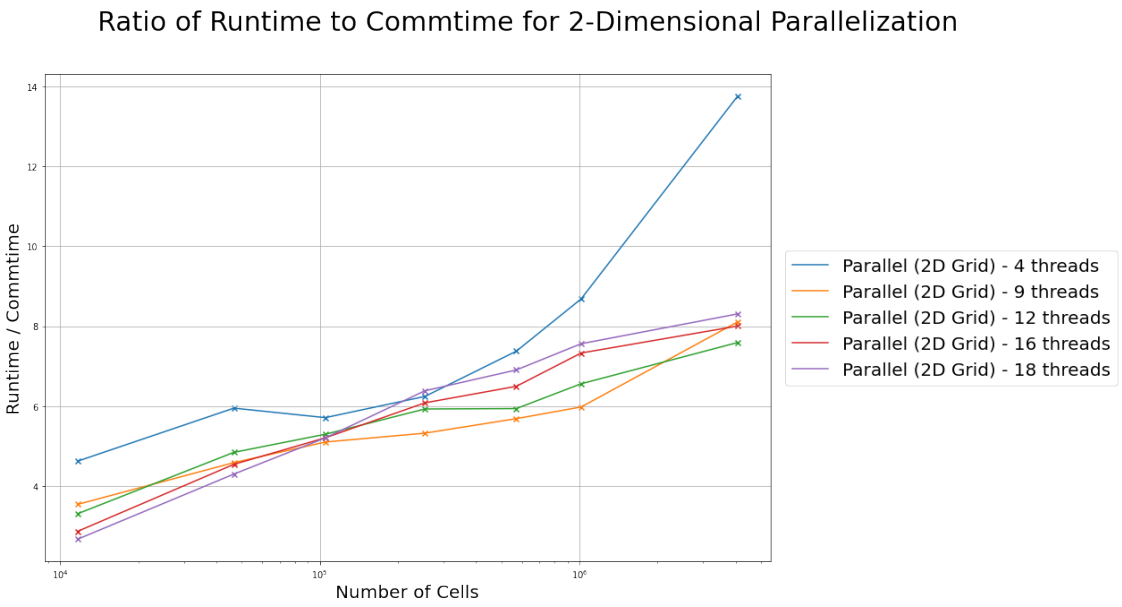
[ ]:
```
def plotCommTime(dataFrame: pd.DataFrame, ax: plt.Axes, label: str):
    threads = dataFrame["threads"].unique()

    for t in threads:
        # Get the parallel data
        data = dataFrame[dataFrame["threads"] == t]
        sizes = data["size"].unique()
        run_times = np.array([
            data[data["size"] == s]["total_runtime"].mean()
```

```
            for s in sizes
        ])
        comm_times = np.array([
            data[data["size"] == s]["comm_time"].mean()
            for s in sizes
        ])

        # Calculate ratio of run_time to comm_time
        ratio = run_times / comm_times

        # Plot the ratio
        ax.scatter(sizes, ratio, marker="x")
        ax.plot(sizes, ratio, label="{} - {} threads".format(label, t))
```

[ ]:
```
fig, ax = plt.subplots(
    nrows=1, ncols=2,
    figsize=(20, 10),
    gridspec_kw={"width_ratios": [4, 1]}
)
fig.suptitle("Ratio of Runtime to Commtime for Sequential Parallelization",␣
 ↪size=30)

# Plot ratio of run_time to comm_time
plotCommTime(parallel_seq_data, ax[0], "Parallel (Sequential)")

# Get the legend handles and labels
h, l = ax[0].get_legend_handles_labels()

# Set labels
ax[0].grid()
ax[0].set_xlabel("Number of Cells", size=20)
ax[0].set_xscale("log")
ax[0].set_ylabel("Runtime / Commtime", size=20)

# Add the legend
ax[1].legend(h, l, fontsize=20, loc="center")
ax[1].axis("off")

# Show the plot
plt.show()
```



[ ]:
```
fig, ax = plt.subplots(
    nrows=1, ncols=2,
    figsize=(20, 10),
    gridspec_kw={"width_ratios": [4, 1]}
)
fig.suptitle("Ratio of Runtime to Commtime for 2-Dimensional Parallelization",␣
 ↪size=30)

# Plot ratio of run_time to comm_time
plotCommTime(parallel_grid_data, ax[0], "Parallel (2D Grid)")

# Get the legend handles and labels
h, l = ax[0].get_legend_handles_labels()

# Set labels
ax[0].grid()
ax[0].set_xlabel("Number of Cells", size=20)
ax[0].set_xscale("log")
ax[0].set_ylabel("Runtime / Commtime", size=20)

# Add the legend
ax[1].legend(h, l, fontsize=20, loc="center")
ax[1].axis("off")

# Show the plot
plt.show()
```


Ratio of Runtime to Commtime for 2-Dimensional Parallelization
Runtime / Commtime
Number of Cells
Parallel (2D Grid) - 4 threads
Parallel (2D Grid) - 9 threads
Parallel (2D Grid) - 12 threads
Parallel (2D Grid) - 16 threads
Parallel (2D Grid) - 18 threads