Chapter 5: Time-Domain Parallelization for Accelerating Cloth Simulation

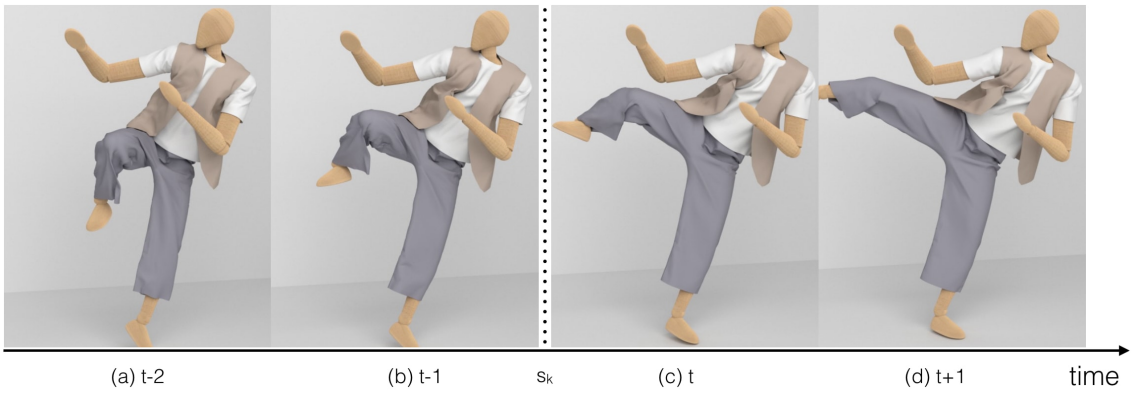


Figure 5.1: Simulated 'Karate' animation using my method. My method parallelizes the simulation workload in time domain using a two-level mesh representation. In the figure, the time domain partition point s_k is between frame t-1 and t, which will be simulated by two different processors. I use an iterative detail recovery algorithm to refine the state of the cloth from low-resolution mesh before the parallel high-resolution simulation begins. As a result, very little visual artifacts can be observed from (b) to (c). In the shown benchmark above, my parallelization method has achieved up to 99x speedup on 128-core systems – an unprecedented level of scalability in distributed CPU systems – compared to at most 47x on a 128-core system [152]. The performance gain is also better than the GPU parallelization [153] on similar benchmarks, while my approach offers the additional flexibility for coupling with adaptively remeshed cloth simulators.

5.1 Introduction

With the proper estimated body parameters, garment geometry, and fabric materials obtained from Chapter 4, it is now possible to synthesize garments on a

given sequence of body motion. One straight-forward way for garment synthesis is using cloth simulation. It offers realistic garment motion and collision-free results at the cost of high latency and throughput. Luckily, given the fact that the try-on system often resides on the cloud, it is possible to accelerate the cloth simulation by making use of manycore and cloud computing.

In this chapter, I propose a novel method for parallelizing cloth simulation. Unlike previous methods, my method divides the workload in *time* domain that minimizes the communication overhead, thereby achieving much better scalability and higher performance gain over previous methods.

The key challenge in time-domain parallelization is to obtain or approximate the simulation states before the time-consuming simulation begins. I use a two-level mesh representation to address this time-dependency issue. Observing that a coarse-level mesh can be simulated at a much higher speed, my method runs a lower-resolution simulation using coarser meshes to approximate the state at each time step. After an appropriate remeshing process, the higher-resolution simulations using finer meshes can be run in parallel. To further refine the simulation results, I propose a practical technique to smooth the state transition from the low-resolution to high-resolution simulations. To recover the lost states, I make use of the coarse-level mesh and run several 'static' simulation steps before the high-resolution simulation starts. Experiments in Sec. 5.6 show that this technique can reduce the visual artifacts between temporal partitions. In order to balance the workload of each processor, I further develop an adaptive partitioning algorithm, which takes into account the varying time consumption of each frame caused by

different contact configurations. I make use of the time measurements of previous frames in both mesh resolutions and determine the partition point based on the current estimation of the total running time.

To sum up, the key contributions of this work include:

- A time-domain parallelization algorithm supporting *adaptive meshes* with minimal communication overhead (Sec. 5.3);
- Load estimation and load balancing techniques that maximize the overall performance acceleration (Sec. 5.4);
- A practical state transitioning algorithm between low- and high-resolution simulations to recover details and ensure the visual quality of the simulated sequences (Sec. 5.5).

On a given set of benchmarks, my method achieves an unprecedented level of scalability in distributed CPU systems when compared to [152, 154]. Its performance gain is also higher than the GPU parallelization [153], while my approach offers the additional flexibility for coupling with adaptively remeshed cloth simulators. I also verify that given sufficient amount of processors, my method can achieve an average performance as fast as the low-resolution simulation, while obtaining simulation results similar to ones using high-resolution meshes. This method can be widely adopted in applications, where runtime performance is much more critical than accuracy, such as rapid design prototyping.

5.2 Related Work

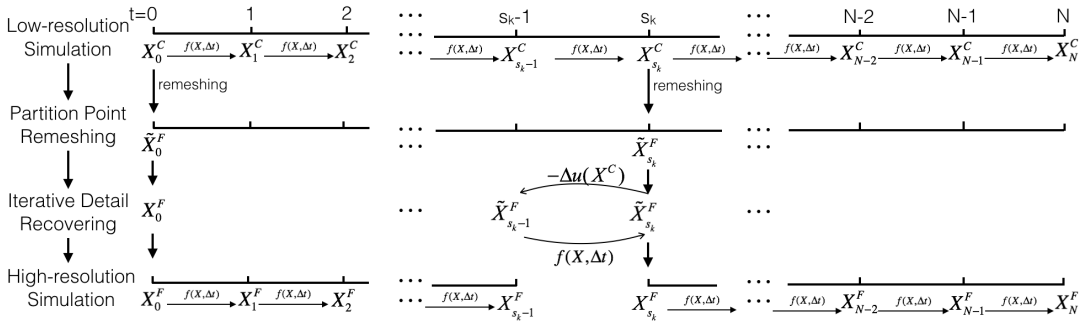


Figure 5.2: An overview of my method. I first simulate the cloth mesh in low resolution, obtaining the approximated states $\mathbf{X}_k^C$. After I select the starting point in time for each processor s_k (Sec. 5.4), I use the upsampling function to generate the initial states $\tilde{\mathbf{X}}_{s_k}^F$ and recover the detail information iteratively (Sec. 5.5). Lastly, I simulate the entire sequence in parallel, given the starting states $\mathbf{X}_{s_k}^F$.

In this section, I survey recent works on cloth simulation, parallelization techniques, and other related acceleration techniques for physics-based simulation.

5.2.1 Cloth Simulation

Simulation of cloth and deformable bodies has been extensively studied for a wide range of applications in different areas, from computer graphics, CAD/CAM, robotics and automation, to textile engineering. Due to their ability to take large time steps, implicit or semi-implicit methods [24, 155, 156, 157] have been widely adopted after the seminal work by Baraff and Witkin [27]. However, most of these works focus on the serial simulation improvement and their runtime performances can be slow. I use one of the state-of-the-art simulation algorithms, ARCSim [74], as the cloth simulator in my prototype implementation, but my parallelization technique does not rely on any specific simulation algorithm.

5.2.2 Time Parallel Time Integration Method

The scientific computing community have thoroughly studied parallelization techniques solving partial differential equations [158, 159, 160]. I refer readers to this survey by Gander et al. [161] for more details. Cloth simulation is similar to the general time-evolution equations. However, there is a gap for these works to be directly applicable. Cloth simulation has coupled other non-PDE factors, such as the collision response due to continuous contacts with the human body. The standard collision response within Physically-based Modeling literature is usually an "empirical" impulse applied mainly on the boundary cases, where the cloth is about to collide with the body or within a pre-defined 'threshold' neighborhood. Traditional solutions [158] use an arbitrary initial guess (e.g.$\mathbf{X}_t = \mathbf{X}_0$) for each of the time step and try to update the overall solution using a fixed point iteration. The discontinuity introduced by collision not only prevents the method from solving the fixed point problem in Newton's method (calculating derivatives of the conditional term determined by variables to be solved), but also prevents most of the collision response algorithm from obtaining stable and correct results (a severe inter-penetration of $\mathbf{X}_t = \mathbf{X}_0$ at time t that can hardly be handled). This special characteristic of cloth simulation makes it challenging to apply methods solving pure integrations (where the solution space is often regular) such as PFASST [158], due to collision-induced discontinuities.

5.2.3 Parallel Cloth Simulation

Several parallelization techniques for cloth simulation have been proposed. [162, 163] proposed GPU-based simulation methods for elastic bodies. [164, 165, 166, 167, 168] proposed different types of spatial parallelization but they all suffer from severe sub-linear scalability due to large communication overhead. [152] improved the work from [169] using Asynchronous Contact Mechanics and reduced the communication by proposing a locality-aware task assignment, which first scaled more than 16 cores. [153] implemented a GPU-based simulation pipeline. Their method has achieved an impressive speedup of 58 times, which is comparable to the performance of my method on a 64-core cluster.

The main difference between other parallelization methods and mine is that I decompose the simulation task in *time* domain. Partitioning in time domain significantly reduces the communication cost in distributed systems, thereby offering a considerable speedup. To the best of my knowledge, my method is the *first time-domain parallelization* algorithm for cloth simulation that can be coupled with *adaptive remeshing schemes*.

5.2.4 Hierarchical Structures and Multi-level Methods

Multi-level algorithms have offered significant performance improvement on various simulation problems. Tamstorf et al. [170] proposed a multi-grid method to speed up the cloth simulation. Bergou et al. [171] developed a tracking solver for rapid interaction in animation. They set up a two-level mesh representation and used

the desired coarse level animation to guide the fine level one by applying constrained dynamics. My method builds on top of their work to ensure the low-res consistency of the results. Recent works [172, 173, 174] generate high-resolution wrinkles from low-resolution cloth. My method is a physically-aware approach; it's more diverse and realistic compared to those work. Mine is more of an intermediate trade-off between time-consuming simulation and physically-unaware wrinkle synthesis. I use a hierarchical mesh representation to approximate the states of the cloth mesh at each time step, before transitioning to computationally expensive high-resolution simulations on fine meshes.

5.2.5 Mesh Upsampling

Mesh upsampling algorithms are widely explored from geometrical approaches [175, 176, 177] to data-driven methods [178, 179]. My method needs a specific mesh upsampling function to transfer the (approximated) state of the simulated cloth from low-resolution to high-resolution. While classic subdivision methods [177] cannot generate high-resolution details, data-driven ones [178, 179] depend largely on the specific configuration in the training data, and as a result, can generate interpenetrations when applying to arbitrary scenarios. For generality, I do not assume any specific upsampling function. Instead, I introduce an iterative detail-recovering approach described in Sec. 5.5 in order to account for the lost details in the low-resolution mesh. In my experiment, I use an adaptive remeshing method in [74] for its flexibility of use and a straightforward, linearly-interpolated subdivision for fast

error computation.

5.3 Overview

In this section I give an overview of my approach. I define the problem formally before I introduce the basic idea of the method.

Problem Statement: Given the initial state of a cloth mesh, $\mathbf{X}_0$ (inclusive of both position and velocity), generate a sequence of cloth states $\mathcal{V} = \{\mathbf{X}_1, \ldots, \mathbf{X}_N\}$ that characterize the cloth interaction with the given environment, using a time step Δt and a simulation function $\mathbf{X}_{k+1} = f(\mathbf{X}_k, \Delta t)$.

Fig. 5.2 shows the overall pipeline of my algorithm. The key idea of this method is to partition the time domain of the cloth simulation rather than the spatial domain of the simulated cloth. In order to obtain the (approximated) mesh state without full simulations, I propose a two-level hierarchy representation. I simulate the cloth mesh $\mathbf{X}^C$ at a coarser level with much lower computation and determine the partition point S (in time) according to the algorithm described in Sec. 5.4 before I simulate the entire high-resolution sequence $\mathbf{X}^F$ at the finer level in parallel.

The fine-level mesh at the starting point of each temporal partition is obtained by the corresponding coarse-level mesh using an upsampling/remeshing function $u(\mathbf{X}^C)$. However, the finer mesh may be quite different from the coarse one after remeshing because high frequency information $\mathbf{X}^D$ is not stored in the coarse-level mesh. Therefore, I design a practical state-transitioning technique to recover the

lost details to the extent possible, before the high-resolution simulation begins. This state-transitioning method will be discussed in Sec. 5.5. I list the notations used in this chapter in Table 5.1.

NOTATION	DEFINITION
$\mathbf{X}_k$	state of the cloth at step k
$\mathcal{V}$	output sequence of states
N	simulation sequence length
Δt	specified time step
$f(\mathbf{X}_k, \Delta t)$	one-step simulation
$f^i(\mathbf{X}_k, \Delta t)$	i-step simulation
$\mathbf{X}^C$	coarse level state
$\mathbf{X}^F$	exact fine level state
$\mathbf{X}^D$	state difference between the two level states
$\tilde{\mathbf{X}}^F$	approximated fine level state
$u(\mathbf{X}^C)$	upsampling function
p	number of processors
S	ordered set of starting points for parallelization
s_j	starting point of the jth processor
K	coarse-to-fine ratio

Table 5.1: **Notations and definition of my method.**

5.3.1 Two-Level Mesh Hierarchy Representation

Ideally I want to divide the whole simulation process into several temporal partitions so that I can simulate each partition in parallel and independently. However, since the mesh state at step k, $\mathbf{X}_k$, is determined by the state at previous step $\mathbf{X}_{k-1}$, I do not know the exact intermediate states until I finish the simulation from step 0 to step k. Here I use the hierarchical mesh representation to address this time-dependency problem. I maintain two sets of simulated meshes, $\mathbf{X}^C$ and $\mathbf{X}^F$, which represent the low- and high-res(olution) simulation states using the coarse- and fine-level meshes, respectively. I can recover the high-res state from the low-res

one by a user-defined upsampling function: $\tilde{\mathbf{X}}^F = u(\mathbf{X}^C)$.

Note that the obtained high-res state from the fine mesh, $\tilde{\mathbf{X}}^F$, is only an approximation of the exact state $\mathbf{X}^F$. But, for simplicity, I assume that $\mathbf{X}^F = \tilde{\mathbf{X}}^F$ in this section. Further state refinement is discussed in Sec. 5.5.

Due to the fact that the simulation using a coarse mesh is significantly faster than the one using a fine mesh, I can obtain low-res states $\{\mathbf{X}_1^C, \ldots, \mathbf{X}_N^C\}$ in a relatively small amount of time. I further choose p starting points $S = \{s_0 = 0, s_1, \ldots, s_{p-1}\}$ in time for p processors, according to my partitioning algorithm to be discussed in Sec. 5.4.1, and run the high-res simulation using the fine mesh in parallel:

$$\mathbf{X}_k^F = \begin{cases} \tilde{\mathbf{X}}_k^F & k \in S \\ f^{k-s_j}(\mathbf{X}_{s_j}^F, \Delta t) & s_j < k < s_{j+1} \end{cases} \tag{5.1}$$

where

$$f^i(\mathbf{X}_k, \Delta t) = \begin{cases} f(f^{i-1}(\mathbf{X}_k, \Delta t), \Delta t) & i > 1 \\ f(\mathbf{X}_k, \Delta t) & i = 1 \end{cases} \tag{5.2}$$

for running i steps of simulation.

5.4 Time Domain Parallelization

In this section I will describe my parallelization technique. I solve the partitioning problem from the simplest case to the most complex one, in order to balance the workload of each processor.

5.4.1 Static Temporal Partitioning

A straightforward approach for the partition problem is to divide the time domain into p temporal segments of the same length:

$$s_j = \lfloor \frac{N}{p} j \rfloor \tag{5.3}$$

Assuming that every simulation step using the fine mesh takes the same amount of time, the overhead of this partition schedule is the time spent in simulation using the coarse mesh. To further simplify the case, I take another assumption that the simulation speed at the low-res level is K times as fast as high-res level. I can estimate the speedup as:

$$\eta_1 = \frac{KN}{K\frac{N}{p} + (p-1)\frac{N}{p}} = \frac{Kp}{K+p-1} \tag{5.4}$$

Note that in the low-res simulation using a coarse mesh there is no need to continue the simulation after I reach s_{p-1}. Therefore, the time spent on low-res simulation is $(p-1)\frac{N}{p}$.

One improvement of the straightforward approach is that I can start the high-res simulation in parallel, as long as the corresponding starting point is ready. Intuitively, I want all processors of the system to finish their jobs at the same time to achieve a good workload balance and the best speedup possible. This objective can be attained by adjusting the starting points so that the processor which starts

earlier takes a longer part to simulate. Taking the same assumption, I arrive at a load-balancing equation:

$$s_j + K(s_{j+1} - s_j) = s_{j+1} + K(s_{j+2} - s_{j+1})$$

Recall that K is the ratio between the high- to low-res simulation time, s_j, s_{j+1}, and s_{j+2} are the starting point for simulation on the processors $j, j+1$, and $j+2$, respectively. This equation yields:

$$s_j = \lfloor \frac{1-q^j}{1-q^p} N \rfloor \tag{5.5}$$

where $q = 1 - \frac{1}{K}$. The speedup can then be expressed as:

$$\eta_2 = \frac{KN}{K(s_1 - s_0)} = K - K(1 - 1/K)^p \approx p - \binom{p}{2}\frac{1}{K} \tag{5.6}$$

This is a tighter bound than Eqn. 5.4, as p approaches to K. The key reason behind the sub-linear speedup is that the overhead ratio to the original computation is $1/K$. In practice, the ratio between high- to low-res simulation time can be controlled by the user and can usually reach 100∼200 using the method described in Sec. 5.4.3, which is sufficient for running on a large distributed system.

5.4.2 Adaptive Partitioning

In the discussion above, I consider K as a known constant throughout the entire simulation process. However, it is highly unlikely that this would be the case. First of all, remeshing in the simulation run leads to a varying number of vertices and thus a dynamically changing size of the linear system. Secondly, the computational cost can vary considerably, even with the same mesh size, due to collision queries. Recent studies [153] show that collision detection and response can take up to 80% of the total running time. Moreover, the difference of per-step runtime is also dominated by the collision response and the size of the adaptive mesh, which are largely related to the object granularity. It has much more impact in the high-resolution than the low-resolution, which K accounts for as well. Therefore, the ratio of high- to low-res simulation time varies and the exact number is usually unknown.

A fixed partitioning scheme can become unstable and sensitive to these variations, resulting in load imbalance. One common solution is to cut down the jobs into more smaller tasks so that the imbalance can be reduced by dynamic job scheduling scheme. This method surely works, but it will have large extra overhead due to job scheduling and required preprocessing time (Sec. 5.5), and extra hand-tuned granularity parameter to optimize the performance. Since I want to avoid any unnecessary computational overhead, I here propose an adaptive partitioning algorithm.

Suppose that I have simulated up to step n using the coarse mesh, when the first high-res parallel simulation with the same starting time has completed m steps,

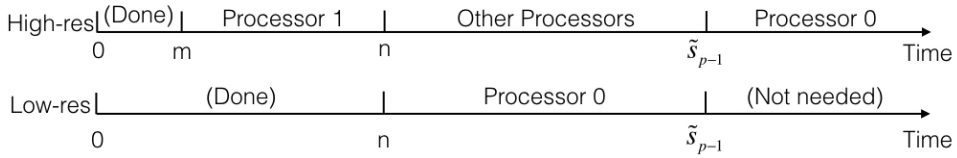


Figure 5.3: Adaptive partitioning Algorithm. I estimate the ratio of high-to-low-res simulation time, $\tilde{K}$, according to the runtime data I observe so far ([0,m] in High-res on Processor 1 and [0,n] in Low-res on Processor 0). The objective is to predict the future running time (marked by 'Processor 0' and 'Processor 1' respectively) to be as close as possible to the actual time.

where $m < n$. Let $T_C(m)$ and $T_F(m)$ denote the running time of the previous m steps using the coarse and fine meshes, respectively. Then, the ratio of the high-to-low-res simulation time, $\tilde{K}$, can be approximated as:

$$\tilde{K} = \frac{T_F(m)}{T_C(m)} = \frac{T_C(n)}{T_C(m)} \tag{5.7}$$

Since these numbers may vary, it is not appropriate to determine the global partition points using current approximations. Instead, I use them to determine if I should perform a cut on step n, i.e. whether n should be s_1 or not. Fig. 5.3 gives a visualization of the process. The objective of the partitioning algorithm is that the total running time on the processor 0, which performs the low-res simulation and the last part of the high-res simulation, is equal to the running time of the current parallel simulation that performs the high-res simulation using a fine mesh from step 0 to step n. This relation can be formulated as:

$$T_C(\tilde{s}_{p-1}) + (T_F(N) - T_F(\tilde{s}_{p-1})) = T_F(n) \tag{5.8}$$

where $\tilde{s}_{p-1}$ is the estimated starting point of the last partitioned segment. I use

the method described in Sec. 5.4.1 to obtain this parameter. I further approximate Eqn. 5.8 to:

$$n = \frac{N}{\tilde{K}} + \frac{\tilde{K}-1}{\tilde{K}}(N - \tilde{s}_{p-1}) \tag{5.9}$$

by assuming stable parameters in the remaining simulation:

$$T_F(j) = \tilde{K}T_C(j) = \tilde{K}T_C(1)j \quad \text{for any j} \tag{5.10}$$

Since n is increasing while $\tilde{K}$ and $\tilde{s}_{p-1}$ can be considered stable compared to n, Eqn. 5.9 can be defined at some point in $1 \leq n \leq N - p$. The remaining cut can be completed recursively. Algorithm 2 shows the pseudocode of this method. $\tilde{K}$ and $\tilde{s}_{p-1}$ here are approximated values used only for this cut. They can vary during the simulation, which will guide my partition algorithm to have adaptive cuts, instead of fixed ones in Sec. 5.4.1.

Algorithm 2: - Adaptive Partitioning

Require: N, p, X_0^C
1: $n \leftarrow 0$
2: start fine level simulation from step 0 on Processor 1
3: **while** true **do**
4: $n \leftarrow n + 1$
5: obtain X_n^C from X_{n-1}^C
6: $m \leftarrow$ steps finished by Processor 1
7: calculate $\tilde{K}$, $\tilde{s}_{p-1}$ from Eqn. 5.5 and 5.7
8: **if** condition of Eqn. 5.9 is met **then** break
9: **end if**
10: **end while**
11: $t_1 \leftarrow n$
12: Control Processor 1 to stop at Step n
13: Recursively partition remaining N-n steps with p-1 processors

In practice, the overall performance using adaptive partitioning is similar to that using static partitioning when the user can manually select the best K value for the simulation scenario. This algorithm generally offers the advantage of dynamically estimating the ratio of the high-to-low-res simulation time, so the user does not need to hand-tune this parameter for the best possible speedup.

5.4.3 Analysis on Performance Scalability

As discussed in the previous sections, the scalability of this time-domain partitioning method for parallel cloth simulation depends largely on the general runtime ratio between the high- to low-res simulation time, K. Since I perform a low-res simulation using a coarse mesh and a parallel one using a fine mesh, the low-res running time is a computational overhead for all processors and thus the speedup before any improvement is $\frac{K}{1+K/p} = \frac{Kp}{K+p}$. The ideal case of perfect workload balance, η_2, is discussed in Sec. 5.4.1, hence the actual performance of Algorithm 2 in a specific scenario, η_3, has the following theoretical bound:

$$\frac{Kp}{K+p} < \eta_1 \leq \eta_3 \leq \eta_2 < K \tag{5.11}$$

Therefore, the higher the K value is, the higher the overall performance gain of my method would be. One common way to increase K is to control the number of total mesh triangles by limiting the smallest possible size of each triangle in the low-resolution level. The other way is to enlarge the time step of the low-res simulation, since it is the common overhead of all processors and should aim for faster speed

rather than smaller discretization errors. A properly chosen large time step can improve the overall performance with minimal impact on the simulation results. With the coarsening techniques in space and time domains, K can be sufficiently large to obtain good scalability in large distributed systems.

5.5 Smooth State Transitioning

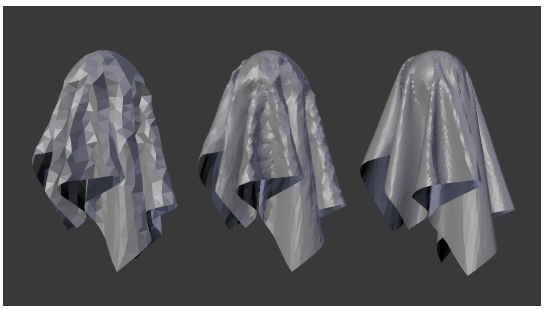


Figure 5.4: An example of the coarse mesh $\mathbf{X}^C$, intermediate mesh, $\tilde{\mathbf{X}}^F$, and the fine mesh, $\mathbf{X}^F$, after iterative corrections.

As mentioned in Sec. 5.4, the high-res simulation state approximation $\tilde{\mathbf{X}}^F = u(\mathbf{X}^C)$ is not the same as the exact state $\mathbf{X}^F$ using the fine mesh, the reason of which is that the high frequency information needed to reconstruct the states of the fine mesh is missing in the estimated states of the simulation using the coarse mesh. Therefore, if I take $\tilde{\mathbf{X}}^F$ directly as the starting state of the parallelized simulation, error $\mathbf{e} = E(\tilde{\mathbf{X}}^F, \mathbf{X}^F)$ will occur, since the high-frequency information is lost. Although $\mathbf{e}$ will vanish as the detail of the mesh is recovered by the simulation, another error will appear at the beginning of the subsequent partition after the end of the current one. (Here I focus on the actual visual effect instead of the L2 distance of each vertex. The error of my specific goal can be defined as the smoothness

of the cloth.) Thus, this error will appear as a 'popping visual artifact' in the final concatenated sequence of the cloth simulation. Fig. 5.4 shows an example of the inaccurate starting mesh (middle) obtained from the corresponding coarse level mesh (left), which causes a popping visual artifact because the error compared to the actual state (right) is large enough to be visible.

One straight-forward method is to apply global smoothing optimization as a post-processing step. However, this space-time optimization is too time consuming to be used in speed demanding applications. As mentioned before, Bergou et al. [171] used constrained dynamics for fine level simulation to match with the coarse level motion. I employ this method to prevent the high-res simulation from diverging too far from the low-res one. However, the high-frequency detail information would be still missing at the transition point. Inspired from the observation that the visual error will be eliminated during the simulation, I propose an iterative refinement technique that can recover as much as possible the high-frequency detail of the cloth from the low-res simulation using the coarse mesh.

5.5.1 Iterative Detail Recovery

Consider the mesh state at the consecutive step points $\mathbf{X}^C_{k-1}$ and $\mathbf{X}^C_k$. The fine-level mesh can be regarded as the sum of the low-frequency coarse mesh and the high-frequency detail:

$$\mathbf{X}^F = u(\mathbf{X}^C) + \mathbf{X}^D \tag{5.12}$$

Assuming that the time step is sufficiently small and the detail does not change

much between two simulation steps, I have:

$$\mathbf{X}_{k-1}^{F} - u(\mathbf{X}_{k-1}^{C}) \approx \mathbf{X}_{k}^{F} - u(\mathbf{X}_{k}^{C}) = \mathbf{X}_{k}^{D} \tag{5.13}$$

The idea here is to approximate $\mathbf{X}_{k-1}^{F}$ using $\mathbf{X}_{k-1}^{C}$, $\mathbf{X}_{k}^{C}$ and $\tilde{\mathbf{X}}_{k}^{F}$. From Eqn. 5.13 I have:

$$\tilde{\mathbf{X}}_{k}^{F} = f(\tilde{\mathbf{X}}_{k-1}^{F}, \Delta t) \tag{5.14}$$

$$\approx f(\tilde{\mathbf{X}}_{k}^{F} - u(\mathbf{X}_{k}^{C}) + u(\mathbf{X}_{k-1}^{C}), \Delta t) \tag{5.15}$$

Note that Eqn. 5.15 can be considered as an updated version of Eqn. 5.14. By subtracting the upsampled change of the state as a backward step and the simulation itself as a forward one, I can compute $\tilde{\mathbf{X}}_{k}^{F}$ iteratively. Algorithm 3 below shows the iterative detail recovery process. I run this algorithm at each of the transition point as a pre-processing step before the high-res simulation begins.

Algorithm 3: - Iterative Detail Recovery

Require: $\mathbf{X}_{k-1}^{C}$, $\mathbf{X}_{k}^{C}$ $(k \in S)$

1: $\tilde{\mathbf{X}}_{k}^{F} \leftarrow u(\mathbf{X}_{k}^{C})$
2: **while** not reaching maximum iteration **do**
3: $\quad \tilde{\mathbf{X}}_{k-1}^{F} \leftarrow \tilde{\mathbf{X}}_{k}^{F} - u(\mathbf{X}_{k}^{C}) + u(\mathbf{X}_{k-1}^{C})$
4: $\quad \tilde{\mathbf{X}}_{k}^{F} \leftarrow f(\tilde{\mathbf{X}}_{k-1}^{F}, \Delta t)$ with constraints introduced by TRACKS [171]
5: **end while**
6: $\mathbf{X}_{k}^{F} \leftarrow \tilde{\mathbf{X}}_{k}^{F}$

5.5.2 Convergence and Continuity

Taking the advantage of the constraint-based tracking solver introduced by Bergou et al. [171], this iterative algorithm can be proved to have convergence guarantee. I show the proof in the following section. It is not guaranteed that the convergence point is exactly the same as the high-res simulation result. However, due to the enforcement of the tracking constraint, the difference compared to the result at the previous step will be $O(\Delta t)$, which means that there will be very little discontinuity and in most practical cases they are invisible.

5.5.3 Proof of Convergence of Algorithm 3

Theorem 1. *Algorithm 3 can reach the convergence point when applying the coarse-level tracking constraints to the system, as long as* $\frac{\partial \mathbf{F}}{\partial \mathbf{X}} = 0$ *for external forces.*

Proof. I assume the whole system is running under the Forward Euler method:

$$\begin{pmatrix} \Delta \mathbf{x} \\ \Delta \mathbf{v} \end{pmatrix} = \Delta t \begin{pmatrix} \Delta \mathbf{v} \\ M^{-1}\mathbf{F}(\mathbf{X}) \end{pmatrix} \tag{5.16}$$

where $\mathbf{F}$ is the force function, and $\mathbf{X} = \begin{pmatrix} \mathbf{x} & \mathbf{v} \end{pmatrix}^T$ is the state of the cloth. Given the assumption that $\frac{\partial \mathbf{F}}{\partial \mathbf{X}} = 0$ for external forces, they have the same contributions for each iteration and are all canceled out by the subtraction $(\Delta u(\mathbf{X}_k^C))$ in Algorithm 3. So I only consider internal forces.

Since I only focus on one high-res simulation step here, I leave off the resolution

superscript and replace the step number subscript by the iteration time. I denote the upsampled coarse-level difference by $\Delta\mathbf{X}_0 = \begin{pmatrix} \Delta\mathbf{x}_0 & \Delta\mathbf{v}_0 \end{pmatrix}^T$. Using the new notation, I have:

$$\begin{pmatrix} \mathbf{x}_i \\ \mathbf{v}_i \end{pmatrix} = \begin{pmatrix} \mathbf{x}_{i-1} - \Delta\mathbf{x}_0 \\ \mathbf{v}_{i-1} - \Delta\mathbf{v}_0 \end{pmatrix} + \Delta t \begin{pmatrix} \mathbf{v}_{i-1} - \Delta\mathbf{v}_0 \\ M^{-1}\mathbf{F} \end{pmatrix} \tag{5.17}$$

I now regard the evolution from $\begin{pmatrix} \mathbf{x}_{i-1} & \mathbf{v}_{i-1} \end{pmatrix}^T$ to $\begin{pmatrix} \mathbf{x}_i & \mathbf{v}_i \end{pmatrix}^T$ as one full simulation step (instead of a backward-forward iteration), and only focus on the velocity equation (since the position can be derived from it):

$$\mathbf{v}_i = \mathbf{v}_{i-1} + \Delta t(M^{-1}\mathbf{F} - \Delta\mathbf{a}_0) \tag{5.18}$$

where $\Delta\mathbf{a}_0 = \Delta\mathbf{v}_0/\Delta t$ is the corresponding acceleration value. Given that the internal forces are negative gradients of the potential energy, I have:

$$\frac{d^2\mathbf{x}}{dt^2} = M^{-1}\mathbf{F} - M^{-1}M\Delta\mathbf{a}_0 \tag{5.19}$$

$$= -M^{-1}\frac{\partial E}{\partial \mathbf{x}} - M^{-1}\frac{\partial M\Delta\mathbf{a}_0 \cdot \mathbf{x}}{\partial \mathbf{x}} \tag{5.20}$$

$$= -M^{-1}\frac{\partial E}{\partial \mathbf{x}} - M^{-1}\frac{\partial E_0}{\partial \mathbf{x}} \tag{5.21}$$

$$= -M^{-1}\frac{\partial \tilde{E}}{\partial \mathbf{x}} \tag{5.22}$$

where I make up a form of potential energy (E_0) with constant gradients to unite the two components.

By computing the dot product with the velocity (of the previous iteration), I have:

$$\frac{d\mathbf{x}}{dt} \cdot M\frac{d^2\mathbf{x}}{dt^2} = -\frac{d\mathbf{x}}{dt}\Big|_{(i-1)\Delta t} \cdot \frac{\partial \tilde{E}}{\partial \mathbf{x}}\Big|_{\mathbf{x}_{i-1}-\Delta\mathbf{x}_0} \tag{5.23}$$

$$= -\frac{d\mathbf{x}}{dt} \cdot (\frac{\partial \tilde{E}}{\partial \mathbf{x}}\Big|_{\mathbf{x}_{i-1}} - \frac{\partial^2 \tilde{E}}{\partial \mathbf{x}^2}\Delta\mathbf{x}_0) \tag{5.24}$$

$$= -(\frac{\partial \tilde{E}}{\partial t} - \frac{\partial^2 \tilde{E}}{\partial \mathbf{x}\partial t}\Delta\mathbf{x}_0) \tag{5.25}$$

$$= -\frac{\partial}{\partial t}(\tilde{E} - \frac{\partial \tilde{E}}{\partial \mathbf{x}}\Delta\mathbf{x}_0) \tag{5.26}$$

$$= -\frac{\partial \tilde{E}}{\partial t}\Big|_{\mathbf{x}_{i-1}-\Delta\mathbf{x}_0} \tag{5.27}$$

or in a discrete form:

$$\mathbf{v}_{i-1} \cdot M\mathbf{a}_i = -\frac{\partial \tilde{E}}{\partial t}\Big|_{\mathbf{x}_{i-1}-\Delta\mathbf{x}_0} \tag{5.28}$$

This equation means that the whole system tends to decrease the sum of the potential energy: when $\tilde{E}$ is decreasing, the acceleration $\mathbf{a}_i$ will have roughly the same direction with the velocity; otherwise it will have the opposite one, makes the velocity direction turn around eventually. The coarse-level tracking constraint here serves as a damping component, which prevents the system from oscillation due to conservation of energy. It also prevents $\tilde{E}$ from infinitely decreasing since the coarse shape of the mesh is strictly preserved [171]. Therefore, after sufficient number of iterations the whole system will reach a balance where $\frac{\partial \tilde{E}}{\partial t} = 0$, and a stable result gives $\mathbf{v}_i = \mathbf{a}_i = 0$.

□

Note that although I have constraints on external forces, in most of the cases,

they can be easily satisfied, such as gravitational forces and user-control impulse forces. Here I consider collision response as part of the constraint system, so it does not have impacts on the practical correctness. I use Forward Euler only for the simplicity of the expression in the proof. Actually I can derive the same form of Eqn. 5.18 using any other integrator (e.g. Backward Euler), during which the extra terms related to $\Delta\mathbf{v}_0$ (introduced by Backward Euler [27]) can be canceled out, eventually leaving $\Delta\mathbf{a}_0$. The main idea of the proof is that the system is conservative, regardless of the actual integrator, before adding extra damping constraints that ensures the final convergence. Upon convergence, the change in the high-res states (i.e. velocities and accelerations) will be the same as the change in the interpolated low-res states. This step, together with the position constraints by TRACKS, ensures the position and velocity difference between the high-res results at the boundary to be $O(\Delta t)$, smoothing out the visual popping artifact.

5.5.4 Iteration Number Estimation

The number of iterations needed for convergence, according to the proof, is largely related to the strength of the coarse-level constraint (in other words, the coarse-to-fine ratio K), since it provides the damping force to the system. Additionally, given a fixed upsampling scale (K), the iteration number is also related to a) the stiffness and density of the cloth, and b) the time step Δt. I estimate my iteration number in a simplified 2-D spring-mass system. Suppose at $t = 0$ a string with length l is hanging horizontally, with both endpoints fixed. It is currently

discretized as one single piece of 1-D string so the middle part of itself will not fall down. However, in the continuous real-world space, it is not in the equilibrium state and it has a residual energy of $O(l^2)$. This continuous case can actually be regarded as a string discretized to infinitely many small pieces. I define the residual energy as the difference of the potential energy between the current discretized one and the continuous one.

Subdividing the spring will bring the entire system closer to the actual continuous case (since the newly introduced vertices will fall down), so the residual energy will decrease. The spring system will start to bounce around upon discretization and I assume that there are damping forces in the system. After discretizing the spring into c pieces of equal length, the new system will have a residual energy of $O(l^2/c)$ when reaching the equilibrium state in the new discretization setting. If the system is in the critical damping condition, the energy will decrease by a factor of e after $t = \sqrt{m_s/\xi}$ seconds, where m_s is the mass of the spring and ξ is the stiffness. Therefore, the recovery time needed from the coarse level to the fine one is $O(\sqrt{m_s/\xi}\ln c)$.

In my case, I have $K = c^{O(1)}$ which depends on the embedded simulator and the collision state. Also I set $\ln K \leq 7$ to cover most of the cases. I use the density and the Frobenius norm of the stretching and bending stiffness matrix in [93] to estimate $\sqrt{m_s/\xi}$. m_s typically ranges from 0.1 to 1, while the value of ξ is between 10 and 100.

Combining all of them above, I have an estimation of $c_0\sqrt{m_s/\xi}/\Delta t$ as the number of iteration steps needed, where m_s is the density and ξ is the Frobenius

norm of the stretching and bending stiffness matrix in [93]. I use $c_0 = 10$ across all of my experiments. In practice, the iteration can also end when no large difference is detected between current and previous results. I found that using my estimation number the difference threshold can be as small as 10^{-3} relative to the scale of the cloth.

In each of the temporal partition, I add an extra simulation steps of $c_0\sqrt{m_s/\xi}/\Delta t$ to refine the starting state, so the total ideal performance gain due to parallelization is

$$\eta = \frac{N}{c_0\sqrt{m_s/\xi}/\Delta t + N/\eta_2} \tag{5.29}$$

Given a cloth material configuration with fixed m_s and ξ, η will have an upper-bound of η_2 if $c_0\sqrt{m_s/\xi}/\Delta t \ll KN/\eta_2$. This can be easily satisfied since the duration $N\Delta t$ is usually from a few seconds to many minutes, and $c_0\sqrt{m_s/\xi}$ is usually smaller than 1.

5.5.5 Implementation Details

There are some minor details in the implementation of the approach. When I take a larger step in the low-resolution simulation, I estimate the change of the state in the corresponding high-res step $u(\mathbf{X}_k^C) - u(\mathbf{X}_{k-1}^C)$ by linearly interpolating the states in between. The same method is also used in the adaptive partitioning method described in Sec. 5.4.2. The recovery iterations also count into the estimation of the current $\tilde{K}$, but do not count into the total number of steps, N, since there is no corresponding step in the low-res level and each processor has the same number of

extra simulation steps, so the system still remains balanced. I regard $\tilde{K}$ as $+\infty$ if the first step of the high-res simulation is not finished at the time I determine n in Sec. 5.4.2. Note that the state $\mathbf{X}$ includes both the position and velocity components. I also refine the velocities in the upsampling phase. When using adaptive remeshing, I obtain the new velocity as the average of the two vertices during edge splitting, following ArcSim [74]. The change of the state is also computed correspondingly.

5.5.6 State Inconsistency

In the extreme cases where the high-resolution mesh is much finer than the low-res one, e.g. 1M versus 100, the shape of the cloth in that case is largely determined by the aggregated effect from details not captured by low-res simulation. Therefore, I cannot recover the exact detail as in the serially simulated one at the transition point, which is referred to as the 'state inconsistency problem'. Enforcing the high-res mesh to match the low-res one using the tracking solver [171] can effectively avoid this problem. So, it can lead the simulation result to follow the movement of low-res one instead, which limits this approach from accuracy-demanding usage in those extreme cases. However, for other usage such as rapid design prototyping, where environmental constraints are mild and K is reasonable, motion difference between two levels is small and I can indeed achieve visually plausible results with high speedup, which are shown in Fig. 5.10 and 5.11. Alternative methods to improve the speedup without harming the accuracy is also discussed later in Sec.5.6.5.

5.6 Results

My method is tested on a large computing cluster with 526 compute nodes, each with 12-core (dual socket), 2.93 GHz Intel processors, 12M L3 cache (Model X5670), and 48 GB memory at 2:1 ratio IB interconnect, MPI for communication. I run one process in each of the cores (compactly assigned). I use up to 128 cores of this cluster to show the linear scalability provided by my theory and up to 512 cores to show the maximum possible speedup in large distributed systems. I could not test on a larger number of cores due to a core limit of 512 per job locally. I use the upsampling function by [74] throughout all of my experiments except in Table 5.4, which uses linearly-interpolated subdivision for fast error computation. As stated in Sec. 5.5.6, this method cannot guarantee the same accuracy as full simulation, which often cannot guarantee the same accuracy as the physical systems. The objective of this work is to generate visually plausible simulation to provide rapid visual feedback for interactive applications, such as rapid design prototyping.

5.6.1 Parameter and Scenario Setting

As mentioned in Sec. 5.4.3, I control the general coarse-to-fine ratio by limiting the smallest mesh size and enlarging the time step of the low-res simulation. Specifically in all of my test cases, the smallest length size of the triangle in the low-res simulation is about 5 times as large as that in the high-res one. The number of iterations in each of the smoothing processes is set to be the same as that in Sec. 5.5.4. I use ARCSim [74] as my base simulator, since it naturally supports

adaptive mesh refinement with an efficient remeshing algorithm. My method can be used in other CPU-based simulators using uniform meshes as well, as long as the upsampling algorithm is specified or implemented. All listed K in the following tables are averaged values across the entire simulation. I show scaling results using figures for clarity.

I use 7 different benchmarks to test the performance and the animation quality of my method: **Blue Dress and Yellow Dress** (Fig. 5.11(a,b)), **Sphere** (Fig. 5.11(c)), **Falling** (Fig. 5.11(d)), **Karate** (Fig. 5.1), **Twisting** (Fig. 5.10(a)) and **Funnel** (Fig. 5.10(b)). The default setting is 20 second simulation at the low-resolution time step of 0.02 sec using 128 cores. I extend the duration to 80 seconds and decrease the time step to make comparisons and validate my theoretical analysis on performance gain. Below are descriptions of each benchmark data.

To the best of my knowledge, previous works did not provide any code or experimental data to public, so the best known practice is to use the reported 'speedup data' in other works with similar scenarios, to minimize the difference due to computing platforms or implementation. I use the timing data of 'Two Cloths Draped' scenario from [152] since it has similar settings as mine (cloth-object interaction), similarly with other benchmarks.

Scenario	Blue Dress	Yellow Dress	Sphere	Falling	Karate	Twisting	Funnel
Original size speedup	74.1	75.0	102	116	96.4	92.3	93.8
4x large size speedup	99.6	109	178	119	103	101	108

Table 5.2: Results on a higher-resolution mesh. I run my system on meshes of higher resolution. Values in the table are the corresponding speedup.

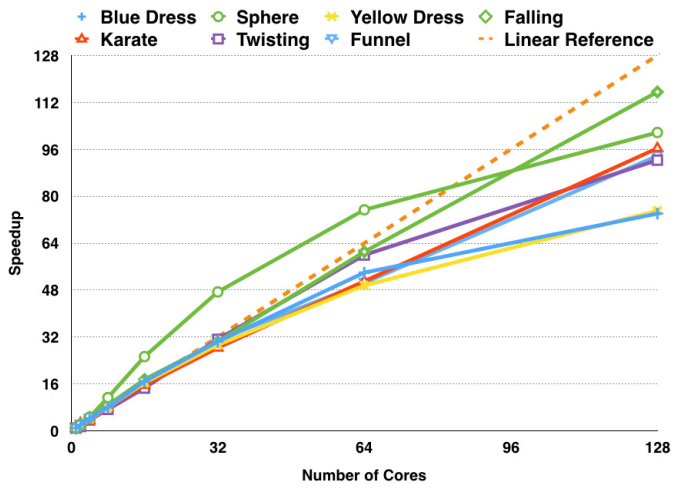


Figure 5.5: Performance scaling result with large low-res time step. A nearly linear scalability is achieved.

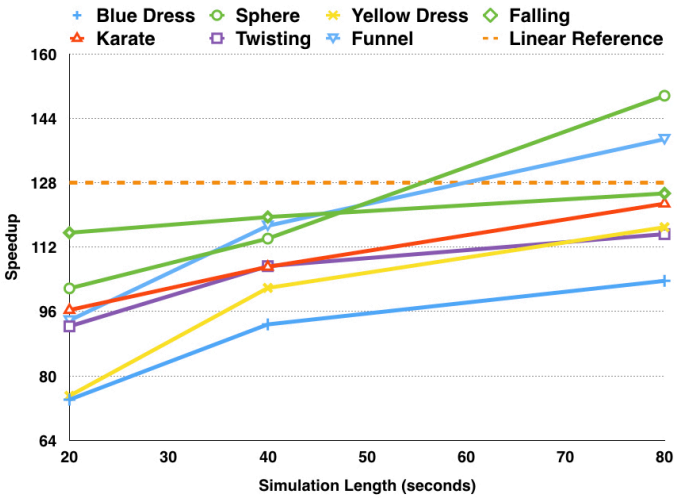


Figure 5.6: Results with increasing length of the simulation. A larger speedup is observed with longer duration of simulation.

5.6.2 Performance

Nearly linear scalability w.r.t. the number of cores. As indicated in Fig. 5.5, my method achieves a good scalability with an increasing number of processors. The reason of the super-linear speedup in the 'Sphere' scene is that it contains rapidly changing contacts with obstacles. When the cloth is free from contact after the sphere passes through, the remeshing algorithm of ARCSim failed

to simplify the mesh effectively, spending an unnecessarily large amount of time simulating simple flat cloth. However, due to the nature of my two-level structure, I maintain a reasonably small number of mesh elements while preserving the quality, and therefore outperform the serial approach significantly. I tested my method on a higher-resolution mesh and observed an even better speedup (Table 5.2) due to the same reason.

Improved scalability with increasing simulation duration. I show in Fig 5.6 that the scalability of my parallelized cloth simulation improves as the duration of the simulation increases. Although the averaging effect of the remaining load imbalance may partially account for it, the most likely reason is from Eqn. 5.29. I have relatively small speedup in 128-core parallelization when simulating a 20-second simulation because the iterative detail recovery algorithm consumes a relatively large amount of time according to Eqn. 5.29. Since the overhead is not dependent on the duration of the simulation and my method is a time-domain parallelization technique, the performance gain improves as the length of the simulation increases due to a smaller portion of the overhead.

Cores	8	16	32	64	128
Uniform partition runtime(s)	5533	3010	1042	684	631
Adaptive partition runtime(s)	4721	2568	928	565	532
Speedup (%)	117	117	112	121	119

Table 5.3: Comparison between different partition schemes. Values in the table are simulation runtime in seconds.

Performance impact on different choices of parameters. To verify my scalability analysis in Sec. 5.4.3 and 5.5.4, I further ran my benchmark with much smaller time steps in low-res simulation. As mentioned in Sec. 5.4.3, increasing

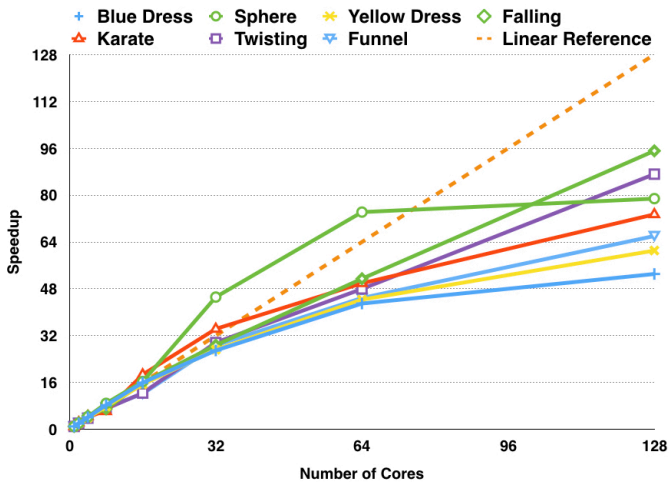


Figure 5.7: Performance scaling result with small low-res time steps. Compared to Fig. 5.5, the speedup for cases with core number larger than 32 is decreased, due to the smaller time steps for low-res simulation.

low-resolution time step is one of the ways to increase the ratio of high-to-low-res simulation time, K. Fig. 5.7 shows that smaller time steps in low-res simulation leads to a sub-linear scaling in all datasets, starting from the 64-core configuration. Although the 'Sphere' dataset has a bigger K due to its simplicity, the scalability starts to degrade at 128 cores as well. The speedup still increases with the simulation duration. However, as it is more closely bounded by K, the gain factor is not as significant as that with large time steps. In practice, a large time step in low-resolution simulations is beneficial to the parallelization performance, but it is limited by (a) the embedded simulation method, (b) the duration of a single frame, and (c) the desired animation quality.

Performance impact on different partition schemes. Table 5.3 shows that by using my adaptive partitioning scheme, I achieve an average of about 120% speedup compared to the uniform partitioned one with the best chosen parameter. In cases such as rapid design prototyping, where the cloth is in continuous contact with

Scenario	Blue Dress	Yellow Dress	Sphere	Falling	Karate	Twisting	Funnel
Time step(low-res)	1/200s				1/100s	1/50s	1/125s
Time step(high-res)	1/200s						1/500s
# of faces(low-res)	5K	6K	8K	6K	4K	4K	4K
# of faces(high-res)	80K	95K	131K	94K	58K	65K	65K
# of triangles(obstacle)	20K	20K	1280	15K	28K	762	4K
K	165	170	172	60	99	188	794
Low-res speed (serial 1-core)	0.6	0.79	1	1.2	0.83	0.22	0.32
High-res speed(OpenMP 12-core)	32.2	44.3	55.9	23.2	27.6	13.7	86.7
My method	0.89	1.14	1.3	1.5	0.91	0.41	1.22
Error before detail recovery	11%	12%	3.2%	22%	29%	46%	16%
Error after detail recovery	4%	6%	0.6%	5%	9%	14%	7%

Table 5.4: Results in the extreme case. I use 512 cores to simulate these scenes. Values in the table are in seconds per frame. The error metric is relative curvature difference compared to serial results in percentage. I use linear interpolated subdivision for fast error comparison.

obstacles, the parameter K remains relatively stable. However, it is still difficult to compute K before simulation begins, since it depends on the specific mesh and collision structure. Furthermore, it is best not to compute the parameter using the first few frames, since the cloth at the beginning can be under constrained without sufficient contact with the obstacles. My adaptive partitioning method here serves as an on-the-fly parameter estimation algorithm in order to achieve good workload balance.

Low-res speed with high-res mesh on a large distributed system. I further test my method in extreme cases where K is relatively small compared to p, which is possible in practice when the computational resources are sufficient. The runtime result is shown in Table 5.4. Although I cannot achieve a speedup as high as 512 due to the limitation of K, I have actually met the upper bound. The serial low-resolution simulation has consumed most of the time so there is very little space to improve in my scheme.

Comparison with previous CPU parallelization work. I compare the performance of my method against other CPU parallelization techniques. Fig. 5.8 shows

that in smaller-scale systems (less than 16 cores), my method can maintain a linear speedup with respect to the single-core system, scaling better compared to previous CPU-based methods using spatial-domain partitioning, e.g. 11x over 16 cores by [154]. For larger-scale systems (Fig. 5.9), I achieved about 50% more efficiency than previous methods such as [152]. In these methods, the processors need to send the information to each other, typically several times, when solving the linear system, resulting in large communication overhead and limited scalability. In contrast, my method only needs to share the states from low-resolution simulations once. Therefore, my method can achieve greater scalability and efficiency in comparison.

In addition, I compare my method with the original embedded OpenMP version of ARCSim. Although a maximum of 2.69x is observed using OpenMP with 2 cores due to a better cache usage in the linear solver, the performance scaling is poor when adding more cores, which results from that the simulation algorithm does not parallelize the remeshing process due to memory access issues. My method disables the OpenMP feature in the ARCSim. Since I parallelize the simulation in time domain, I can avoid memory access control problems, thereby achieving a better speedup.

Method	Speedup over sequential ArcSim [74]	Supports Adaptive Mesh?
Tang et al. [153]	47-58x	No
My method(64-core)	50-75x	Yes
My method(128-core)	75-115x	Yes
My method(512-core)	91-214x	Yes

Table 5.5: **Comparison with GPU method [153].** Other than the scalable speedup gain with more cores, the method is able to naturally support adaptive mesh during the simulation.

Comparison with GPU-based parallelization. Using similar benchmarks as [153],

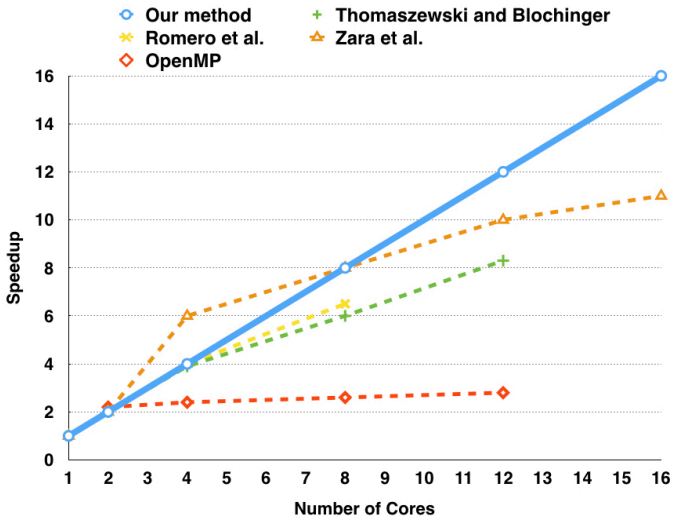


Figure 5.8: Small scale parallelization comparison. My method (in blue solid line) achieves a linear speedup, while others are limited by the communication overhead due to spatial domain partitioning.

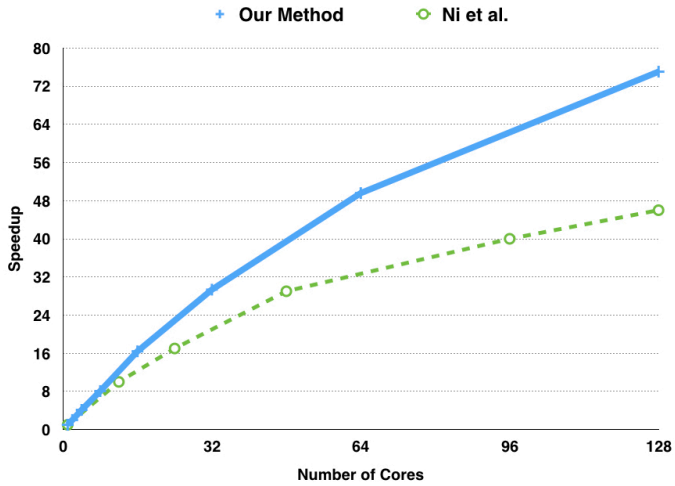
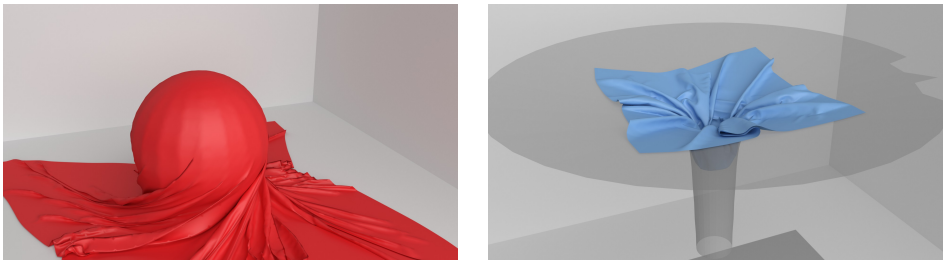


Figure 5.9: Large scale parallelization comparison. My method (in blue solid line) achieves about 50% higher efficiency than [152] using dynamic workload balancing.

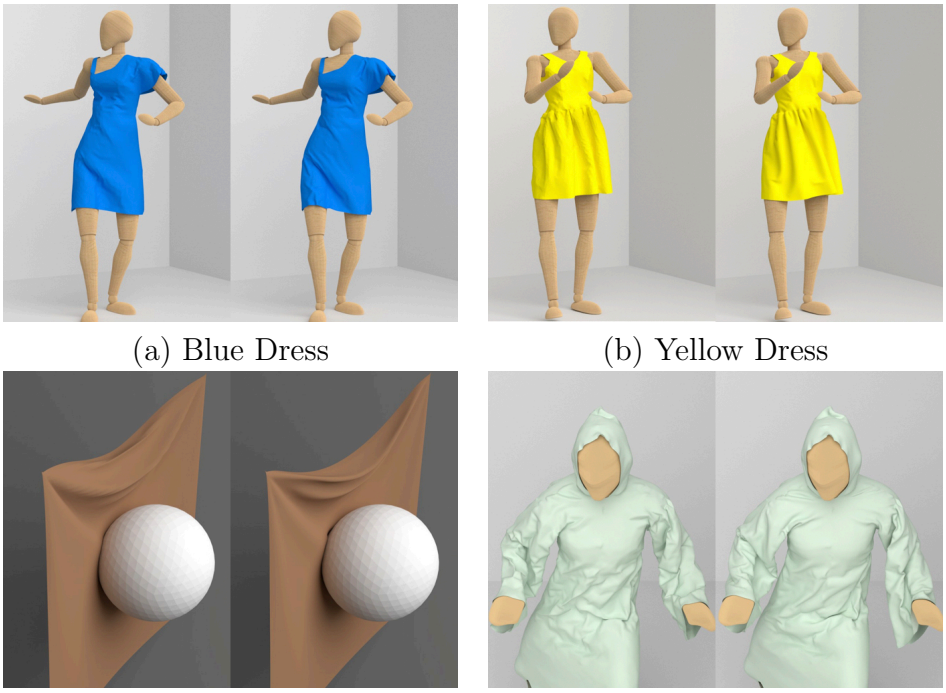
the speedup of my method in a 64-core system configuration is up to 54x in practical scenarios compared to the original ARCSim implementation on a single-core system and achieves a performance gain comparable to the GPU parallelization of [153] (Table. 5.5). However, my method has other distinctive strengths compared to the GPU method. Mine is the first work that can couple an adaptive mesh of varying dimensions during the simulation. I use the same number of triangles for



(a) Twisting (b) Funnel

Figure 5.10: More simulation results (best view with zoom-in in PDF). I have achieved visually plausible and smooth results even in challenging cases involving frequent contacts.

performance comparison, but in practice I can produce similar visual granularity with much fewer triangles using adaptive mesh [74], thereby making my method even faster. Moreover, my performance can be further improved using more cores and a longer simulation sequence, as shown in Fig 5.5 and 5.6.



(a) Blue Dress (b) Yellow Dress

(c) Sphere (d) Falling

Figure 5.11: Refining results (best view with zoom-in in PDF). The left image in each of the example is the upsampled mesh **without** detail recovery, which lacks high frequency details and causes 'popping' artifacts. The right one is the corresponding mesh **using my method**.

5.6.3 Smoothness

Fig. 5.11 and Table 5.4 shows the results before and after the refining algorithm is applied. If directly using the results from the upsampling algorithm, the detail of the cloth is significantly different from the correct one and therefore introduces popping artifacts. After applying the iterative smoothing algorithm, the high frequency information is recovered. I use average curvature distance defined in Eqn. 5.30 to measure the error between the recovered mesh and the original, high-res one simulated using ARCSim on a single core.

$$E = \frac{\sum_{f_1,f_2 \in F} |curv(f_1, f_2) - curv(\tilde{f}_1, \tilde{f}_2)|}{\sum_{f_1,f_2 \in F} |curv(f_1, f_2)|} \tag{5.30}$$

where f_1, f_2 are two adjacent faces in the original mesh, and $\tilde{f}_1, \tilde{f}_2$ are two corresponding faces in my simulation result. I disable remeshing and use linearly-interpolated subdivision for fast comparison. A larger value of the curvature error indicates a sharper edge in the corresponding position and thus a potential artifact. Before my recovery method, a relative error up to 46% is observed, which can cause large 'popping' artifacts in the result animation (Fig. 5.11). By using my technique, the error has decreased by 2-5 times, which is a significant improvement.

5.6.4 Memory and Render Latency

The extra memory footprint introduced by my method is small compared to the high-res mesh. In my experiments, the low-res mesh storage is 5.5% of the high-

res one. I do not render the low-res simulation in my method, and it actually starts at the same time with the first partition of the high-res one. Therefore, my method does not introduce any latency compared to the full-res simulation. In fact, I have achieved a 'pre-fetch' effect for the subsequent partitions due to the very fast, low-res simulation, thereby reducing any potential latency introduced by non-real-time simulation.

5.6.5 Limitations

There are some limitations with this method. First of all, the performance gain is bounded by the ratio of low- to high-resolution simulation time. Other than accelerating the simulation through parallelization in the temporal domain, I can additionally employ GPU implementation to further improve the overall gain. With a factor of $50x$ speedup from GPU [153] and a sufficient number of processors to parallelize the high-resolution simulation, it is possible to accelerate the performance even further. Secondly, the runtime of my method is bounded by a single-step high-resolution simulation time. This implies that at least one simulation step must take place in order to see the result. However, my method accelerates the overall performance, so I can actually achieve 'pseudo-interactivity', where the user can have a very fast visual feedback in parallel. Another possible direction is to implement a hybrid domain decomposition scheme, allocating some processors for spatial-domain parallelization to accelerate the single-step runtime. My approach provides plausible visual results in practical real-time applications, like rapid design prototyping.

However, as stated in Sec. 5.5.6, This approach may not be suitable in applications requiring high precision. In practice, the resulting cloth can sometimes appear slightly stiffer than the original one.

5.7 Conclusion and Future Work

In this chapter, I introduce a novel temporal-domain parallelization method for practical cloth simulation such as rapid design prototyping. Taking the advantage of faster simulations on coarser meshes, I parallelize the cloth simulation in time with accelerated computation and minimal communication overhead. I also proposed an iterative detail recovery algorithm to minimize the visual artifacts due to the state transitioning from coarse to fine meshes. My method outperforms existing CPU- and GPU-based parallelization techniques on a diverse set of benchmarks. It offers high efficiency and nearly linear scalability on large distributed systems, while maintaining high-fidelity visual simulation of the cloth. The scalability of my method is dependent on the ratio of low- to high-resolution simulation time, the length of the simulation, and persistence of contacts with obstacles. Since this method utilizes only time-domain parallelization, a natural extension would be a hybrid decomposition scheme that may provide a potential usage in short-duration simulation or in circumstances with memory constraint.

This work has been published in the proceedings of ACM SIGGRAPH/Eurographics Symposium on Computer Animation (SCA) 2018.