12-2018

A Scalable, Chunk-based Slicer for Cooperative 3D Printing

Jace J. McPherson
University of Arkansas, Fayetteville

A Scalable, Chunk-based Slicer for
Cooperative 3D Printing

A thesis submitted in partial fulfilment of the requirements for the degree of
Bachelor of Science in Computer Science with Honors

by

Jace McPherson
University of Arkansas
Bachelor of Science in Computer Science, 2018

This thesis is approved for recommendation to the Honors College.

Dr. Wenchao Zhou
Thesis Director

Dr. Matthew Patitz
Committee Member and Honors Advisor

Dr. Dale Thompson
Committee Member

Acknowledgements

I'd like to acknowledge Dr. Wenchao Zhou for his continued mentorship over the past two years as this research has matured. His guidance, expertise, and understanding have helped me grow as a researcher and writer, while also giving me valuable, intriguing topics on which to spend my time. I'd like to thank Chandler Bair, Laxmi Powell, and Dr. Zhenghui Sha for their contributions to the research detailed in Chapter 3 of this thesis. Austin Williams also played a big role in the development of various parts of Chapter 4, so I'd like to extend my appreciation to him as well.

Dedication

The past three and a half years have been very formative for me, and there are many people in my life who have made a significant impact on my successes and learning experiences. Firstly, I'd like to thank my parents, Keith and Julie McPherson, for giving me the room to exercise my independence, while continually supporting my decisions as I progressed through my college career. Without their love, support, understanding, and life advice, I would not be able to stand where I am now, and as firmly as I can now. I'd also like to thank my sisters, Lilly and Evynn McPherson, for working so hard for their goals at such young ages, inspiring me to push myself just as hard as they have. I'd like to thank my close group of friends, Naseer Naseem, Sophia Scalise, David Darling, Christopher Faulkner, and Mary Kathryn Rockett for frequently keeping me company as I worked on this project over the past two years, providing much-needed and entertaining break time. I would also thank Dr. Matthew Patitz for serving as a model of a truly passionate researcher. His dedication to rigor, intellectual intrigue, and quality, meaningful work has strongly influenced my perspective as a productive person. Finally, I'd like to thank Dr. Wenchao Zhou, my advisor and mentor for the past two years. With Dr. Zhou, what started as an extra-curricular research project ended up defining the majority of my college focus. He has encouraged me to do high quality, high yield work and opened doors for me in the world of research. His generosity, understanding, and perspective, both as a research advisor and as a thoughtful friend, have shaped my own outlook and dedication to the development of truly ground-breaking technology.

Abstract

Cooperative 3D printing is an emerging technology that aims to increase the 3D printing speed and to overcome the size limit of the printable object by having multiple mobile 3D printers (printhead-carrying mobile robots) work together on a single print job on a factory floor. It differs from traditional layer-by-layer 3D printing due to requiring multiple mobile printers to work simultaneously without interfering with each other. Therefore, a new approach for slicing a digital model and generating commands for the mobile printers is needed, which has not been discussed in literature before. We propose a chunk-by-chunk based slicer that divides an object into chunks so that different mobile printers can print different chunks simultaneously without interfering with each other. In this paper, we first developed a slicer for cooperative 3D printing with two mobile fused deposition modelling (FDM) printers. To enable many more mobile printers working together, we then developed a framework for scaling to many mobile printers with high parallel efficiency. To validate our slicer for the cooperative 3D printing process, we have also developed a simulator environment, which can be a valuable tool in visualizing and optimizing a cooperative 3D printing strategy. This simulation environment was also developed to export the visualization in a generic format for use elsewhere. Results show that the developed slicer and simulator are working effectively.

Table of Contents

1 Introduction 7

1.1 Background 7

1.2 Cooperative 3D Printing 9

1.3 Problem Formulation 11

1.4 Summary 13

2 Chunk-based Slicer 14

2.1 3D Printing and Slicing 14

2.2 Cooperative 3D Printing 16

2.2.1 Chunking 19

2.2.3 Slope Determination 20

2.2.4 Chunking Plane Determination 21

2.3 Efficiency Concerns 23

2.4 Slicing for Cooperative 3D Printing 24

2.4.1 Printing Sequence 25

2.4.2 Tool Path Generation and Transition between Chunks 25

2.4.3 Command Generation and Communication 29

3 Scalable Cooperative 3D Printing 33

3.1 Scaling Strategy 33

3.1.1 Chunking 34

3.1.2 Scheduling 37

3.1.3 Encoding a Printing Strategy 40

3.2 Evaluation Framework 43

3.2.1 Estimated Execution Time (EET) 44

3.3 Implementation Details 46

3.3.1 Scaled Chunker 50

3.3.2 Single-Chunk Simulation 53

3.3.3 Scaled Simulation 54

3.3.4 Print Time Estimation 60

3.4 Simulation Results 64

4 Implementation of Chunk-based 3D Printing 70

4.1 Real-world Implementation 71

4.2 Portable Simulation 77

5 Conclusion 81

5.1 Future Work 82

6 Appendix 83

Appendix A – UML Diagram 83

7 Glossary 84

8 References 86

1 Introduction

1.1 Background

Additive manufacturing (AM) is a rapidly developing field of technology that encompasses any processes and mechanisms by which 3D objects are produced layer-by-layer based on digital models. In general, additive manufacturing processes involve two components: a printhead for delivering energy, materials or both, which operates with three axial degrees of freedom and a chamber in which to "print" an object. Additive manufacturing machines are often referred to as "3D printers". Numerous developments in the AM world have given 3D printers the capability of printing materials like plastics, metals, and even foods.

Theoretically, we should be able to 3D print most objects with full automation, but 3D printers as they exist face severe limitations that hinder the mass-market adoption the technology foretells. Some important problems are related to lack of scalability and a lack of manufacturing automation involving 3D printers.

In this paper, we are focused primarily on the 3D printer scalability problem. Firstly, examining the current state of 3D printers reveals an obvious, fundamental problem to traditional 3D printing processes.

(a) (b)

Figure 1-1. Traditional 3D printer machines, both consumer-grade and industrial-grade. (a) MakrBot Replicator+ . (b) ProdWays ProMaker P2000

Taking a glance at the two printers in Figure 1-1, we can see that the size of a print job is limited to the size of the printer. More specifically, an object cannot be printed if it is wider than the full horizontal movement range of an extrusion nozzle or if it is taller than the maximum height of the extrusion nozzle above the printing surface (i.e., the "print bed"). In this case, the size limitation to print jobs limits the long-term goal of fully dynamic technology.

A large body of research has developed workarounds to the lack of physical scalability. Some of the workarounds involve printing multiple parts to be attached with adhesives or with creative joints, which requires additional assembly process and forces changes in design and increases the manufacturing cost.

Secondly, 3D printing technology lacks *time* scalability. Most 3D printers have a single printhead. This means the length of time needed to print an object is limited by the movement speed of that printhead. There are two potential fixes to this: 1) speed up the printhead, and 2) add additional printheads.

There are no easy implementations of the fixes mentioned in the previous paragraph. Speeding up a 3D print head is limited by other physical processes (e.g., heat transfer, vibriation, inertia, accuracy, etc.). Adding printheads to an existing machine is also a difficult mechanical challenge, introducing problems such as complex extruder tracks, printhead collision avoidance, printable area overlaps, and the need for a cooperation process between print heads.

In conclusion, the 3D printing scalability problem arises from two limitations: the print bed size, and the single printhead mechanism. The purpose of this paper is to propose solutions to these two problems.

1.2 Cooperative 3D Printing

In this paper, we propose a solution called "Cooperative 3D Printing" to address the limitations of classical 3D printing. By this proposed mechanism, large print jobs that were not previously possible to complete without human interaction can be autonomously produced in a single piece without interruptions, as depicted in Figure 1-2.

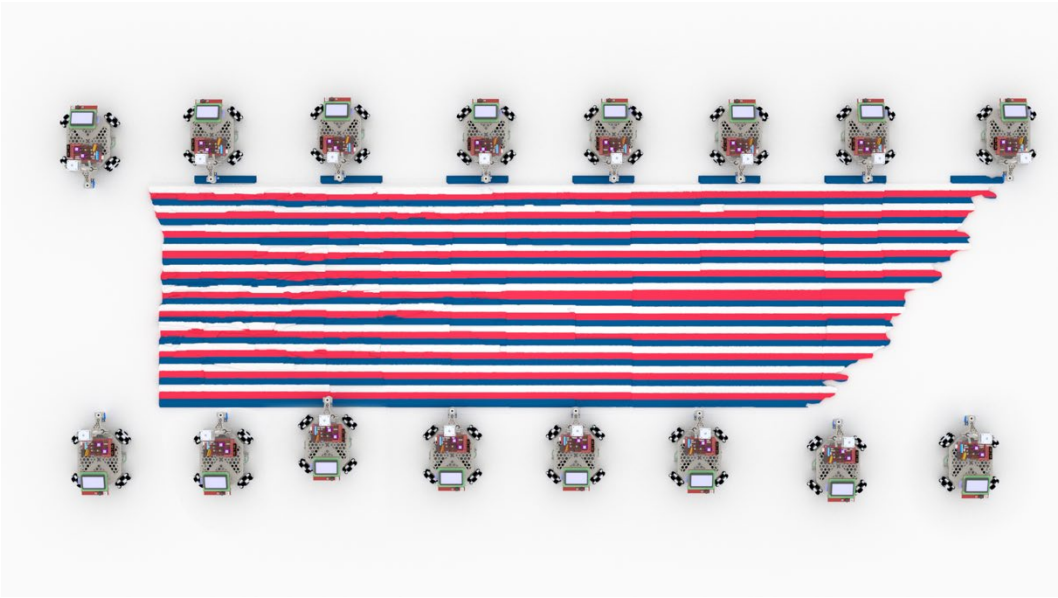

Figure 1-2. Large scale cooperative 3D printing. Many robots cooperate to produce a single object that does not require assembly upon completion. The final product in this figure is a topographical map of the state of Arkansas.

To summarize the mechanism, cooperative 3D printing is composed of any number of independent, autonomous, free-roaming 3D printers which are given directions to print part of a whole object. These parts are printed *on top of each other* such that they are joined during the printing process, as opposed to afterwards.

Cooperative 3D printing solves physical scalability with the premise that multiple independent 3D printers can be used to produce a single object. These printers need to "cooperate" to produce objects that would normally exceed the size limitation of a traditional 3D printer. They must have the freedom to navigate a large area, such that their print range is limited only by the size of the print surface, as opposed to a fixed range imposed by the extrusion nozzle's mechanism. To summarize, assuming the print surface is easy to scale, the potential print size will also be highly scalable.

This new mechanism also solves time scalability assuming new 3D printers that enter the fray can decrease the overall print time. Given that the number of printers is dynamic, we can quantify the time scalability as a function of the parallel efficiency from using any number of robots.

1.3 Problem Formulation

The advent of cooperative 3D printing presents many engineering challenges and it is important to highlight the specific problems that will be addressed. In this section, we will summarize the high-level components of a cooperative 3D printing ecosystem and specify which of those components are implemented in this thesis.

Figure 1-3. Cooperative 3D printing high-level process overview. This thesis is concerned with the processes labelled with an orange color.

Figure 1-3 shows a process overview for Cooperative 3D printing. This thesis is concerned with adressing the orange processes. These processes involve various algorithms and subprocesses that allow real-world robots to print the original input CAD model.

The chunker's design purpose is to properly subdivide a model into "chunks" that are distributed amongst the robots to print.

The slicer is designed to convert the distinct "chunks" into print commands so that the robots know how to print their chunks. The slicer must also figure out the appropriate commands for each robot to enable inter-robot communication.

The simulator uses the commands generated by the slicer to create a visual that will accurately display how the commands would be carried out by real robots. An accurate visual shows the user if the print will be carried out properly or not. It is also used as a validation step to our algorithms for the chunker and slicer. As a result, the simulator must be carefully designed, as the accuracy of the rest of the process depends on the accuracy of the simulator.

To summarize, the Chunker, Slicer, and Simulator present the following questions:

- Chunker: What is a realistic, optimal way to subdivide a model given the physical constraints of a 3D printing robot? How can this subdivision scale to multiple robots? What's the proper way to assign an individual chunk to a robot? How does one chunk's printing depend on the presence of the other chunks?
- Slicer: As the chunks are converted to printer commands, how should the commands differ from classical 3D printing to accommodate multiple robots printing distinct chunks? Will new routines need to be developed, and if so, what are they? What new commands needed to enable robot coordination, e.g. collision avoidance?
- Simulator: What visualization elements are necessary to validate the real-world feasibility of a print? How should the actions of multiple robots be visualized on a single visual timeline? What important conclusions can the simulation allow us to make about a single cooperative 3D print? Once a simulation is generated, what is the best way to ensure its utility and portability?

1.4 Summary

Having examined the problem and having described the requirements for its solution, we will describe the cooperative 3D printing solution as a useful candidate to the scalability problems of classical 3D printing.

Chapter 2 describes the chunk-based slicer that forms the cornerstone of the technology. This unique slicing process enables the subdivision of 3D printed models such that multiple chunks can be printed in parallel, while eliminating robot-material collisions. The concept presented in this chapter avoids robot-robot collisions by demonstrating the process with only two robots.

Chapter 3 demonstrates a method for scaling the chunk-based slicer two more than two robots. By examining more specific physical constraints, the chunking process can scale with two degrees of spatial freedom, as opposed to just one as presented in chapter 2. This chapter provides an evaluation framework that aids in determining the effectiveness of the scaling strategy. Finally, the simulation and corresponding results for the scaling strategy are shown.

Chapter 4 takes a closer look at the implementation of chunk-based 3D printing. This chapter focuses on the simulation environment we developed and addresses the implementation requirements of a useful cooperative 3D printing system.

Chapter 5 summarizes the contribution of this paper to the new field of cooperative 3D printing. By examining the findings, we take a look at the significance and implications of this work as it relates to rapid development, evaluation, and implementation of cooperative 3D printing systems. This chapter also explains future opportunities within this field and areas of research interest that can serve as next steps to enhancing cooperative 3D printing.

2 Chunk-based Slicer

Although additive manufacturing has become increasingly popular in recent years, it has been significantly limited by its slow printing speed and the size of the object it can print. Cooperative 3D printing is an emerging technology that aims to address these limitations by having multiple printhead-carrying mobile robots (or mobile 3D printers) work together on the same print job on a factory floor. With the traditional layer-by-layer 3D printing approach as defined by the ASTM F42 committee [1], it would be difficult for the mobile 3D printers to cooperate without interfering with the already printed part and with each other, which calls for a different approach of 3D printing.

2.1 3D Printing and Slicing

In traditional 3D printing, a CAD model needs to be sliced into layers and the path of the printhead movement needs to be planned to deposit materials for each layer. A slicer usually works by intersecting a plane at different Z-heights with the CAD model and calculating the boundary segments on each layer. The movement path of the printhead is then determined to infill the region within the boundary at each layer. Many different slicers have been developed, such as Slic3r [2], Cura [3], Kisslicer [4], and Skeinforge [5]. C Kirschman et al. has developed a parallel slicing algorithm to improve the slicing speed [1, 6]. E Sabourin et al. presented an adaptive slicing algorithm for layer-based 3D printing that can slice with different layer thickness [7]. S. Lefebvre et al. reported a GPU accelerated slicer [8]. However, slicing for the emerging cooperative 3D printing technology has not been investigated before.

A slicer is usually accompanied by a visualizer for the user to see the slicing results. A G-code viewer is one of the most common visualizers, such as the built-in viewer in Repetier Host [9]. Because cooperative 3D printing involves multiple robots, a simulator that can visualize the dynamic path of each mobile robots and how the materials are deposited over time will be beneficial for validating the printing path and optimizing the printing strategy for cooperative 3D printing. Many different simulators have been developed for mobile robots, such as Gazebo [10], EyeSim [11], UberSim [12], and Simbad [13]. These robot simulators can effectively simulate the interaction of multiple robots in 2D or 3D for evaluation of the design and the behavior of the robots. However, simulators for visualizing the dynamic 3D printing process of mobile 3D printers have not been reported.

In this chapter, to address the possible geometric interference arising from the layer-by-layer based approach with multiple mobile 3D printers, we present a chunk-based slicing approach so that each mobile 3D printer only needs to print a small chunk at a time, which can effectively separate the mobile 3D printers. The chunk-based printing can also keep 3D printing localized and therefore potentially avoid the large temperature gradient and internal stresses that are common with 3D printing of large objects. With proper scheduling of each individual mobile printer, this approach can be scaled to a very large number of mobile printers without interference. To simplify the problem, the slicing algorithm in this chapter will be limited to the cooperative 3D printing between two mobile 3D printers that carry a fused deposition modelling (FDM) extruder. This chunk-based slicing algorithm ensures good bonding between the chunks and smooth transitioning when a mobile robot moves from one chunk to another.

It is worth noting that the positioning and alignment of multiple mobile printers in the physical world is a non-trivial problem, which deserves separate research. Therefore, instead of validating the chunk-based slicing algorithm on the physical mobile printers, we created a simulator environment to simulate the dynamic printing process over time and the communication between mobile printers using the sliced results as an input. This simulator environment makes it much easier, faster, and less expensive than actually executing a print job when validating the slicing results, which provides a valuable tool to understand and optimize the printing strategies before submitting a print job. In addition, the simulator environment takes similar inputs as the physical mobile printers, which would make it effortless to submit the printing job to the physical mobile printers after validation in the simulator environment. Our results show that our chunk-based slicer works effectively for the two-robot printing strategy, as validated by the simulator. This new chunk-based slicer and the new simulator environment presented in this thesis represent a significant step towards cooperative 3D printing where multiple independent 3D printers can work together. The implementation of this simulator is detailed in Chapter 3.

2.2 Cooperative 3D Printing

At the core of the cooperative 3D printing platform is a mobile 3D printer as shown in Figure 2-1, which replaces the XY stage on a regular 3D printer with a set of omnidirectional wheels to translate the printhead in XY plane. This design enables unlimited printing in the X direction, but the Y direction is limited by the distance between the printhead and the front wheels (termed as "build depth" in this thesis) if a layer-by-layer based approach is used because the printed material in the previous layers will block the path of the wheels in Y direction.

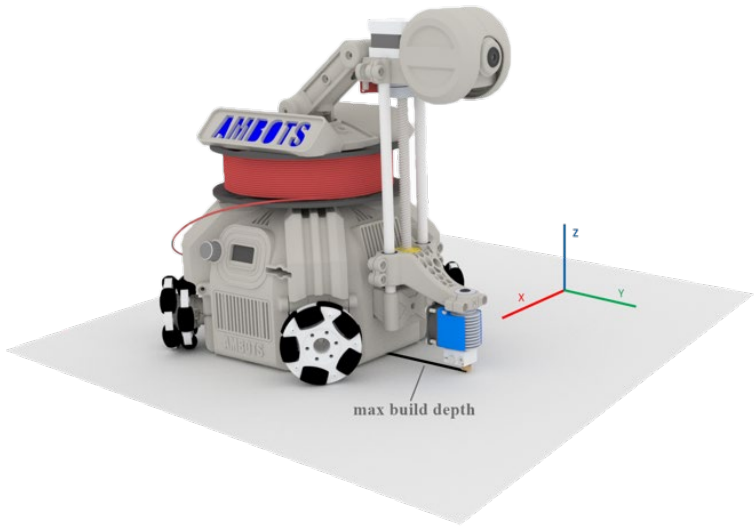

Figure 2-1. Illustration of a mobile 3D printer, which can print indefinitely in the X direction, but is limited in the Y direction

In this thesis, we present a chunk-based printing strategy, where the mobile printer finishes all the layers of one chunk before it moves to print another chunk, effectively solving the problem of the blocked path by the printed materials to enable the mobile printer printing unlimited in both X and Y directions. Similar partition-based printing strategies already exist, mainly for printing parts that can be assembled into a single model in post-processing. Examples include Chopper [14], curvature-based partitioning methods [15], and skeletonization [16]. While these methods are efficient at dividing model meshes for post-assembly, the chunking method for cooperative 3D printing requires that all chunks be printed such that they are bonded during the printing process without post-assembly, requiring a new, different process. One issue that arises is the bonding between the chunks. Our solution to this issue is to use a sloped interface (and/or an angled printhead) to allow more bonding surface between the chunks. A general slicing strategy for cooperative 3D printing is illustrated in Figure 2-2:

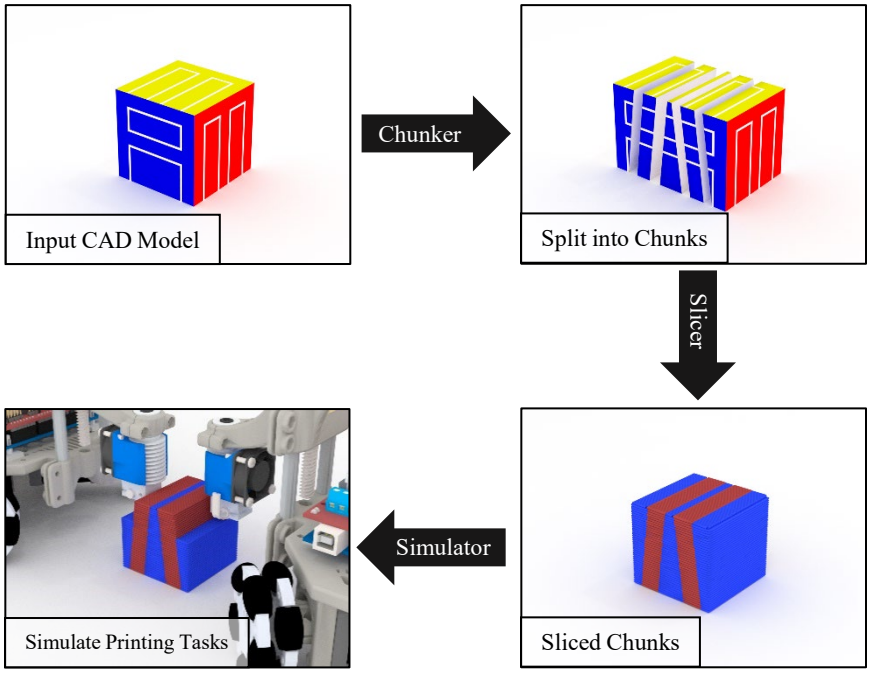

Figure 2-2. Illustration of the slicing strategy for cooperative 3D printing: (1) The chunker splits the printing job into chunks and ensures feasible printing of each chunk and good bonding between chunks; (2) The slicer slices the chunks into layers, generates commands for printing the chunks, schedules the sequence of printing the chunks among multiple robots, and inserts communication commands to enable necessary communication among multiple robots; (3) The simulator visualizes the dynamic printing process using the commands generated by the slicer.

The following provides overviews of the behavior of the "Chunker", "Slicer", and "Simulator" processes, shown using black arrows in Figure 2-2:

- Chunker: A CAD model of the print job will be first input into a "chunker", which splits the CAD model into chunks based on a set of criteria to ensure feasible printing of each chunk and good bonding between chunks.

- Slicer: The chunks will then be sliced into layers using a slicer, which generates commands for printing the chunks (e.g., tool paths, material extrusion, temperature control, etc.), schedules the sequence of printing the chunks among multiple robots,

and insert communication commands to enable necessary communication among multiple robots.

- Simulator: The commands generated by the slicer is interpreted by a simulator, which visualizes and animates the dynamic printing process over time to provide a tool for evaluating the chunking and slicing parameters and results.

2.2.1 Chunking

The objective of chunking is to divide the printing job into chunks such that they can be assigned to as many robots as possible to increase the printing speed. Therefore, the overall chunking strategy is highly dependent on the geometry of the print, the number of available robots, and how the robots will be scheduled. To simplify the problem, we will begin by considering the chunking problem for exactly two robots, then by covering the methodology for scaling in chapter 3. The method by which we split a print job between two robots will later be applicable for many robots using a "divide and conquer" strategy.

To chunk for two robots, we will split the object into multiple chunks along one direction (the Y direction in Figure 2-1) with sloped planes to ensure good bonding between chunks. Two robots start from the center chunk and print along the $+Y$ and $-Y$ directions, respectively, to finish each chunk. To calculate the geometries of these chunks, we simple bisect the original geometry multiple times around multiple planes. Because we have constrained the problem to chunking in the $+Y$ and $-Y$ directions, each plane can be defined by two things: its slope and Y position.

2.2.3 Slope Determination

A sloped interface between chunks is needed for this chunk-by-chunk 3D printing strategy. The angle of the sloped plane needs to be carefully determined due to conflicting objectives:

1. A maximum slope angle will maximize the volume of each chunk and increase printing efficiency;
2. A minimum slop angle will maximize the area of the bonding interface and increase the bond strength.

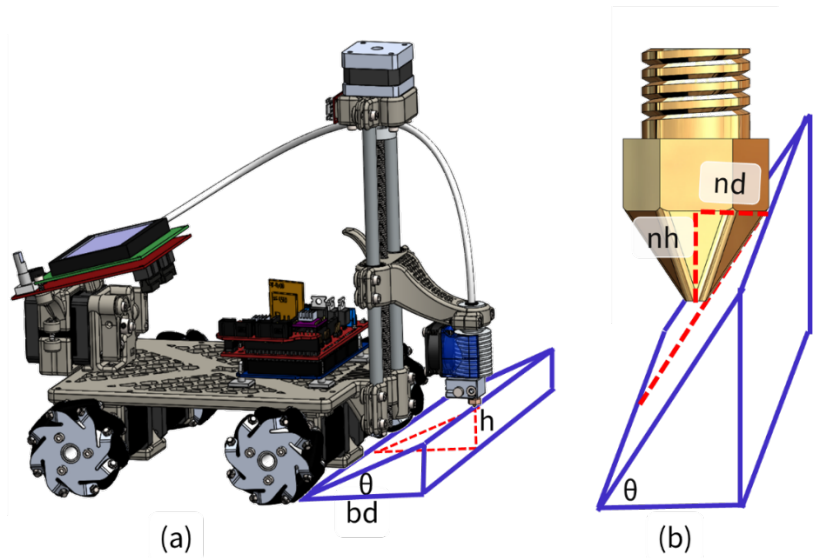

*Figure 2-3. Illustration of robot build limits: (a) The smallest slop angle of a chunk depends on the ratio of the object height, **h**, and the robot build depth, **bd**; (b) The largest slope angle of a chunk is limited by the ratio of the nozzle height, **nh**, and the nozzle depth, **nd**.*

In addition, the range of the slope angle is limited by the robot parameters as illustrated in Figure 2-3, which should be determined by:

$$\theta_{max} = \tan^{-1}\left(\frac{nh}{nd}\right) \tag{1}$$

$$\theta_{min} = \tan^{-1}\left(\frac{h}{bd}\right) \tag{2}$$

Here, θ_{max} and θ_{min} are the limits of the slop angle, nh and nd are the nozzle height and nozzle diameter, h is the height of the object to be printed, and bd is the build depth of the printer, as illustrated in Figure 2-3.

If the angle is too large or too small, either the front wheels of the robot or the nozzle will interfere with the printed material. It should be noted that the range of the angle is dependent on the printer design and the limits can be easily changed with a tilted nozzle or a printer with a variable build depth. Tests should be performed to choose an appropriate slope angle. In this paper, we use the calculated θ_{max} for our subsequent calculations.

2.2.4 Chunking Plane Determination

With a determined slope, we will also need to know where we want to split the object. For the chunking strategy with two robots, we first need a center chunk, which can only be printed by one robot. After the center chunk is completed, the two robots will finish the chunks on the left and the right sides, respectively. The center chunk's chunking planes can both be represented as tuples in the form $(\bar{n}, \bar{p})$, where $\bar{n}$ is the plane's normal vector and $\bar{p}$ is any point on the plane. This is the most convenient representation because the bisecting algorithm we used (specifically, the bisecting algorithm in Blender [17]) required only these two values to bisect a 3D geometry around the plane. The left and right chunking planes for the center chunk can be determined by:

$$Plane\ \boldsymbol{L_0} = \left(\bar{n}_{L_0}, \overline{p_{L_0}}\right) = \left(\left[\bar{c} \times \left|(0,0,h) + \frac{h}{\tan\theta} \cdot \perp (\bar{c})\right|\right], \left[\bar{p_c} + \frac{h}{\tan\theta} \cdot \perp \bar{c}\right]\right) \quad (3)$$

$$Plane\ \boldsymbol{R_0} = \left(\overline{n_{R_0}}, \overline{p_{R_0}}\right) = \left(\left[\bar{c} \times \left|(0,0,h) - \frac{h}{\tan\theta} \cdot \perp (\bar{c})\right|\right], \left[\bar{p_c} - \frac{h}{\tan\theta} \cdot \perp \bar{c}\right]\right) \quad (4)$$

where $\bar{c}$ is the vector representing the center line of the object (in our case, a line that varies only in the X direction), $\overline{p_c}$ is a point on the center line, θ is the angle of the chunking plane (see equations (1) and (2)), and

$$\perp (x, y, z) \coloneqq (-y, x, z) \tag{5}$$

After calculating these two planes, we can iteratively shift those planes outward by some shift amount, s, from the center chunk by iterating $\overline{p_{R_{i+1}}} \leftarrow \overline{p_{R_i}} + s \cdot \bar{c};\ i \geq 0$. We can use these planes to slice the model into subsequent "left" and "right" chunks. Figure 2-4 demonstrates the iterative chunking process, starting with the center chunk, then shifting the chunking planes $\boldsymbol{L_0}$ and $\boldsymbol{R_0}$ to the left and right, respectively.

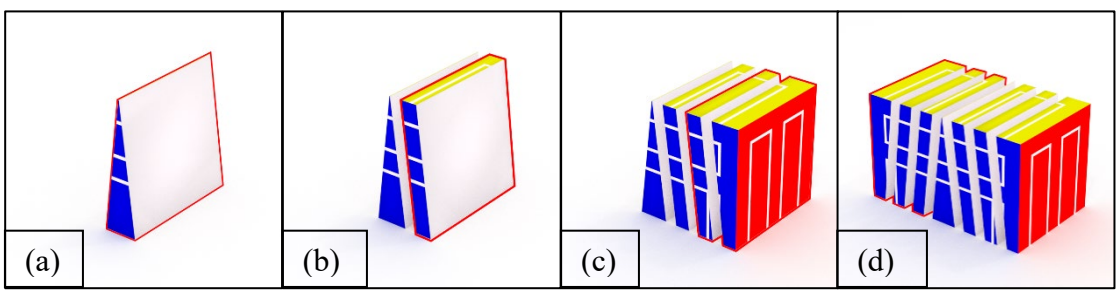

Figure 2-4. Iterative chunking results. Planes $\boldsymbol{L_0}$ and $\boldsymbol{R_0}$ are reused and shifted to split further chunks on the left and right of the center chunk: (a) center chunk; (b) shifted plane $\boldsymbol{R_0}$ to the right by one chunk; (c) shifted plane $\boldsymbol{R_0}$ to all the right chunks; (d) shifted plane $\boldsymbol{L_0}$ to all the left chunks.

We have applied this chunking algorithm to two different geometries using different chunking settings to demonstrate its effectiveness, including a cylinder and a car model, as shown in Figure 2-5. The yellow chunk is the center chunk. As we can see, the chunker works effectively with complex geometries and different settings.

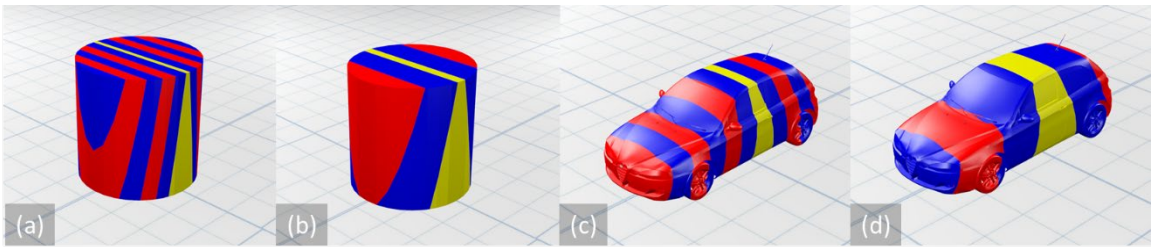

Figure 2-5. Chunker results with two different objects, a cylinder and a car model: (a) Cylinder with short build depth and steep chunking slope; (b) Cylinder with deep build depth and moderate slope; (c) Car with short build depth and steep chunking slope; (d) Car with deep build depth and moderate slope.

2.3 Efficiency Concerns

To determine the speedup gains of cooperative 3D printing, we can adapt the Amdahl's law used for parallel computing. The print job can be split up into two parts:

1. A part that can only be printed by one robot (i.e., the center chunk); and
2. A part that can be printed simultaneously by multiple robots (i.e., the rest of the chunks).

Assuming the average build speed is b $\mathrm{cm}^3/\mathrm{hour}$, the total volume of the print is V_{total} and the volume of the center chunk is V_{center}, the total printing time with one single printer would be:

$$T_1 = \frac{V_{total}}{b} = \frac{V_{center}}{b} + \frac{V_{total} - V_{center}}{b} \tag{6}$$

As a general discussion, assuming the printing job of the non-center chunks can be split among N printers, the total printing time would become:

$$T_N = \frac{V_{center}}{b} + \frac{V_{total} - V_{Center}}{b \cdot N} \tag{7}$$

So, the printing time reduction, r, with N printers is:

$$r = \frac{T_N}{T_1} = \frac{V_{center}}{V_{total}} + \frac{1}{N}\left(1 - \frac{V_{center}}{V_{total}}\right) \tag{8}$$

And the speedup gain is $s = 1 / r$. It is clear that the V_{center} / V_{total} needs to be minimized to maximize the speedup gain. In our current two-robot printing scenario ($N = 2$), if we assume $V_{center} / V_{total} = 0.05$, for example, the speedup gain would be:

$$s = \frac{1}{0.05 + \frac{1}{2}(1 - 0.05)} = 1.905 \tag{9}$$

To maximize speedup, we have already set up the calculations of the center chunk to minimize the center chunk volume by using the maximum slope angle possible for the chunk faces, θ_{max}.

2.4 Slicing for Cooperative 3D Printing

The objective of the slicer is to make sure the robots can work together to finish printing according to the printing strategy. Unlike a regular slicer that only generates a tool path for a single print head, the slicer for cooperative printing need to accomplish three functions:

1. Assign chunks to each robot and determine their printing sequence;
2. Generate tool paths for each chunk and the tool paths for transitions between chunks;
3. Generate commands based on the tool path for the robots to execute and provide a mechanism for the robots to communicate with each other in case one robot's printing task is dependent on the status of the printing task of another robot.

2.4.1 Printing Sequence

To determine the path for a robot to follow, the robot must first know the chunks it will print and their sequence. As we only consider two robots in this chapter, we can use the following simple strategy to assign the chunks to the robots, where C_A represents Robot A's chunks, and C_B represents Robot B's chunks:

$$C_A = [\text{center chunk, left chunk 1, left chunk 2, } \dots] \tag{10}$$

$$C_B = [\text{right chunk 1, right chunk 2, } \dots] \tag{11}$$

Where Robot A is assigned the center chunk and all the chunks on the left, and Robot B is assigned all the chunks on the right. The chunks then need to be ordered based on the scheduling strategy for the print job. Because the chunks were generated in order by the chunker, there is no need to order the chunks for the simplified two-robot printing in this section.

2.4.2 Tool Path Generation and Transition Between Chunks

With the ordered chunks assigned to each robot, we need to generate a sequential tool path for each robot to finish its assigned chunks. This task can be accomplished in steps as illustrated in Figure 2-6: (1) Generate the tool path for each chunk; (2) Generate the tool path between chunks (transition); (3) Combine the tool paths in sequence.

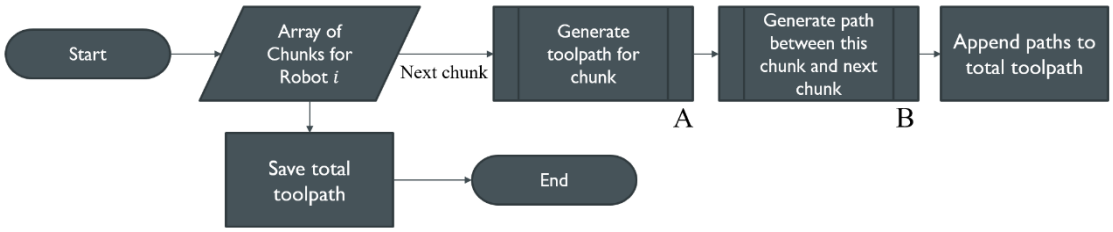

Figure 2-6. Path generation for each robot: generate tool path for each chunk and generate path to transition between chunks.

The tool path generation for a chunk is essentially the same process a normal CAD slicer uses for printing an object. The general process is illustrated in Figure 2-7. Instead of starting from an STL file (or other 3D file format), the tool path generation algorithm starts with triangular meshes generated by our chunker. Based on the specified layer thickness, a list of horizontal planes is generated to split the model into multiple layers. The horizontal planes are generated to split the model into multiple layers. The horizontal planes are then intersected with the triangular mesh to calculate the intersection line segments at each layer. Figure 2-8 shows an algorithm we used to calculate the line segments at each layer. The line segments are then ordered into a ring to form a perimeter for each layer. Infill paths are then generated for the parameters at each layer. Because this process has been well-establish in current slicers, we are omitting most of the details here, although the slicing process is by no means a trivial task. Building a general-purpose slicer involves accounting for oddities in models such as multiple isolated meshes, non-closed geometries, holes in closed geometries, and more. The development of a proprietary slicer consumed much of the work of this research.

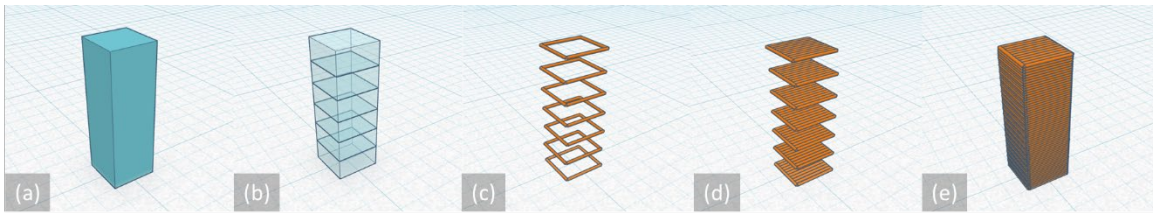

Figure 2-7. Illustration of the tool path generation process: (a) Original model; (b) Model split into horizontal layers; (c) Perimeters calculated for each layer by intersecting the triangular mesh with a plane at each layer; (d) Generate infill path for the perimeter at each layer; (e) Combine all the tool path generated at each layer.

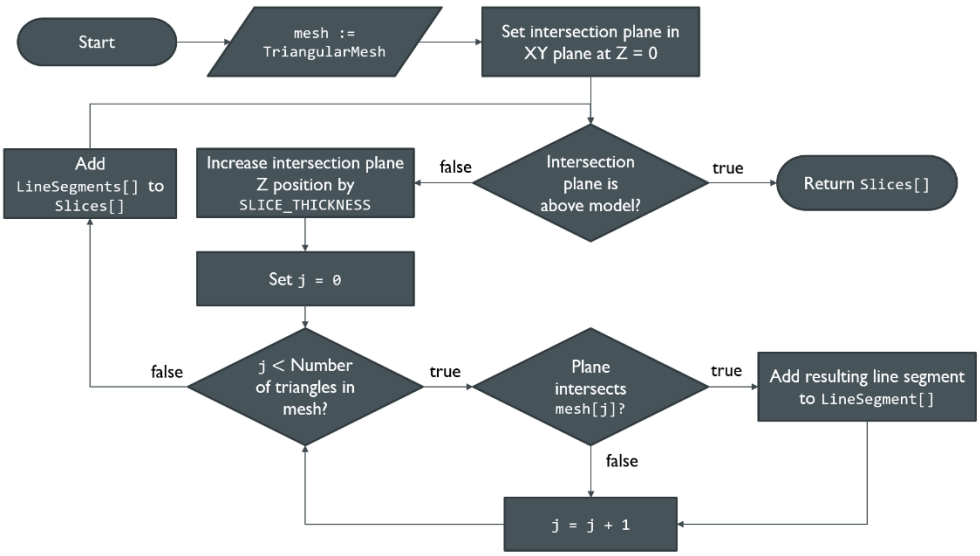

Figure 2-8. Process for calculating the line segments for the perimeter path of a layer.

The results of the tool path generation can be seen in Figure 2-9, which shows the slicing and infill algorithms are working correctly.

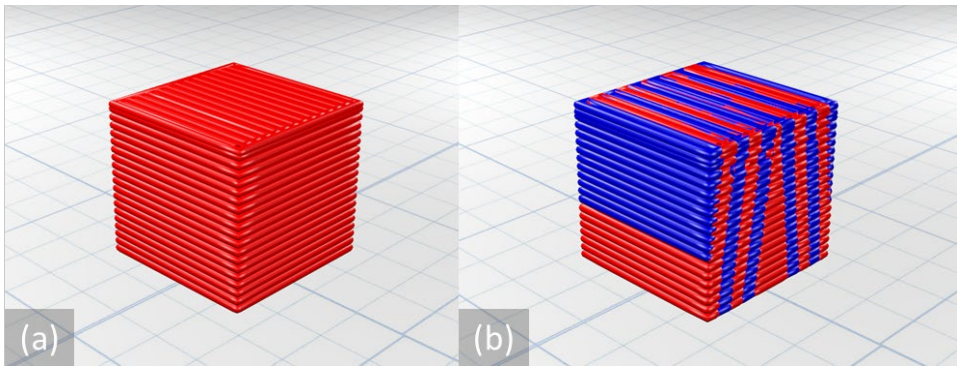

Figure 2-9. Tool path generation from the slicer algorithm (thick filament was used for better visualization): (a) without chunking; (b) with chunking first.

In addition to the print path for each chunk, the robot must have a way of moving from one chunk to the next. A simple direct line would not work, as the robot could knock against previously printed materials. Instead, we generate a separate path from the endpoint of the current chunk to the starting point of the next chunk. There are possible ways to optimize this path to save printing time, but we are using a simple approach that is not optimized but always works. Figure 2-10 visually demonstrates how we generate this transition path. Four points comprise the path: the endpoint of the current chunk $p1$, the start point of the next chunk $p4$, and two points in between. For $p2$, we simply shift the printhead upwards a small amount. For $p3$, we move the extruder to the boundary of the chunk (i.e. to the same x and y position as $p4$). The path generation process is detailed in Figure 2-11.

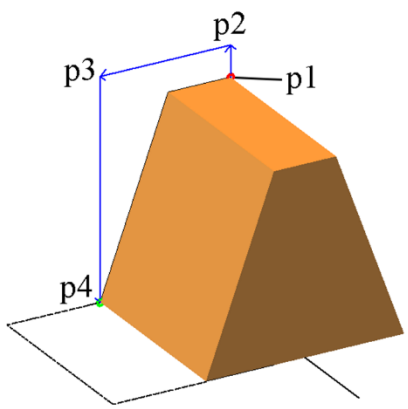

Figure 2-10. Transition path between chunks ($p1 \rightarrow p2 \rightarrow p3 \rightarrow p4$): the printhead moves slightly upward from previously printed materials and navigates to the start of the next chunk. The endpoint of the current chunk is $p1$ and the start point of the next chunk is $p4$. $p2$ and $p3$ are points used to generate the transition path.

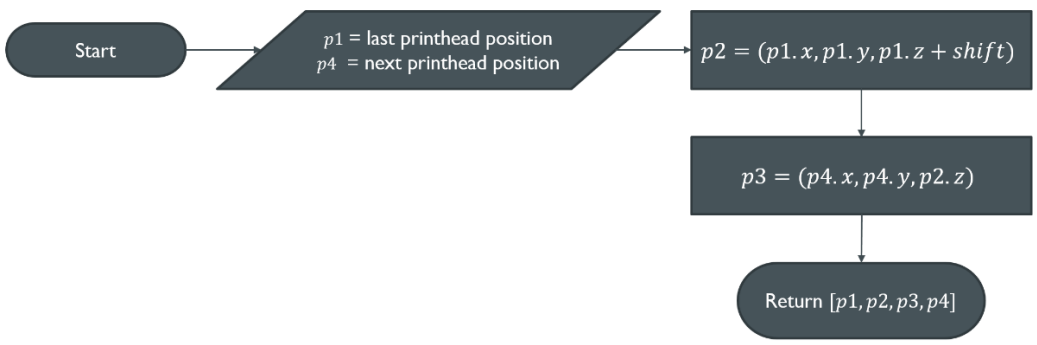

Figure 2-11. Process of generating transition path between two chunks

2.4.3 Command Generation and Communication

Until this point, we have generated the entire tool path for each robot, which are organized in a multi-level hierarchical data structure as illustrated in Figure 2-12.

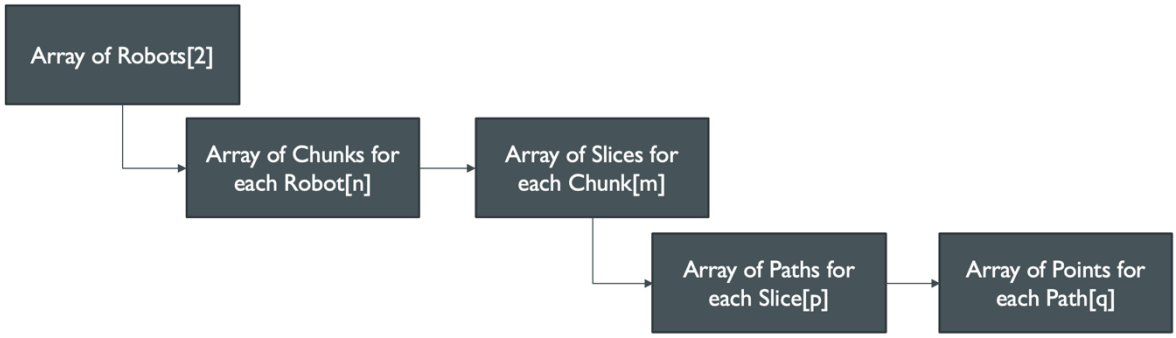

Figure 2-12. Data structure of the tool paths for the robots.

With the entire tool path generated, the slicer needs to generate commands that can be interpreted by the robots to execute the movements. One of the most common type of commands used in a regular 3D printer is G-code commands. In this paper, we use similar commands for our simulator to interpret. For example, we use a "`MOVE X Y Z`" command, which moves the robot from its current position to the specified position (X, Y, Z). It is equivalent to a G1 command in G-code and thus make it easy to output the commands to a real robot. Since the tool path is just a list of points, this one command will be sufficient to instruct the robot to move along its tool path.

Now that the robots know where to move based on the generated commands, they also need to know when to move. In the situation when one robot's next move is dependent on another robot's printing status, it is necessary to provide a mechanism for the robots to communicate. For example, in our two-robot situation, the second robot must wait until the first robot finishes printing the center chunk to start printing and thus has to know when the first robot finishes printing the center chunk. One direct way is for the robots to have real-time constant communication with each other, but it would significantly increase the complexity when many robots are involved. Luckily, the interdependence of the printing tasks can usually be pre-determined in the chunking or slicing stage. Therefore, we can pre-implant a communication command at the stage when communication

is needed. In this thesis, we implemented a "NOTIFY" command, which is inserted behind a MOVE command to notify the other robot that a certain movement has been finished. Because our situation only involves the center-chunk waiting period, only one NOTIFY command is needed for the entire print. The second robot will not begin along its toolpath until it receives the NOTIFY command from the first robot. Before receiving the NOTIFY command, the waiting robot must be executing a simple WAIT command that forces the robot to stall until the other robot sends a NOTIFY command. The hardware implementation of these commands isn't important, as long as they coordinate as described.

In addition to the MOVE, NOTIFY and WAIT commands, the robot also needs to know when the materials should be deposited. This is because the robot does not print materials along all the tool paths. For example, when the robot is transitioning for one chunk to another, no materials need to be printed. Therefore, we implemented a "TOOL ON/OFF" command to indicate whether the robot should print materials. When "TOOL ON" command is issued, materials will be printed along all the following tool path until a "TOOL OFF" command is issued. Based on the tool path data and how we want to robot to communicate and print materials, we can translate the tool paths into commands for the robots to execute the printing process using the three commands we have implemented. This process is shown in Figure 2-13.

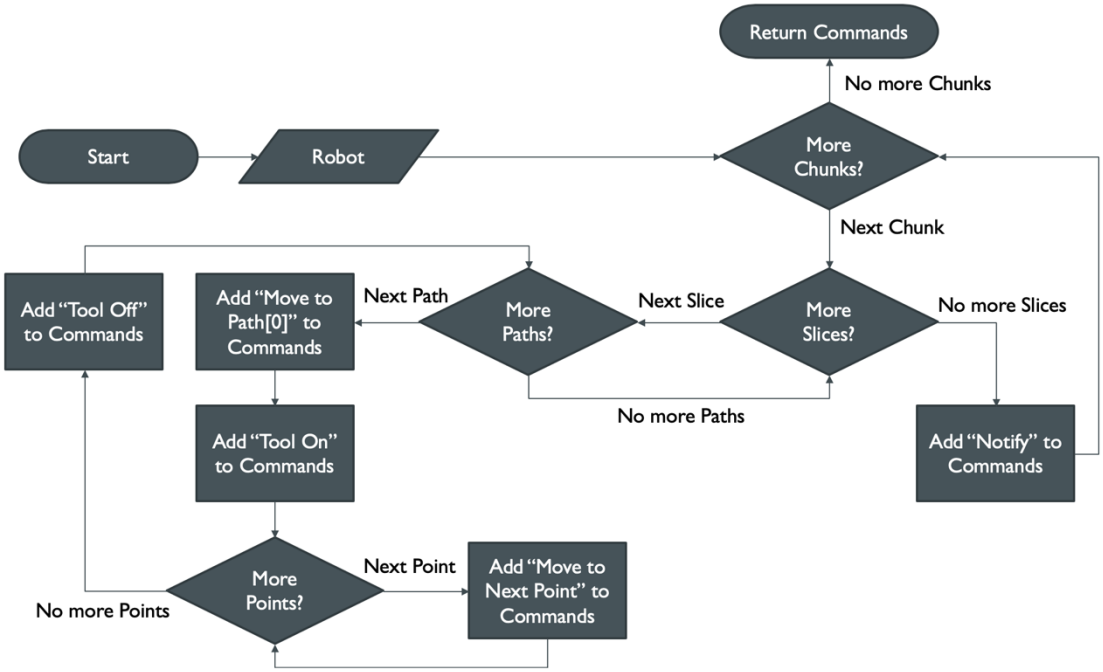

Figure 2-13. Conversion from the tool paths to robot commands.

After the tool path that was generated by the slicer has been converted to a sequence of "commands", these can be used in two ways: (1) To be converted directly to G-code for real-world robots to execute, or (2) to be simulated to visualize the behavior of the commands.

3 Scalable Cooperative 3D Printing

The motivation of this chapter is to take the findings of chapter 2 and expand the possibilities of the technology. Chapter 2 describes the chunking and slicing process for printing a model in parallel with two robots. In theory, we should be able to amend the two-robot process to facilitate the use of many more printers. This chapter introduces such a strategy, named the Scalable Parallel Array of Robots for 3D Printing (SPAR3).

The biggest problem this chapter will address is the need for a generalized framework for describing the chunking and slicing process. This framework will include a generalized definition of a "printing strategy" and a method for evaluating printing strategies. This framework will lay the foundation for the development of SPAR3. Finally, section 3.3 will provide SPAR3 simulation results with a brief discussion.

3.1 Scaling Strategy

A printing strategy is composed of two distinct stages: *chunking* and *scheduling*. As it relates to the two-robot printing process, these two stages take the place of the previous, trivial *chunking* stage. The new chunking stage achieves a similar goal: divide a large model into "printing tasks" (i.e. "chunks"). The scheduling stage assigns the chunks to individual robots, along with a schedule (i.e., a printing sequence) for the printing of those chunks, such that the robots print the chunks in an order that does not lead to collisions. In this section, we will first discuss the chunking and scheduling strategies, and then present a mathematical framework describe the strategies using a directed dependency tree (DDT).

3.1.1 Chunking

An object to be printed needs to be divided into smaller chunks so that the printing work can be distributed to multiple robots. The chunking process used in this paper chunks only along a horizontal plane, with sloped interfaces, much like the method used in chapter 2.

The key difference that separates scaled chunking from two-robot chunking is that the chunking process happens along more than one axis. In this case, we will study the characteristics of chunking in a grid pattern. The chunker separates the main model into pieces lengthwise, then separate each of those pieces width-wise. For example, Figure 3-1 provides visualizations of the SPAR3 chunking pattern.

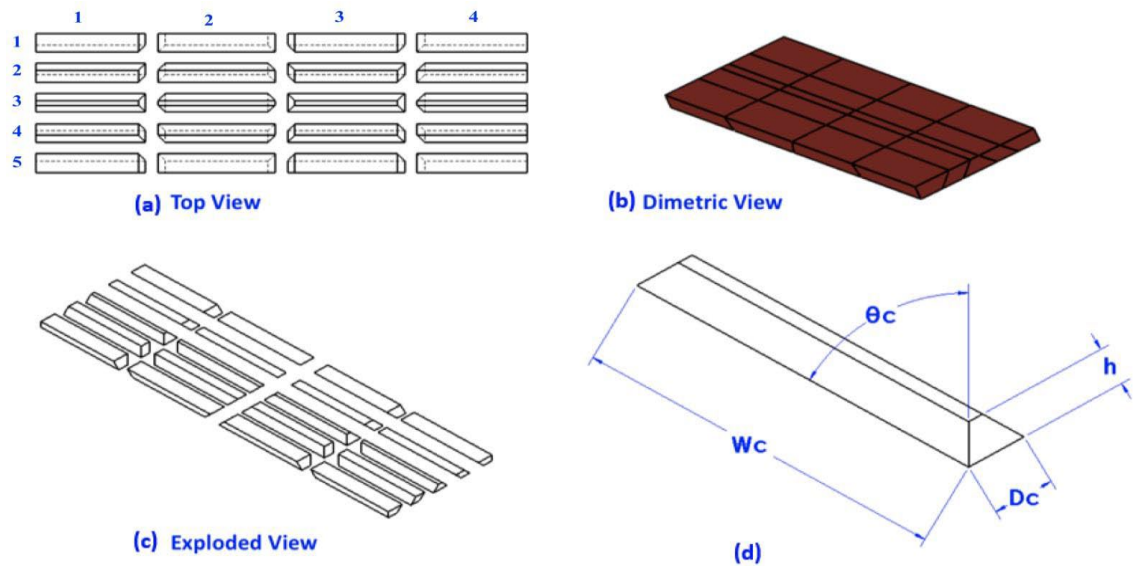

Figure 3-1. (a) Top exploded view of a part showing individual chunks (both rows and columns are numbered). Chunks are reference by row-first ordered pair notation (e.g. $(4, 2)$ *refers to the chunk in the 4th row and the 2nd column. (b) Dimetric view of the part showing chunk's boundary. (c) Dimetric exploded view. (d) Dimension of a center chunk.*

Notably, the sloped interface along each "slice" (i.e. each row or column) is the same for every chunk in that slice. In addition, along a particular slice, a chunk with a positive-sloped

face is produced, as well as a chunk with a negative-sloped face. This results in every chunk having at least one sloped interface (in this example, every chunk has at least two).

In order for each chunk to be feasibly printed on its own, the chunk's dimensions must satisfy the following constraints:

1. As shown in Figure 3-2(a), given the angle of the sloped interface between chunks, θ_c, the angle of the exterior of the extruder nozzle from the horizontal, θ_e, the maximum height of the chunks, h, and the absolute depth of each chunk, D_c, then θ_c must satisfy the following inequalities.

$$\theta_c \leq 90 - \theta_e \tag{12}$$

$$\theta_c \geq \tan^{-1}\left(\frac{2h}{D_c}\right) \tag{13}$$

Equation (12) must be satisfied, otherwise the nozzle will interfere with the printed part of the chunk, introducing an upper bound on θ_c. Equation (13) must be satisfied because the shallowness of the angle disrectly affects how deep the chunk will be, D_c. If D_c is too large, then the printer will not be able to reach the entire depth of the chunk, and eventually collide with printed material. Thus, D_c and h determine the minimum angle allowed for the sloped interface.

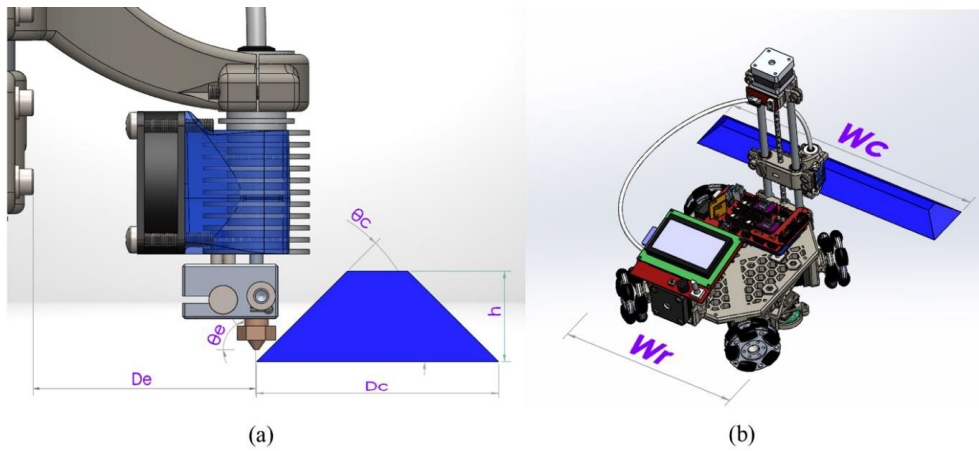

Figure 3-2. (a) Illustration of chunk's dimensions and printing limitations on the slope, and (b) Comparison of chunk width with the width of the robot.

2. Given the reach of the printhead arm D_e, defined by the lateral distance between the point of extrusion and nearest of the robot's wheels and/or chassis, and given the depth of the chunk D_c, the following equation must also be satisfied.

$$D_c \leq D_e \tag{14}$$

This ensures that the printing nozzle can reach the full width of the chunk without collisions with the wheels or chassis.

3. Given the width of the robot, W_r, and the width of the chunk, W_c, as illustrated in Figure 3-2(b), the following must be true to avoid collisions between robots that are printing adjacent chunks in the same row. This constraint only matters for SPAR3 printing.

$$W_c \geq W_r \tag{15}$$

3.1.2 Scheduling

Scheduling consists of two processes: *chunk assignment* and *chunk scheduling*

1. Chunk assignment: An example of a fully chunked model is shown in Figure 3-1(a). Each chunk is assigned to an individual robot. Each row-wise pair of chunks is assigned to the same robot, such that there is a gap between the active robots (robots that are printing) at any given time to prevent collisions during printing. For example, as illustrated in Figure 3-3(a) and (b), chunks $(3, 1)$ and $(3, 2)$ (as represented in Figure 3-1(a)) are assigned to one robot, while chunks $(3, 3)$ and $(3, 4)$ are assigned to the second robot. Additionally, each row of chunks is assigned to the *most appropriate* row of robots, i.e. the row of robots that is closer. This prevents inter-row collisions. For example, the chunks of row 5 are only assigned to robots on a single side of the print: the bottom side.

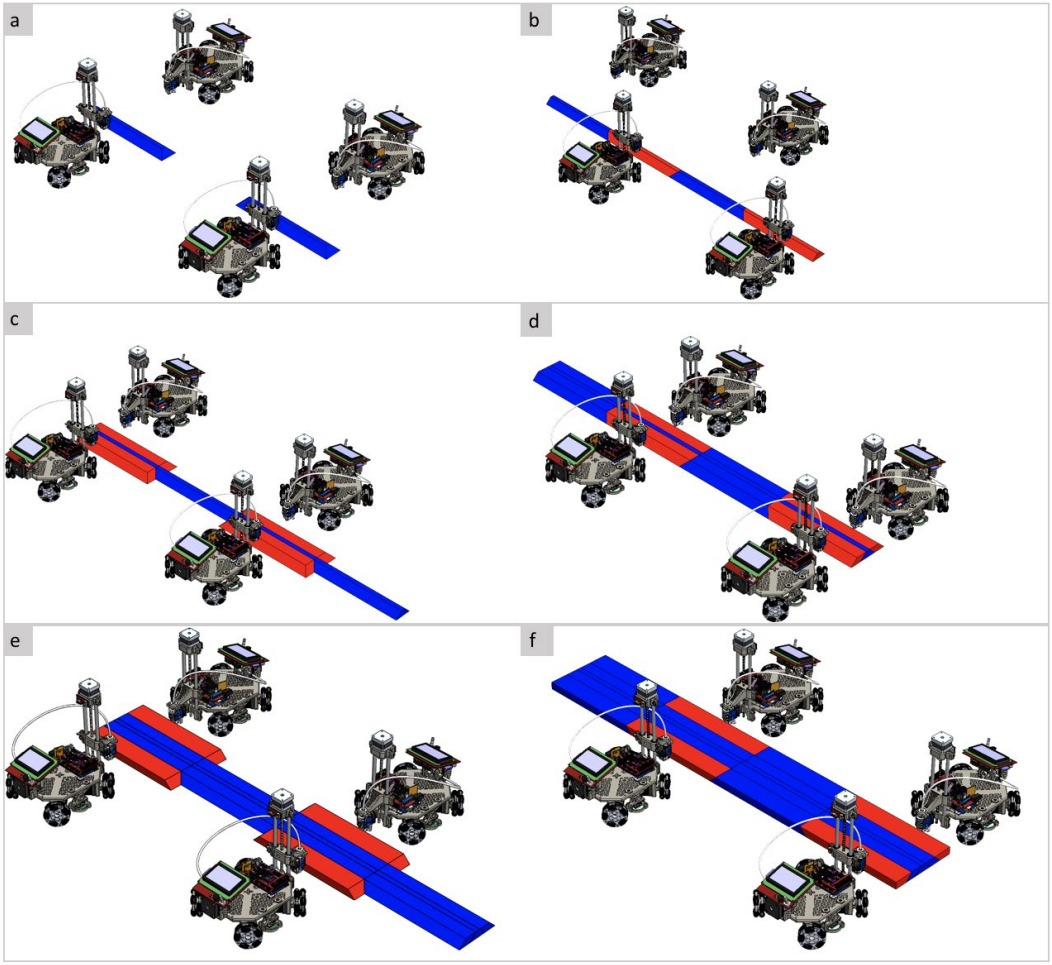

Figure 3-3. Illustration of a rectangular prism being printed using the SPAR3 strategy with four robots. Printing starts with the center chunks seed chunks. Chunks that are being worked on at each step are represented in red color whereas the completed ones are represented in blue.

2. <u>Chunk scheduling</u>: After the completion of chunk division and chunk assignment, a print sequence is generated based on the dependency relationship between chunks. Based on the dependency, the chunks can be labeled by three overlapping types:
    a. Seed Chunk: Seed chunks are the chunks that are printed first in a print job and have a positive bonding slope on all sides unless they are the end chunk. In Figure 3-1, the chunks at $(3, 1)$ and $(3, 3)$ are seed chunks.
    b. Parent Chunk: Parent chunks are chunks that need to be printed prior to

printing some set of other chunks. In Figure 3-1, the seed chunks at $(3, 1)$ and $(3, 3)$ are parent chunks of chunks $(3, 2)$ and $(3, 4)$.

c. Child Chunk: Child chunks are those that cannot be printed until after their respective parent chunks are completed. Child chunks can either be a gap chunk or a dependent end chunk. Chunks that are physically located between two parent chunks are called "gap chunks", such as chunk $(3, 2)$. Chunks that are at the end of a row but are dependent on the completion of their parent are called "dependent end chunks", such as chunk $(3, 4)$. In this case, chunks $(3, 2)$ and $(3, 4)$ are child chunks. Chunk $(3, 2)$ has parents $\{(3,1), (3,3)\}$, while chunk $(3,4)$ has a single parent, $(3, 3)$.

Using the printing object from Figure 3-1 as an example, the printing scenario as a result of the scheduling in conjunction with the sloped surface chunking method is depicted in Figure 3-3. Chunks are assigned to four 3D printing robots and printing begins at the center of the printing area and then expands into two opposing rows of robots. The following describes the execution order of printing for the SPAR3 process.

- Firstly, one row of robots prints the seed chunks, while the other stand by at a safe distance to avoid collisions with the active robots.
- After both seed chunks are fully printed, the active robots move over simultaneously by one chunk to print the gap chunks and dependent end chunks on the same row.

- The active robots retreat slightly to begin printing an adjacent row of chunks. At the same time, the previously inactive robots become active and begin printing the other row adjacent to the center row.
- The parallel sets of active robots independently and iteratively print rows of chunks by (1) printing the parent chunks for that row, then (2) printing the gap/dependent end chunks for that row.

3.1.3 Encoding a Printing Strategy

With a proper description of a single, scalable printing strategy from sections 3.1.1 and 3.1.2, we are now well-equipped to encode this information in a computationally useful way. This encoding allows the performance and correctness of a strategy to be evaluated with ease.

The encoding for a printing strategy uses a directed dependency tree (DDT), a graph that represents the dependency of objects towards each other, specifically in a strictly hierarchical structure. In this tree graph, a node (or vertex) will represent a chunk to be printed, and edges will be directed at a parent that is strictly above the source node in order of execution priority. Every chunk is assigned one node, and every parent-child relationship is assigned one edge. Finally, the DDT must be in transitive reduced form; this means that there are no redundant dependencies encoded in the tree. For example, if chunk A depends on chunks B and C, and chunk B depends on chunk C, only two edges are needed to encode these relationships, namely the relationships $A \rightarrow B$ and $B \rightarrow C$.

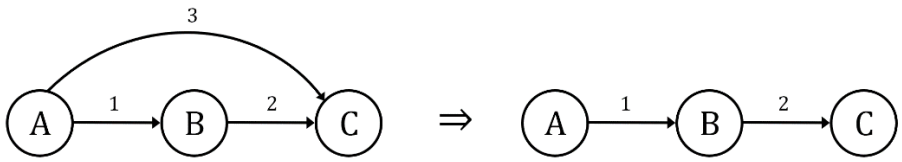

Figure 3-4. Visualization of a simple transitive reduction. Edge 3 is redundant since it is implied that node A depends on node C via A's dependency on node B.

The formal description of this model isn't necessarily visually intuitive, so it is useful to examine a few examples. The simplest cooperative 3D printing example involves only two robots operating in parallel, as shown in Figure 3-5.

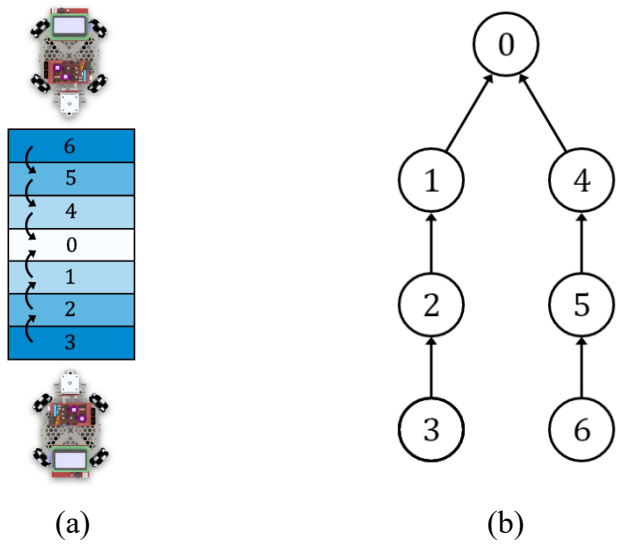

Figure 3-5. Simple two-robot chunking (a) and dependency tree (b)

The DDT of Figure 3-5(b) encodes the following information for the chunk layout from Figure 3-5(a):

- Seed Chunk: The root node of the tree represents the seed chunk. Since seed chunks do not depend on the completion of other chunks, the node representing a seed chunk should only have inbound edges.
- Parent-Child Relationships: Any instance of an edge encodes a child-parent relationship from the source node to the destination node. That is to say, the chunk represented by the source node is dependent on the completion of the chunk represented by the destination node.

From this information, not only is it clear what the execution order of the chunks should be, it is also *computable*. In section 3.2, we dive deeper into the utility of the dependency tree.

Similarly, the SPAR3 process can also be encoded by a dependency tree.

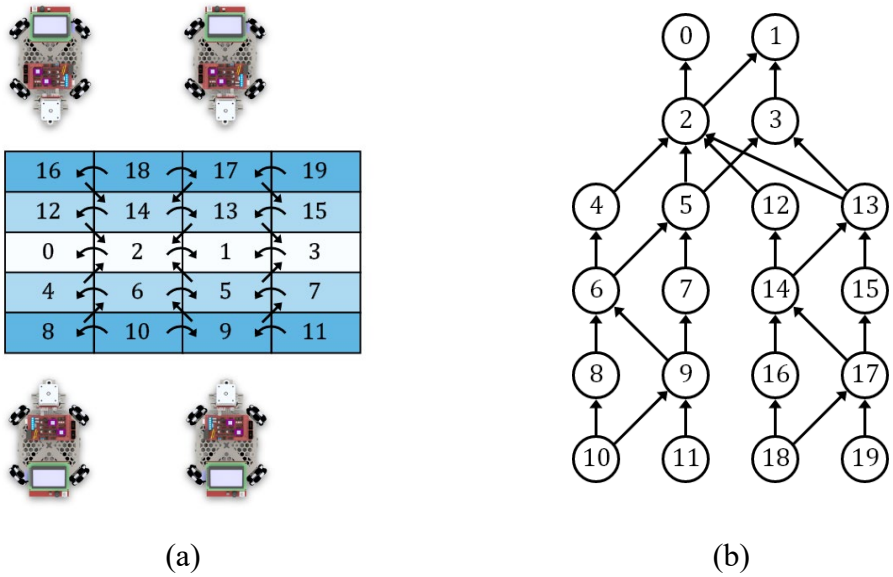

Figure 3-6. Four-robot SPAR3 chunking (a) and its corresponding dependency tree (b)

The DDT in Figure 3-6(b) encodes the same type of information as the DDT in Figure 3-5(b), albeit with a bit more complexity. Captured in this DDT is the following information:

- Seed Chunks: The root nodes represent the two seed chunks, 0 and 1 (i.e. chunks $(3, 1)$ and $(3, 3)$ from Figure 3-1(a)).
- Parent-Child Relationships: Edges represent dependencies from source nodes to destination nodes. In this case, we can see a couple of chunks, for example chunk 2, which depend directly on the completion of multiple chunks before they can be printed.
- Row Dependencies: Examining chunk 5, for example, we can tell from the chunk layout that chunk 5 doesn't *physically* depend on chunks 2 and 3, however this dependency is necessary to encode collision avoidance between rows. Printing chunk 5 simultaneously to chunks 2 or 3 would result in a robot collision. Thus, we introduce additional dependencies to force robots to work on a single row at a time. Chunk 5 does not have a labelled dependency on chunk 1 because, as stated at the beginning of this subsection, a DDT encoding a printing strategy must be in transitive reduced form.

3.2 Evaluation Framework

An evaluation framework, in this thesis, refers to a set of equations and methods for deriving pre-print metrics and validation checks from a set of chunks and a corresponding DDT. Such a framework will prove to be very useful for future development of this field, as it provides an extensible library of well-defined functionality standards for cooperative 3D printing.

As part of the development of a strategy evaluation framework, it's important to name the metrics we're most concerned with. Most importantly, this framework should tell us, given a proper dependency tree, how long a print should take according to the schedule encoded by the tree, called the *estimated execution time (EET)*. Without knowing the EET of a schedule, we can't know if the print is going to be more efficient than using a single printer. In manufacturing environments, validating this technology's promise of speedup is essential.

The EET will be the metric of focus in this thesis, while further metrics and validation steps are left as topics for future work. For example, determining the maximum parallelizability of a DDT, validating that a DDT does not produce collisions, etc.

3.2.1 Estimated Execution Time (EET)

The EET of a DDT can be calculated according to the equations presented in this subsection, assuming the following is given:

- D, a proper DDT. Refer to the beginning of section 3.1.3 for a full description of a proper DDT.
- C, a set of unique, non-intersecting chunks. These chunks should correspond one-to-one with the nodes of the DDT. Each chunk should carry enough information to determine the chunk's printing time if it were to be printed by itself.

For a given DDT, D, and set of chunks C, let c_i be the chunk corresponding to node i. For chunk c_i, let $t(c_i)$ represent the amount of time a robot takes to print c_i. We will use the abbreviation $t(c_i) \rightarrow t_i$. Assuming the entire print begins at $t = 0$, let $T(D, c_i)$ represent the

time after $t = 0$ to complete printing chunk c_i. For example, if c_0 is a seed chunk, then $T(D, c_0) = t_0$. If c_1 depends on c_0, then $T(D, c_1) = T(D, c_0) + T(c_1) = t_0 + t_1$.

More formally, the definition of $T(D, c_i)$ is as follows:

$$T(D, c_i) = \max(\{T(D, c_m) \mid m \in c_i.\text{deps}\}, 0) + t_i \tag{16}$$
$$\textit{where} \quad i \in D.\text{nodes}$$
$$c_i.\text{deps} = \{m \mid (i, m) \in D.\text{edges}\}$$

For example, for the DDT shown in Figure 3-6(b), D, the time at which chunk c_5, which has chunks c_3 and c_2 as dependencies, finishes printing is given by:

$$T(D, c_5) = \max(\{T(D, c_2), T(D, c_3)\}, 0) + t_5$$

$$T(D, c_5) = \max(\{\max(\{T(D, c_0), T(D, c_1)\}, 0) + t_2, \max(\{T(D, c_1)\}, 0) + t_3\}, 0) + t_5$$

$$T(D, c_5) = \max\{\max\{t_0, t_1\} + t_2, t_1 + t_3\} + t_5$$

$$T(D, c_5) = \max\{(t_0 + t_2), (t_1 + t_2), (t_1 + t_3)\} + t_5$$

Given the definition of $T(D, c_i)$, we can express the total estimated execution time of the entire print represented by D as the maximum of every possible value of $T(D, c_i)$. More formally:

$$EET_D = \max\{T(D, c_i) \mid i \in D.\text{nodes}\} \tag{17}$$

3.3 Implementation Details

The scaled chunking process follows the same overall process as the two robot chunker, however, simulating scaled cooperative 3D printing requires more complexity. Many robots simulated in the same environment need to behave similarly to how they would in real-life, meaning they need to robustly execute a respond to "wait" and "notify" commands. Generalizing the simulation process for many robots should maintain compatibility with 2-robot slicing, such that any 2-robot or n-robot simulation can be simulated by the same process.

At a very basic level, each robot is simply executing a linear set of G-code commands. Were we to simulate a single robot executing commands, the simulation process would take a very simplistic form, like that shown in Figure 3-7.

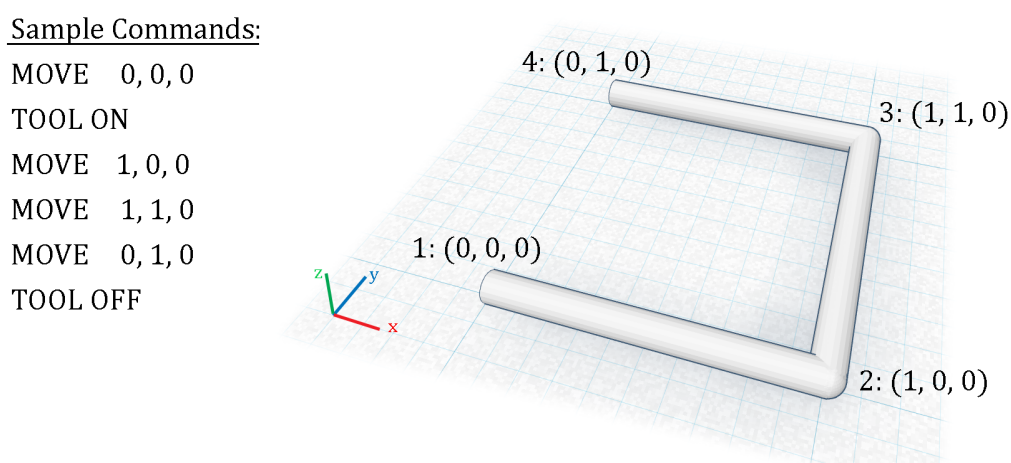

Figure 3-7. Simple simulation example. The extruded cylinder resulting from the commands can be represented as a "Material" object in memory.

Given that the only events that take place in a single-robot simulation are robot movements and material placement, it would be sufficient to represent a simulation over time with

"frames", where each frame contains a robot state (i.e. the robot's position) and a list of all "Material" objects that were newly extruded during the duration of that frame.

A multi-robot simulation, however, requires more processing than a simple linear progression through a set of commands. Each robot needs to be responsive to the state of every other robot. The purpose of this section is to describe the mechanisms that were developed to simulated a cooperative 3D print.

To thoroughly cover the implementation details from a language-agnostic standpoint, this section will cover high level processes and data structures, then procedurally expand the implementation details of the broad concepts.

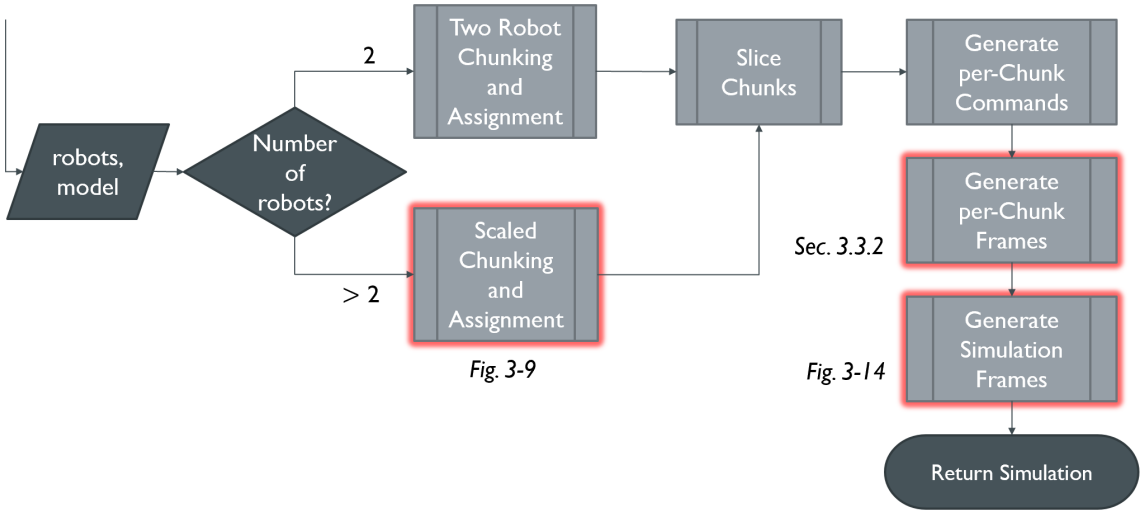

Figure 3-8. Scalable Cooperative 3D Printing process overview. New additions are highlighted red. The point of the process is to accept a list of robots and a 3D model and to return a fully calculated simulation in the form of "video frames".

Figure 3-8 shows the high-level overview of the scalable 3D printing process. The goal of this process is to accept a list of robots and a printable 3D model and return the "video frames" for the simulated printing of the input model using the input list of robots. These video frames

are not actual images made up of pixels, but rather descriptions of the state of the system at a specific point in time. Hence, a “frame” in this research is defined as the minimal representation of a new scene state given an older scene state. In our case, a scene state only corresponds to the positioning of robots and the structure of static material.

Notably, the slicer and command generation processes will remain the same as the two-robot printing process. Like with two-robot printing, every chunk must be sliced then converted to a collection of commands needed to print that chunk. Each robot knows which chunks it is responsible for and can therefore execute the commands for each of its chunks in order. Otherwise, the robot has predefined routines for printing movements that are not directly related to printing a chunk, e.g. transitioning the print head between two chunks.

Before expanding the implementation details of the new components in follow-up flowcharts, it’s important to be aware of the data structures that will be used. In order to maintain chunk printing order, robot-chunk ownership, etc., the data will be stored and managed according to the UML diagrams in Figure 3-9.

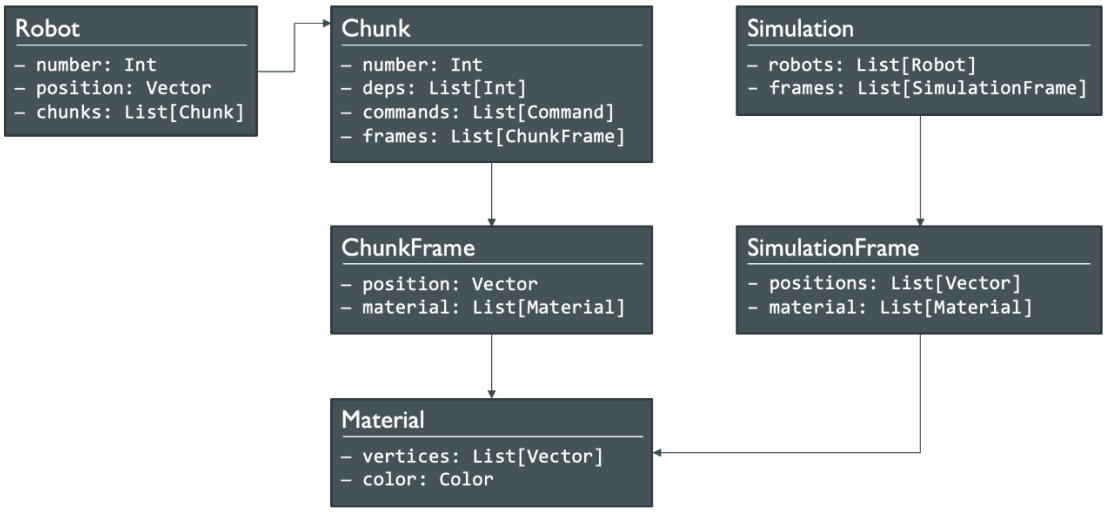

Figure 3-9. Simplified UML Diagrams for the main data structures. Full diagram given in Appendix A. Robots know their current location (in a scene), and the chunks they've been assigned to print. Chunks know their dependencies, commands, and "chunk frames". Simulations know the robots in the simulation and the frames that comprise the time-varied positions of robots and material.

The description of Figure 3-9 briefly explains many of the important structural relationships for these classes. A final, important note relevant to this figure is related to the different types of frames. There are two ways to represent a frame – one that applies only to the position and material placement relevant to a single robot-chunk pair, and another that applies to the positions of all robots and the placement of all material within a given frame. The mechanism behind the formation of SimulationFrames will be explained later in this section.

The ultimate product of the simulator is a single "Simulation" object, shown in Figure 3-9. This object can then be interpreted frame by frame by any renderer capable of producing visuals corresponding to the "Simulation" object – for example, a video file, a Blender scene filled with 3D objects, or a scene in any other 3D CAD program.

3.3.1 Scaled Chunker

If more than two robots are present in the scene (specifically an *even* number of robots greater than two), then the scaled chunker is used to subdivide a model into chunks.

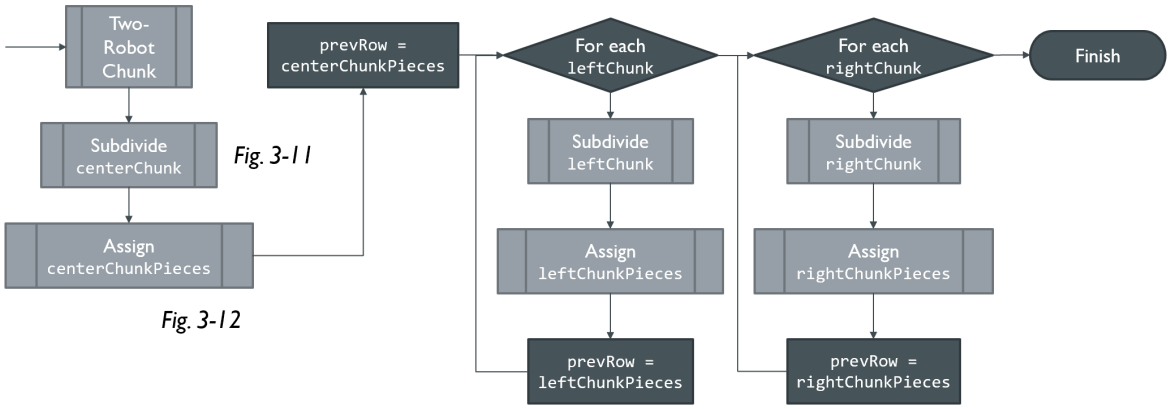

Figure 3-10. SPAR3 chunking process overview. The chunker is responsible for (1) subdividing the main model, (2) assigning the newly created pieces to the appropriate robot, and (3) calculating the chunk dependencies.

Referring to Figure 3-10, notice that the SPAR3 chunking process is simply an extension of the Two-Robot Chunking process. The Two-Robot process splits a model into parallel rows of chunks. The purpose of the Scaled process is to further divide these rows into adjacent chunks. The beginning of section 3 covers this similarity in more detail.

The gray processes *Subdivide* and *Assign* refer to the processes in Figure 3-11 and Figure 3-12.

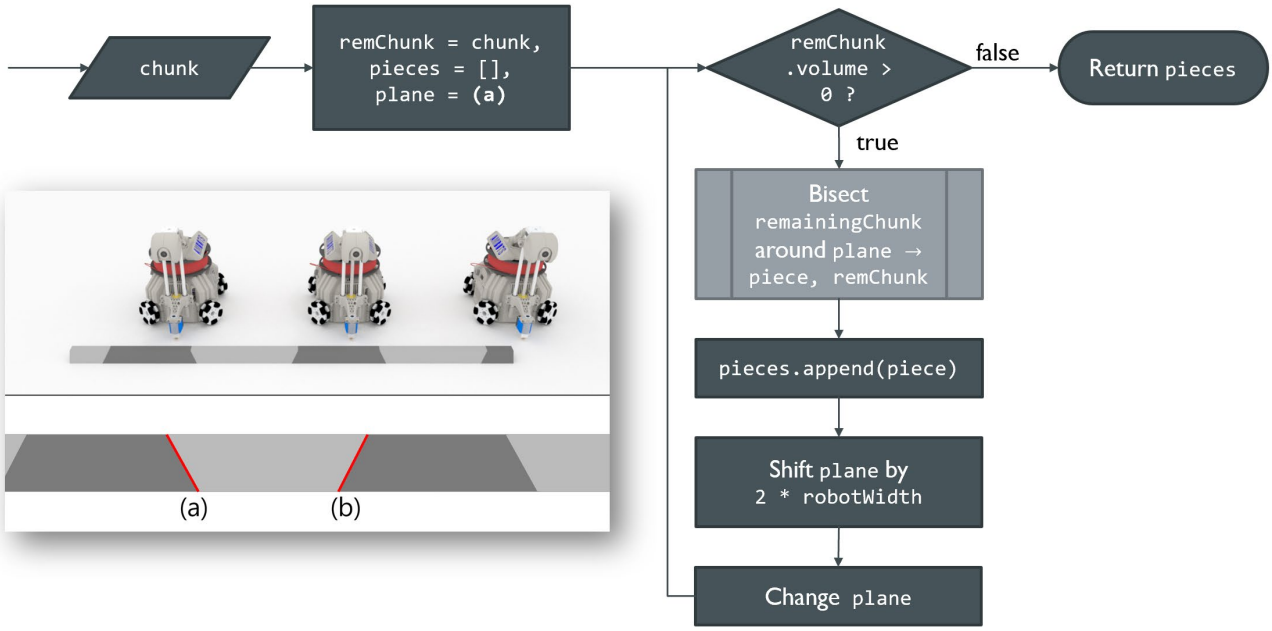

Figure 3-11. Chunk subdivide process overview. Given an output chunk from a two-robot chunking process, this process subdivides it using the two arrays (a) and (b).

The Chunk Subdivide process is responsible for splitting a single chunk row (produced by the Two-Robot Chunker) into smaller length-wise chunks. It makes use of two alternating planes, (a) and (b), that are iterated over the length of the chunk. As each plane is iterated, the row chunk is *bisected* using the current plane as the separator. More about bisection is discussed in section 2.2.4.

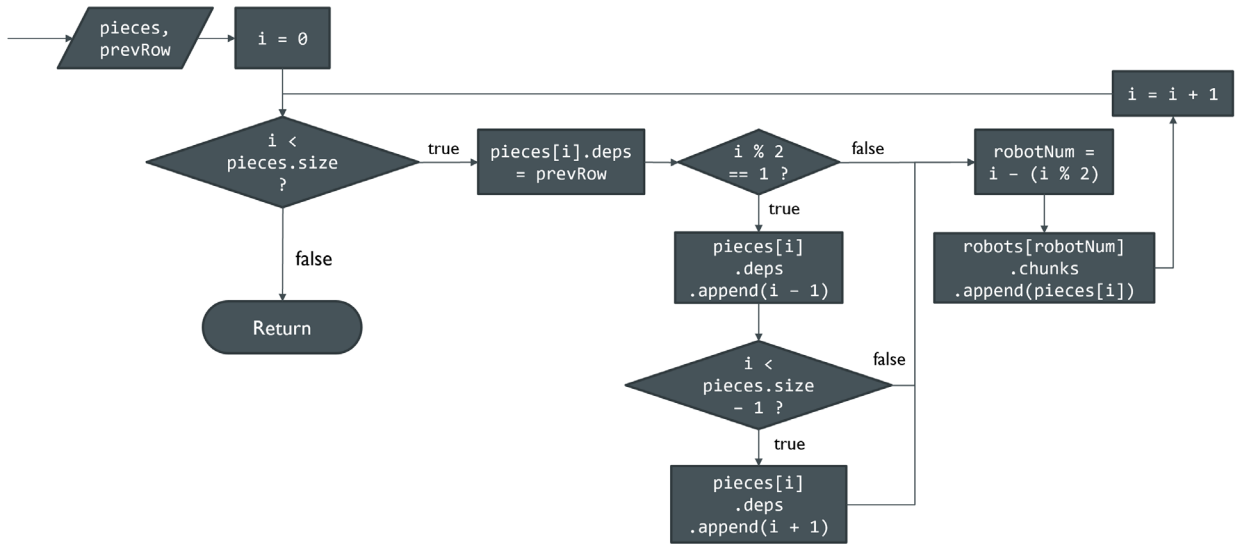

Figure 3-12. Chunk assignment process overview. The idea is that every robot will receive two subsequent chunks after the execution of this process and that the dependencies of each chunk include the previous row and any dependencies enforced by SPAR3. Refer to the Glossary for definitions of shortened terms in this flowchart.

The Chunk Assignment process is responsible for assigning each finalized SPAR3 chunk to the appropriate robot, as well as assigning each chunk the correct chunk dependencies, as defined by the SPAR3 strategy. This step is crucial for collision avoidance. The robot to which a chunk belongs can be determined simply by a chunk's order in the row. The dependencies of a chunk are determined by the *chunk assignment* and *chunk scheduling* definitions described in detail in section 3.1.2.

3.3.2 Single-Chunk Simulation

As mentioned in the beginning of section 3.3, a single robot in an isolated environment can be easily simulated in such a way that each frame of a hypothetical video of the simulation can be represented by a simple data object, as shown in Figure 3-13 below.

Commands:

```
TOOL ON
MOVE  1.5,  0, 0
MOVE  1.5, -1, 0
MOVE    1, -1, 0
TOOL OFF
MOVE  0.5, -1, 0
TOOL ON
MOVE    0, -1, 0
...
```

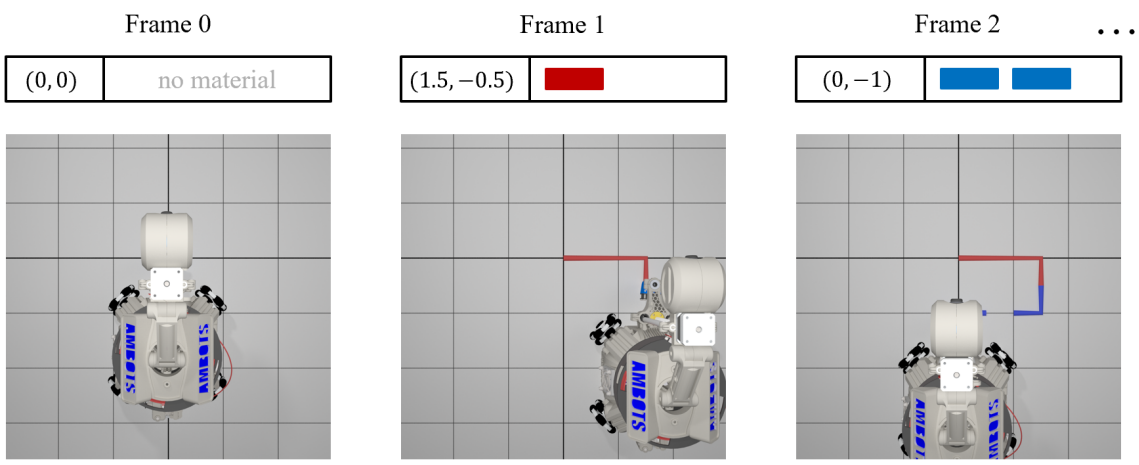

Figure 3-13. Example of a single-robot simulation. The robot starts at position $(0, 0)$*. Commands are executed in order. The robot is assumed to move at a speed of 2 units per frame. Commands that are not completed within a single time frame generate multiple pieces of material for the same command. The differing colors in material are for visualization purposes only. Notice the small piece of material beneath the robot's printhead in frame 2. The coordinates used by the MOVE commands are absolute coordinates.*

Following this example, a set of commands has supposedly been generated by the slicer. Any commands between consecutive "TOOL ON" and "TOOL OFF" commands should extrude material at the robot's extruder following the robot's "MOVE" commands. If a time

frame contains multiple such groups of “MOVE” commands, multiple “Material” objects are generated for that frame. In the event that a robot would not fully complete a command before the end of any given frame, the robot’s position at the end of the frame is linearly interpolated to the position where the robot is located at the end of the frame. In addition, the material that is extruded for any unfinishable commands is linearly interpolated to the robot’s position at the end of the frame.

The commands generated by the slicer for each chunk can independently be converted to an array of frames, following the example from Figure 3-13. Each chunk keeps this array of “ChunkFrame” objects (see Figure 3-9) in its “frames” property, in preparation for further processing by the scaled simulator.

3.3.3 Scaled Simulation

The generalized simulation process is responsible for making sure all chunks are printed according to the commands generated by the slicer. The simulation process is also designed to ensure no robot begins printing its next chunk before the dependencies for that chunk are satisfied. The process generates a list of “simulation frames”, each of which contains the position of every robot for that particular video frame and a list of “material” objects that were generated (or “printed”) in the time interval represented by that frame.

In order to achieve these goals, the process needs to keep track of various markers. The states of these markers will determine the actions that occur in the frame that is currently being evaluated. These markers include:

- The finished state of each robot,

- the list of finished chunks,
- the current chunk a robot is printing, or ready to print, and
- the current progress a robot has made through its current chunk

These markers are sufficient to create a state machine that accurately simulates the progression of the cooperative 3D print job. Figure 3-14 demonstrates the process overview for the simulation generation.

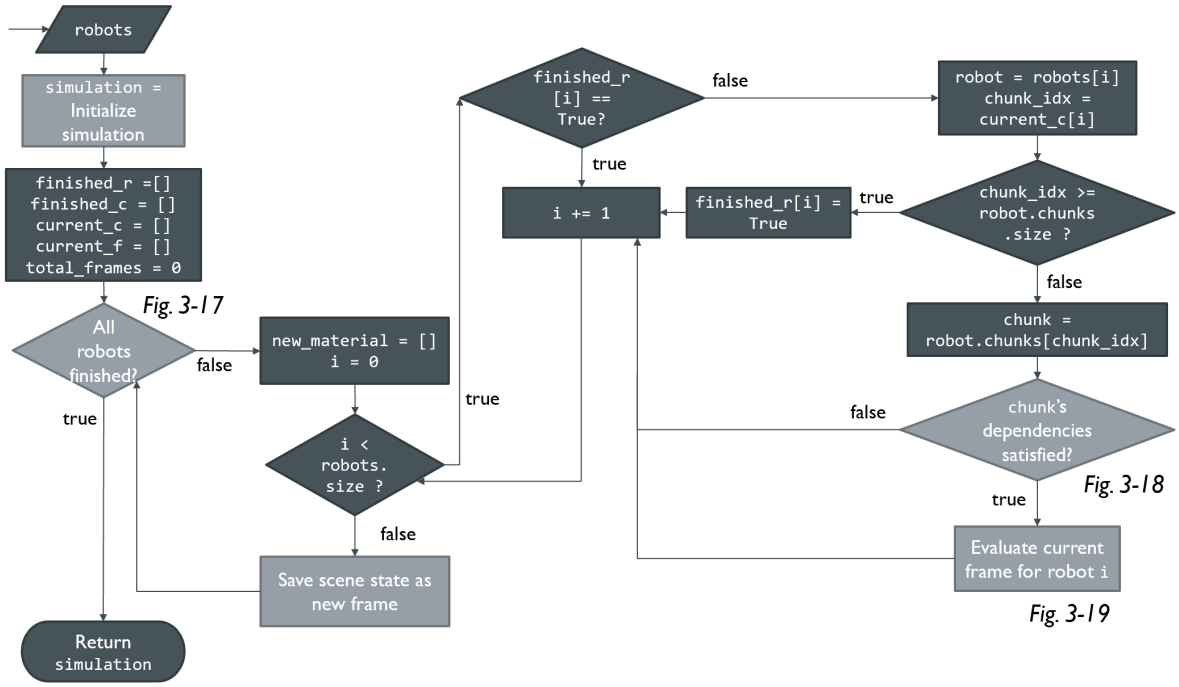

Figure 3-14. Simulation generation process overview

The point of this algorithm is to join the individual chunk frames into a single consecutive list of simulation frames in such a way that a video produced from the simulation frames would show no robot collisions and no floating material. Most of these safeguards are ensured by previous processes related to chunk dependency assignment and chunking. However, it is this

process's responsibility to produce the final set of simulation frames cohesively. Figure 3-15 and Figure 3-16 help visualize the function of this process.

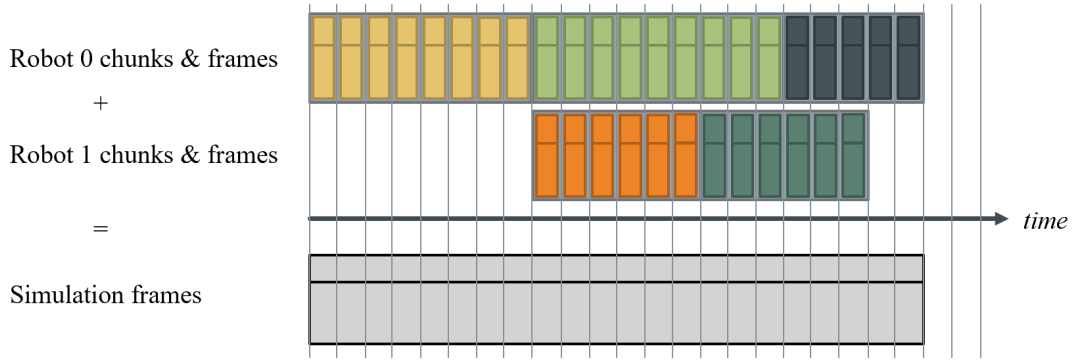

Figure 3-15. Visualization of chunk frames being joined into simulation frames. The gray vertical lines separate frames. The colored rectangles represent the data in each chunk frame. The joining of multiple chunk frames into a single simulation frame is not finely visualized here.

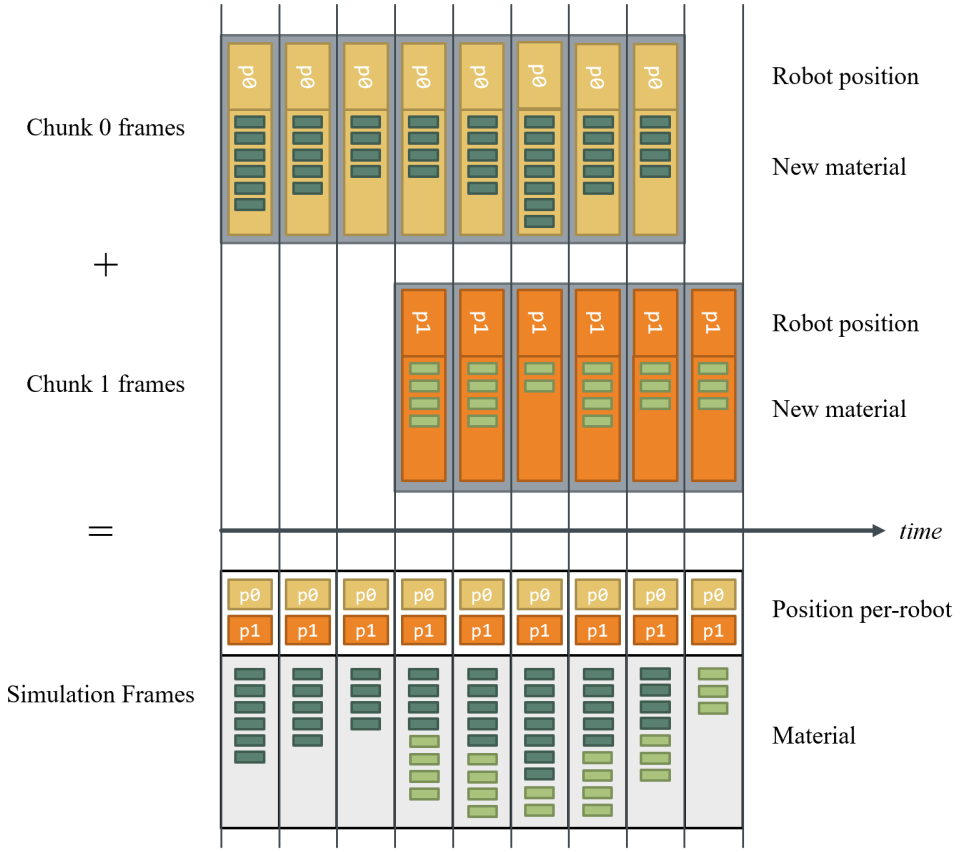

Figure 3-16. Finer resolution of chunk frames being joined into simulation frames. The simulation frame possesses the position information from each chunk frame. All material from multiple chunk frames are joined into the resulting simulation frame.

The details of the process in Figure 3-14 are designed to achieve the results visualized in Figure 3-15 and Figure 3-16. The main branch, *All robots finished?*, is the main loop of this process. As long as this evaluates to *false*, then there are still machines that need to evaluate a new state for the upcoming frame. The following figure demonstrates the process for determining if all robots have finished printing based on the marker object "`finished_r`".

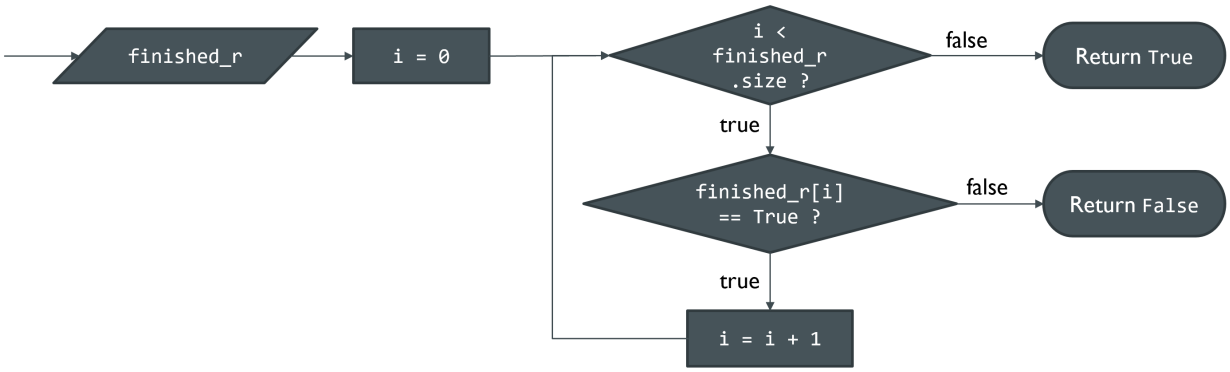

Figure 3-17. Detect all robots finished process overview. "`finished_r`" (short for "finished robots") is an array with the same length as the number of robots. The boolean values in the array indicate whether the machine corresponding to that index is finished or not.

In the generalized simulation process overview, the next important branch is a loop that iterates over every robot. For each robot, some logic needs to occur to determine what should happen for that robot in the current frame. For example, if a robot is marked as "finished", it doesn't need to be evaluated, so it is skipped. If a robot is not *marked* as finished, but it has completed all of its chunks already, it is appropriately marked as "finished", then skipped. Finally, if a robot is determined to be "in progress" (i.e., the opposite of "finished"), then the final branch detects whether the robot's current chunk has all of its dependencies satisfied. If not, the robot is skipped. However, if the current robot's current chunk's dependencies are

satisfied, then a different subroutine, *Evaluate frame for current robot* `i`, is called for that robot. The logic for detecting whether dependencies are satisfied is demonstrated in Figure 3-18.

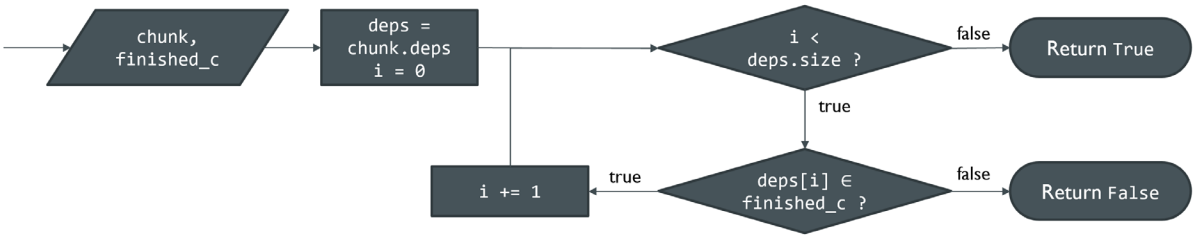

Figure 3-18. Dependency satisfaction process overview.

The *Evaluate frame* subroutine, as shown in Figure 3-19, is responsible for manipulating the current scene (i.e. moving the robot and adding new material) according to the state markers mentioned above.

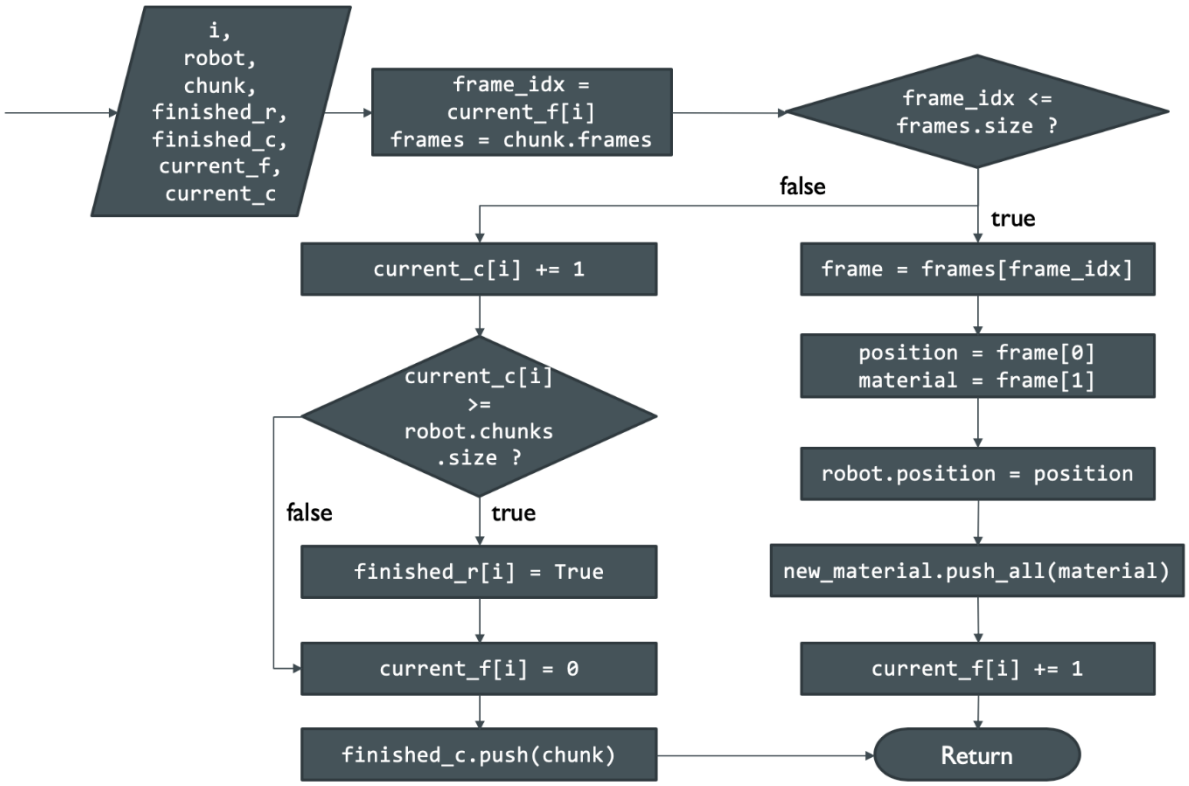

Figure 3-19. Evaluate frame process overview. "`frame_idx`" maintains the current chunk frame a robot needs to evaluate. If this index is larger than the actual number of frames for that chunk, then the process needs to set the robot's current chunk to the next chunk. Otherwise, the simulation frame needs to be updated with the chunk frame's data.

The process in Figure 3-19 evaluates the new state of a machine if it is currently working on printing a chunk. If it is working on a chunk, the right branch is followed, and if not, the left branch is followed. The right branch relocates the robot to the position held by the current frame object and adds the material in the frame object to the global "`new_material`" object. The left branch sets the robot's current chunk to its next chunk once it has finished all of the frames of a given chunk.

Once every machine has had its state evaluated, a new simulation frame is generated and added to the end of the list of simulation frames (stored in the "`simulation`" object). Once all the machines are marked as finished, the simulation has completed.

3.3.4 Print Time Estimation

Estimating the print time of a cooperative 3D print was formally defined in section 3.2.1. This section serves as the implementation details of the evaluation of EET_D in Equation (17). As stated previously, it is useful to know *before* executing a simulation roughly how long the print will take. Examples use cases include stopping the process early before an unsatisfactory print is evaluated any further, amongst other user-specific needs.

The function $t(c_i)$ is evaluated based on the total length of time taken to execute all commands for chunk c_i. Therefore, estimating the print time requires the command-generation of all chunks. Because each robot's speed is known, it is possible to calculate the time taken to print an individual chunk. In fact, the "per-chunk frame" generation process already calculates the length of time needed in time units of "frames". In this sense, it is sufficient to describe the length of time needed to print a chunk as a multiple of the number of frames needed to *simulate* the chunk. Thus, the print time estimation function will very closely approximate both the simulated print time and the real-world print time.

The implementation of this print time estimation function depends on augmenting the existing directed dependency tree (DDT) with an additional node value, the *execution time*. For each node in the graph, to estimate the time to finish printing that node and every one of its dependencies, each of its dependencies must be queried for their execution times, hinting at an intuitive recursive algorithm.

The estimated execution time of the printing of a single chunk is directly proportional to the number of chunk frames that a chunk possesses, so the evaluation of $t(c)$ runs in constant time, $O(1)$, since the number of frames for a chunk is assumed to be known at this point. The

estimation algorithm has the same time complexity of walking the dependencies of every node in a transitive reduced graph, $O(n^2k)$, where n is the number of chunks and k is a placeholder for the complexity of the operations at each visit to a node. Because each visit to a node only executes constant time operations – i.e. $O(1)$ – the whole algorithm runs in $O(n^2)$. Again, this is assuming the DDT is, indeed, in transitive reduced form. If not, then it must first be transitively reduced, which has a worst-case complexity of $O(n^{2.37})$ (i.e. the time complexity of matrix multiplication [18]). The transitive reduction step isn't necessary, but it is the only way to guarantee a well-defined runtime complexity. Otherwise, a highly connected dependency tree could cause exponential runtime. It should be noted that there are rarely going to be enough chunks such that runtime will be prohibitively long. There are very few potential use cases involving chunks numbering in the thousands.

The process diagrams for the estimated execution time algorithms are shown in Figure 3-20 and Figure 3-21. Before these processes execute, it is assumed that a DDT has been constructed from the generated chunks.

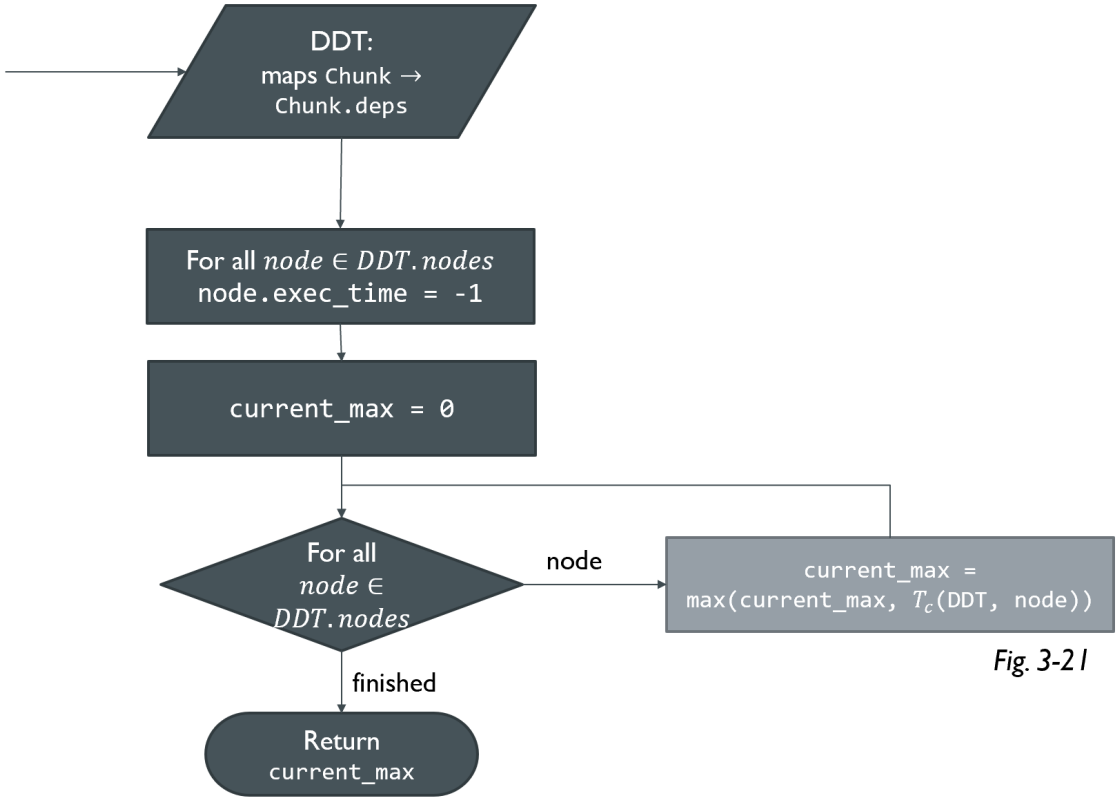

Figure 3-20. Estimated execution time process overview. This is the main function. Each node's execution time is initialized to "-1", then the function T_c is called n times, once for each node. The maximum value of T_c is returned as the estimated execution time of the print job represented by DDT.

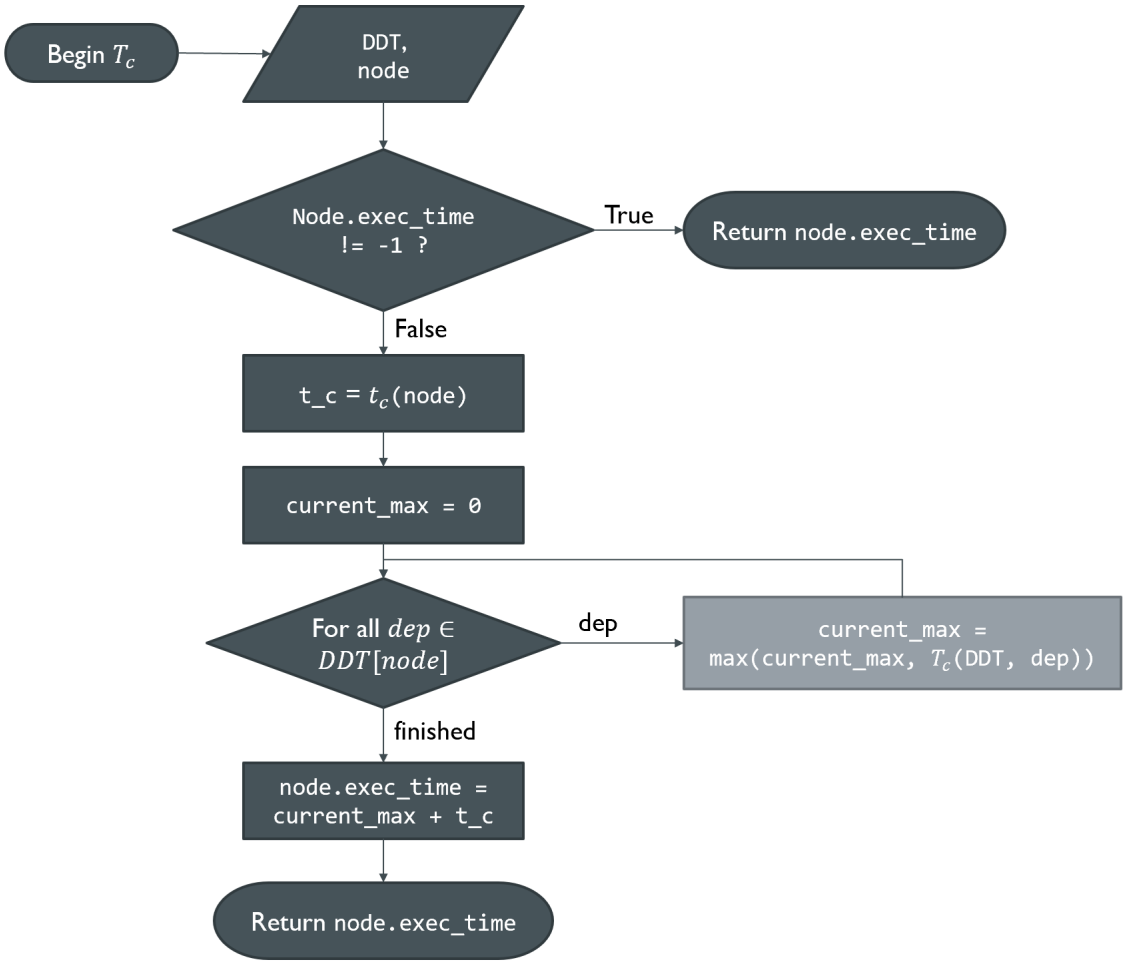

Figure 3-21. Process overview for the function T_c. If "exec_time" has already been calculated (i.e. it is not -1), then return that value for T_c. Otherwise, calculate $T_c(DDT, node) = max\{T_c(DDT, i) \mid i \in DDT[node]\} + t_c$.

3.4 Simulation Results

The SPAR3 strategy was tested on two different 3D models, the first a simple rectangular prism and the second a complex topographical map of Arkansas. The results in this section compare the estimated time (as discussed in section 3.3.3) to the actual simulated time. The time value (in hours) is determined by multiplying the number of frames in the estimation/simulation by the amount of time represented by each frame.

Because the simulator executes commands as if they were real-world G-code commands, the simulated time to print is very close to the real printing time. In order to reduce the amount of time needed to render, each frame represents 140s of real-world print time. The testing methodology is to compare the number of frames needed to fully print a 3D object across three strategies: (1) Single robot printing, (2) Two robot printing, (3) the SPAR3 strategy with up to 16 robots, while measuring both the estimated time (described in section 3.2.1) and simulated time.

Table 3-1 shows the constant parameters for the simulation environment.

Table 3-1. Parameter settings for the simulation

Robot width	16cm
Robot build depth	4cm
Robot printhead slope	60°
Slice thickness	1.6mm
Infill type	Solid
Time per frame	140s

The first model to be tested is a low-height rectangular prism with dimensions 280cm × 24cm × 2cm (total volume of 13,400 cm^3). Periodic snapshots of the printing process are shown in Figure 3-22, numbered 1 through 6. The second model is a topographic map of the state of Arkansas, approximately 232cm × 87cm × 2.5cm (total volume of ~19,524 cm^3). The printing process of this model is illustrated in Figure 3-23 and is numbered 1 through 6.

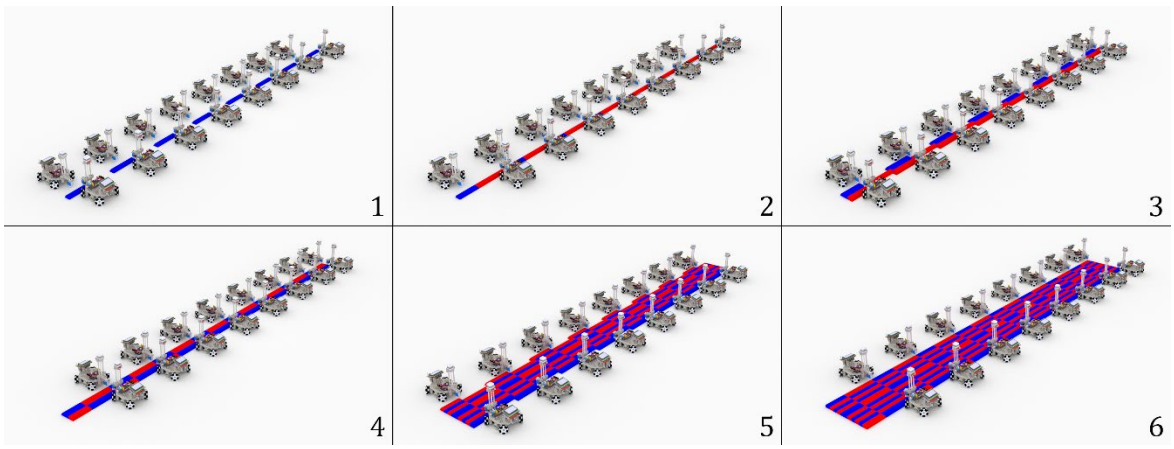

Figure 3-22. The rectangular prism being printed from start to finish.

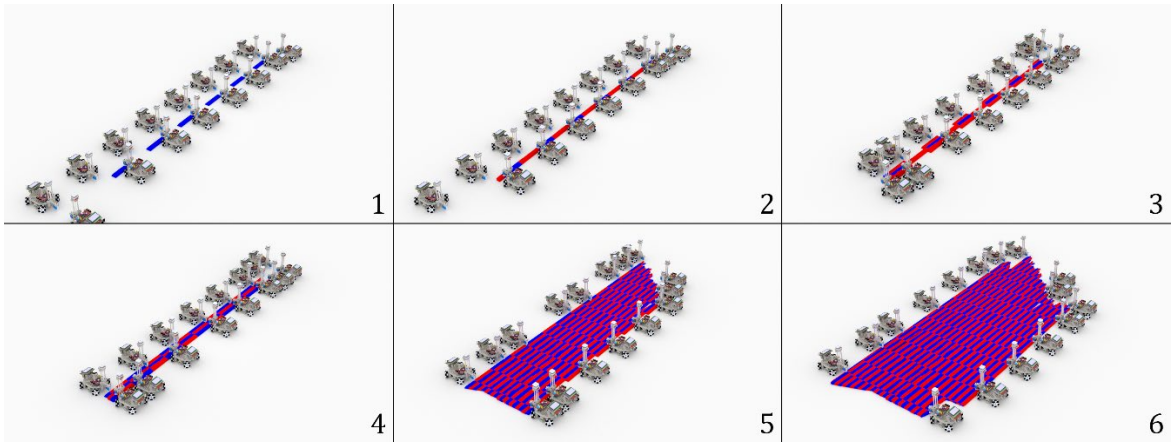

Figure 3-23. The Arkansas topographic map being printed from start to finish.

Table 3-2 shows the estimated and simulated time in hours taken for both printing jobs. The estimated time is calculated based on Equation (17), whereas the simulated time is the total time the robots took to complete the print job in the simulation.

Table 3-2. Estimated time vs simulated time of printing a rectangular prism model (left) and Arkansas model (right). The first column provides the number of printing robots.

	Rectangular Prism Model			**Arkansas Model**		
Robots	Estimated Time (h)	Simulated Time (h)	Speedup	Estimated Time (h)	Simulated Time (h)	Speedup
1	188.46	190.63	N/A	257.02	258.00	N/A
2	120.09	120.09	1.59	162.24	162.21	1.59
4	61.41	60.98	3.13	101.81	101.7	2.54
6	41.49	41.34	4.61	74.9	72.99	3.53
8	31.34	31.31	6.09	60.16	58.14	4.44
10	25.51	25.39	7.51	49.89	47.76	5.40
12	21.39	21.39	8.91	42.93	40.13	6.43
14	18.51	18.51	10.03	37.88	36.59	7.05
16	16.57	16.53	11.53	37.68	35.58	7.25

Table 3-2 reveals two important trends. First, the results indicate that the SPAR3 printing process significantly speeds up the time taken to print. For example, if 6 robots are used to print a rectangular prism instead of 1, the print time shortens from almost 191 hours (nearly 8 days) to roughly 41 hours (less than 2 days). With only two robots, we already see a 60% speedup in print time. However, with the new SPAR3 scaling strategy, we are able to parallelize the workload to at least 16 robots, potentially more given a sufficiently large model to print.

The speedup grows linearly with the number of robots in the case of the rectangular prism, as shown in Figure 3-24. The same growth also shows linear growth for low numbers of robots for the Arkansas model, but the growth slowly decreases as the number of robots increases. We expect the rectangular model to drop off similarly as the number of robots increases past 16. This is expected because there isn't any added benefit in adding robots beyond a certain point for a model of a fixed size. Once the number of robots to fully parallelize is reached, adding more robots will not result in a reduction of print time as the additional robots will not be utilized for printing. The upper limit (number of robots) at which the speedup stops improving can be obtained using the modified Amdahl's law from section 23.

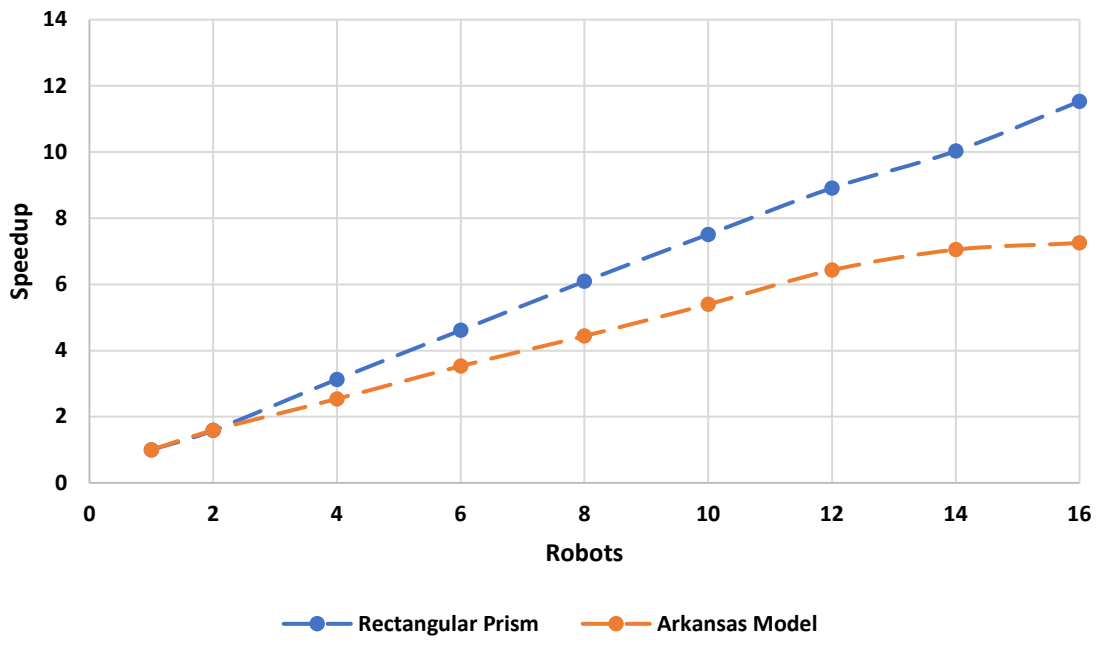

Figure 3-24. Graph showing the total number of robots used vs the speedup of execution of printing task

The rectangular prism works very well with the SPAR3 strategy due to the fact that all the chunks have roughly the same volume and every robot has the same number of chunks to print.

This minimizes the amount of time that some robots spend waiting for their dependencies to be satisfied. The Arkansas model, however, has chunks of varying volume (some chunks being completely empty with 0 volume). This causes multiple robots to wait for long periods of time before they can begin new chunks. As a result, the Arkansas model sees worse speedup. This leads to the conclusion that if we want to minimize the total print time using the SPAR3 strategy, it is ideal to have more uniform volumetric sizes of chunks, since a chunk's printing time is linearly correlated with its volume.

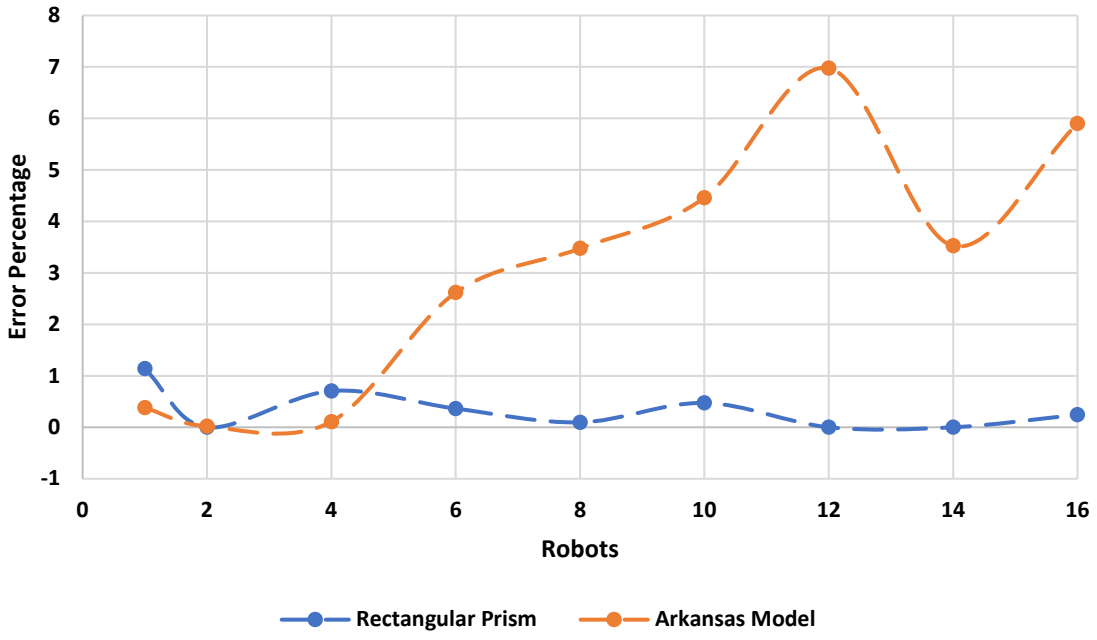

Figure 3-25. Graph showing the total number of robots used vs the error percentage.

The second important trend revealed by Table 3-2 is the relationship between estimated print time and simulated print time. Figure 3-25 provides the graphical representation of the error percentage calculated using estimated time and the simulated time against the total number of robots used. The estimated time for the rectangular prism shows a much lower error

percentage (at most, 1.15%) and stays relatively steady as the number of robots increases. However, the Arkansas model fluctuates frequently, and is generally higher as the number of robots increases.

The scaling discrepancy when evaluating the Arkansas model's scalability is also related to the non-uniform volume of chunks. This is evident when we consider that any given chunk cannot be printed until its dependencies are fulfilled. If some chunks have a larger volume, then they will take longer to print than other chunks. This means that there are some chunks that cannot begin printing due to a single dependency with a high volume. This effect cascades through the entire dependency tree, extending the overall time of the print due to a few chunks with a disproportionate material volume.

Finally, the error trend for the Arkansas model is not as obvious as that of the rectangular prism. Regardless, the print time estimation algorithm described in this chapter provides a faster approach to estimate the printing time with reasonable accuracy.

4 Implementation of Chunk-based 3D Printing

As it stands, the processes and software outlined in the sections leading up to this chapter provide a solid theoretical foundation for real-world implementation, as well as an accurate evaluation framework for gathering pre-print metrics. However, there are questions and challenges that the research up to this point does not address. These questions and challenges serve as a principled guide to the topics of this chapter, with the goal to address them as reasonably as possible given the theoretical scope of this research.

Firstly, the bridge between the Cooperative 3D Chunker and real-world robots is missing. At this point in the story, we are missing the crucial element that makes this research meaningful: it's applicability to parallelized, cooperative, real-world printing. The following subsection addresses this issue with discussion of software that has been developed in conjunction with this research.

Secondly, this research hasn't addressed the usability of the simulation output. Specifically, no mention has been made about the exportability of a simulation calculation to other software formats. Without addressing this issue, the simulation's usability is directed solely by the developer of the software, unless a user reverse-engineers their own simulation solution. In the final subsection of this chapter, we will discuss this issue further and offer a solution.

The portability of the simulation results and the applicability of this research to the real world serve as the foci of this chapter. Notably, these two problems are deeply related, and this connection will be made clear as they are addressed in the coming sections.

4.1 Real-world Implementation

It's important to introduce the constraints of working with real-world robots as compared to our comfortable simulation environments. Challenges like periodic position calibration, consistent print head levelling, power delivery, etc. are monumental in their own right. Even so, these solutions are only related to this research to the extent that they are interested in maximizing the viability of the technology and deserve publications of their own (some of which currently have been published).

The constraints we are interested in for the purposes of this research are related to the software behaviors of real-world 3D printing robots. The standard way to issue commands to extrusion-based 3D printers is with a simple programming language (or "numerical control language") called G-code. G-code was briefly introduced in section 29 as a parallel to the basic movement command language developed in this research. It is this parallel that provides a direct mapping between the proprietary command language and usable G-code that can be interpreted by the microprocessor of a 3D printer.

Recalling from chapter 3, each chunk is responsible for storing the commands related to the printing of that chunk, and all other extraneous commands are pre-programmable. However, in the real-world, robots will need to be given a single set of G-code commands and are expected to cooperate according to our wait-notify paradigm. This means that robots that were previously unaware of the concept of a "chunk", let alone the concept of cooperating with other robots as they print their chunks, need to have an entirely new system for managing this cooperation.

The fundamental information needed by real-world robots to perform cooperative 3D printing is as follows:

- The G-code files for each robot, one file per robot,
- The dependency network, encoded in its own file or formatted in a custom G-code command, and
- The chunk ownership of each robot, encoded in its own file or formatted in a custom G-code command.

Alongside this research, a user-friendly web interface has been developed to ease the process of issuing a print job to cooperative robots. Every step of the process is automatable, from chunking to issuing G-code to the robots. As part of this automation, the following process is used:

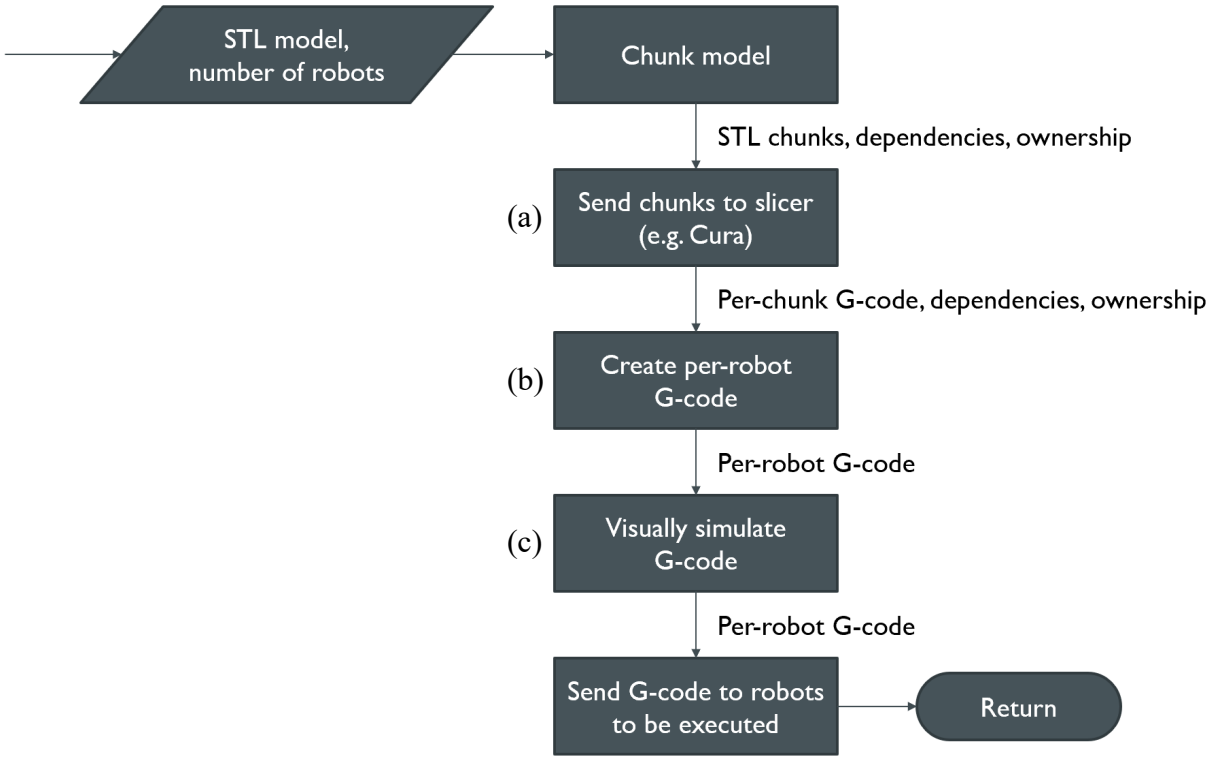

Figure 4-1. The website automation process. (a) replaces the proprietary, simplistic slicer developed for this research. (b) is a nontrivial step requiring further explanation. (c) refers to the cooperative 3D printing simulation processes from chapter 3.

Figure 4-1(a) is the first modification of the cooperative 3D printing process. As mentioned in chapter 2, a proprietary slicer was developed for the purposes of this research. The process was time-consuming and did not yield a particularly useful slicer in that it was incapable of handling most oddities present in 3D geometries (for example, holes in the middle of a vertical slice of a model). Cura [3] is a widely used, frequently updated slicer. It is extremely powerful with complex 3D geometries, accounting for hundreds of edge cases in models to make them printable. The chunker performs the chunking, then exports the chunked 3D pieces to STL files. These STL files are then passed through Cura's slicer individually to produce a set of G-code commands for each chunk.

Figure 4-1(b) is a new step that receives the per-chunk G-code, the chunk dependencies, and the robot-chunk ownership information with the intent to produce a single set of custom G-code for each robot. For each robot, this G-code contains all the commands for printing each of a robot's chunks, and also markers that signal the robot to either "notify" another robot that it has finished a chunk, or "wait" for a notification from another robot that a specific chunk has been finished.

The result of this G-code will have a format similar to Figure 4-2, with embedded wait and notify commands in the form of "#" comments.

```
G0 X12.0 Y0.2 Z0.0 F3200
...                          } Chunk 0 commands

# N_C 0
# W_C 2                      } Notify/wait commands

G0 X24.0 Y3.4 Z1.2 F3200
...                          } Chunk 1 commands
```

Figure 4-2. Example of a G-code file for a single robot. The "#" comments are custom commands that the robot uses to communicate with other robots about the completion of chunks. In this small example, Chunk 0 has no dependencies and Chunk 1 depends on Chunks 0 and 2.

In this example, the unique addition to this G-code is with the "#" comments. "`# N_C`" is a notify command. The numbers after the command are the chunk numbers that are completed. This notification is broadcast to every other robot printing the model. "`# W_C`" is a wait command. The numbers following the wait command are the chunks that need to be notified by other robots before continuing with the G-code. The chunks for which to wait will always be the subsequent chunk's dependencies minus any dependencies already taken care of by this robot.

Figure 4-1(c) refers to the existing simulation process, but rather than performing the chunking and command generation steps as shown in Figure 3-7, the process can begin at the "Generate per-chunk Frames" step. Cura has already generated the per-chunk commands, so those can be parsed into the internal G-code representation with which the existing software is already familiar. In order to extract the chunk dependency and ownership information, the `#N_C` and `#W_C` commands (from Figure 4-2) must be leveraged. The following flow chart demonstrates how to parse the per-robot G-code into an array of chunks for each, complete with dependencies.

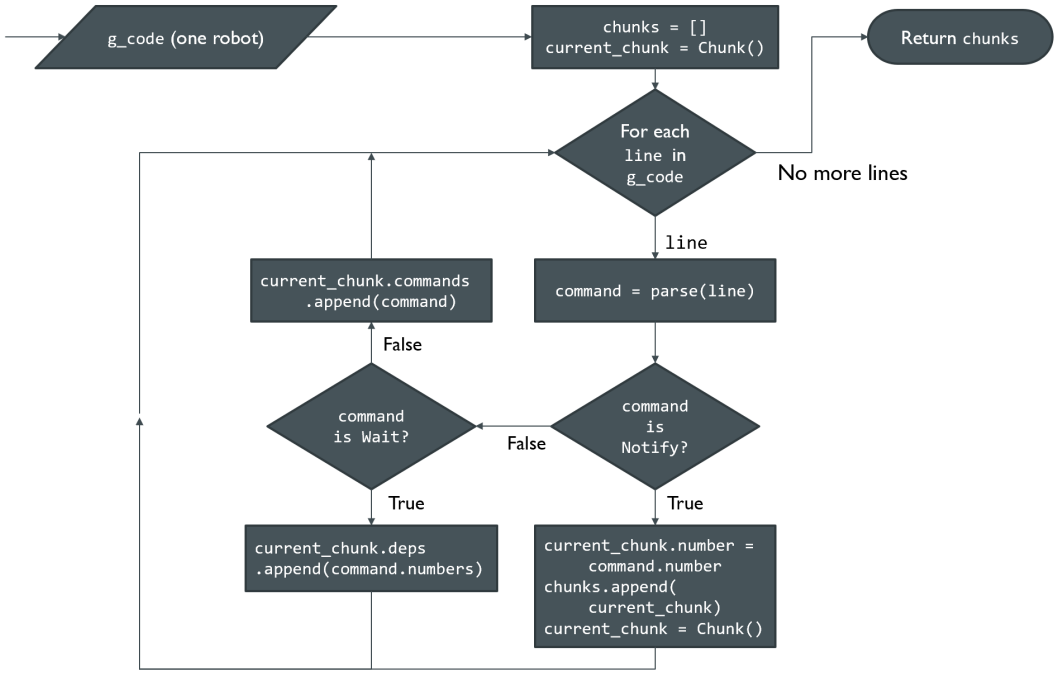

Figure 4-3. Algorithm for converting per-robot G-code to Chunk objects for use in the Simulator.

The process diagrammed in Figure 4-3 can be executed for every robot to convert its raw G-code commands into the data models used by the Simulator. This process ensures that the

simulation and the real-world robots are using the same information, which is necessary to properly validate the cooperative 3D printing algorithms.

Once these commands have been parsed, they can then be simulated. To this end, a frontend tool has been developed, as shown in Figure 4-4, to make the cooperative 3D printing experience more manageable. The underlying infrastructure makes use of the software from this thesis to carry out the conversion of a 3D geometry to G-code files for multiple robots. The frontend also has the capability to communicate with any connected AMBOTS so that they can carry out the per-robot G-code commands.

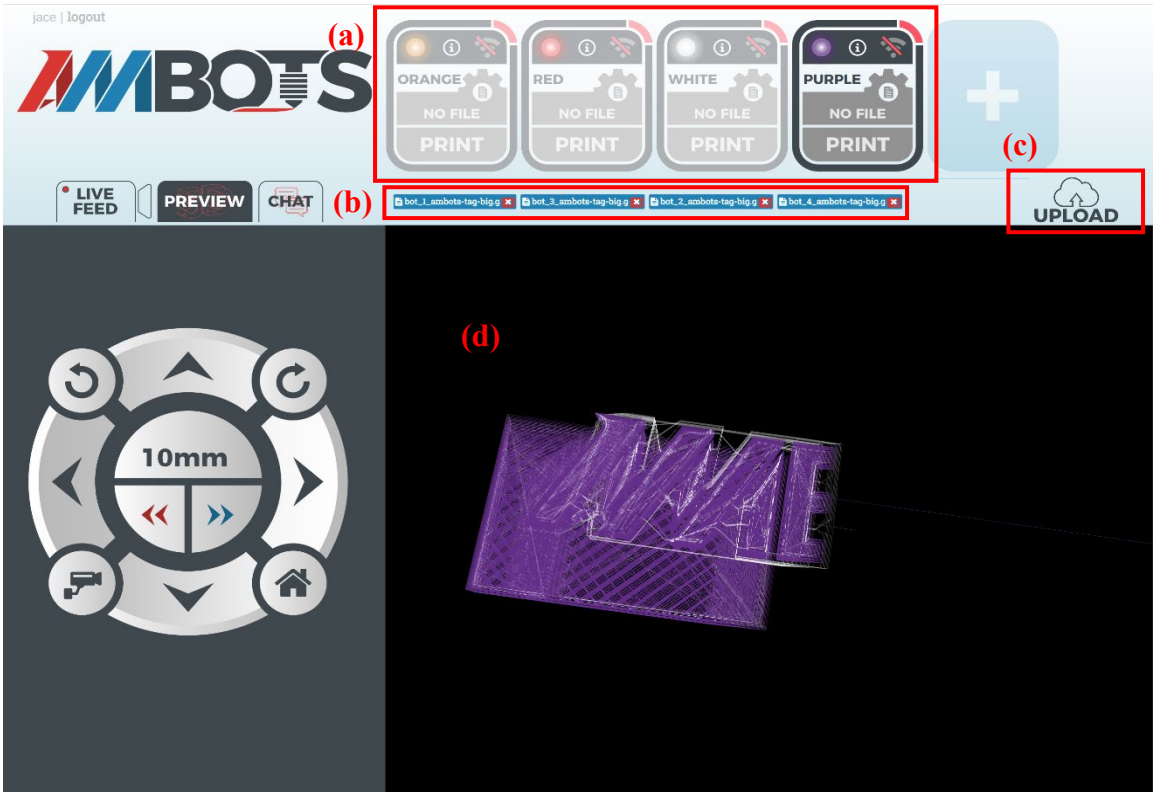

Figure 4-4. Screenshot of a prototype web frontend for cooperative 3D print management. (a) Robot setup. Four robots are initialized. (b) The resulting per-robot G-code files. (c) Upload STL file. STL models are automatically chunked, sliced, and simulated with the number of robots in (a). (d) A basic G-code visualizer (not a simulator).

4.2 Portable Simulation

The next problem this software needs to address is the need for the simulation visualization to exist outside of our proprietary software, or, at the least, make the format generic enough that a visualizer is easy to build around the simulation format.

A popular choice for open file formats (that is, formats with open-source descriptions without intent to obscure the format from users) is JavaScript Object Notation (JSON) [19]. JSON is a text format that is easy for machines and humans alike to parse and generate. Its format is based upon paradigms in object-oriented programming. Another benefit of this choice is that JSON parsing libraries exist for nearly every major programming language, including Python, Java, JavaScript, C++, and many more.

In Figure 3-9, the only object needed to visualize a simulation is the Simulation object (and SimulationFrame, which shows up in the members list of Simulation). Because this information is both sufficient and all of it is necessary, it make sense to simply serialize the Simulation object in JSON format. Looking at the format description in Figure 4-5, that's essentially how the format was derived, with a couple of minor modifications.

```
{
    "init": {    // Contains all initialization data (in this case,
                 // just robot positions)
        "robots": [    // An array of all the robots in the scene
            {    // Each robot has a number, rotation, and position
                "n": 0,              // number
                "r": 0.0,            // rotation (radians)
                "v": [x1, y1, z1]    // position
            },
            {
                "n": 1,
                "r": 3.14159,
                "v": [x2, y2, z2]
            }
        ]
    },
    "frames": [    // An array of all SimulationFrames
        {  // frame 0
            "robots": [       // An array of new states per-robot
                ...    // Each robot format matches that of the
                       // "init"
            ],
            "material": [    // An array of all placed material in
                             // this frame
                [   // Placed material is represented by a sequence
                    // of coordinates.
                    [x3, y3, z3],
                    [x4, y4, z4]
                ],
                [
                    [x5, y5, z5],
                    [x6, y6, z6],
                    [x7, y7, z7],
                    ...
                ]
            ]
        },

        {  // frame 1
            ...
        },
        ...
    ]
}
```

Figure 4-5. Portable simulation file format. Comments are meant as a description to aid with understanding the format in this paper. They should not be included in any legitimate portable simulation JSON file.

This format includes all of the information held in the Simulation object, as mentioned, but is not necessarily a direct serialization of a Simulation object. For example, the variables representing a machine are “n”, “r”, and “v”. These are shortened variable names for the purpose of saving space when storing simulation files. On the topic of space-saving, it is always ideal to save simulation files without any whitespace (which is still valid JSON). The white space can frequently consume as much as 75% of the simulation file size, especially when the white space is comprised of space characters as opposed to tab characters.

As a proof-of-concept for the utility of the simulation file, a standalone Javascript-based visualizer was developed as part of this research to demonstrate the simplicity of using the simulation file. Creating the software took 6 hours and 500 lines of JavaScript code to have a fully functional, interactive, frame-by-frame visualizer of a simulation file exported directly from the scaled simulation process. A few screenshots of the simulator are shown in Figure 4-6.

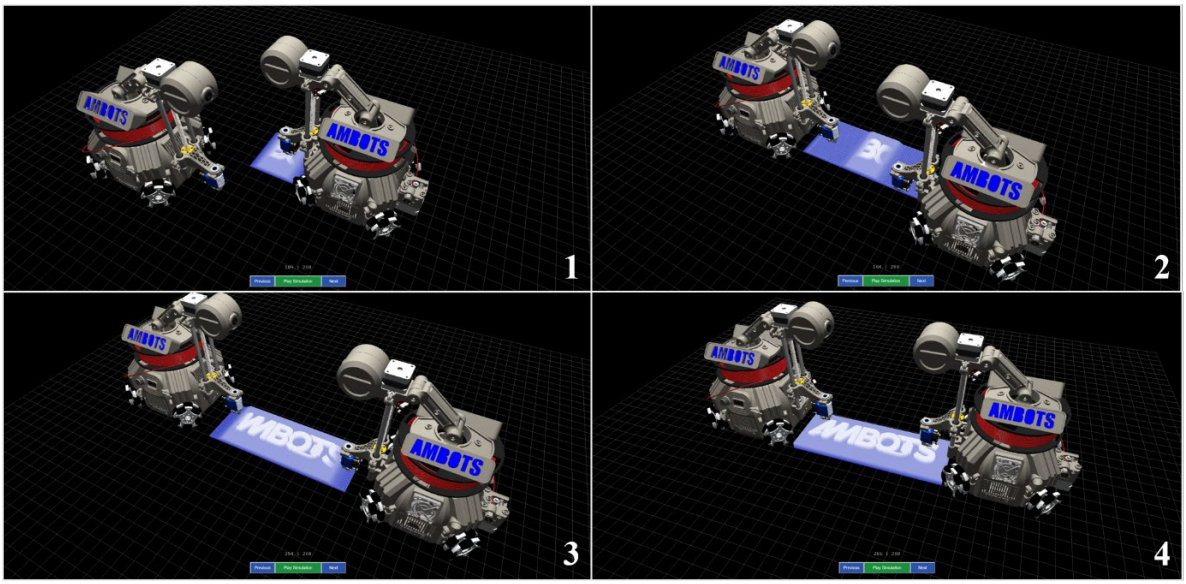

Figure 4-6. Screenshots of proof-of-concept visualization software. All robot movements and materials depicted are derived purely from the raw data encoded in a portable simulation JSON file. This simulation only used two robots and had a total of 289 frames. The simulated print time is roughly 2.7 hours. The whole simulation is viewable within ten seconds.

This simulator was written using a general-purpose, open-source 3D graphics library called Three.JS [20]. The robot models are predefined 3D models that can be modified at the whim of the developer. That is to say, the robot models are not related to the simulation JSON file. The material placed in the scene is all derived from the simulation JSON file, but stylistic choices like the coloring of the material are also at the developer's discretion.

Again, the purpose of this proof-of-concept software was to demonstrate the ease of development and flexibility of design afforded to a developer when the encoded simulation file is human-readable and highly accessible. The visualization software handles the job simply and quickly, with little development time and few hiccups. This bolsters our confidence that future visualizers will be similarly easy to develop.

5 Conclusions

This project aimed to lay the foundation for scalable Cooperative 3D printing, a new research direction that hasn't been addressed before in the literature. Cooperative 3D printing aims to solve the scalability issue with existing 3D printing technology. This thesis has presented, in detail, a feasible process for managing N 3D printing robots operating in parallel on a single print job, taking into account the geometric constraints, the communication requirements between robots, and the necessary pre-processing needed to properly subdivide a model for chunk-based printing.

This thesis also presents validation for a scaled printing strategy with a simulator (ignoring hardware anomalies that could occur, which are not within the scope of this thesis). Not only were validation considerations made, but also those for the utility of the simulation algorithm; simulations generated by the algorithms in this thesis are highly portable using the open, readable JSON format for use in any environment, be it visual or computational.

Beyond the simulation, the methods for producing executable G-code have also been developed alongside this work, solidifying the software suite as a fully-featured Cooperative 3D Slicer, comparable to that of popular user-interface slicing softwares.

It is our hope that this technology goes on to revolutionize the way manufacturing processes are structured, focusing on developing automated, independent robots that can, under a reliable mechanism, produce complete, finished products with minimal human post-processing. In addition to printing robots, we envision this technology supporting a wide variety of manufacturing robots, including pick-and-place robots, circuit-printing robots, and more.

5.1 Future Work

To further develop this work and enable real-world 3D printing, real-world testing is necessary. The logical continuation of this work is to take the G-code intended for real-world printers and test the printing results of many models, varying in geometric complexity. The results of such testing could yield a few important organizational problems. For example, the chunker used in this thesis subdivides models without consideration to the possibility for chunking at very thin parts of a model, which could make for a weak chunk boundary. If this comes to be an issue, it will be necessary to modify the chunking algorithm to maximize the surface area of chunk boundaries. This is just one example of a potential problem that can only arise from real-world testing.

This thesis's research can also be extended with the development of more scaling strategies, i.e. strategies besides SPAR3. As part of this research, many alternative strategies were considered, for example, strategies involving robots moving circularly around the centerpoint of a model, working their way outwards as more material gets placed. There is potential for many more methods of printing, but they were not explored in this research and will make great additions to this foundational piece.

6 Appendix

Appendix A – UML Diagram of the Chunker, Slicer, and the Simulator

7 Glossary

This section is used to document abbreviations used in flowcharts and code examples throughout this thesis.

`chunk_idx`	"Chunk index". An integer value, representing the index of a Chunk in an array.
`current_c`	"Current chunks". An array with one value for each robot. A value at position i in the array specifies the Chunk number that robot i is currently printing.
`current_f`	"Current frames". An array with one value for each robot. A value at position i in the array specifies the current index in a Chunk's "`frames`" property that robot i is currently simulating.
`DDT`	"Directed Dependency Tree". This name is used to represent any graph-like object whose purpose is to encode a DDT for chunk dependencies. Chunk objects themselves are graph-like, in that the "`dependencies`" property of Chunks points to the chunk numbers upon which a given Chunk is dependent.
`deps`	"Dependencies". Used in flowcharts to save space. This property corresponds to the "`dependencies`" property of Chunks.
`EET`	"Estimated Execution Time". Defined as the length of time needed to print a given Chunk or Cooperative 3D print.
`exec_time`	"Execution Time". This nomenclature is used only as a property of Chunk objects to store the length of time taken to print that Chunk.
`finished_c`	"Finished Chunks". A set of all the chunk numbers for chunks that have been printed at a given point in a simulation.
`finished_r`	"Finished Robots". An array of boolean values with one value for each robot. A value at position i indicates whether or not robot i has finished printing. When every value in the array is "True", the simulation has finished.
`frame_idx`	"Frame Index". An integer value, representing the index of a ChunkFrame in a Chunk's "`frames`" property.

N_C "Notify Command". The token "N_C" is used in G-code files as a custom command. It is always followed by a single integer: the chunk number which the robot has just completed.

pieces Synonym for "smaller chunks", relative to some larger chunk. Used in the SPAR3 definition to refer to chunks that are produced from row-wise chunks, which are themselves produced by two-robot chunking.

prevRow "Previous Row". An array of all the chunk pieces from the row of chunks directly preceding a given "current row". This is used so that the "current row" of chunks being produced can refer to the previous row's chunks for the purposes of creating dependencies in the SPAR3 strategy.

remChunk "Remaining Chunk". Used to refer to one of the two chunks produced by a single bisection around a chunking plane. The "remaining chunk" is the chunk that must be further subdivided, while the other chunk produced by a bisection is considered a "finalized chunk".

W_C "Wait Command". The token "W_C" is used in G-code files as a custom command. It is always followed by either a single integer or multiple comma-separated integers: the chunk numbers that musts be completed before the robot is allowed to continue executing commands.

8 References

[1] ASTM-F42-Committee, "Standard Terminology for Additive Manufacturing Technologies," in *ASTM International*, West Conshohocken, PA, 2012.

[2] A. Renellucci, "Slic3r: G-code generator for 3D printers.," 2015. [Online]. Available: http://slic3r.org/about. [Accessed: 16-Apr-2018].

[3] D. Braam, "Cura (software)," *Wikipedia*, 2016. [Online]. Available: https://en.wikipedia.org/wiki/Cura_(software).

[4] "KISSlicer," 2016. [Online]. Available: http://www.kisslicer.com/.

[5] "Skeinforge," *RepRap Wiki*, 2015. [Online]. Available: http://fabmetheus.crsndoo.com/overview.php.

[6] C. Kirschman and C. Jara-Almonte, "A parallel slicing algorithm for solid freeform fabrication processes," *Solid Free. Fabr. Proceedings, Austin, TX*, pp. 26–33, 1992.

[7] E. Sabourin, S. A. Houser, and J. Helge Bøhn, "Adaptive slicing using stepwise uniform refinment," *Rapid Prototyp. J.*, vol. 2, no. 4, pp. 20–26, 1996.

[8] S. Lefebvre and L. I. N. Grand-Est, "A GPU accelerated CSG modeler and slicer," in *18th European Forum on Additive Manufacturing (AEFA'13)*, 2013.

[9] Repetier, "Repetier-Host," *RepRap Wiki*, 2017. [Online]. Available: https://reprap.org/wiki/RepRap.

[10] N. Koenig and A. Howard, "Design and use paradigms for gazebo, an open-source multi-robot simulator," in *Intelligent Robots and Systems, 2004.(IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on*, 2004, pp. 2149–2154.

[11] T. Bräunl, "The EyeSim Mobile Robot Simulator," *Electron. Eng.*, 2000.

[12] B. Browning and E. Tryzelaar, "Ubersim: A realistic simulation engine for robot soccer," in *Proceedings of Autonomous Agents and Multi-Agent Systems, AAMAS'03*, 2003.

[13] L. Hugues and N. Bredeche, "Simbad: an autonomous robot simulation package for education and research," in *Proceedings of The Ninth International Conference on the Simulation of Adaptive Behavior (SAB'06)*, 2006, pp. 831–842.

[14] L. Luo, I. Baran, S. Rusinkiewicz, and W. Matusik, "Chopper: partitioning models into 3D-printable parts," *ACM Trans. Graph.*, vol. 31, no. 6, p. Article 129, 2012.

[15] J. Hao, L. Fang, and R. E. Williams, "An efficient curvature-based partitioning of large-scale STL models," *Rapid Prototyp. J.*, vol. 17, no. 2, pp. 116–127, 2011.

[16] W.-P. Xu, W. Li, and L.-G. Liu, "Skeleton-sectional structural analysis for 3D printing," *J. Comput. Sci. Technol.*, vol. 31, no. 3, pp. 439–449, 2016.

[17] Blender Online Community, "Blender - a 3D modelling and rendering package. URL: http://www.blender.org/," *Blender Foundation*, 2013. [Online]. Available: http://www.blender.org/.

[18] A. V. Aho, M. R. Garey, and J. D. Ullman, "The Transitive Reduction of a Directed Graph," *SIAM J. Comput.*, vol. 1, no. 2, pp. 131–137, 1972.

[19] JSON.org, "Introducing JSON," *Json.Org*, 2014. [Online]. Available: http://www.json.org/.

[20] Three.JS Authors, "Three.JS," 2010. [Online]. Available: https://threejs.org/.