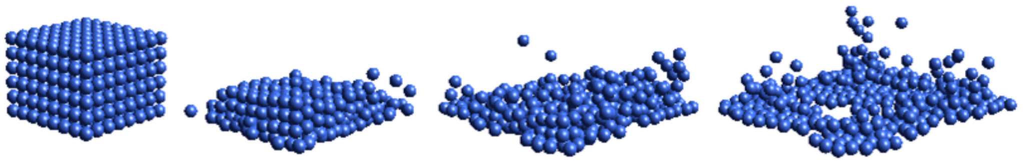
Parallel Fourier Parameterization and Impulse-based Cluster Collisions in Incompressible Particle-based Fluid Dynamics

Shane Transue and Shannon Steinmetz
University of Colorado Denver
Department of Computer Science and Engineering



Abstract

In this work we introduce and construct a framework of parallel algorithms that facilitate the parameterization and simulation of incompressible particle-based fluids. This parallel simulation framework is composed of three components: (1) the parameterization of the initial states of the simulated particles derived from Fourier analysis, (2) a set of discrete and continuous collision detection algorithms for particle and convex object interactions based on a newly introduced cluster-based event system, and (3) an impulse-based collision response algorithm for rigid-particle interaction. In the implementation of this framework we utilized both grouped SIMD facilitated through the Compute Unified Device Architecture (CUDA) and MIMD techniques through OpenMP to provide parallel implementations of efficient algorithms for each of these components. The collected results are demonstrated through the parameterization of waveforms to drive particle visualizations and the interactive simulation of particle-based fluid animation. Through the parallel implementations of our algorithms, we have been able to reduce our overall simulation time by approximately 80% demonstrating a 5.37x increase in performance compared to the sequential implementation.

Keywords: Fourier parameterization, physically-based simulation, parallel computation, continuous collision detection, impulse-based collision resolution

1 Introduction

Representing a well established domain within computer graphics, the simulation of particle-based fluid dynamics is a form of simulation that can be coupled with the power of parallel processing for the purpose of vastly improving the performance of large-scale particle systems that can be initialized using parameterized waveforms to approximate the simulation of non-viscous fluids [1] for the generation of abstract visualizations. In the implementation of this parallel particle simulation framework, we present several parallel algorithms that facilitate the parameterization and physical simulation of particles with the aim of improving the overall temporal performance of the simulation compared to the equivalent sequential implementation. By specifically looking at GPU stream processors and MIMD architectures, variants of many sequential algorithms in particle dynamics and collision detection/resolution can be implemented within these parallel constructs. This allows us to develop efficient parallel algorithms for rendering complex visualizations using standard features presented by modern parallel architectures.

Through our research we have identified that while some of the required algorithms lie within the embarrassingly parallel domain, others require several additional algorithms to formalize them within a parallel context. Considering the potential performance gain derived from implementing embarrassingly particle-based simulation algorithms, we introduce alternative techniques that are required for ensuring that additional non-trivial algorithms for parameterization and collision resolution can be implemented within the parallel computational domain. This includes the utilization of row/column variant parallel matrix multiplication for waveform-based parameterization and cluster-based contact modeling for parallel collision resolution. Utilizing these techniques, we have illustrated that almost all of the algorithms presented in our simulation domain can be implemented in parallel and provide a reliable improvement in performance.

The development of our parallel particle-based fluid simulation framework is defined by the use of the Compute Unified Device Architecture (CUDA) and the OpenMP parallel MIMD execution model through the C++ programming language. In this work we present both the sequential and parallel versions of our algorithms for temporal analysis in the execution time of the physical simulation. The basis of our analysis is the comparison between the sequential and parallel versions of our algorithms. To effectively perform this analysis we utilized several different NVIDIA GPU-based systems including personal computers (for visualization rendering) and the Hydra Cluster within the Parallel and Distributed Systems Lab. These systems represent the primary hardware that was targeted in this research.

Through the implementation of these parallel algorithms in waveform parameterization, particle dynamics, and collision detection/resolution we have constructed the ability to visualize the proposed particle-based fluid dynamics and also illustrate that these algorithms outperform the equivalent sequential algorithms on a variety of hardware. The implementation of our parallel particle-based simulation framework is completely independent and does not have any library dependencies (with the exception of GLUT for the OpenGL viewport management if the visualization source is used). In the implementation we have also included various demonstrations of the resulting parallel algorithms both in visualization and in temporal analysis. In this paper we present the introduction of these parallel algorithms and the technical analysis of the runtimes reported by each algorithm on various forms of hardware. The results of our work are presented in the form of the improved parallel execution times as well as the resulting recorded visualization that represents the original intent of this work.

2 Approach Overview

Parameterized control of physically-simulated particles exhibiting non-viscus fluid behaviors is derived from the correspondence of several algorithms. Specifically, we define the physical pipeline that our implementation exhibits through the introduction of four stages: (1) Parallel Fourier Parameterization, (2) Parallel Collision Detection, (3) Parallel Impulse-based Collision Resolution, and (4) Parallel Contact Resolution. These components form the basis upon which our visualization and analysis is based. Here we provide a brief overview of each of these components and how they relate to the physical states illustrated by the particles within our simulation.

Parallel Particle Simulation The introduction of our parallel particle simulation is based on the simple definition of a physical particle state p based on the position, velocity, mass, and external force of each individual particle. In the formulation of our simulation we assume that there are n particles that are simulated in parallel based on the conventional initial state problem used to define the dynamic behavior of the particle in 3D space over time. Particles within the simulation represent the core state of the simulation and all presented algorithms operate by modifying the physical state of each particle. This includes parameterization, collision detection and resolution.

Parallel Fourier Parameterization Parameterization of simulated particles based on waveform analysis provides an extensive number of ways that sonic waves can define the physical behavior of a simulated object. Based on performing Fourier analysis on a set input data source, we have formed a generalized parametrization algorithm that extracts the initial parameters of the simulated particles and uses that information to define the behavioral characteristics of the particles through the simulation. In our implementation this parameterization function is executed in parallel and initializes the physical states of all particles within the simulation. This forms the initial state of our formal initial state problem presented in our particle dynamics formulation.

Parallel Collision Detection Representing a large component of the physical pipeline, the process of performing collision detection is an inherently time consuming process. Theoretically we observe that the process of analyzing all possible collision pairs within a simulation of n particles is in the order of n^2 operations. Therefore to significantly increase the performance of this required process, typically two alternatives are proposed: (1) reduce the number of potential collision checks through a pruning algorithm and (2) perform the collision detection process in parallel. In our implementation we utilize both of these approaches to significantly reduce the execution time of the collision detection algorithms we have implemented. Specifically we look at how we can use displacement-based potential collision pruning, discrete collision detection (DCD) with singly-linked collision chains, and continuous collision detection (CCD) generalized for all convex shapes using conservative advancement (CA).

Parallel Impulse-based Collision Resolution The resolution of inter-particle collisions must be performed to ensure the physical correctness of the implemented simulation. To effectively resolve the collisions that are detected during the simulation, we implement an iterative impulse-based collision resolution algorithm. The implementation of an iterative impulse-based resolution method is non-trivial in the parallel computation domain. This is because this process cannot inherently provide the correct result in parallel, therefore to introduce the parallel component in this algorithm, we introduce a parallel cluster-graph collision resolution technique. This technique allows us to perform several iterative impulse resolution loops based on the connectivity exhibited between the simulated particles. The formal proof of correctness for this technique is defined within our singly-linked collision detection algorithm that ensures collision clusters through transitivity.

Parallel Contact Resolution Due to the natural external forces acting upon particles during a simulation (ex. gravity) we also provide a parallel contact resolution algorithm that prevents interpenetrations between simulation time-steps. Building upon the cluster-graph introduced in the parallel collision resolution algorithm, the parallel implementation of the contact resolution algorithm also utilizes these cluster contacts to iteratively resolve resting contacts between simulated particles. This iterative process ensures that resting contact constraints are upheld during the simulation time-step.

3 Parallel Particle Dynamics

The simulation of individual particles within a physical model is inherently computationally inexpensive. This is specifically true for the dynamics model that can be utilized for the simulation of s Newtonian-based particle model. The simulation is numerically defined as an initial state problem that depends only on the initial velocity, initial position, external force, and mass of the simulated objects. Additionally we define the time-step of the simulation as h. This constant defines the rate at which the simulation is progressed (typically the time-step value is $\frac{1}{30}$ or $\frac{1}{60}$ for 30 or 60[fps] respectively). Here we define the state of a particle P to represent these values: $p = m, x, y, z, \vec{v}, f$. From this primitive form we can define the initial state that provides the basis of our physical simulation given the mass of the particle and the external force acting upon the particle:

Eq. 1 $$\frac{\mathrm{d}\vec{V}(t)}{\mathrm{d}t} = \frac{\vec{F}(t)}{m}, \vec{V}(t_0) = \vec{v_0}$$

Similarly we utilize the new velocity information derived from 1 to define the new position of an individual particle during the simulation. The solution to both of these initial state problems for all n particles will completely define the new physical state of simulation for the next time-step.

Eq. 2 $$\frac{\mathrm{d}X(t)}{\mathrm{d}t} = \vec{V}(t), X(t_0) = x_0$$

The approximation of these solutions is obtained using Euler's method for forward integration (here we focus on the parallelization not accuracy). From the discretization of the simulation, the constant time-step and Euler's method we derive the new velocity of each particle using Equation 3.

Eq. 3 $$\vec{v_{i+1}} = \vec{v_i} + \mathrm{h}\frac{\vec{F}(t_i)}{m}$$

Using the same discretization process with Euler's method we can also efficiently derive the new position of the particle by forward integrating again using Equation 4.

Eq. 4 $$\vec{x_{i+i}} = x_i + \mathrm{h}\vec{V}(t_{i+1})$$

In this implementation we present a common method for the approximation of non-viscus fluids by utilizing an extensive number of independent particles. The objective of this technique is to introduce a large number of particles into a simulation to naturally derive fluid-like behavior. We classify this behavior as physically *Premoze03*, but not physically accurate. The physical model of these particles is derived from the original impulse-based simulation technique proposed in [2] and is directly related to techniques introduced in [3]. Here we note that it is possible to improve the visual fidelity of the resulting fluid surface utilizing the standard Marching Cubes algorithm [4], however it does not affect the parallelization of the proposed simulation algorithms.

4 Fourier-based Parameterization

The introduction of the initial state problem presented in the previous section is explicitly defined by the analysis of waveforms provided from an external data source. Based on the formulation of our particle system, we define a Fourier-based transformation that defines the initial position, velocity, and mass of the simulated particles. Our particle system is represented by a set, P, where $p \in P p = m, x, y, z, \vec{v}, j$ such that x, y, z are the current position value m is the mass of the particle and j is a collision property which will be discussed later. We use $\vec{v}$ to describe the current velocity of the particle at any time step. The particles initial values must be constructed from some properties, thus we have chosen to utilize the frequency values of a sound wave as the initial parameters. In order to extract the frequencies we must utilize a transform from the time domain to the frequency domain. Admittedly there are better techniques, however we found it challenging to parallelize the Discrete Fourier Transform utilizing its result as our particle values.

4.1 Discrete Fourier Transform

The *DFT* is defined by $X_k = \sum_{t=0}^{N} X_t e^{-i2\pi kt}$ where k is a particular frequency and t is a time value over N samples. We implemented a parallelization of this algorithm using a very simple equation from the work of Tevs et al. [5]

$$\textbf{Eq. 5}\ \ F_{tk} = e^{-i2\pi kt}$$

This equation provides us with a blueprint for performing the transform in a sequence of matrix multiplication stages. The exponent matrix in 5 is the second half of the summation but we still require a coefficient matrix containing all the original amplitude values of a sound wave. This matrix is constructed by aligning the entire sample data for each row to the respective column in the exponent matrix via $C_k = X$. We now have the tools to perform the DFT with a simple matrix multiplication $C * F = T$ where $C, F, T \in \mathbb{R}^{NxN}$ and each row T_k contains a sample space in which each value is a correlation to the respective frequency for that time. This data will become our parameterizations for the simulation particles.

4.2 Parallelization of the DFT

We face a serious issue with regard this this technique and our experimentation has demonstrated that although it is mathematically sound it does not lend itself well to the software implementation. This technique requires at least 1.4GB of RAM for the three matrices required to transform a single sample. A sample ranges from 22Khz to 44Hhz on average (which is standard for most sound files). Foregoing this limitation we proceeded to successfully parallelize the transform and disregard the memory footprint due to the fact that the initialization of our simulation is a one time only event. Our parallelization takes advantage of three separate techniques using a *sequential, CUDA, OMP* approach. We developed the sequential technique using a standard matrix multiply, we then developed a CUDA kernel multiplication which leverages $N/32$ blocks at 32 threads per block, the exact size of a warp plus the remainder $N\%32$ elements in the case our matrix is not 2^k sized.

It should be noted that our sizes are 1K and not the original 22K or 44K, this is due to the fact that our simulation was struggling to render such a large amount of data and resolving this issue will be done as part of our future optimizations. It is also worthy to note that the construction of the DFT coefficient and exponent matrices had to be done in a careful way. The sample data is organized as a single vector so in order to properly populate the matrices in both row and column order we devised an iteration parameter $0 \leq t \leq N$ which can place the correct value into a one dimensional array at the correct two dimensional column ordered matrix location. To index any position we use $M_{ij} = M[i+j*M]$ therefore the following equations provide us with the ability to iterate over a sample vector and properly populate the 3D matrix in a row ordered, and column ordered fashion. It is noteworthy that the equations are designed to populate a three dimensional matrix should we decide to take advantage of such construct in the future.

$$\textbf{Eq. 6}\ \ R_{ix} = \frac{t}{M}\,\%\,M + (t\,\%\,M) * M + \left\lfloor \frac{t}{MN} \right\rfloor MN$$

$$\textbf{Eq. 7}\ \ C_{ix} = (t\%M) + (\frac{t}{M}\,\%\,M) * M + \left\lfloor \frac{t}{MN} \right\rfloor MN$$

We now must produce the actual parameterizations for the particles within the simulation. In order to ensure that values do not fall outside an acceptable range we use a simple normalization equation $R(f, min, max) = min + (f/max - min) + max - min$ which ensures $min \leq f \leq max$.

Item	Value
`fp`	`rf/RFMax`
`fp2`	`rf/LastRF`
v_x	`R(fp,VMax,VMax)`
v_y	`R(fp - Last RF,VMin,VMax)`
v_z	`R(rf,VMin,VMax)`
`Position`	`R(random value,PosMin,PosMax)`
`Mass`	`R(fp2,MassMin,MassMax)`

5 Parallel Collision Detection

Producing the correct fluid-based behavior within the simulation particles is completely dependent upon the process of detecting and resolving collisions between the particles. Collision detection occupies a large amount of time during the physical simulation time-step and is commonly reduced through the reduction of potential collisions through the process of culling (or pruning). Additional considerations for collision event representation and parallelization can have a significant affect on the n^2 potential collision checks that must be performed for each simulation time-step. In this section we introduce several techniques to reduce the overall collision detection processing time using these techniques.

Parallel Discrete Collision Detection (DCD) The basis upon which we implement all of our collision detection techniques is based on the original n^2 checks between the particles within the simulation. Since our particles are represented by spheres with radii r, we can deduce the intersection between two individual particles during a simulation time-step. To ensure this process takes the minimal amount of time (roughly 8[ms]), we devised a technique and a partial proof of correctness which allows us to represent particles in a system with a forwardly linked list. Normally, collision detection requires that one creates a mapping where each particle would list all the other particles it collides with. This has an upper bound of N^2 memory use as well as performance. With this in mind we devised a system in which each particle only requires a reference to the first particle it collides with. We can do this by assuming that any particle that collides with another is transitively associated with any particles that has collided with and so on. We have the start of our proof:

Defn 1 *Let $S = p_1, p_2, ..., p_n$ represent the set of particles in the system and let $p \equiv p_0$ be defined by $|p - p_0| \leq 0$.*

If $p \equiv p_0$ and $p_0 \equiv p_1$ then $p \equiv p_1$ by the transitive property. From this we can conclude that if any particle is associated to another other particle we only need represent a forward linking to represent all possible collisions. In order to prove equivalence we must show that $|p - p_0| \leq 0$ is reflexive, symmetric and transitive.

Since S is a vector space it is commutative therefore $|p - p_0| \leq 0 \Rightarrow |p_0 - p| \leq 0$.

Since $|p_0 - p_0| \leq 0 \Rightarrow |0| \leq 0$ our relation is reflexive.

Finally we must show transitivity which is slightly more involved. Let us assume $|p - p_0| \leq 0$ and $|p_2 - p_0| \leq 0$

$$\text{Then } |p - p_0| \leq |p_2 - p_0|$$

Without loss of generality we can show

$$|p - p_0| \leq |p_2 - p_0| \Rightarrow$$

$$p^2 - p_0^2 - p_2^2 + p_0^2 \leq 0 \Rightarrow$$

$$\sqrt{p^2 - p_0^2 - p_2^2 + p_0^2} \leq \sqrt{0} \Rightarrow$$

$$|p - p_0| \leq 0$$

We therefore can claim that our relation is an equivalence relation, therefore this will provide an adequate representation of the collisions detected using this method. Because of our theorem we are able to be confident that our structure can adequately represent all collisions, however in the world of animation we can fudge the results to allow for some extra functionality. If we take our proof as a road map we can slightly extend its meaning by substituting a value of ϵ for zero. This nullifies our proof of correctness, however if we set ϵ to a very very small value we essentially maintain the behavior in the eyes of the user. The human eye can't detect very tiny differences and therefore allowing us to perform collisions using a bounding sphere of extremely small size.

```
Particle [] A;    /* Particle Array */

For i:N
  A[i].Collide = −1;

For i:N−1
  For j=i+1: N−1
    If Collides(A[i], A[j])
      A[i].Collisde = j;
```

Next we parallelize our implementation. The pseudo-code above demonstrates the approach to our playfully named *swap-NStack 2000* algorithm. The formulation of this algorithm is embarrassingly parallel and simply uses an OMP for loop to check each element pair going forward.

The sequential execution time for this algorithm runs about $T = \frac{N(N+1)}{2}$. Our parallel speedup is roughly $S_p = T_1/T_p = \frac{N(N+1)/2}{(N+1)/2P} = P$. Additionally, we implemented a full N^2 everything to everything collision algorithm for comparison as well as a CUDA based implementation using the same $N/32$ blocks with 32 threads per block.

Parallel Continuous Collision Detection (CCD) The implementation of a collision detection algorithm also requires additional considerations regarding the objects within the simulation. Since our simulation controls a set of potentially fast moving particles, we build upon our collision detection algorithm from the previous section to provide a continuous variant. While the discrete collision detection algorithm checks the interpenetrations between particles at each individual time-step, our continuous detection algorithm utilizes conservative advancement [6, 7] to check for interpenetrations between simulation time-steps. This will prevent incorrect collision responses and remove the potential for tunnelling. As a requirement of the impulse-based collision resolution algorithm, this CCD technique will provide the instant in time between rendered frames when the particles collide. This is achieved by determining the relative velocity of two particles during the simulation time-step: $v_{rel} = a_{vel} - b_{vel}$ in the surface normal direction: $v_{rel,n} = v_{rel} \cdot \vec{n}$. This operation will completely cancel out the movement of particle b such that when this relative velocity is applied to particle a, we can increment its position to determine if it will collide with particle b. During this process we ensure that the interval at which particle a is moved is fixed. This will guarantee that no collision between these two particles will be missed.

In the parallel implementation of this conservative advancement CCD technique we utilize the same OpenMP parallel for loop. If we consider that the application will require additional functionality for more complex shapes, we can utilize a collision map that is accessed in a synchronized manor by the threads performing the parallel CCD. This allows our technique to be used for all convex objects if the required adjacency information can be tracked using the GJK (Gilbert-Johnson-Keerthi) algorithm. Utilizing this form of the implemented collision detection is generally not suitable for particle collisions due to the existence of a closed form solution for the first contact time in the continuous form.

6 Impulse-based Collision Resolution

Collision Resolution Techniques Throughout the field of computer graphics, several forms of collision resolution techniques have been developed. Namely these techniques include (1) penalty-based methods, (2) constraint-based methods, and (3) impulse-based methods. Recent developments in penalty-based methods produce reasonable results for most simulations, however they still require accurate penetration depth estimation techniques. Typically, penalty-based methods are not utilized where visible interpenetrations degrade the visual fidelity of the simulation. While slight interpenetrations between particles in a fluid-based simulation can be essentially ignored, the resolution process is embarrassingly parallel due to the direct force application of other surrounding particles. Considering constraint-based methods we can largely focus on the system of non-linear equations that requires a set of impulses to properly correct the current collisions in the system. These methods formulate this problem in a form of the Linear Complementary Problem (LCP). When solved for a simulation time-step, the applied impulses will ensure that there are no further penetrations in the next simulation time-step. Parallel implementations of this algorithm exists. The last consideration is to iteratively solve for these impulses [8], hence the iterative impulse-based methods. While this technique is extremely fast, it also presents several challenges: (1) the algorithm may be slow to converge for large systems, (2) the process is not inherently parallel, and (3) requires a CCD method to provide the input state of the particles for each time-step.

Impulse-based Collision Resolution The process of resolving a collision utilizing two impulses is defined by the particles involved in the collision, the mass of each particle, and the collision normal between the two particles. The properties of the particles are immediately known and the collision normal is provided by the CCD algorithm. To define the change in velocity of both particles we introduce the notion of an instantaneous impulse:

Eq. 8 $$J = \int_{t_1}^{t_2} F(t)\,\mathrm{d}t \quad s.t. \quad t_2 - t1 = 0$$

Given that this form can be simplified and expressed in terms of the velocities of the particle and its associated mass we can write this as the following with respect to both particle a and particle b involved in the collision.

Eq. 9 $$P'_{a_v} = P_{a_v} + \frac{J}{P_{a_m}} \cdot \vec{n} \quad P'_{b_v} = P_{b_v} - \frac{J}{P_{b_m}} \cdot \vec{n}$$

The change in velocity is defined by the magnitude of the applied impulse J and the mass of each particle. This will simply calculate the change in velocity that will prevent the collision in the next simulation time-step. Below we define this impulse:

Eq. 10 $$J = \frac{-(1+\epsilon) v_{ab} \cdot \vec{n}}{\vec{n} \cdot \vec{n} (\frac{1}{P_{a_m}}) + (\frac{1}{P_{b_m}})}$$

Cluster-graphs The iterative resolution of the collisions detected during a single time-step in the simulation incurs dependencies between the particles in contact, therefore this process cannot be inherently parallelized. To counteract the introduction of these interdependencies that are incurred during the simulation, we introduce the notion of a cluster graph. A cluster graph is an undirected graph that describes the collision contacts (as a cluster of particles in a contact state) during an individual simulation time-step. Any particle that has one or multiple contacts qualifies as a node within a contact graph and the edges represent the contact. The smallest cluster graph is defined as a pair-wise collision between two particles. From the construction of the set of mutually exclusive contact graphs representing different clusters in 3D space, we can assert that each can be solved independently utilizing the parallel iterative impulse method.

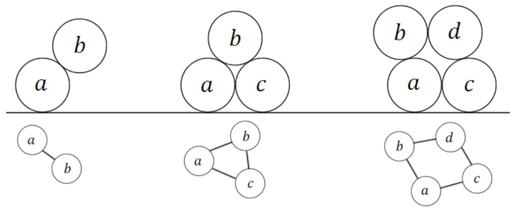


Figure 1: Illustration of three individual particle clusters. The equivalent cluster-graph is illustrated below each potential contact configuration. Each of these groups will be iteratively resolved using impulses.

Due to the sporadic nature of these clusters we have opted to use OpenMP to iteratively solve each of the clusters generated during a simulation time-step. This process is also degrade to a sequential implementation given the correct circumstances: stacks, Newton's Cradle, enclosed volumes, etc. In these instances we cannot avoid the sequential calculation of the individual cluster graph.

7 Impulse-based Contact Resolution

The process of iteratively applying impulses to the particles within the physical simulation will result in preventing penetrations; however this does not consider the case when particles remain in contact with one another during an entire simulation step. When this is the case we must provide an alternative impulse magnitude. The explicit difference between a collision and a contact resolved with an impulse is the duration. A collision impulse is instantaneous and a contact impulse must be large enough to prevent an interpenetration in the next simulation time-step. Therefore we can determine, based on the external force applied to the particle, how far it will penetrate the object it is in contact with. This is determined using the following equation:

$$\textbf{Eq. 11}\quad J = P_m\left(\frac{penetration_{\vec{n}}}{h}\right)$$

The application of this impulse to the affected particle during the current time-step h will completely prevent the interpenetration in the next time-step due to the new velocity of the object incurred by this impulse. The parallel implementation of this process relies upon the iterative process much like the collision resolution algorithm. Additionally this contact resolution technique can also utilize the cluster graphs that are generated for the input to the collision resolution algorithm. Therefore the cluster graphs that are generated are used for both the collision and contact resolution algorithms.

8 Results

The results collected for all of the individual components within our implementation has been relatively positive, in that, while we generally have an incurred overhead in employing parallel algorithms, overall we notice that the performance has been increased when compared to the sequential equivalent algorithms. In this section we introduce the collection of simulation timings that illustrate the runtime performance of several of our algorithms. These results include the execution runtime for each of the following components: (1) the parallel DFT for particle initial state parameterization, (2) the parallel implementation of the collision detection algorithm, (3) the parallel implementation of the cluster-based iterative impulse-based collision resolution algorithm and (4) the impulse-based contact resolution algorithm.

Test Architectures The primary computer architectures that we tested the implementation of our collision response algorithms on are defined by their ability to facilitate both MIMD in a shared memory environment and provide CUDA support through an NVIDIA graphics card. The main desktop computers that we tested our algorithms on were Intel-based Core i7 CPUs with NVIDIA GeForce GPUs. Additionally we have tested our algorithms on the Hydra Cluster within the Parallel and Distributed Systems Lab.

DFT Parallelization Results In the development of the of the parallel DT implementation, we have implemented a very straightforward *Open MP* multiplication technique using the local CPU. This technique leveraged a parallel for loop with default scheduling and chunk size. This algorith produces the initial states that are loaded into each of the particles before the simulation begins. Here are the results of our experimentation again using an Intel Dual Core i7 3.5Ghz custom built system with SDD 32GB RAM and an NVIDIA GTX 650 GPU with 384 CUDA Cores. The results of our experimentation using an Intel Dual Core i7 3.5Ghz custom built system with SDD 32GB RAM and an NVIDIA GTX 650 GPU with 384 CUDA Cores is shown below:

Type	Size	Time (ms)
Sequential	1024	6966.87
OMP	1024	6637.94
CUDA	1024	277.253

Parallel Collision Response Results For the duration of 200 simulation frames, we have recorded the total number of cluster events that occur during each time-step. This illustrates that the parallel collision resolution algorithm performs several parallel iterative impulse-based resolutions during each of these time-steps. This is how we have introduced parallel computation into the process of computing the collision resolution between particles. A note about this result is that the number of cluster events varies drastically during the course of the simulation. Furthermore, we also note that the configuration of the simulation may influence this result (as in the case where all particles are touching and the computation is reduced to the sequential algorithm). The following results in this section have been recorded using a Core i7-4770K with 4 cores and 8 threads (@ 3.5Ghz). Additionally this test bed utilizes 16GB of memory and contains a GeForce 770 GTX (SM Count: 8, Total Cuda Cores: 1536, Device Memory: 2048[MB]).

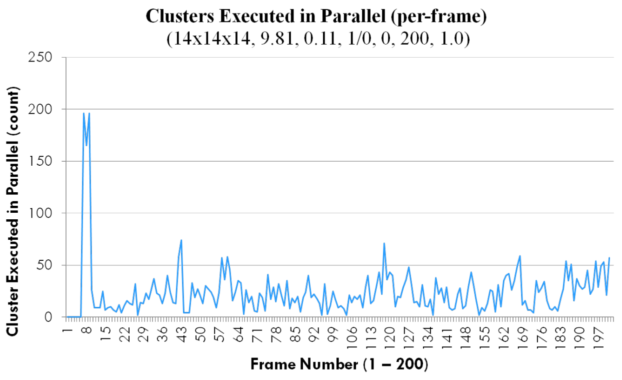


Figure 2: Through the duration of 200 simulation frames, the number of cluster events have been recorded. This illustrates the level of parallelism introduced in the parallel collision resolution algorithm during the simulation. During an individual time-step each of the cluster events are resolved in parallel.

Illustrating the distribution of the frame time for each of the 4 components (dynamics, CCD, collision response, contact response), we clearly see that the collision detection algorithm takes the largest amount of time. Further reducing the cost of the collision detection algorithm would substantially increase the overall performance of the simulation.

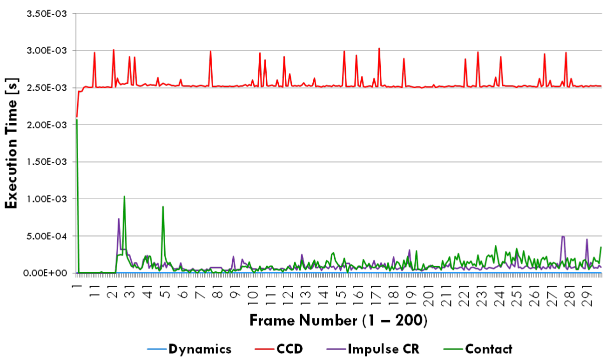


Figure 3: Plot of the execution time of each component in the physical pipeline including the particle dynamics, collision detection, collision resolution and contact resolution. This illustrates that the collision detection clearly dominates our allocated frame time.

We have performed additional experiments using our collision detection algorithm again using an Intel Dual Core i7 3.5Ghz custom built system with SDD 32GB RAM and an NVIDIA GTX 650 GPU with 384 CUDA Cores.

Algorithm	Size	Time (ms)
(Sequential) Brute force	1024	73.84
(Parallel OMP)	1024	.9963
(Parallel CUDA)	1024	90.087
(Sequential) Brute force	10000	4933.54
(Parallel OMP)	10000	90.66
(Parallel CUDA)	10000	119.013

The following plot illustrates the plot above, however the collision detection runtime has been removed. This illustrates the correspondence between the iterative collision and iterative contact algorithms and their associated runtimes. Each of these algorithms utilize the same cluster events, however particles in contact may not necessarily require a collision impulse.

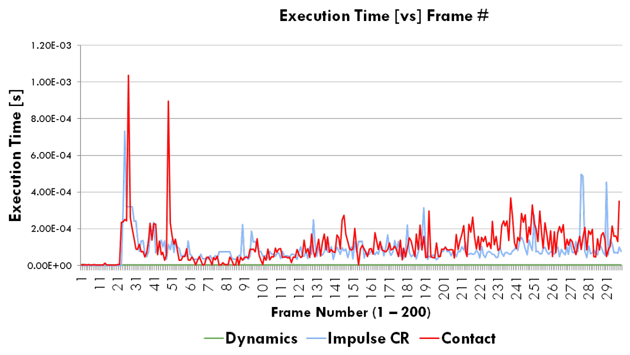


Figure 4: This plot illustrates the difference in the execution time between the iterative collision resolution and iterative contact resolution algorithms. Here we note how negligible the particle dynamics is (almost no contribution to execution time).

Illustrating the execution time of the parallel collision response algorithm based on the number of cluster events in the simulation provides a fairly sporadic runtime analysis. This wild fluctuation in the runtime is determined by the number of cluster events that are generated at each simulation time-step. However in this instance we have illustrated that while the number of cluster events can vary, the parallel implementation of our algorithm outperforms our sequential collision response algorithm.

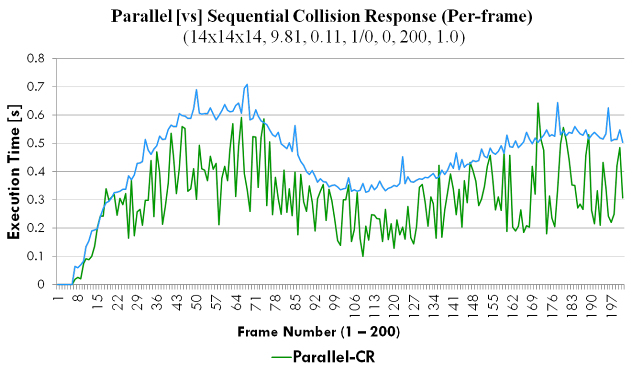


Figure 5: Sequential [vs] Parallel execution times of the collision response algorithm during the course of 200 simulation frames. In this plot we see the execution time of the parallel algorithm is wildly sporadic, however it still provides a higher performance than the sequential algorithm that we have implemented.

Overall we can evaluate the total execution time for both the collision and contact resolution for the total number of particles simulated. In the following plot we illustrate that the total collision response time has been reduced through the use of the parallel cluster-based resolution algorithms.

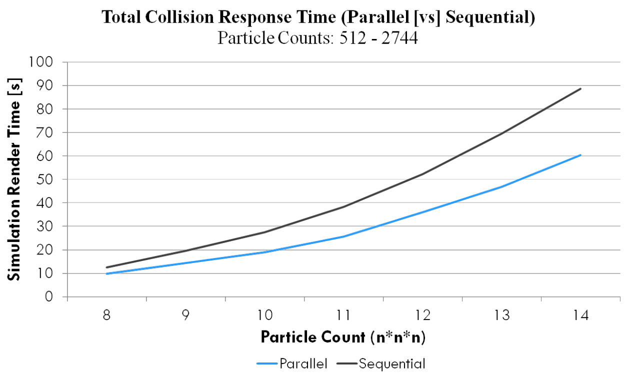


Figure 6: The addition of the collision and contact resolution times for various numbers of particles. As we add particles to the simulation we observe a larger render time required to generate all 200 frames of the simulation.

The following is an excerpt from the provided implementation that executes on the Hydra cluster for 200 frames of the simulation. The results of this execution for 600 particles is provided:

Parallel Algorithm	Time (s)
Collision Detection (CCD)	5.34
Collision Resolution (Impulse)	0.0116
Contact Resolution (Impulse)	0.0141
Dynamics Time	0.004987

We compare this result with the execution of our sequential algorithm implementations to illustrate the benefit gained from the introduced parallelism. The following chart provides the runtimes of the 200 rendered frames on the Hydra cluster. Here we also note the performance penalty of performing the conservative advancement CCD technique for convex objects.

Sequential Algorithm	Time (s)
Collision Detection (CCD)	20.50
Collision Resolution (Impulse)	8.6204
Contact Resolution (Impulse)	0.2363
Dynamics Time	0.03391

9 Conclusion

In this paper we presented several parallel algorithms that have been used to effectively model an approximation of a non-viscus particle-based fluid. These algorithms include the parallel parameterization of the particle initial parameters, the particle dynamics, collision detection and collision resolution algorithms. Each of these parallel implementations provided higher performance than their sequential counterparts, although some of the more complex algorithmic designs lead to a smaller increase in performance; however in general we observed that the implementations of our parallel algorithms were more efficient than relying on the sequential versions.

References

[1] E. Moeendarbary, T. Y. NG, and M. Zangeneh, "Dissipative particle dynamics: Introduction, methodology and complex fluid applications - a review," in *International Journal of Applied Mechanics*, vol. 1, pp. 737–763, 2009.

[2] B. Mirtich, *Impulse-based Dynamic Simulation of Rigid Body Systems*. PhD thesis, University of California, Berkeley, December 1996.

[3] M. Muller, D. Charypar, and M. Gross, "Particle-based fluid simulation for interactive applications," in *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pp. 154–159, Eurographics Association, 2003.

[4] W. E. Lorensen and H. E. Cline, "Marching cubes: A high resolution 3d surface construction algorithm," in *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, vol. 21 of *SIGGRAPH '87*, pp. 163–169, 1987.

[5] A. Tevs, A. Berner, M. Wand, I. Ihrke, M. Bokeloh, J. Kerber, and H.-P. Seidel, "Animation cartography - intrinsic reconstruction of shape and motion," in *ACM Transactions on Graphics*, 2011.

[6] X. Zhang, "Reliable and fast conservative advancement for physically realistic rigid body simulation," in *Simulations, Serious Games and Their Applications*, pp. 105–120, Springer Singapore, 2014.

[7] T. Brochu, E. Edwards, and R. Bridson, "Efficient geometrically exact continuous collision detection," *ACM Trans. Graph.*, vol. 31, pp. 96:1–96:7, July 2012.

[8] J. Bender and A. Schmitt, "Constraint-based collision and contact handling using impulses," in *19th International Conference on Computer Animation & Social Agents*, July 2006.