Project Lake: Efficient Water Surface Simulation with MPI and CUDA Parallelism

Peilin Rao, Zheng Zhong
Language Technologies Institute
Carnegie Mellon University
Pittsburgh, PA 15213
{peilinr, zhengzho}@cs.cmu.edu

1 Motivation

Just like most of the people, Peilin Rao is a huge lover for lakes. He enjoys all the beautiful afternoons of the last summer at the shoreline of Homer lake, Illinois. He was sitting in an armchair, holding his favorite fishing rod, and watching birds flying over the shiny water surface with just right amount of sunlight. Now he misses Homer lake so much. As a computer scientist, he wants to find out a way to rebuild that piece of precious memory of Homer lake and save it into his computer. After doing some research, he finds out that realistic simulation of water surface is actually very computational expensive. He asks his best friend Zheng Zhong to help him. Luckily, they are both taking parallel programming this semester and have some ideals on the optimizations. Therefore, this project has a simple goal: how to utilize parallel programming to efficiently simulate the motion of water surface?

2 Summary

In this project, we use a method of simulating water surface with statistical model Tessendorf (2004). Using statistical model gives more realistic result but it is much more computational expensive than the approximation model, which tends to oversimplify the water surface to a linear combination of sin waves. After we implement this statistical model with the brute force algorithm, we modify it to get a good vectorization version of SIMD optimization. Then, we find out the calculation in the model can be decomposed into two Fourier transforms, which can be accelerated with Fast Fourier Transform algorithm. We implement a parallel version of Fast Fourier Transfer with Message Passing Interface. Finally, we add CUDA supports in our simulation which greatly reduces the computation time. With MPI and CUDA, our implementation achieves a significant speedup compared to the

sequential version. We would like to elaborate our implementation and discussion experimental results on different inputs. Figure 1 shows a example of our simulator. You are more than welcome to run our code and generate an gif simulation or have a look at the sample gif simulation on our Github page [1].

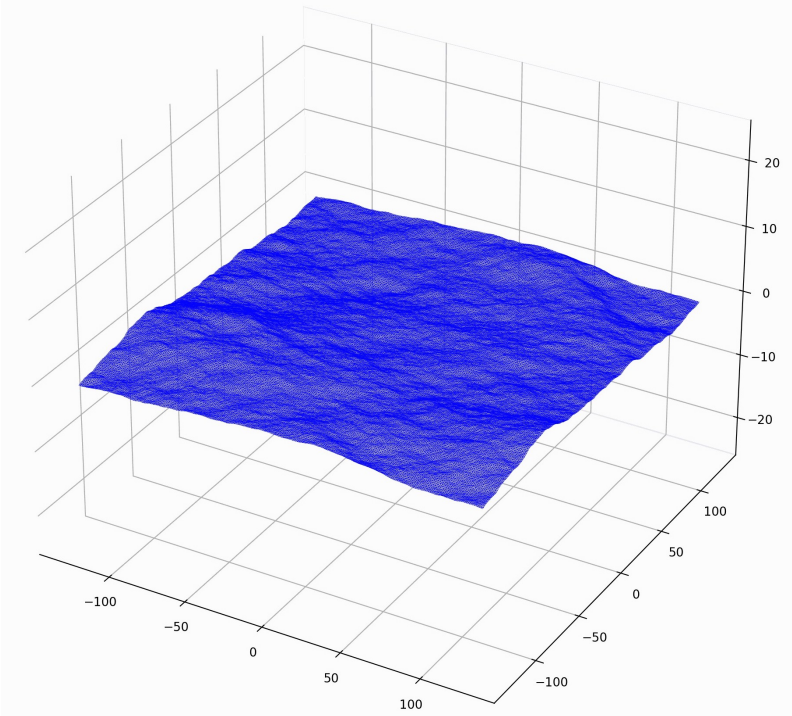


Figure 1: An example screenshot of our water surface simulation.

3 Background

3.1 Statistical Water Model

We would like to begin by giving some characteristics and constraints considered in the statistical model before talking about its implementation, as the theory behind this model is entirely provided by this paper Tessendorf (2004). We would not focus on any rendering problem such as lighting or coloring since this is not a project for computer graphics course. Instead, we only consider the height (y) change of a tile of points in 3D dimension with fixed x and z values. The output of our program is the triangular mesh of those points moving over time.

In one simulation, we keep the following tune-able parameters as constants: (1) the sample size: the x-z tiles of all the points considered (2) total frames: the number of times of generating a screenshot. (3) wind direction (4) wind speed (5) initial amplitude (6) mean and standard derivative of the normal distribution for sampling random wave heights. Those parameters determine how the final simulation

[1] `https://github.com/peilinrao/parallel-computing-project-lake`

looks and need to be tuned carefully to make the water surface look realistic. We provide some decent sets of parameters in the result section.

Back to the statistical model, since there are a lot of math and physics concepts discussed in the statistical model that is too complex to be implemented in a parallel programming course, I made some variation of that model. First, the Fourier transform based model of a wave height field assumes the height of the points at any horizontal position x and z is a sum of sinusoids with complex amplitudes that only depends on time.

$$h(\bar{x}, t) = \sum_{\bar{k}} \tilde{h}(\bar{k}, t) \exp(i\bar{k} \cdot \bar{x})$$
$$\bar{x} = (x, z), \bar{k} = (2\pi x/N, 2\pi z/N)$$

We set x dimension, z dimension, x sampling distance, z sampling distance to the same value N for simplicity. We can see that at any given time t, to calculate the height of one point, we need to calculate $\tilde{h}$ for every point. In the following equations, k is the length of vector $\bar{k}$.

$$\tilde{h}(\bar{k}, t) = \tilde{h}_0(\bar{k}) \exp(i\omega(k)t) + \tilde{h}_0^*(-\bar{k}) \exp(-i\omega(k)t)$$

Here $\tilde{h}_0$ is the Fourier amplitude and $\omega(k)$ is the dispersion relation. There are many possible ways to define those and here is the formula used in our implementation. In the following equations, V is the wind speed, g is the gravitational constant, $\hat{\omega}$ is the unit vector of wind direction, ξ_r and ξ_i are sampled from a normal distribution, A is the wave amplitude. We use a Philips spectrum to model the wave.

$$\tilde{h}_0(\bar{k}) = \frac{1}{\sqrt{2}}(\xi_r + i\xi_i)\sqrt{P_h(\bar{k})}$$
$$P_h(\bar{k}) = A\frac{\exp(-1/(V^2/Lg)^2)}{k^4}\|\hat{k} \cdot \hat{\omega}\|^2$$
$$\omega(k) = \sqrt{g\|\bar{k}\|}$$

3.2 Fast Fourier Transform

Our implementation benefits greatly from Fast Fourier Transform algorithm. In this section I would like to provide some context about FFT.

The Discrete Fourier Transform can be described as the following equation.

$$X_k = \sum_{n=0}^{N-1} x_n * e^{-\frac{2\pi i}{N}nk}$$

Although we can just solve this summation as a simple matrix multiplication, we can also observe that a lot of terms in $e^{-\frac{2\pi i}{N}nk}$ are duplicated. Therefore, we are able to make use of radix-2 decimation-int-time case of Cooley-Tukey FFT algorithm to accelerate this. The equations and graphs in this section are borrowed from FFT for clarity.

$$X_k = \sum_{m=0}^{N/2-1} x_{2m} * e^{-\frac{2\pi i}{N}(2m)k} + \sum_{m=0}^{N/2-1} x_{2m+1} * e^{-\frac{2\pi i}{N}(2m+1)k}$$
$$X_{k+N/2} = \sum_{m=0}^{N/2-1} x_{2m} e^{-\frac{2\pi i}{N/2}mk} - e^{-\frac{2\pi i}{N}k} \sum_{m=0}^{N/2-1} x_{2m+1} e^{-\frac{2\pi i}{N/2}mk}$$

Then we can rewrite X_k and $X_{k+N/2}$ in terms of the FFT of even and odd terms of X_k. Note that Fourier transform is defined recursively so we can implement it with dynamic programming with a good vectorization.

$$X_k = E_k + e^{-\frac{2\pi i}{N}k} O_k$$
$$X_{k+N/2} = E_k - e^{-\frac{2\pi i}{N}k} O_k$$

4 Approaches

4.1 Vectorization

As we can see from the equations, calculating the height of every point in our square tile needs to know the $\tilde{h}$ value for every points. Therefore, if the length of our square tile is N, the total number of points is N^2, we have an $O(N^4)$ algorithm, which is terrible. However this is not the end of the world. We find that our equations can be benefit from FFT and Vectorization. By decompose and reordering, we have the following equation:

$$h(\bar{x}, t) = \sum_{\bar{k}} \tilde{h}(\bar{k}, t) \exp(i\bar{k} \cdot \bar{x})$$
$$h(x, z, t) = \sum_{a=-N/2}^{N/2} \exp(2\pi ax/Ni) \sum_{b=-N/2}^{N/2} \tilde{h}(\bar{k}, t) \exp(2\pi bz/Ni)$$
$$\bar{k} = (a, b)$$

With a little more changes to the summation, we get:

$$h(x, z, t) = (-1)^{x+z} \sum_{a=0}^{N} \exp(2\pi ax/Ni) \sum_{b=0}^{N} \tilde{h}(\bar{k}, t) \exp(2\pi bz/Ni)$$
$$\bar{k} = (a - N/2, b - N/2)$$

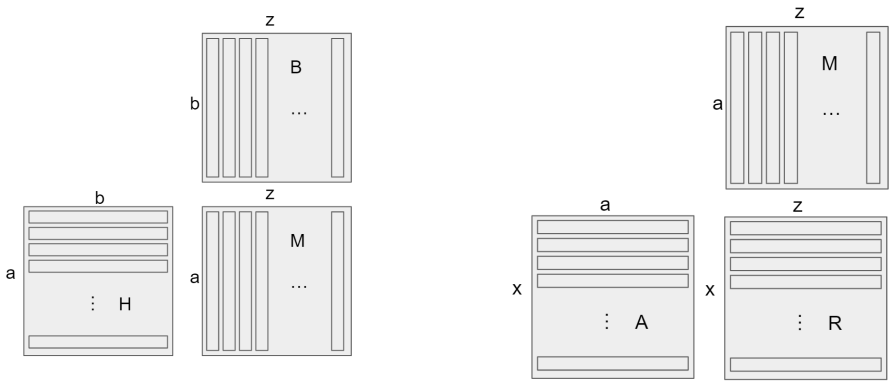
We claim that this new equation can be decently vectorize if we precalculate $\tilde{h}(\bar{k}, t)$ for every possible $\bar{k}$. Although in the original model, ξ_r and ξ_i should be sampled differently when updating height for every point, in our experiment, reusing the precalculated $\tilde{h}(\bar{k}, t)$ performs just as good as re-sampling visually.

Before we perform vectorization at time t, We need make several definition. First we define matrix F, matrix H, matrix A, and matrix B.

$$F_{x,z} = (-1)^{(x-N/2)+(z-N/2)}, x \in (0, N), z \in (0, N)$$
$$H_{a,b} = \bar{h}(a, b, t), a \in (0, N), b \in (0, N)$$
$$A_{x,a} = e^{2\pi a(x-N/2)i/N}, a \in (0, N), x \in (0, N)$$
$$B_{b,z} = e^{2\pi b(z-N/2)i/N}, b \in (0, N), z \in (0, N)$$

At the end, we need to update the height for every x and z coordinate using a matrix Y of size N^2, where $Y_{x,z}$ is the updated height for horizontal point (x, z) at time t. We claim that the updated matrix R can be expressed as the following equation, where $\odot$ is an element-wise multiplication. Also we claimed that we can use Fast Fourier Transform to accelerate the calculate. Let $M = H \cdot B$ and $R = A \cdot M$, we will show that FFT can be applied for the calculation of M and R.

$$Y = F \odot (A \cdot (H \cdot B))$$



(a) $M = H \cdot B$ (b) $R = A \cdot M$

Figure 2: Graphic representation of vectorization

Let's first look at $M = H \cdot B$, as shown in 2a. Take an arbitrary column from M, say we take the m-th column. Let column m in M be θ and column m in B be β , we can see that the following equation is a Fourier transform equation. We can vectorize this calculation for every column of B with SIMD since they all perform the same instructions.

$$\theta[a] = \sum_{b=0}^{N} H[a,b] * \beta[b] = \sum_{b=0}^{N} H[a,b] * e^{2\pi b(m-N/2)i/N}$$

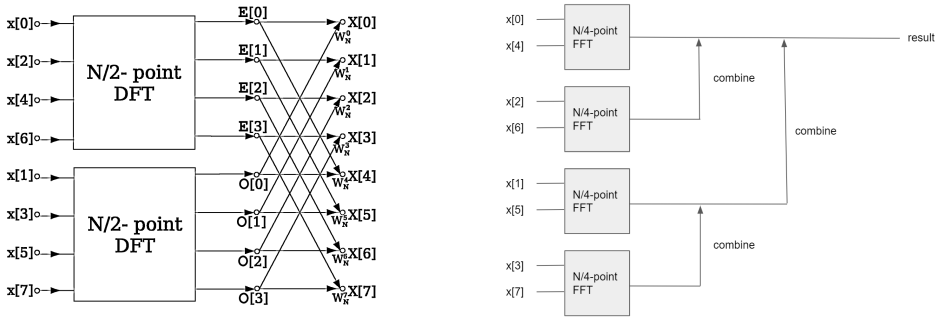
Similarly, we can then look at $R = A \cdot M$, as shown in 2b. Take an arbitrary row from R, say we take the r-th row. Let row r in R be γ and column r in A be α , we can see that the following equation is also a Fourier transform equation. We can vectorize this calculation for every row of A with SIMD since they all perform the same instructions.

$$\gamma[z] = \sum_{a=0}^{N} \alpha[a] * M[a,z] = \sum_{a=0}^{N} e^{2\pi a(z-N/2)i/N} * M[a,z]$$

Now we would like to introduce Fast Fourier Transform and our implementation of parallelized FFT with MPI in the following sections.

4.2 Parallel FFT with MPI

As mentioned in the previous section, FFT can reduce the computation needed using dynamic programming. Now we present a parallelize version of FFT using Message Passing Interface that could further reduce the computation time.



(a) FFT with size 8 on 2 cores, borrowed from FFT

(b) Parallel FFT with size 8 on 4 cores

Figure 3: Parallel FFT implemenation

We start by consider a simple situation 3a. Here we have input x of size 8 and 2 cores. We could let the first core to compute E[0:4] using even terms of x, and let the second core to compute O[0:4] using odd terms of x. Then they can exchange E and O to each other and calculate all X terms. However, in our implementation, there is no need to compute X terms on both cores since we only need one core to report the result. Therefore, we only need the second core to send O to the first core and halt, while the first core need to compute all X after it receives O from the second core.

Now, let's consider a more complex case 3b where we have 4 cores instead of 2. Each core first uses FFT to compute 1/4 of all inputs. Then the communication starts. Core 4 sends its result to core 3 while core 2 sends its result to core 1. Then core 4 and core 2 hang until further notice. Core 1 and core 3 combine the received data with their own calculations. Core 3 sends its combined data to core 1. Core 1 performs another combination of data and return it as the results.

As you can see, such implementation is recursive with the assumption that the size of input data and number of cores are both power of two. Since there are two FFT stages in our implementation and the first result of FFT is passed into the second, the root core broadcast its first result to all the other cores to make the calculate consistent. While the data is very large, such parallel algorithm should become more efficient since the communication cost in MPI becomes marginal compared to the great gain of computation speed from parallelism.

4.3 CUDA Acceleration

As mention in the earlier section, we find that precalculate H matrix allows us to perform FFTs. However, computation of H matrix is also very expensive. When working with multiple cores for

MPI, it is a better practice if we can efficiently compute H matrix in root core and broadcast it to the other ones. Luckily, such computation can be parallelized with CUDA. Recall that:

$$H_{a,b} = \bar{h}(a, b, t), a \in (0, N), b \in (0, N)$$
$$\bar{k} = (2\pi a/N, 2\pi b/N)$$
$$\tilde{h}(\bar{k}, t) = \tilde{h}_0(\bar{k}) \exp(i\omega(k)t) + \tilde{h}_0^*(-\bar{k}) \exp(-i\omega(k)t)$$
$$\tilde{h}_0(\bar{k}) = \frac{1}{\sqrt{2}}(\xi_r + i\xi_i)\sqrt{P_h(\bar{k})}$$
$$P_h(\bar{k}) = A\frac{\exp(-1/(V^2/Lg)^2)}{k^4}\|\hat{k} \cdot \hat{\omega}\|^2$$
$$\omega(k) = \sqrt{g\|\bar{k}\|}$$

We can see that for any pair of (a,b), the $\bar{h}(a, b, t)$ does not require any information about other points. Therefore, we can make a CUDA kernel with N block per grid and N thread per block to parallelly compute $H_{a,b}$. Note that normal distribution sampling for variables ξ_r and $i\xi_i$ in python cuda kernel seems to be problematic. We solve this by passing in a large set of normal distribution samples into the kernel.

Since we are using a mix of CUDA and MPI for parallelization, making sure they do not conflict with each other becomes very important. We only run CUDA on root core of MPI and broadcast the H matrix to other cores.

5 Results

5.1 Parameters

There are some environment parameters in our model that can be tuned to draw more realistic water waves visually. Those parameters would not affect the computation cost in any way.
(1) W_d: wind direction
(2) V: wind speed
(3) A: wave amplitude
(4) (μ, σ): wave randomness

Also, there are some performance parameters that basically determine the speed of our program.
(1) N: sample size, we need adjust the height of N^2 points in our mesh of square tile every frame
(2) T: frames, how long we want our animation to be
(3) M: method: which is the method to run our code
(4) C: number of core used in Parallel-MPI methods

There are a total of 5 methods we implemented.
(1) BF: Brute Force without any vectorization

(2) DFT: Vectorized Discrete Fourier Transform

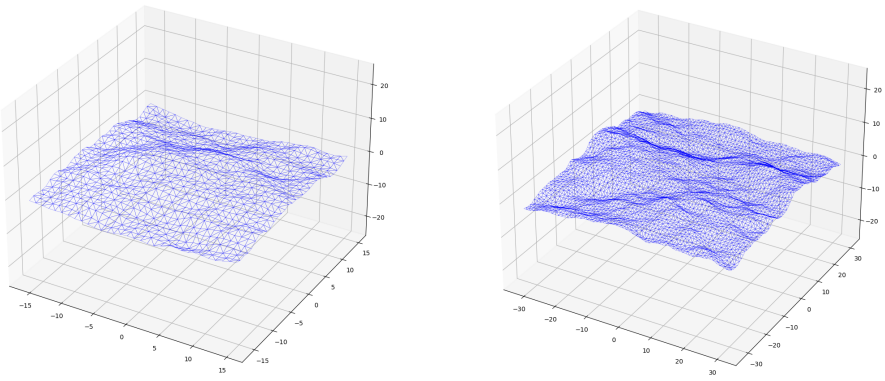
(3) FFT: Vectorized Fast Fourier Transform

(4) FFT_P: Vectorized MPI-Parallel Fast Fourier Transform

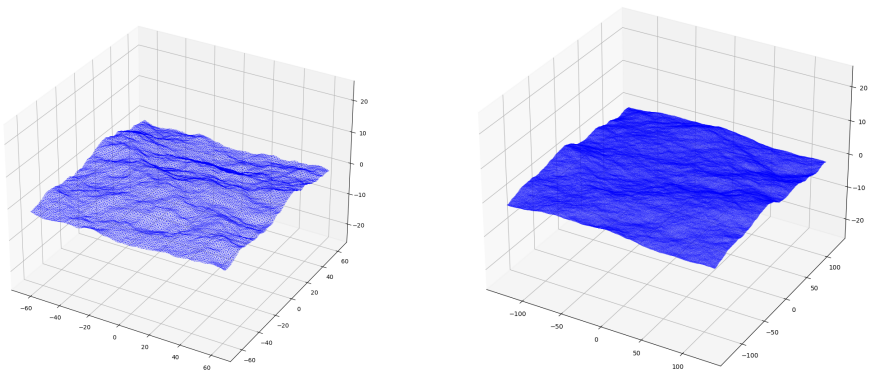
(5) FFT_PC: Vectorized MPI-Parallel Fast Fourier Transform with Cuda Acceleration, the best and default method of our program

5.2 Experiement 0: Qualitative Evaluation

In this experiment, we would like to see how our animation looks like for different sample sizes (N = 32, 64, 128, 256). We carefully tunes the environment parameters so they all roughly looks the same. You are strongly encourage to look at the gif of those animations in our github pages. All animations shows beautiful realistic water waves. The screenshot of those visualization results4 are shown below.



(a) N = 32 and N = 64



(b) N = 128 and N = 256

Figure 4: Lake Visualization with differnt sample sizes

5.3 Experiment 1: Parallization Speedup

In this experiment, we would like to examine the speed up we gain from using MPI-Parallel FFT, with or without CUDA. We setup the sample size to be 256, frame to be 20 and fix all the environment parameters. We change the number of cores used (C = 1, 2, 4, to 8) and record the computation time needed for two methods: Vectorized MPI-Parallel Fast Fourier Transform without CUDA and Vectorized MPI-Parallel Fast Fourier Transform with CUDA. Then, we compute the speedup based on the computation time. As shown in figure5, we have two important observations. First, there are a huge performance gap between two methods. The method with CUDA implementation reaches more than 2 times speedup compared to the method without CUDA. Second, the computation time decreases and speedup increases sublinearly as the number of cores increases for both methods, with and without CUDA acceleration. The sublinear speedup is caused by heavier communication cost between cores as the number of cores increases. Also there are additional data transfer cost into and out from GPU for CUDA implementation that potentially limits the speedup.

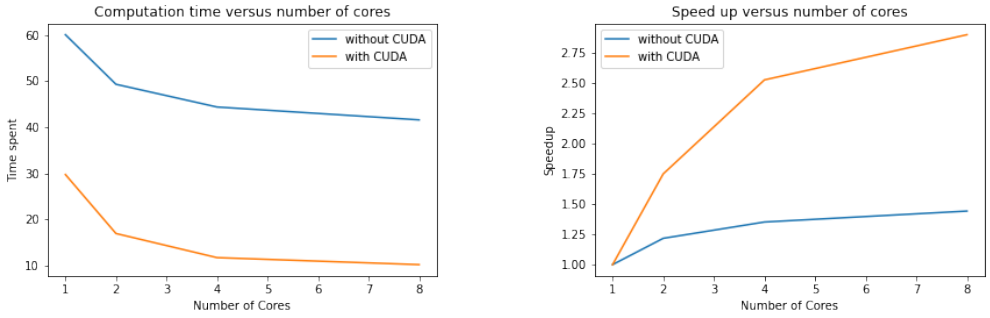


Figure 5: Experiment 1

5.4 Experiment 2: Computational Complexity

In the experiment, we want to evaluate the computational complexity of our implementation with respect to sample sizes. Note that the sample size here is different with problem size. With sample size of N, the problem size is N^2. We set number of cores to be 8, frame to be 20, and fix all environment parameters. We change the sample sizes (N = 32, 64, 128, 256) and record the computation time on two methods: Vectorized MPI-Parallel Fast Fourier Transform without CUDA and Vectorized MPI-Parallel Fast Fourier Transform with CUDA.

In the following graphs, we show that the computational complexity of both methods are roughly linear with respect to problem size. This should be considered as a great achievement since the brute force method has quadratic computational complexity with respect to the problem size. Another observation is that the method with CUDA is at least twice faster than the method without CUDA.

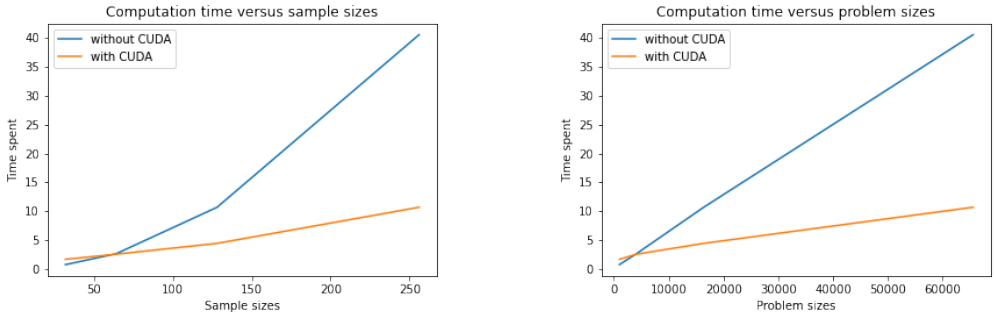


Figure 6: Experiment 2

5.5 Experiment 3: Method Efficiency

This experiment compares the efficiency of five methods while keeping all the other parameters as constant. However, since the unvectorized brute force method is too slow, we would like to examine it separately.

To show the effect of vectorization, we run the experiment with the unvectorized brute force, Discrete Fourier Transform, and Fast Fourier Transform method 7. For the sake of time, we run the experiment for one frame under a very simple setting with sample size of 16 and constant environment parameters. We can see that the brute force method is more than 100 times slower than the vectorized Discrete Fourier Transform and Fast Fourier Transform solution. M Since our input size is not large enough, the problem does not scale well enough. Therefore, FFT is slightly slower than DFT, but both of them are much faster than the unvectorized method.

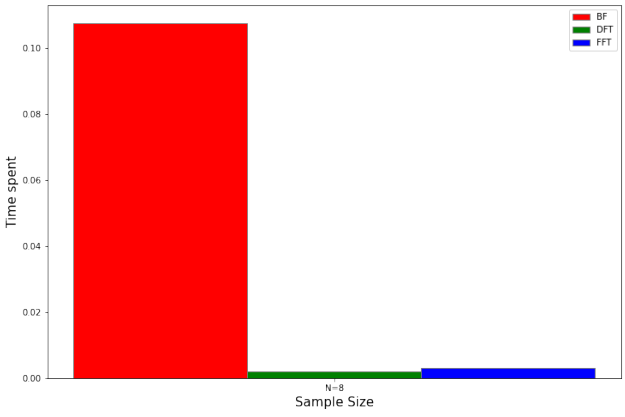


Figure 7: Time with unverctorized BF solution, DFT, and FFT.

Then we conduct a new round experiment 8 with four methods with verctorization for sample size of 256, frame of 20, and constant environment variable. As expected, the methods from lowest to the fastest are: DFT (Vectorized Discrete Fourier Transform), FFT (vectorized Fast Fourier Transform),

FFT_P (Vectorized MPI-Parallel Fast Fourier Transform), FFT_PC (Vectorized MPI-Parallel Fast Fourier Transform with CUDA Acceleration).

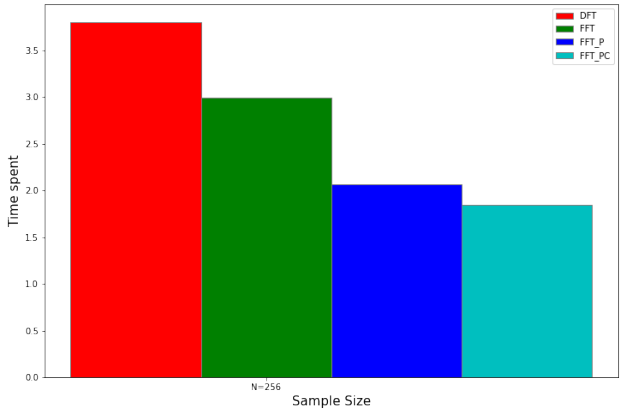


Figure 8: Time with DFT, FFT, FFT_P, and FFT_PC

5.6 Experiment 4: Execution Time Breakdown

In this experiment we hope to examine the execution time breakdown of MPI and CUDA in order to find possible improvements of the project. We only use FFT_PC method and fix every parameter except number of cores. The following graph 9 shows the percentage of execution time for the CUDA task and the MPI task. We find that when there is no MPI parallelization, CUDA task takes half of the execution time. As number of cores increase, the MPI task becomes faster so that CUDA task takes more percentage of execution time. We plan to improve CUDA algorithm as it might become the bottleneck of the program.

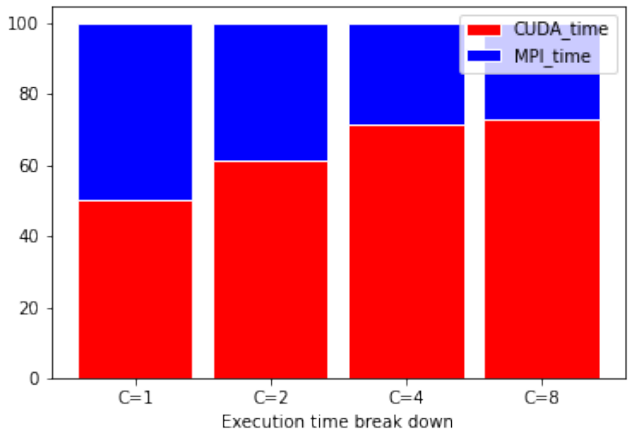


Figure 9: Execution time breakdown

6 Work Assignment

Peilin Rao [60% distribution] did computation of the math/physics on the model, wrote most of the code, designed good vectorization methods and MPI algorithm with FFT, implemented CUDA acceleration and wrote first half of the report.
Zheng Zhong [40% distribution] did research on the different approaches to the original problem. She subtracted important information from papers about the statistical model, helped with MPI implementation part of the program, helped with debuging, analyzing problems, and suggesting fixes. She also wrote the last half of the report.

References

Cooley–tukey fft algorithmg. URL https://en.wikipedia.org/wiki/Cooley%E2%80%93Tukey_FFT_algorithm.

Jerry Tessendorf. Simulating ocean water. pp. 26, 2004.