A Distributed 3D Print Driver

Master-Thesis von Suman Bidarahalli aus India
Tag der Einreichung:

1. Gutachten: Prof. Dr. Arjan Kuijper
2. Gutachten: Alan Brunton, PhD.
A Distributed 3D Print Driver

Vorgelegte Master-Thesis von Suman Bidarahalli aus India

1. Gutachten: Prof. Dr. Arjan Kuijper
2. Gutachten: Alan Brunton, PhD.

Tag der Einreichung:
Abstract

Determining material arrangements to control high-resolution multi-material 3D printers for reproducing shape and visual attributes of a 3D model (e.g. spatially-varying color, translucency and gloss) requires large computational effort. Today’s resolution and print tray sizes allow prints with more than $10^{12}$ voxels each filled with one of the available printing materials (today up to 7 materials can be combined in a single print). Cuttlefish, a 3D printing pipeline, processes the input in a serial fashion leading to increased computation time for higher number of models. Distributed computing is one way of achieving better performance for large computations. Through this master thesis, we have developed a distributed version of the cuttlefish printer driver in which the computational task is distributed amongst multiple nodes in the cluster and the resulting partial output is merged to generate the full slices. The architecture supports streaming, which is required to rapidly start the print before the full computation is finished, as cuttlefish processes the input in small parts and generates chunk-wise output. Finally, the comparison of the performance achieved by the distributed vs the non-distributed cuttlefish version is established to get a better understanding of the advantages and the challenges of distributed computing.
Ehrenwörtliche Erklärung


Darmstadt, den April 18, 2017

Suman Bidarahalli
Acknowledgment

I would first like to thank my thesis supervisors Alan Brunton and Marco Dennstädt for guiding me throughout the thesis. They consistently allowed this thesis to be my own work, but steered me in the right direction whenever they thought I needed it. The door to Prof. Dr. Arjan Kuijper's office was always open whenever I ran into a trouble spot or had a question about my research or writing. I would also like to thank Philipp Urban for his continued support and encouragement. I offer my sincere appreciation for the learning opportunities bestowed on me by them which helped me to mould myself to go beyond my limits. It would not have been possible for me to accomplish my thesis without the help and cooperation of my colleagues (Can, Tejas, Andrea, Chaitra, Rutwik) who not only provided me with their workstations for running the test-cases but also motivated me during my thesis. My heartiest thanks to my family and Daniel for their unfailing love and support during my thesis.
# Contents

1 Introduction ................................................................. 9
   1.1 Problem Statement .................................................. 10
   1.2 Thesis Organization ................................................ 12
   1.3 Summary .............................................................. 12

2 Related Work ............................................................. 13
   2.1 Introduction to distributed systems and its types ................. 13
      2.1.1 Use of Distributed Computing in fields involving large computations .............................................. 15
   2.2 Existing solutions for 3D Printing ................................ 15
   2.3 Summary .............................................................. 16

3 System Model ............................................................. 17
   3.1 Non-distributed Cuttlefish Pipeline Components .................. 17
   3.2 System Model for Distributed Cuttlefish Pipeline ................. 18
      3.2.1 Distributed system for cuttlefish printer driver ........ 18
      3.2.2 Components of distributed cuttlefish printer driver .. 19
   3.3 Summary .............................................................. 22

4 Design Analysis .......................................................... 23
   4.1 Prototype I- Input/Output using Network File System ............ 23
      4.1.1 Master sub-task creation and distribution ............... 23
      4.1.2 Slave Partial Output ........................................ 29
      4.1.3 Partial Output Merging at Master ......................... 31
   4.2 Prototype II- Input/Output independent of Network File System . 33
      4.2.1 Selection of the prototype .................................. 35
      4.2.2 Printing Software Components and Printer Specification ......... 35
      4.2.3 Distribution of the work load to the slave nodes ....... 35
      4.2.4 Work Load Collection and Computation .................... 36
      4.2.5 Creation of the full slices ................................ 37
      4.2.6 Analysis of Prototype II design ......................... 37
   4.3 Serialization and Deserialization of Data ......................... 38
   4.4 Communication medium ............................................ 39
   4.5 Summary .............................................................. 39

5 Implementation .......................................................... 40
   5.1 Native Cuttlefish Application ..................................... 40
   5.2 Prerequisites for Distributed Cuttlefish Application Development ......................................................... 40
   5.3 Installation and Cluster Set-up ................................... 41
   5.4 Introduction to New Components .................................. 42
      5.4.1 Introduction to the New Classes .......................... 42
      5.4.2 Code Organization ............................................ 43
   5.5 Communication Specifics ........................................... 43
      5.5.1 MPI APIs used in the Implementation ...................... 44
      5.5.2 Communication Abstraction Layer ......................... 47
      5.5.3 How does MS-MPI work internally? ....................... 48
   5.6 Comparison of MPI to traditional socket programming .......... 49
   5.7 External Libraries ................................................. 50
   5.8 Debugging ......................................................... 52
   5.9 Logging ........................................................... 52
   5.10 Project Management .............................................. 52
   5.11 Summary .......................................................... 54
6 Results and Analysis

6.1 Overview of SDLC Testing Phase ................................................................. 55
6.2 Implementation Assessment ................................................................. 56
  6.2.1 Manual Tests ................................................................................. 56
6.3 Results ................................................................................................. 59
  6.3.1 Comparison of distributed vs non-distributed cuttlefish performance ........ 59
  6.3.2 Comparison of Distributed Cuttlefish Prototypes ................................ 65
  6.3.3 Distributed Cuttlefish Prototype II Performance Evaluation .................. 69
6.4 Summary ............................................................................................. 74

7 Conclusion .............................................................................................. 75

8 Future Work ............................................................................................. 77
## Listings

3.1 *BitmapProducerObjet* Component Configuration ........................................... 18

4.1 Configuration file submitted by the user ...................................................... 25
4.2 Slave Configuration file Node 2 ......................................................................... 25
4.3 Slave Configuration file Node 1 ......................................................................... 26
4.4 Distribute Tasks- Greedy Approach ................................................................... 28
4.5 Calculate full slice height .................................................................................. 32
4.6 Calculate full slice width .................................................................................... 32

5.1 CUDA free configuration ...................................................................................... 40
5.2 mpiexec syntax .................................................................................................... 42
5.3 Flag to enable distributed computing solution ..................................................... 43
5.4 find_package use for MPI libraries ...................................................................... 43
5.5 Initialization of Execution Environment ................................................................ 44
5.6 MPI Query Functions ......................................................................................... 45
5.7 Blocking MPI Send ............................................................................................... 45
5.8 Blocking MPI Receive ......................................................................................... 47
5.9 Explicit Synchronization via Barrier .................................................................... 47
5.10 Shutting down the MPI environment .................................................................... 47
5.11 Open profile from File ....................................................................................... 51
5.12 Save profile to Memory ...................................................................................... 51
5.13 Open profile from Memory ................................................................................ 51
5.14 API to compress the data ................................................................................... 51
5.15 API to uncompress the data ................................................................................ 51
List of Tables

5.1 MS-MPI Environment Variable ................................................................. 41
5.2 Class specific for Prototype I ................................................................. 43
5.3 Classes specific for Prototype II .............................................................. 43
5.4 Common Classes For Prototype I and II ...................................................... 44
5.5 Projects for Distributed Computing .......................................................... 45
5.6 Cmake Directives ....................................................................................... 46
5.7 Component-wise Information Logged ......................................................... 53

6.1 Speed-Up and Efficiency: Distributed Cuttlefish Prototype I vs Non-Distributed Cuttlefish For Same Models 61
6.2 Speed-Up and Efficiency: Distributed Cuttlefish Prototype I vs Non-Distributed Cuttlefish For Different Models ................................................................. 61
6.3 Speed-Up and Efficiency: Distributed Cuttlefish Prototype II vs Non-Distributed Cuttlefish for Same Models 62
6.4 Speed-Up and Efficiency: Distributed Cuttlefish Prototype II vs Non-Distributed Cuttlefish for Different Models ................................................................. 65
6.5 Prototype I vs Prototype II Measurements for 10 Head Models ..................... 69
6.6 Prototype I vs Prototype II Measurements for Different Models Input .......... 70
6.7 Prototype I Performance For Varying Model Size ....................................... 70
6.8 Prototype II Performance For Varying Model Size ..................................... 70
6.9 Prototype II: Design I vs Design II Different Models Input Set .................... 74
6.10 Prototype II: Design I vs Design II Same Models Input Set ....................... 74
1 Introduction

Additive manufacturing (AM) is a generic term used to describe a set of technologies that fabricate 3D Objects directly from 3D model data [1]. American Society for Testing and Materials (ASTM) International defines AM as the process of joining materials to make objects from 3D model data, usually layer upon layer, as opposed to subtractive manufacturing methodologies [2]. 3D Printing is one of the technologies encompassed by AM [3]. 3D Printing is a manufacturing method wherein materials like plastic, polymer or metal, are deposited on one another in layers to produce a three dimensional object [4]. The prices of the materials used and the cost of the machinery involved in AM is dropping rapidly as well as the better engineered materials allow for high quality builds leading to the application of this technology in various fields.

Working of 3D Printing can be defined roughly as a 3-step process, depicted in Figure 1.1, which includes creating a virtual design of the object (either by using designing software or through 3d scanners), slicing the 3d model before it can be printed followed by printing the object, and final step is the finishing process in which the printed object is cleaned up, polished or sanded, painted to complete it as intended.

![Figure 1.1: 3-Step 3D Printing Process](image)

The material chosen to print the object determines the underlying technology for the actual printing process, for example, if the material chosen is plastic then Fused Deposition Modeling (FDM) technology is used, for photo sensitive resin material the technology named Photo-polymerization is used, whereas SLS (Selective Laser Sintering) is the technology used if print material is powder (Alumide). Photo-polymerization is a technique that involves the use of UV light to solidify the photosensitive material. This technique is used by different 3d printing processes like Stereo-lithography (SLA), Digital Light Processing (DLP) and MultiJet printers. MultiJet printers begin by spraying the tiny droplets of photopolymer in the shape of the first layer followed by cross-linking the polymer by using the UV light from the lamp attached to the print head thus locking the shape of the layer in place [5]. Multi-material printers enable fabrication of 3D prints which are composed of diverse materials i.e. materials which are fused to form the object may differ in their chemical properties, and/or physical properties [6].

The step 2 of the 3d Printing process involves the use of a software which enables conversion of the input (i.e. virtual design) to a form which is understood by the print machine. The software fundamentally performs couple of actions enlisted below before the data is sent to the printer:

- Parses the input and converts it to an internal representation
- Performs slicing using the internal representation - Slicing is the procedure of dividing the 3D model into 2D layers where in each layer is a slice- a 2D representation of printer parameters for varying (x,y) coordinates and a constant z coordinate. Slicing can be thought of as similar to integrals i.e the 3D object can be approximated by it’s derivatives i.e. 2D slices, Figure 1.2 [7].
- Assign materials to the 2D-Slices, if the printer supports multi-material prints.
- Convert the 2D slices into machine specific code or representation.

The last step of 3D Printing is finishing up the printed object. Although optional, the importance of this step should not be underestimated as it is worth the enormous effort to make the prints/prototypes look like objects made from real world materials (figure1.3). The finishing techniques for the 3D printed objects can be referred as the craftsman’s workshop, where patience, skills and experience can help transform the raw products of the printers into fully realized models (figure1.4). Majorly, finishing the prints involves sealing, polishing and painting which helps to bring out the smooth surface and fine feature details. For example, production grade plastic is used by FDM technology to print models which can be sanded, drilled or glued just like any another plastic part.
1.1 Problem Statement

Cuttlefish [8] is a powerful print driver, which performs the transformation of the digital geometric representation of the design into machine-specific code or representation that eventually drives the 3D printer. Cuttlefish supports high-resolution multi-material printers and allows printing large sized multiple objects. High-resolution multi-material prints consist of huge amount of voxels (easily up to $10^{12}$ as today's printers allow to combine 7 materials in a single print), a voxel is 3D equivalent of 2D pixel [9]. To reproduce the shape and attributes of the 3D model with high precision, Cuttlefish performs material assignment at the voxel level. To do so large amount of computational effort is needed which cannot be achieved efficiently by a single processing entity for higher number of objects. Moreover, cuttlefish processes the input in serial fashion leading to increased amount of computational time for multiple objects.

To improve the performance for large computations, distributed computing can be used [10] [11]. Distributed computing is concept where in multiple machines of a distributed system work together on a single problem domain [12]. A distributed system can be defined as the group or cluster of autonomous computers which are connected via network and communicate primarily through message passing [13]. Figure 1.5 depicts the components of the distributed system. The ultimate goal of distributed computing is to maximize performance by enhancing resource utilization in a cost effective, transparent and reliable manner.

The state of the art multi-jet 3D printers allow high resolution printing of multiple print objects occupying large volume at the same time. To efficiently exploit the offered functionality of these printers, appropriate digital fabrication software
needs to be developed. Through this master thesis, we have implemented a distributed version of Cuttlefish 3D Printing pipeline by applying the various concepts of distributed computing. Pre-processing of the large sized multiple objects can be distributed among the nodes of the distributed system so as to utilize the processing power of each node in order to increase the efficiency by limiting the computational effort and time for each node. Distribution of the tasks among the
nodes of the distributed system parallelizes the pre-processing of the input thereby reducing the waiting time for each object to be processed.

1.2 Thesis Organization

This document consists of seven chapters, starting with the introduction which provides the overview of the domain and problem statement addressed in the thesis, followed by the related work which summarizes the various fields where distributed computing has been used and its application in 3D Printing. Chapter 3 and 4 provide a detailed description of the proposed solution, the system model and various design decisions made during the thesis. Implementation of the proposed solution is described in Chapter 5 and comparison of the results is discussed in the Chapter 6 followed by the conclusion.

1.3 Summary

The current Chapter provides the fundamental background of the 3D Printing Technology, the problem statement addressed through the thesis and the solution chosen to solve the problem. The next Chapter provides information regarding the existing solutions in the 3D Printing technology as well the various distributed systems in place.
2 Related Work

With science and technology progressing rapidly, there is a good chance of availability of already implemented solution for a given problem identified in a target system. This chapter highlights the existing systems in place which provide solutions designed to achieve similar goals to the thesis under discussion and comparison to how the solution presented in the current document differs from them. This chapter is divided into two sections with the first section briefly describes the existing distributed systems which can be used for cluster computing followed by the second section focusing the use of distributed systems in the field of 3D printing.

2.1 Introduction to distributed systems and its types

A distributed system consists of autonomous systems (AS) connected through network which interact (if necessary) through message passing and most importantly give the user an illusion of being single large system [14]. The main goal of distributed system is to connect the users and the available infrastructure i.e. IT resources, in a transparent, open, cost-effective, reliable, scalable way. Transparency in distributed computing environment (DCE) means that the user is unaware of the underlying system complexities in terms of location of the AS, heterogeneity of the system hardware, concurrency and replication of the AS mainly to provide reliability and fault tolerance. Transparency property of the distributed systems makes it appear as a single system dedicated to perform some tasks to the user [15]. Scalability refers to the property by which the number of AS in the DCE can be increased (scaled in) or decreased (scaled out) so as to provide best suited infrastructure to handle the fluctuating load (i.e. the amount of work to be done). The shared resources of the distributed system can be broadly classified as physical resources and virtual resources. The Figure2.1 details the resources included in the two categorizes.

Distributed Computing is a vast domain which consists of various different types of computing environments like utility computing (grid computing and cloud computing), cluster computing, and peer-to-peer computing (Figure 2.2) which are described briefly as follows.

- **Peer-to-Peer computing**: Each node in the DCE acts as both client and server, providing as well as consuming part of the system resources. Peers are machines simply connected to the internet where in each peer can freely join and leave the DCE without affecting the whole system making the system self-organizing and also implying no particular role (either master or slave) for a peer (Figure 2.3).

- **Utility computing**: Utility computing is based on a model for service provisioning, which allows the users (consumers) to pay the providers for using the service only when they need to. It emphasizes on a business model, by
which service providers make available the requested/paid resources to the customers. All grid/cloud platforms are regarded as utility service providers. Grid computing enables coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organizations where as cloud computing is a computing paradigm that involves outsourcing of computing resources with the capabilities of replaceable resources and scalability i.e on-demand provisioning with little or no up-front IT infrastructure investment costs. Cloud computing offers its benefits through three types of delivery models namely- infrastructure-as-a-service (IaaS), platform-as-a-service(PaaS) and software-as-a-Service (SaaS)

- **Cluster Computing**: A cluster is a set of stand-alone computers and a network interconnecting them. The computers of the cluster work cooperatively together as a single integrated computing resource. A cluster is local if all of its component subsystems are supervised within a single administrative domain, usually residing in a single room and managed as a single computer system. The components of a cluster are connected to each other through fast local area networks.

As mentioned in the brief description of different distributed systems, each node is similar to the other in a peer-to-peer system and it can join/leave the distributed system without affecting the user. As the requirement of the thesis design needed designated roles for the nodes, a peer to peer system does not fulfill the requirement of specific node roles in distributed system. A Peer-to-Peer system mostly uses internet to maintain connectivity amongst the peers. The distributed system to be used in the thesis has nodes which were connected through the enterprise network and it was not possible for a node outside the institution to join the system, which makes peer to peer system not an ideal choice. Grid computing essentially has systems which are offered by multiple institutions as resources, which dedicated to perform part of computation required for solving huge problems. The institutions which offer their systems form a virtual organization and only these institutions have shared access rights on the resources [15]. The distributed system in demand for the thesis is dedicated to be used for a specific enterprise and other institutions not part of the same network cannot contribute computes nodes. This makes grid computing not suitable as a distributed system for the thesis under discussion.
2.1.1 Use of Distributed Computing in fields involving large computations

Distributed systems offer various advantages like computational speed-up, robustness through duplication of the resources, cheaper and scalable infrastructure etc. Using distributed systems allows to overcome the limitation of resources for solving problems with large computation, making it possible to get correct results within in a shorter span of time. The various domains where distributed computing has been used involves astronomy: for example Cosmology@Home is a project started to design models which describe the universe [16], data analysis and machine learning: the DistributedDataMining project involved data analysis for stock market prediction and analysis of medical information [17], molecular biology: RNA world distributed system uses bio-informatics software to study the structure of RNA [18], climate study projects started in Oxford University use distributed system to study the climate changes which help to draw models for more accurate climate predictions [19], etc.

Most of the above mentioned projects run on BOINC software platform which facilitates volunteer computing or grid computing. It is a middleware system which allows public computers to contribute compute resources by running a client application and also provides mechanisms to limit the amount of resources to be used along with selection of the project to which the compute resources are dedicated [20]. Each such computer which is contributing some computational power, also uploads the output/results periodically to the server. It uses internet for communication with compute nodes i.e. sending the tasks to be performed and output collection. The projects which run on the BOINC platform have large computation requirement, storage requirement etc. but the main requirement from these applications to be suitable to run on the platform is that task which the applications perform be divisible into sub-tasks which can run independently. For the project to run on BOINC and use volunteer computing, it should have low data to compute ratio i.e. they should consume / produce less than a gigabyte of data per day of CPU time, this is due to the fact that input and output data is exchanged via commercial internet which might turn costly for large amount of data. Due to the restriction of low data/compute ratio, BOINC is not suitable for the approach presented in the thesis as the partial slices which are communicated can be large for large models which might exceed the data limit. As the compute nodes which are participating in volunteer computing are unknown and are not accountable, they could be unreliable and insecure nodes, which is not acceptable as the cuttlefish application can be used to create 3D prints of the confidential designs and sending the designs to unknown compute nodes would compromise the confidentiality.

2.2 Existing solutions for 3D Printing

As described in the Introduction, 3D Printing is one of the upcoming additive manufacturing process and a favorite research topic. There are various solutions available in the market for varying kinds of requirements in terms of material, appearances, cost etc. Out of the many interesting solutions available, there are three solutions which are most relevant with respect to the current thesis topic as they use distributed system as an integral part of their solution. ‘3dPrinterOS’ is a cloud-based management software for 3D printers which provides tools for making 3D printing user-friendly [21]. The user can connect the local 3D printer to the internet using the software provided by 3dPrinterOS thereby allowing the cloud-based software to drive the connected printer. It allows three input file formats (obj, amf and STL). The management software provides a tool to perform orientation of the print object on the print bed and to scale the size of the object as well as the slicer to perform slicing by allowing the user to set the slice thickness. The 3dPrinterOS software can be used only for the printers which are supported by it and does not act as a universal printer driver. The software allows to drive the connected printer but does not provide a way to generate output which can be then used to control the printer i.e. in our case bitmaps or STL output. During installation of the software to connect the local printer to the cloud, there are numerous device drivers which are installed on the client machine which might not always be ideal as there could be various trust issues associated with installation unknown device drivers from random sources.

‘3DHubs’ is an online services which allows the user to provide digital input of the print object, select the material to be used to print the input and last but not the least select the print lab (where the object is printed locally and shipped to the user) by comparing the prices for printing the object [22]. It is a commercial solution to enable 3D Printing as a service where in the user does not need to own a 3D printer. It provides easy access to different kinds of material as well as helps to avoid setting up the required infrastructure to make use of the 3D Printing technology. It is quite beneficial and cheaper for users who do not print quite often but might not be an ideal solution for frequently printing customers. To use the solution the 3d model is uploaded to the server, which might not be desirable for companies where the design needs to be highly-protected. Also, there are no additional tools to provide dimensions if the digital input needs to be scaled up or down.
The ‘Collaborative Cloud Printing Service’ is a cloud-based service for utilization of the 3D Printing resource developed by Institute of Computer-aided Product Development Systems at University of Stuttgart [23]. The solution provided is within the academic context and not a commercial solution. The focus of the project was to facilitate sharing of 3D Printing resources, provide platform for developing or extending and testing BPMS (Business Process Management Systems). The underlying distributed system used is the internet where as the approach in the current thesis is to use the enterprise network and computers within the network as distributed system which would be more desirable for companies who have privacy of their print models as the primary concern. The output for the cloud service under discussion is a printed object where as for the distributed cuttlefish the output is not necessarily a 3D printed object.

2.3 Summary

The 2 Chapter provides a brief overview about the existing 3D Printing solutions and other domains which use distributed computing to perform large computations. It also explains the different types of distributed systems and analysis of which type of distributed system is suitable for the proposed solution. The distributed system architecture designed to be used throughout the thesis is discussed in the Chapter 3.
3 System Model

Cuttlefish, as mentioned earlier, is a 3D printing pipeline, which processes one or more 3D digital models and outputs control code to drive a particular 3D Printer to realize the object. The digital model of the object to be printed is created either using a modeling software or acquired by 3D scanners. Cuttlefish allows numerous input file formats like STL, OBJ, PLY, WRL.

The whole pipeline has a streaming architecture, which means that the input can processed in smaller parts so as to generate the output sooner thus letting the printing process to be started as early as possible i.e. even before the whole object has been processed. The output is generated in parts called as chunks- each chunk is a collection of 2D layers, where each layer is referred to as a slice.

3.1 Non-distributed Cuttlefish Pipeline Components

The pipeline also has a component-based architecture. The composition of the pipeline varies as per the components assembled to form the pipeline. The components of the pipeline can be chosen on the basis of factors like the type of generated output needed to drive the particular printer, for example BitmapProducer component if the desired output is bitmaps, where as STLProducerObjet component is included when the desired output is an STL file per print material, i.e. a mesh representation, or if the output required is a GCode then the component used is GCodeProducer, e.g. for FDM printers. Another factor for deciding the components of the pipeline is what attributes of the object should be reproduced: shape, color, translucency, internal structure, etc.

The pipeline performs the following fundamental steps in a serial fashion irrespective of the components forming the pipeline 3.1. Let us now examine each of the stages outlined in Figure 3.1:

- **Input Parsing**: The input given to the pipeline, mostly mesh files and textures, is parsed and internal mesh representations are created for them. Internal mesh representations of multiple objects can be grouped into print jobs, together with each having it's own unique identifier and is referred to as a print object. At present FileParsePj is the component which performs this task.

- **Print object arrangement**: The print objects are arranged within the print volume to lower the material consumption and the print time. The arrangement of the print jobs is done by first optimizing the orientation of each print object to reduce print time and support requirements, followed by a greedy approach where in to arrange the print objects (approximated by their bounding boxes) compactly within the print volume. PrintJobOrganizer is currently handling this task.

- **Rasterization and Voxelization**: Rasterization is the process of converting vector graphics, e.g. polygons or curves, to raster images, i.e. pixels or dots for output to printer or display device. In our context, the input surface representation is converted to a voxel representation- a voxel is 3D equivalent of 2D pixel, and each voxel intersecting the surface is assigned attributes like color, occupancy etc. Currently the component doing this is PrintJobVoxelizer. The algorithm used for voxelization can be varied using the component configuration and further details are beyond the scope of the thesis.

![Cuttlefish pipeline component functionality](image-url)
• **Tonal Value Calculation:** This step involves calculation and assignment of tonal values from the optical features (RGB or BRDF data) of surface voxels. The tonal value computation is done with the help of a calculated ICC-profile belonging to the specific printer. The component doing this is referred to as *TonalValueCalculator*.

• **Surface Distance Evaluation and Tonal Analyzer** (optional): The component evaluates the distance of voxels to the surface and assigns values of a given attribute (of the surface voxel) to inner voxels within the specified maximum distance. The Tonal value analyzer component reports summary statistics, comparing the half-toned signal to the original tonal values in easy to interpret graphs.

• **Layer Extraction and Half-toning:** The component *LayerExtractor* extracts the layers below surface which are subsequently half-toned by the component *LayerHalftoner*. Half-toning is the process of creating an image comprised of discrete dots rather than continuous tones. When viewed from a distance, the dots blur together, creating the illusion of continuous tones. Details can be found in [24].

• **Output Generation:** The generated slices are then used to create the desired output, e.g., bitmaps or STL files. The output files are then used to drive the print machine.

The input given to the printer driver is a main configuration file in a JSON format - JSON (JavaScript Object Notation) is an open-standard format that uses human-readable text to transmit data objects consisting of attribute-value pairs, describing various input fields. The main configuration file lists the files describing the printer specification, component configuration, geometry, appearance, logger type and output folder path. The printer specification file consists of the details related to the resolution of the printer, print volume, material names and unique identifier for each material, ICC profile - it is a file which contains data that characterizes any color input/output devices. The component configuration enlists the components forming the pipeline and configuration parameters for each component. Listing 3.1 is an example of the *BitmapProducerObjet* component configuration description.

```
Listing 3.1: BitmapProducerObjet Component Configuration

1 {  
2   "type":"BitmapProducerObjet",  
3   "OutputPath": "Output/",  
4   "PrinterTextFileName": "printer_config.txt",  
5   "computeType":"multi-threaded",  
6   "pitchRequirement": "32",  
7   "unusedMaterial":"VeroClear"  
8 }
```

The `type` field denotes the name of the component and `computeType` sets the component to run in a single-threaded or multi-threaded fashion. `OutputPath` is used to set the path at which the component generates the bitmaps and `pitchRequirement` value is used to make the bitmap width a multiple of the specified number. The component which parses the input main configuration file creates a *PrintingSoftware* for the given component configuration.

### 3.2 System Model for Distributed Cuttlefish Pipeline

The non-distributed cuttlefish pipeline runs on a single machine and therefore there is no particular machine specific action needed to be performed depending on the role of the machine. For an application to run on a cluster of machines, sometimes it is necessary to perform additional steps which are specific to the role of the node in the cluster. Before understanding the architecture of the distributed cuttlefish printer driver it is necessary to understand the architecture of the cluster, referred to as a distributed system, on which the driver is designed to run.

#### 3.2.1 Distributed system for cuttlefish printer driver

The distributed cuttlefish printer driver is developed using a cluster of the computers(nodes) which are heterogeneous in their hardware as well as software configuration and are connected via a standard workplace network. The initial set up used during the development had three nodes out of which one node is the master node and remaining nodes are the slave nodes. The names master and slave depict a predetermined role the nodes have and type of work they do. The nodes also have access to a shared networked file system (figure 3.2).
As mentioned earlier, the role of the node determines the type of work they do. The master node is the computer in the cluster where the user submits his request, namely the main configuration file (figure 3.3).

The master node then performs some computation and distributes the sub-tasks to be performed by the slave nodes (figure 3.4).

The slave nodes perform the assigned tasks and report back to the master (figure 3.5). The cluster nodes follow a classic master-slave communication paradigm.

### 3.2.2 Components of distributed cuttlefish printer driver

The distributed cuttlefish pipeline components may differ depending upon the role of the node in the cluster. The components put together form a printing software in which the components run on the node in a loop, with each iteration of the loop processing chunk-wise data. To apply distributed computing as a solution for stated problem of large computation on big data set, there are three additional steps, as enlisted below, some of which need to be performed either on the master or the slave node:

1. Division of task into sub-tasks
2. Awaiting the sub-tasks and doing the needed work
3. Submitting back the partial output

Master Node Printing Software Components

As per the model discussed in the section 3.2.1, the master node performs the step 1 from the above steps. The division of the task among the slave nodes needs to be performed by the master efficiently so as to make sure that distributed load is balanced and none of the slaves are over-worked. To do so, the master needs to evaluate the input and perform some measurements so as to make a wise decision regarding the task split. Hence, the master printing software needs
to at least parse the input into an internal mesh representation i.e. print jobs and compute the necessary parameters on the basis of which an informed decision can be made. The component MasterDistributor is responsible for division of the submitted task and takes the print jobs as input and generates sub-tasks as the output. After the successful generation of the sub-tasks, the distribution of the sub-tasks among the slaves is done. The form in which the tasks are distributed may vary leading to different designs of the component which is discussed in detail in the next chapter.

On receiving the sub-tasks from the master, the slave nodes start doing their share of work. The master after submitting the sub-tasks has to wait for the slaves to generate the partial output. So as to maintain the streaming architecture, the slaves perform the assigned work in chunks and report the partial output to the master. The master then collects the partial output and performs some computation using received data and provides the final output to the user. MasterMerger component in the master printing pipeline is responsible for receiving the partial output from all the slaves and combining it before sending it on to the output component. The output from the slave nodes are the chunks of partial slices which are later merged into chunks of full slices. Figure 3.6 shows the components of the printing software running on the master node for Prototype I and Figure 3.7 for prototype II.

Slave Node Printing Software Components

The slave nodes in the cluster (i.e. all nodes other than the master are considered slave nodes) perform the assigned task and report back the results to the master node. To do so, the slaves nodes have to first wait until the master has divided the task submitted by the user into smaller sub-tasks. Once receiving the input, the slave nodes run the whole printing software (similar to the non-distributed cuttlefish version) with just one minor change. As the slave nodes report the partial output to the master, they do not need to run the output component, for example the Bitmapproducer. The partial output which is reported by the slaves to master is in form of slices and hence, the slave printing software running on the slave nodes does not contain the output component, instead it runs a component called the SlaveReporter which, as the
name suggests, reports the partial output in a chunk-wise fashion back to the master. Figure 3.8 shows the components of the printing software running on the slave node for Prototype I and Figure 3.9 for Prototype II.

3.3 Summary

This Chapter explains the system model i.e. the architecture of distributed system, the existing cuttlefish application streaming architecture as well as the changes in the printing software pipeline introduced as a part of the solution. Using this system model, various possible designs for the components of the pipeline are discussed in the next Chapter 4.
4 Design Analysis

An algorithm can be defined as a set of precise (i.e., unambiguous) instructions that specify how to solve a given problem. An algorithm can be implemented in multiple ways leading to varying performance and efficiency of the program, for example, using different data structures, data storage options i.e. if the data should be stored in memory or on file system, implementation language, library functions used etc. A design is a result of such numerous decisions made with the goal to achieve a balance between efficiency and optimization. Very rarely is it possible to find the perfect design which would lead to a program performing with the same efficiency for the entire input space. Hence, while making the decisions it is important to evaluate the advantages and disadvantages of the chosen approach. This chapter is therefore dedicated to analyze the design choices made during the implementation.

As discussed in Section 3.2, the task submitted by the user to the master node is divided into sub-tasks and distributed among the slave nodes of the cluster. The slave nodes perform the allocated work and report the partial output which is merged at the master node. The Figure 4.1 summarizes the process. The *MasterPrintingSoftware* follows the steps denoted in the left branch of the flow chart where as the *SlavePrintingSoftware* follows the steps of denoted in the right branch of the flow chart. The *MasterPrintingSoftware* and *SlavePrintingSoftware* design decisions include:

1. Format of the sub-tasks allocated to the slaves nodes
2. Distribution of sub-tasks by the master
3. Format of the slave partial output
4. Merging of the partial output

Each of the design possible from the various combinations of the above decisions yields a different behavior of the component responsible for that particular task. The following sections describes the two designs implemented through this thesis in detail.

4.1 Prototype I- Input/Output using Network File System

Prototype I is the base prototype wherein the nodes communicate amongst each other the input and the partial output using the shared network file system. The synchronization amongst the nodes is done via message passing.

4.1.1 Master sub-task creation and distribution

The user submits the job to the master node using a configuration file in JSON format - JavaScript Object Notation is a syntax for storing and exchanging data. The JSON document contains a member called as PrintObjectFiles whose member values indicate the geometry, texture, orientation etc of the object to be printed. The FileParserPJ component of MasterPrintingSoftware parses the set of input objects and converts it into set of print objects. Each print object contains a mesh representation of the geometry and the information related to it’s appearance. The print objects are grouped together into a print job. Each print job is then organized in the print tray and then passed to the MasterDistributor component which is in charge of sub-task creation and distribution.

The arrangement of the print objects on the print tray is done using a simple approach. For each print object the bounding box is calculated and the print object bounding boxes are sorted in the descending order i.e. largest to smallest. The whole print tray is seen as one big cell which contains the partitioned cells holding the print objects. For the first object i.e. is the one with the largest bounding box, the cell which fits the object is found and placed in the cell representing the print tray whose volume is equal to the print volume. The print volume is calculated by multiplying the height, width and length of the print tray. After placing the smaller cell, the larger print tray cell is split thus creating multiple smaller cells which can be chosen for the remaining objects. For each print object, the placement is recorded by the offset values of the cell in which it is placed starting from the origin.

The sub-tasks distributed by the component can be already created print jobs or they can be in the same format as the received input i.e. configuration file. In this prototype, the MasterDistributor creates a configuration file for each
Figure 4.1: Distributed Printer Driver Flow Chart
slave with the exactly same format as the configuration file parsed by the FileParserPJ with a few changes i.e. the PrintObjectFiles contains details of only the print models allocated to that particular slave and the OutputFolder value is the folder location for the slave to write the partial output. In the Listing 4.1, the configuration file received by the master node has the PrintObjectFiles containing four inputs models which are distributed by the master as seen in the Listing 4.3 for the slave with rank 1 and Listing 4.2 for slave with rank 2.

**Listing 4.1: Configuration file submitted by the user**

```json
{
    "ComponentConfiguration": "//pc2215/SharedFolder/layered_halftoner_CMYK_PrintJob.json",
    "PrinterSpecs": "//pc2215/SharedFolder/printer_specs_layer_profile_CMYK.json",
    "PrintObjectFiles": [
        {
            "GeometryFile": "//pc2215/SharedFolder/3DModels/Head/HeadMTL/Head_Level3.obj",
            "TextureFile": "//pc2215/SharedFolder/3DModels/Head/HeadMTL/HeadColour.jpg",
            "fixRotation": false
        },
        {
            "GeometryFile": "//pc2215/SharedFolder/3DModels/Head/HeadMTL/Head_Level3.obj",
            "TextureFile": "//pc2215/SharedFolder/3DModels/Head/HeadMTL/HeadColour.jpg",
            "fixRotation": false
        },
        {
            "GeometryFile": "//pc2215/SharedFolder/3DModels/Head/HeadMTL/Head_Level3.obj",
            "TextureFile": "//pc2215/SharedFolder/3DModels/Head/HeadMTL/HeadColour.jpg",
            "fixRotation": false
        },
        {
            "GeometryFile": "//pc2215/SharedFolder/3DModels/Head/HeadMTL/Head_Level3.obj",
            "TextureFile": "//pc2215/SharedFolder/3DModels/Head/HeadMTL/HeadColour.jpg",
            "fixRotation": false
        }
    ],
    "OutputFolder": "//pc2215/SharedFolder/MRepo/3DPrinterDriver/",
    "Loggers": [
        {
            "type": "ScreenLogger",
            "verbosity": "INFO"
        },
        {
            "type": "FileLogger",
            "verbosity": "TRACE",
            "OutputPath": "text.log"
        }
    ],
    "gnuPlotPath": "",
    "validationFileName": "validation.json",
    "DistributionPrototype": "1",
    "CommunicatorType": "NetworkCommMPI"
}
```

**Listing 4.2: Slave Configuration file Node 2**

```json
{
    "ComponentConfiguration": "//pc2215/SharedFolder/layered_halftoner_CMYK_PrintJob.json",
    "PrinterSpecs": "//pc2215/SharedFolder/printer_specs_layer_profile_CMYK.json",
    "CommunicatorType": "NetworkCommMPI",
    "DistributionPrototype": "1",
    "Loggers": [
        {
            "type": "ScreenLogger",
            "verbosity": "INFO"
        },
        {
            "type": "FileLogger",
            "verbosity": "TRACE",
            "OutputPath": "text.log"
        }
    ],
    "gnuPlotPath": ""
}
```
This prototype is possible only if the cluster nodes have read/write permission to the shared network file system where the geometry, texture files and input/output folders are stored /created. The MasterDistributor creates a unique path for each slave using the rank of the slave and writes the configuration file to this path. It then communicates to the slave only the path where the slave can find the configuration file. The sequence diagram in the Figure 4.2 summarizes the communication between the master and the slaves.

The advantages of distributing the sub-tasks through the configuration file are:

- **Design simplicity**: The most important aspect of this design is its simplicity as the SlavePrintingSoftware pipeline does not need to be modified greatly with respect to the serial 3D Printing pipeline, which means that it is almost similar to running a single instance of the non-distributed version of the 3D Printing pipeline.

- **Lower Communication Overhead**: Slave nodes are blocked until the master communicates the configuration file path to each slave. Communication of the file path is faster as it is less number of bytes to be exchanged among the nodes as compared to the number of bytes to be sent when print jobs are distributed.
**Figure 4.2: Master-slave input communication**

- **Faster Disk I/O**: The configuration file read and writes i.e. disk I/O (both at the master and slave) are much faster than sending the print job as series of bytes because of the additional steps (refer to section detailing the second prototype) that need to be performed while sending the print jobs. It is simpler implementation as communicating the print jobs as series of bytes would require serialization and deserialization of the print jobs with an additional effort of communicating the size along with texture information for each print object.

- **No serialization and deserialization**: As the configuration files are in JSON format, there are very efficient library functions which allow to read and write the files with ease. As the format of the files is quite standard, there is no need for serialization and deserialization of the files even if the nodes have varying underlying system architecture.

The disadvantages of distributing the sub-tasks via configuration file are:

- **Dependency on shared network file system**: This prototype has a strong requirement that the nodes have a shared network file system and each node must have read access to the model texture/geometry file along with write access to the output folder to dump the partial output. If the print jobs are communicated as stream of bytes, it is possible to get rid of this requirement.

- **Performance overhead**: The configuration file, model geometry and texture files are parsed twice, once by the master to create the print jobs needed for cost computation and once by the slaves for actual computation. This leads to redundant execution of the `FileParserPj` and `PrintJobOrganizer` at both master and slave nodes which accumulates to a significant performance overhead (refer to the Section).

- **Performance bottleneck at the master**: As the files are read and written to the shared network file system, the network bandwidth limits the speed of the disk I/O. With increase in number of slave nodes, the load at the master to write the configuration file per slave increases which may become a performance bottleneck.

---

**Sub-Task Distribution**

The sub-tasks are distributed by the `MasterDistributor` component based on a cost function. The goal of the distribution is to allocate equal work to each slave i.e. none of the slaves are over-worked and all of them finish the work almost at the same time. One of the main reasons which makes it important to achieve load balancing is to ensure better utilization of the cluster resources, a cluster as a whole will perform well only if each of the nodes do equal work leading to none of the nodes being the bottleneck.

The cost function used for distribution of the tasks evaluates the cost of each work item and allocation of work item among the slave nodes is done such a way that total cost of allocated work items is equal for each node. The cost for each work item may vary depending on the work item, for example for print objects, the size or the print volume of the print objects could be seen as cost of the work item, or priority assigned to the print objects can be another way to evaluate the cost.
Naive cost function implementation

Each print object \( O_i \) has an axis-aligned bounding box \( B(O_i) \). The minimum bounding box \( \text{min}(B(O_i)) \) and maximum
bounding box \( \text{max}(B(O_i)) \) is calculated. Using \( \text{min}(B(O_i)) \) and \( \text{max}(B(O_i)) \), the width (\( W \)), height (\( H \)), and length (\( L \)) of
each print object is calculated as per equation 4.1

\[
\begin{align*}
W &= \text{max}(B(O_i)).y - \text{min}(B(O_i)).y \\
H &= \text{max}(B(O_i)).z - \text{min}(B(O_i)).z \\
L &= \text{max}(B(O_i)).x - \text{min}(B(O_i)).x
\end{align*}
\] (4.1)

The volume \( (V^i) \) of the bounding box \( B(O_i) \) is calculated using the equation 4.2.

\[
V^i = WHL
\] (4.2)

The total sum of the volumes of \( k \) print objects calculated using equation 4.3

\[
V_{\text{Total}} = \sum_{i=1}^{k} V^i
\] (4.3)

Threshold (for each node in a cluster of size \( n-1 \)) calculated using equation 4.4 is used for allocating the sub-tasks to
the slaves. The cluster size considered is \( n-1 \) as the master is not allocated any sub-task and is hence excluded from the
cluster of slaves.

\[
T = \frac{1}{n-1} \sum_{i=1}^{k} V^i
\] (4.4)

The allocation of the sub-tasks is done using the greedy approach. The list of volumes calculated using equation 4.2 is
sorted in the descending order. Then the allocation of the object is done in a round-robin fashion to each slave until the
allocated total volume for the node is greater than or equal to the threshold. If the threshold is already reached for the
node, then the node is skipped. The allocation of the print object stops when each node's capacity has been consumed or
all the print objects are assigned. The listing 4.4 summarizes the distribution logic.

Listing 4.4: Distribute Tasks- Greedy Approach

```cpp
DistributeTasks(VolList, threshold, NumNodes) {
    std::sort(VolList.begin(), VolList.end(), std::greater<>()); // sort in descending order
    int listOfSum(NumNodes-1)={0};
    int vlSize= VolList.size();
    int nodeInd= 1;
    int allocatedIndexList(NumNodes-1);
    for (size_t ind = 0; ind < vlSize;)
    {
        if(listOfSum[nodeInd] < threshold)
        {
            sumList[nodeInd] = sumList[nodeInd] + tempList[ind];
            indList[nodeInd].push_back(ind);
            ind++;
        }
        else
        {nodeInd++;
        }
    }
}
```

The solution found using the greedy approach is \( 4/3 \)- approximation to the optimal solution of the optimization, when
the distribution is done among two nodes. The approximation can be estimated using the Equation 4.5, where \( k \) is the
number of subsets created through partition[25]. For \( n-1 \) slave nodes in a cluster of size \( n \), the number of partitions done
is equal to the number of slave nodes. By replacing \( k \) with \( n-1 \), we get the Equation 4.6. As the number of slave nodes
increases in the cluster, the solution found using this approach has better approximation to the optimal solution as seen in the Figure 4.3.

\[ A = \frac{k + 2}{k + 1} \]  \hspace{1cm} (4.5)

\[ A = \frac{n + 1}{n} \]  \hspace{1cm} (4.6)

4.1.2 Slave Partial Output

The SlaveReporter component receives the container of slices whose size is equal to the chunk size, as input from the previous component in the pipeline. At each of the slaves, the container actually contains the partial slices which are sent to the master for merging. Each slice internally stores the width, height and the buffer which contains the material id of the assigned material at varying \((x,y)\) coordinate. As the cluster may contain heterogeneous machines with varying system architecture, to ensure proper interpretation of the slices at the master, the slices are serialized. Serialization of the slices includes serializing the width, height (unsigned integers to same size), buffer to an output stream (std::ostream). The behavior of the SlaveReporter component is summarized in the Figure 4.4.

Analysis of component behavior

The SlaveReporter component is multi-threaded and uses a synchronized queue for inter-thread communication. The runInternal method of each component is the main thread of execution i.e. the cuttlefish process, whereas in the first run of this method a worker thread is created which executes asynchronously to perform certain tasks. The main thread interacts with the next and previous components of SlavePrintingSoftware whereas the worker thread interacts with the main thread and worker threads of MasterPrintingSoftware-MasterMerger component.

The interaction between the main thread and the worker thread can be viewed as the classic producer consumer problem, see Figure 4.5, where in the main thread produces the partial slices which are consumed by the worker thread and serialized. Each chunk of partial slices is one work item produced by the main thread and inserted in the buffer. Depending on the type of buffer, there might be varying number of synchronization points between the threads. In the current design, the buffer is implemented as a queue of limited size i.e. a bounded buffer. As the buffer is of fixed size, there are two conditions which lead to blocking either the main thread or the worker thread. As the main thread acts as the producer, it blocks when the buffer is full (i.e. slow consumer) and waits on a condition indicating at least one free
slot in the buffer. The worker thread acts as the consumer and blocks when there is no work item to consume and waits until there is at least one full slot.
The worker thread performs the task of serialization and once each chunk of slices is serialized, it communicates with worker thread of MasterPrintingSoftware-MasterMerger component the chunk size- it acts as a way of synchronization of the worker threads between the components. The worker thread sends some meta-data only for the first chunk of slices which includes slice height, slice width, chunk size, total number of slices. This meta-data is used by the main thread of MasterPrintingSoftware-MasterMerger component for creating full slices.

The advantages of SlaveReporter component design are:

- **Separation of concerns**: The main thread is responsible for assigning the work to the worker thread and returns to continue with execution of the pipeline with the next chunk whereas the worker thread is responsible for communication and serialization.

- **Asynchronous communication**: The worker thread performs the blocking calls for synchronization with worker threads of MasterPrintingSoftware-MasterMerger component. As it does so asynchronously with respect to the main thread, the communication will never lead to blocking the main thread. Both the main thread and worker thread execute in parallel.

The disadvantages of SlaveReporter component design are:

- **Synchronization Overhead**: The main thread and the worker thread need to synchronize access to the shared buffer so as to avoid race condition along with blocking calls on buffer full/empty condition. The size of the buffer influences the performance of the solution i.e. if unlimited buffer is used the producer will never block whereas if no buffer is used there is no need for explicit synchronization as producer produces only one item and the consumer consumes it after it is produced.

- **Increased implementation complexity**: Synchronization between the threads leads to added implementation complexity and if not managed properly might lead to deadlock.

### 4.1.3 Partial Output Merging at Master

The partial slices serialized by the slaves are de-serialized at the MasterMerger component. The MasterMerger component creates a worker thread per slave node in the cluster which interacts with the worker thread created by the SlaveReporter component. The main thread of the MasterMerger component synchronizes with worker threads at various points during the execution. The worker threads of MasterMerger component first wait to receive the meta-data...
information from the SlaveReporter component worker thread. After they have received the slice height, width, total number of slices and chunk size, they synchronize with the main thread and indicate the receipt of the meta-data. The worker threads then wait to receive the chunk size of the serialized data from the worker threads of SlaveReporter component, the receipt of the chunk size is the indication that slices are serialized and ready for deserialization. On receiving the chunk size, they start with deserialization of the slices and create a container of deserialized partial slices.

In the meantime, the main thread uses the received meta-data information from the slaves and then creates the container of the full slices. The worker threads at the master node continue to deserialize the chunk they received from the slaves and block until the container of full slices is created by the main thread and ready to be written. On signaling of availability of the container by the main thread, the worker thread then proceed with writing the partial slices to the full slices.

Each worker thread has a dedicated section in the full slice where it writes to, which helps to avoid write conflicts on the shared storage. The main thread waits until all the worker threads have finished writing the partial slices to the full slices and has container of full slices ready. The main thread then continues to proceed with the pipeline with fully merged slices whereas worker threads continue to deserialize the slices or communicate with worker threads of the slave or write the partial slices to the full slice in the container provided by the main thread. The activity diagram in the Figure 4.6 shows the activity of the main thread and worker threads of MasterMerger component.

The main thread at the master computes the full slice height ($H_f$), full slice width ($W_f$), offset ($O_f$) for each worker thread ($W_i$), and total number of full slices ($N_f$). The full slice height $H_f$ is computed as the addition of all the partial slice heights. If the full slice height $H_f$ is exceeding the print bed height, then full slice height $H_f$ is set to the print bed height $\max(H)$. The height is calculated as the addition of partial slice heights because the partial slices from the slaves are written one below the other in y-axis until either the print bed height is reached or each of the slave node partial slices are placed. If the print bed height $\max(H)$ is reached and there are still pending slave partial slices to be fit in the full slice, then the maximum partial slice width is taken as the x-offset and the remaining slices are then places along the y-axis again with each having the x-offset. The listing 4.5 summarizes the logic of full slice height calculation.

### Listing 4.5: Calculate full slice height

```cpp
1 int fullsliceHt(std::vector<int> listOpartiSliceHt)
2 {
3     int maxAllowedHt = getPrintBedHeight(); // get the print bed height
4     int totalHt(0);
5     for(size_t i =0;i<listOpartiSliceHt.size(),i++)
6         { // to indicate that x-offset will be added to some slaves
7             totalHt = listOpartiSliceHt[i] + totalHt;
8         }
9     if(totalHt > maxAllowedHt)
10         {
11             totalHt= maxAllowedHt;
12             m_xOffsetNeed =true; //
13         }
14     return totalHt;
15 }
```

The full slice width ($W_f$) is calculated as the maximum of the all the partial slice widths. If the total height calculated exceeds the print bed height, then total slice width is calculated as the sum of all slice widths. If the total slice width exceeds the print bed width $\max(W)$, then there is no way to fit all the partial slices in the given print bed and hence some of the slaves need to be dropped. If the total slice width $W_f$ does not exceed the print bed width $\max(W)$, the full slice width ($W_f$) is equal to the sum of individual partial slice widths. The full slice width calculation is summarized in the listing 4.6.

### Listing 4.6: Calculate full slice width

```cpp
1 int fullsliceWd(std::vector<int> listOpartiSliceWd)
2 {
3     int maxAllowedWd = getPrintBedWidth(); // get the print bed width
4     int totalWd(0);
5     if(m_xOffsetNeed) // this is set to true if the total sum of slice heights > print bed height.
6         {
7             for(size_t i =0;i < listOpartiSliceWd.size(),i++)
8                 { //
9                 }
```
totalWd = listOfpartilSliceWd[i] + totalWd;
}
if (totalWd > maxAllowedWd)
{
  m_DropSlaves=true; // to indicate that some slaves need to be dropped
}
else
{
  for(size_t i =0;i < listOfpartilSliceWd.size(),i++)
  {
    totalWd = totalWd > listOfpartilSliceWd[i]? totalWd : listOfpartilSliceWd[i];
  }
}
return totalWd;

The total number of full slices is the maximum value from the received total number of slices from the slaves. The main thread needs to keep track of the individual total number of slices from each slave node so as to ensure that the master does not block for a worker thread to write to the full slices when there are no more partial slices left.

The slice container to be provided by the main thread is a synchronized slice container which means that while removing the individual slices or subset of slices or adding the slices to the container, each thread needs to obtain the lock over the container and modify the container thereafter. The use of synchronized container ensures no data race on the shared data structure.

The advantages of the MasterMerger component design are:

- **Separation of concern**: A worker thread for each slave node helps to maintain the separation of the responsibilities among the worker threads as each worker thread is responsible for the assigned slave node.

- **Asynchronous communication and exploiting the possible parallelism**: The worker threads for each slave node communicate with the worker thread of the SlaveReporter component asynchronously without blocking the execution of the main thread. The worker threads perform deserialization of partial slaves and write to their assigned section in the full slices simultaneously without write conflicts amongst them. The main thread also continues to work on computing the full slice height, width etc while the worker threads are deserializing and when the worker threads are done writing to the full slice container, return the full slice container to proceed with next component execution in the pipeline in a chunk-wise fashion. This design maintains the streaming architecture of the original non-distributed version of the pipeline.

The disadvantages of the MasterMerger component design are:

- **Synchronization Overhead**: As there are many synchronization points during the execution where either the worker threads block or the main thread block, it might lead to overhead in the execution run time. Also the synchronization leads to added code complexity.

- **Limited possibilities of partial slice arrangement**: As the arrangement of the partial slices in the full slices is done, there is a possibility of not achieving the optimal arrangement leading to the waste of the print area along with incapability of accommodating each slave. This might lead to dropping one or more slaves from the merging process to making the best possible use of the work done by the slaves.

- **Possibility of deadlock**: The main thread at the master blocks until each slave has received the meta-data or written to the full slices. The main thread might remain blocked if one of the slaves crashes which will eventually lead to a deadlock as other slaves might get blocked until the main thread refills the container with new empty slices. To avoid this, use of timeouts could be done. If the worker thread for a particular slave does not respond within the timeout period, then slave could be dropped. The timeouts shouldn't be too small leading unnecessary dropping of the slaves or too long leading to increased execution time overhead even when the blocking slave has crashed. The use of timeouts adds further code complexity.

### 4.2 Prototype II- Input/Output independent of Network File System

As discussed in the Section 4.1, the prototype I functions only if there is a shared network file system amongst the cluster nodes. The network file system is primarily used to communicate between the nodes the work load (i.e. the configuration file) and the partial output. So, to get rid of this dependency the communication between nodes must be done by sending
the data through the network as a stream of bytes and not via disk I/O. The prototype II involves two major changes as compared to Prototype I:

- Independence from Network File System
- Format of the work item allocated to the slaves
The architecture of the distributed cuttlefish pipeline remains almost the same with two minor changes, the MasterDistributor component is replaced by another component called as MasterPJ Distributor in the MasterPrintingSoftware, and in the SlavePrintingSoftware, the component SlavePJ Collector replaces the components FileParserPJ and PrintJobOrganizer. These changes in the printing software are explained in the following subsections.

4.2.1 Selection of the prototype

The algorithm for performing the distributed computing progresses in the same fashion as described in the Figure 4.1. The configuration file submitted as input by the user to the cuttlefish application consists of a tag called as DistributionPrototype whose value indicates the distribution prototype to be used. The valid values for this tag are 1 for prototype I and 2 for prototype II. The master node parses the configuration file and depending on the value of the DistributionPrototype tag, it creates the components of MasterPrintingSoftware and sends the prototype value to the slave nodes. The slave nodes wait to receive the prototype value and then add the components of SlavePrintingSoftware depending on the received value.

4.2.2 Printing Software Components and Printer Specification

In the prototype I, the slave nodes receive the path where they could find the configuration file containing the information about of the assigned print models, texture information, component configuration for the printing pipeline and the printer specification. Just sending the prototype value is not sufficient for the slave nodes to create the SlavePrintingSoftware. The master needs to send to each slave node all the information which was earlier contained in the main configuration file along with the information from the other files included in the main configuration file. So master should send the JSON data describing the component configuration and printer specification. Each slave node after receiving the prototype value decides what is the next information that it should wait for from the master node. For prototype I, the slave nodes wait for the main configuration file path and for prototype II, they wait for the component configuration and the printer specification description. The component configuration description depends on the user’s choice of components and the configuration of each component. As for each component the number of parameters and their values may vary, the master first sends the size of the data followed by the data describing the component configuration. Similar to the component configuration, the printer specification description depends on the printer settings provided by the user via the printer specification file. As the description of the printer specification is subject to vary depending on the available 3D printer, size of the data is sent to the slave nodes followed by the data describing the printer specification.

4.2.3 Distribution of the work load to the slave nodes

After communicating the information related to the prototype, component configuration and printer specification, the next step for the master node is to create the work load for each slave. The MasterPrintingSoftware starts with the steps similar to prototype I i.e. parse the configuration file provided by the user and create the print job. The components responsible for doing these tasks namely, FileParserPJ and PrintJobOrganizer, remain unchanged. The print job is then passed to the MasterPJ Distributor component. From this point onwards the behavior of the pipeline varies for prototype II. The MasterPJ Distributor component is responsible for distributing the print jobs to the slave nodes instead of writing the configuration files per slave node. The cost function used to distribute the load amongst the nodes is the same as prototype I. The difference is the format in which the work load is assigned to the slave nodes. For each slave, a new print job containing the print objects (distributed as per the cost function) is created. Each newly created print job is a set of the print objects such that, the sum of the volumes of the print object bounding boxes is less than or equal to the threshold calculated using the cost function. The goal of the cost function is to have equal work load for each slave node so that each one of them finishes the chunk-wise computation almost at the same time.

The steps performed by MasterPJ Distributor component are as follows:

1. The PrintJobOrganizer creates the print job and passes the print job to the MasterPJ Distributor component.
2. In the MasterPJ Distributor component, the cost function is calculated using the print object bounding boxes. The calculation of the cost function is same as described in the subsection 4.1.1.
3. Using the threshold calculated in the step 2, the distribution of the print objects is done such that sum of the volumes of the print object bounding boxes (assigned to each slave) is less than or equal to the threshold.
4. The set of print objects assigned to each slave is then packed into a print job per slave.

5. The print job is then serialized to a stream of bytes and sent to the slaves. As the size of the print job may vary, the size of the serialized print job is sent to the respective slave first followed by the print job data.

6. Each print object has texture information which is used to generate the desired appearance. The texture data is stored in the master node GPU memory when the print job is created. For the slave nodes to perform the computation and generate the partial slices correctly, the slave nodes need the texture information which should be sent by the master. The texture data size may also vary per print object and hence, the size of serialized texture data is sent first followed by the serialized texture data.

So to summarize the information sent by the master node to each slave node:

- prototype value indicating the chosen distribution type
- size of the component configuration description
- component configuration description
- size of the printer specification description
- printer specification description
- size of the serialized print job
- serialized print job
- for each print object in the print job, size of texture data and texture data

### 4.2.4 Work Load Collection and Computation

At the slave nodes, the first step performed is receiving the type of distribution i.e. the prototype value set by the user via the configuration file submitted to the master node. After it receives the prototype value, the slave node decides which is the next information it should wait for. For prototype II, the slave nodes wait for the component configuration. Each slave node in the cluster receives the same component configuration from the master. As the component configuration is set by the user and the size of the JSON string describing the component configuration may vary in size, the slave nodes first receive the size of the string and allocate the buffer of the appropriate size to receive the actual component configuration string. Followed by the component configuration, the slave nodes waits to receive the printer specification, similar to component configuration information, first the size of the JSON string holding the printer specification followed by the actual string.

The slave nodes then proceed to create the printing software containing the needed components. Similar to the prototype I, the SlavePrintingSoftware does not contain any output component. As the work load received by the slaves nodes in the prototype II is a print job, the components FileParserPJ and PrintJobOrganizer are not included in the SlavePrintingSoftware, instead a new component called as the SlavePJCollector is included. The SlavePJCollector component receives the size of the serialized print job sent by the master and allocates the buffer to receive the actual serialized print job as stream of bytes. It then deserializes the received data to create the print job. For each print object contained in the print job, it waits to receive the texture information. The slaves receive the size of the texture data, allocate the buffer for the texture data and then receive the texture data. The received texture data is then deserialized and saved in the memory of the node. The rest of pipeline remains unchanged until the SlaveReporter component.

The SlaveReporter component, similar to the prototype I, receives the partial slices and reports them to the master node. As the nodes within the cluster do no have a shared network file system, the slices which are reported to the master node are in form of stream of bytes instead of files written to the disk. The value of the selected distribution type is set at the time SlavePrintingSoftware creation as persistent data- it is a piece of information accessible by all the components of the pipeline. SlaveReporter fetches the prototype value from the persistent storage before it proceeds with rest of the computation.

The SlaveReporter component sends the meta-data information which is the partial slice height, width, total number of slices for the assigned work load and first chunk size. For the prototype II, it skips sending the path where the master node would find the partial slice file. For each chunk of partial slices to be sent to the master, the slave begins by informing the
size of the chunk followed by serialization of the slices. Through serialization of the slices it converts the slices to stream of bytes. The slave nodes send the size of the serialized slice to the master which allows the master to allocate the buffer of appropriate size and then send the serialized slice data. The \textit{SlaveReporter} component design remains unchanged and is as described in the subsection 4.1.2.

4.2.5 Creation of the full slices

At the master node, after assigning the work load to the slave nodes, the \textit{MasterMerger} component is executed. It is in charge of collecting the partial slices from the slave nodes and merging them to form full slices. The \textit{MasterMerger} component, similar to the \textit{SlaveReporter} component, fetches the prototype value from the persistent storage which is saved when the instance of the \textit{MasterPrintingSoftware} is created. There is no huge difference in how the component performs the merging of the partial slices but rather how it receives the partial slices from the slave nodes. The component design remains unchanged and is as described in the subsection 4.1.3. Each worker thread begins by receiving the meta-data information from their respective slave node. For the prototype II, the worker thread does not receive the path where the partial slices files are written by the slave. After receiving the chunk size, the worker threads receive the size of the serialized slice, allocate the buffer and receive the serialized slice as stream of bytes. They then deserialize the stream of bytes to partial slice and store them in the partial slice container. The rest of the steps until the merging of partial slices to full slice and passing them to the output component remains unchanged.

The information communicated between the master and slave nodes is summarized in the Figure 4.7.

4.2.6 Analysis of Prototype II design

As stated earlier, the changes introduced in the prototype II are with respect to the medium used for communication between the cluster nodes. Due to these changes, some of the challenges of the prototype I are resolved. Although, prototype II has a few shortcomings as well. The advantages and disadvantages presented for the component design of \textit{MasterMerger} and \textit{SlaveReporter} apply for prototype II as well as these component's design is unchanged. This section details the analysis of the prototype II in comparison to prototype I.

The advantages of the prototype II are:

- **Independence from the shared network file system:** The prototype II does not require the cluster nodes to have a shared network file system which makes it possible to perform distributed computing in a cluster of nodes connected via network.

- **Avoids redundant execution of certain components at the slave nodes:** The workload assigned to the slave nodes in the cluster are print jobs. As there is no configuration file to be parsed by the slave nodes (as done in the prototype I), the slave nodes do not need to run the \textit{FileParserPJ} and \textit{PrintJobOrganizer} components again. This improves the performance of the \textit{SlavePrintingSoftware} significantly if the geometry and texture files of the input print models are huge.

- **Lesser overhead due to no disk I/O:** For each slave node, the master node had to create the configuration file per slave node and communicate the path. The slave nodes had to write the serialized slices to the disk which were then again read and deserialized by the master node. The amount of disk I/O increases with the increase in the cluster size for the prototype I. In prototype II, the input and the output is communicated as stream of bytes which helps to avoid disk I/O.

The disadvantages of the prototype II are:

- **Increased number of send and receives:** The master sends all the information needed by the slaves as stream of bytes via send and receive primitives. Each send and receive done by the master or the slave is a blocking call. So if one of the nodes is slow in sending/receiving the information, it remains blocked until the call is done. With increase in the size of data to be sent, the overhead of blocking calls increases. If the calls are asynchronous, meaning that the node is not blocked until the respective receive is done by the recipient node, then the required synchronization needs to be additionally implemented by the programmer to ensure there is an order maintained between the sends and also handle the cases of failed communication.

- **Increased computation due to frequent serialization and deserialization:** As the cluster nodes can be heterogeneous, the information communicated between the nodes needs to be serialized at the sender and deserialized at the receiver. The process of serialization and deserialization is explained in the section 4.3. Due to this, for each send and receive there is an additional computation to be performed. As the size of byte stream is not fixed, the size needs to be communicated first followed by the actual data.
4.3 Serialization and Deserialization of Data

The nodes forming the cluster used for distributed computing may not always be homogenous, which means that each node may be different in terms of the system architecture, computation capacity and memory. Due to such heterogeneity, the representation of the user defined data types (i.e. in our case objects) may vary as the “endianness” of the nodes might differ [26]. To ensure that each node interprets the data correctly, the objects are “flattened” to stream of bytes or a string before sending it the destination, also known as serialization of the objects. At the destination node, the received stream of bytes are “unflattened” to reconstruct the object, which is known as deserialization [27]. The Figure 4.8 summarizes the serialization and deserialization process used.

Figure 4.7: Communication between slave and master nodes
4.4 Communication medium

The cluster nodes are connected via network and communicate through message passing. The communication protocol for passing the messages via the network can be TCP/UDP, Message Passing Interface (MPI), REST (Representational State Transfer) or HTTP (Hypertext Transfer Protocol). Irrespective of the protocol used, the two prototypes must function without any need for implementation or logical modifications. To achieve this and to provide the user the convenience of choosing the desired communication protocol, the user can set the protocol of choice through the configuration file which would be used to generate the communicator using the set value as communicator's underlying protocol. The implementation details of how the messages are sent or received are abstracted away as the communicator provides basic communication primitives as the functions of the interface. The Figure 4.9 summarizes the process of how the communicator is selected.

4.5 Summary

This Chapter explains the various possible designs for the components of the pipeline for the two prototypes. Each of the design is discussed in details, in terms of the advantages, disadvantages as well possible impact of the design on the performance. Distributed computing specific computation like serialization and deserialization are discussed as well. The design of Abstraction Layer to different communication mechanism are also detailed in the current Chapter. The next Chapter 5 describes the procedure followed for implementing the discussed design.
5 Implementation

The design for the prototype I and prototype II detailed in the chapter 4 were implemented through iterative development process where the design for prototype I was drafted and implemented first, followed by the prototype II. This chapter details the procedure which was followed during the development and implementation of the thesis.

5.1 Native Cuttlefish Application

The native cuttlefish application is a 64-bit windows application, developed in C++ language using the Microsoft Visual Studio 12 SDK and the compiler used to build the code is Microsoft C/C++ V120 compiler. A project could be compiled either as static library or an executable in release or debug mode. The input for the application i.e. the path of the main configuration file, is presented as command line argument. The output generated by the application depends on the type of output component chosen in the component configuration. For each component, the behavior of the component is logged in the performance report and a log file for overall pipeline is written to the working directory at the end of the execution. This helps to analyze the component performance as well as to know the possible error by looking into the pipeline execution trace in case of a failure. Some of the projects of the application use external libraries (like zlib, little cms, poly2tri, boost, itk etc) which are either used as headers only or compiled for windows platform and linked as static library.

Some parts of the computation performed by the application is on the GPU (graphics processing unit) and the code performing this computation is written and compiled using CUDA Toolkit. Another library used by the application is OpenGL(Open Graphics Library) wherein various powerful visualization features of the library are used for rendering, texture mapping etc. The minimum OpenGL version that is needed by the application to run is 2.0 as one of the APIs used- glCreateProgram, is available only in version 2.0 or higher. The native application has a strong dependency on availability of graphics library and CUDA.

5.2 Prerequisites for Distributed Cuttlefish Application Development

The goal of the thesis was earlier to provide a cloud service which could take the input from the user and do the computation using the cloud instances and provide the bitmaps as the output. With this goal, the native application had to be modified such that it could run on the cloud instances provided by the AWS (Amazon Web Services) as PAAS (Platform as a Service). The cluster of instances which formed the ‘cloud’ were the basic Amazon EC2 instances which were virtual machines with Linux operating system. As the basic instances were virtual machines which were not GPU accelerated i.e. they did not have access to a GPU, and therefore the code which performed the GPU computing had to be changed such that only when the GPU is available, the code is compiled and used. So, this change to the native application code was done by creating a new configuration, other than release and debug, called as the CUDA_FREE_CONFIG which got rid of the dependency on availability of CUDA on the system running the application. The code performing the GPU computing is defined under if-defs to get the CUDA_FREE_CONFIG as seen in the Listing 5.1

```c
//...
#ifndef CUDA_FREE
    inline operator std::shared_ptr<VoxelSliceGPU>()
    {
        if (m_channel == m_invalidIndex)
            return nullptr;
        validate();
        return std::dynamic_pointer_cast<VoxelSliceGPU>(m_pContainer->m_slices[m_channel][m_slice]);
    }
#endif
//...
```

The CUDA_FREE_CONFIG configuration was still only for windows platform and the projects needed to be compiled such that they were compatible with Linux and it was necessary to get rid of the windows specific dependencies. To achieve portability and make the cross-platform build of the application possible, a configuration file called as the ‘CMakeLists.txt’ was written for each project of the application. Using CMake, an open-source system that manages the
Table 5.1: MS-MPI Environment Variable

<table>
<thead>
<tr>
<th>Environment Variable</th>
<th>Value</th>
<th>Significance</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSMPI_INC</td>
<td>..\Microsoft SDK\MPI\Include</td>
<td>Path to find the headers</td>
</tr>
<tr>
<td>MSMPI_LIB64</td>
<td>..\Microsoft SDK\MPI\Lib\x64\</td>
<td>Path to find the libraries for 64 bit system</td>
</tr>
<tr>
<td>MSMPI_BIN</td>
<td>..\Microsoft MPI\Bin\</td>
<td>Path to find the binaries</td>
</tr>
</tbody>
</table>

5.3 Installation and Cluster Set-up

To begin with, a cluster with 2 nodes was created such that the nodes were connected to each other via a common network. To check if the cluster nodes could communicate, a simple ping test was performed. The next step was to write a program which performed a simple task in a distributed fashion and to run it on the cluster. For the cluster to work as system where the processing can be shared, there needs to be a way in which the cluster nodes can be named and the work to be done by each node could be distinguished i.e. assign roles to the nodes. There are many types of hardware/software solutions for a cluster which enable naming and role distinction among cluster nodes. The decision to use MPI-Message Passing Interface, was made to achieve naming and role assignment amongst the cluster nodes. MPI is the open standard specification for message passing libraries which provide apis to enable communication, role distinction and naming amongst the cluster nodes. There are various implementations of the specification available which basically perform the same thing i.e. message passing amongst the cluster nodes but are different with respect to the platform they support, commands to compile and launch the program, debugging and developed by different institutions. Some of the implementations are free-ware and some aren’t. As currently, cuttlefish is a windows application developed using Microsoft visual studio, the Microsoft’s implementation of MPI called as MSMPI is used.

To use MSMPI, on each node of the cluster, the MS-MPI SDK is downloaded and installed. To verify if the installation and setup is done correctly, the environment variables are checked using the command ‘set MSMPI ‘. The environment variables which are particularly interesting are $MSMPI_INC$, $MSMPI_BINS$, and $MSMPI_LIB64$. The Table 5.1 summarizes the details about these variable. To compile a program which is designed to run on the cluster where the nodes communicate using MPI, the directories which help to resolve the reference to header files specific to MPI and path to the libraries which resolve linking to the MPI functions are incorporated in the project. Before running the program, a process management system which allows to manage the parallel processes on the cluster nodes should be executed on each of the cluster node. For MPI, the process management system used is called as SMPI (Simple Multipurpose Daemon) which is part of the software component of the process manager provided by Intel. It allows starting and managing parallel jobs on the cluster nodes. SMPI is used as a windows service and is set during the MPI installation. Ideally, the version of SMPI running on each node in the cluster should be same, although if there are different versions of SMPI, there is a possibility to disable the check for the versions. As the SMPI service is set during MPI installation, the version of SMPI is compatible with the mpiexec version. To run the program, mpiexec command is used with the arguments specifying the number of hosts, host names, the directory where the executable for application to be launched is located.
is available, the name of the executable followed by arguments to the executable. The listing 5.2 is an example of the command syntax to launch the application using mpiexec command.

**Listing 5.2: mpiexec syntax**

```bash
mpiexec -hosts 2 PC2215 PC2286 -wdir \pc2215\bin\Release\ Cuttlefish.exe \pc2215\mainconf.json
```

5.4 Introduction to New Components

After the set-up and installation, the next step was the implementation of the prototype I followed by the implementation of prototype II. The components of both the prototypes are discussed in detail in the Chapter 4. Each of the new component is inherited from the class called as the *PrintingComponent* and is registered using the component name. During the creation of the pipeline i.e. when the component configuration is parsed, each of the component is created and initialized with the required parameter values. The parameter values for each component are set in the configuration file. The Figure 5.1 summarizes the new components (underlined component names) introduced during the implementation of the thesis.

![Component Diagram For New Components](image)

**Figure 5.1: Component Diagram For New Components**

Each component has an input it receives from another component and an output that it generates which may be used by the following component in the pipeline (if any). The dependency is depicted in the Figure 5.1 by the ‘lollipop’ symbol wherein the half circle depicts interface exported by the component and the circle represents the interface that is imported by the component, for example, the component *MasterDistributorPJ* uses the print job created by the *PrintJobOrganizer* and provides the print jobs with distributed print objects used by the *SlavePJCollector*.

5.4.1 Introduction to the New Classes

The new classes which were implemented during the thesis are enlisted in the Table 5.2 for prototype I, in the Table 5.3 for prototype II and common classes used by both the prototypes are listed in Table 5.4.
Table 5.2: Class specific for Prototype I

<table>
<thead>
<tr>
<th>Class</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>MasterDistributor</td>
<td>Class for slave work item creation and distribution. The work item created is the main configuration file containing the assigned number of print object files.</td>
</tr>
</tbody>
</table>

Table 5.3: Classes specific for Prototype II

<table>
<thead>
<tr>
<th>Class</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>MasterPJDistributor</td>
<td>Class for creation of print jobs with the print objects distributed using the cost function and serialization of the print jobs</td>
</tr>
<tr>
<td>SlavePJCollector</td>
<td>Class for collection of the serialized print job, de-serialization of the print jobs and loading the texture data to the node memory.</td>
</tr>
</tbody>
</table>

5.4.2 Code Organization

The solution for the application is generated using the CMAKE tool with each project having its own CMakeLists.txt file. The CMakeLists.txt file specifies the properties of the project such as the project would be built as an executable or static library or dynamic library, the libraries it needs to be linked to, the include directories etc. The code written throughout the thesis is organized in 3 different projects namely P3DDistributedCuttlefish, P3DDistributedUtility and P3DNetworkComm. The Table 5.5 describes code organization in the projects named. Some of the directives of CMake which influence the generator output of a project are add_library, add_subdirectory, target_link_library and add_dependencies. The Table 5.6 details the directive used.

While using CMake tool to generate the solution for distributed cuttlefish application, the flag P3D_DISTRIBUTED should be enabled which allows the inclusion of the projects which have the distributed computing implementation. The flag is defined so as to help easy solution generation of both distributed and native cuttlefish application. When the flag is set, the sub-directories are added by using the add_subdirectory directive in the CMakelists.txt. The Listing 5.3 shows how the sub-directories are included and the use of the flag.

Listing 5.3: Flag to enable distributed computing solution

```cpp
if(DEFINED P3D_DISTRIBUTED)
  add_subdirectory(P3DDistributedCuttlefish)
  add_subdirectory(P3DDistributedUtility)
  add_subdirectory(P3DNetComm)
endif()
```

The MPI dependencies are managed by using the find_package directive in the CMakeLists.txt of the target project. The Listing 5.4 is shows how to find about availability of MPI and if MPI is available, include the headers as well as link to the library.

Listing 5.4: find_package use for MPI libraries

```cpp
if(MPI_FOUND)
  include_directories(${MPI_INCLUDE_PATH})
  target_link_libraries( P3DDistributedCuttlefish ${MPI_LIBRARIES})
endif(MPI_FOUND)
```

5.5 Communication Specifics

Microsoft Message Passing Interface (MS-MPI) is Microsoft’s windows specific implementation of Message Passing Interface Specification (MPI-2 version). It provides 160 APIs which allow communication in a distributed computing environment by use of message passing paradigm and help to build application which perform parallel computation. This section details the API’s used in the implementation followed by a brief explanation of how MS-MPI works internally and last but not the least comparison with other communication mechanisms.
Table 5.4: Common Classes For Prototype I and II

<table>
<thead>
<tr>
<th>Class</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>MasterPrintingSoftware</td>
<td>Creates and manages the printing software for the master node</td>
</tr>
<tr>
<td>SlavePrintingSoftware</td>
<td>Creates and manages the printing software for the slave node</td>
</tr>
<tr>
<td>P3DNetworkComm</td>
<td>Interface exposing the methods for communication</td>
</tr>
<tr>
<td>P3dNetworkCommMPI</td>
<td>Implements the P3DNetworkComm interface methods using MPI as the message passing library</td>
</tr>
<tr>
<td>DistributedComponentParser</td>
<td>Parser for the JSON files used for distributed computing</td>
</tr>
<tr>
<td>DistributedComponentInitiator</td>
<td>Initiator class for distributed computing. It manages the cluster creation, execution and shut down of the printing software</td>
</tr>
<tr>
<td>SyncVoxelSlicesContainer</td>
<td>Synchronized voxel slice container allowing synchronous access to multiple threads</td>
</tr>
<tr>
<td>MasterMergerWorker</td>
<td>Interface exposing methods for performing collection of metadata, de-serialization and writing partial slices to full slices</td>
</tr>
<tr>
<td>MasterMergerWorkerImpl</td>
<td>Implementation of the MasterMergerWorker interface with a thread per slave</td>
</tr>
<tr>
<td>DistributedCostFunction</td>
<td>Class for calculation of the threshold and cost function used for load distribution</td>
</tr>
<tr>
<td>SlaveReporter</td>
<td>Class which performs serialization of the slices and reports them to master</td>
</tr>
<tr>
<td>MasterMerger</td>
<td>Class to collect the partial slices from slave nodes and generation of merged full slices</td>
</tr>
<tr>
<td>SlaveDataBuffer</td>
<td>Synchronous queue used by the SlaveReporter threads class</td>
</tr>
</tbody>
</table>

5.5.1 MPI APIs used in the Implementation

A parallel program written using MPI first should initialize the MPI execution environment which allows to perform the following:

- Create, initialize and set up a message passing layer
- Launch the stated number of processes
- Allocation of resources such as shared memory (if required), local inter-process communication channels, network communication channels and to allow communication with specialized network hardware it allocates ‘special’ memory.

If the initialization of the MPI execution environment fails due to failure in one of the above steps, the command hangs or aborts depending on the chosen implementation. For MS-MPI, there some warning messages shown on the command prompt. The initialization is done using MPI_INIT method as seen in the Listing 5.5.

Listing 5.5: Initialization of Execution Environment

```c
#include <mpi.h>

void MPI::Init(int & argc, char ** argv)

void MPI::Init()
```

The thread that calls the MPI_INIT should also be the thread which destroys the set-up by calling MPI_FINALIZE. MPI_INIT should ideally be the first method to be called before any kind of communication can be done within the distributed computing environment, although MPI_INITIALIZED - checks if MPI environment is initialized by calling MPI_INIT, MPI_FINALIZED - checks if the set-up has been destroyed by calling MPI_FINALIZE, MPI_GET_VERSION - gets the MPI standard followed by the current implementation, can be called before MPI_INIT.

The set of processes launched during the initialization process form a communicator and each process has rank within the communicator. The query functions like MPI_COMM_SIZE and MPI_COMM_RANK are local functions which do not require coordination and communication outside the local MPI process as they are local look ups. MPI_COMM_SIZE returns the size of communicator created during the MPI_INIT method execution and MPI_COMM_RANK returns the rank of process, executing the instruction, in the communicator. The implementation decides the order in which the
### Table 5.5: Projects for Distributed Computing

<table>
<thead>
<tr>
<th>Project</th>
<th>Components / Classes</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>P3DDistributedCuttlefish</td>
<td>MasterMerger, MasterDistributor, MasterPJDistributor, SlaveReporter, SlavePJCollector</td>
<td>1. Project contains the components which are implemented for distributed computing. 2. The project builds as a dynamic linking library. 3. The project uses the Zlib, rapidjson external libraries.</td>
</tr>
<tr>
<td>P3DDistributedUtility</td>
<td>MasterPrintingSoftware, SlavePrintingSoftware, DistributedComputingInitiator</td>
<td>1. Project contains the classes implementing the printing software and the initiator class for distributed computing. 2. The project builds as a static library. 3. The project uses rapidjson external library.</td>
</tr>
<tr>
<td>P3DNetworkComm</td>
<td>P3DNetComm, P3DNetCommMPI</td>
<td>1. Project contains the interface exporting the methods used for communication over the network and an MPI specific implementation of the interface. 2. The project builds as a dynamic linking library. 3. MS-MPI library is the external library used in the project.</td>
</tr>
</tbody>
</table>

Ranks are assigned to the processes and generally the first hostname in the mpiexec command has the rank 0 which is the rank of the root/master process. The Listing 5.6 shows the syntax of the MPI_COMM_SIZE and MPI_COMM_RANK methods.

#### Listing 5.6: MPI Query Functions

```c
#include <mpi.h>

int Comm::Get_size() const
int Comm::Get_rank() const
```

Message passing is done through fundamental directives called as send and receive. The name suggests, the send directive is used to send information and receive directive is used to receive the information. The communication when done using the send and receive directives is point to point communication as there is one sender and one receiver for each pair of send and receive. The communication can also be a broadcast where in there is one sender and rest of the processes in the communicator are receivers. In the implementation of the thesis, point to point communication is done using MPI_SEND and MPI_RECEIVE methods.

The directives for point-to-point communication come in two different flavors, blocking and non-blocking. The blocking send and receive can have 3 additional modes of communication namely, buffered, synchronous and ready. The blocking send does not return from the call until the message data and envelope has been safely stored and the message buffer is ready to be modified again [28]. The blocking send does not provide a status of message being sent or received but rather just provides a guarantee that the message buffer is available again. The Listing 5.7 provides the syntax of a blocking send call. The first three arguments- pointer to the buffer holding the message data, count of elements stored in the buffer and type of each element in the buffer form the message content whereas the last three arguments namely the rank of the sending process, the tag-a unique number identifying the message and communicator handle form the message envelope.

#### Listing 5.7: Blocking MPI Send

```c
int MPI_Send(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
```

The blocking receive consists of 6 parameters, the first three include the pointer to receiver's buffer, the size of the data to be received also the size of the buffer and the type of data to be received and the following 3 parameters include rank of the sender process i.e. the source of the sender, tag is the unique number identifying the message and communicator to which the process belongs to. The receiver can receive message from 'any' source whereas the sender always has to specify the receiver of the message which is known as the 'push' mode. If the receiver had to always mention a specific sender then such a communication mode is called the 'pull' mode. MPI_STATUS can used to know the status of the
<table>
<thead>
<tr>
<th>Cmake Directive</th>
<th>Purpose</th>
<th>Example</th>
</tr>
</thead>
</table>
| add_library      | 1. Adds a library with the stated target name to be built from the listed source code.  
2. The library could be static library, dynamic linking library or modules.                                      | add_library(P3DDistributedCuttlefish SHARED SRC_FILES SRC_HDRS DEPEND_INCLUDES)                                                                                                                     |
|                  |                                                                                                                                                                                                            | The project P3DDistributedCuttlefish is added as dynamic linking library (SHARED keyword specifies creation of dynamic linking library to the cmake tool), followed by the source files, header files and other includes needed to build the project. |
|                  |                                                                                                                                                                                                            | add_library(P3DDistributedUtility STATIC SRC_FILES SRC_HDRS DEPEND_INCLUDES)                                                                                                                     |
|                  |                                                                                                                                                                                                            | The project P3DDistributedUtility is added as a static library specified by the use of STATIC keyword followed by the source files, header files and included needed to build the project. |
| add_dependencies | 1. Adds the dependency among the projects.  
2. It helps to ensure that the projects on which the current project is dependent on is built before the current project.       | add_dependency(P3DDistributedCuttlefish DEPEND_INCS)                                                                                                                                                    |
|                  |                                                                                                                                                                                                            | The project P3DDistributedCuttlefish is dependent on the projects P3DInfrastructure, P3DUtility, P3DModuleInterface and P3DNetComm. add_dependency ensures that these projects are built before P3DDistributedCuttlefish is built. |
| add_subdirectory | 1. Adds the specified subdirectory in the solution.  
2. The name of the directory specifies the location of the source CMakeLists.txt and source code files.                            | add_subdirectory(P3DDistributedCuttlefish)                                                                                                                                                             |
|                  |                                                                                                                                                                                                            | The directive specifies that the project P3DDistributedCuttlefish should be added to the current solution generated and the CMakeLists.txt file required to include the project is present in the directory with same name as the project. |
| target_link_libraries | 1. Links the specified libraries to the named target.                                                                                           | $(DEPEND_LIBS ../zlib.lib;../msmpi64.lib) target_link_libraries(P3DNetComm DEPEND_LIBS)                                                                                                           |
|                  |                                                                                                                                                                                                            | The project P3DNetComm is included in the solution is built as a dynamic link library and the target dynamic linking library generated will be linked to the zlib library and msmpi lib 64 bit version. |
receive call and if in a buffer under run error case the MPI_STATUS value is MPI_ERROR which can be used for further investigation.

**Listing 5.8: Blocking MPI Receive**

1. `int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)`
2. `void MPI::Comm::Recv(void* buf, int count, const MPI::Datatype& datatype, int source, int tag, MPI::Status& status)`

The blocking send and receive work in tandem. For every send there needs to be corresponding receive matching in the following triples: source (rank), tag and communicator. MPI specification 2.0 provides the following properties for the point-to-point communication paradigm.

- **Order**: The message ordering is needed to maintain the requirement of matching send and receives. The messages are non-overtaking which makes the code for message exchanges deterministic in a single-threaded implementation.

- **Progress**: This guarantees completion of send and receive operations if there is matching pair of send and receive operation.

- **Fairness**: No guarantee of fairness in managing the communication is provided.

When there is no longer communication needed among the processes, the communicator can be closed. MPI library provides `MPI_FINALIZE` method to perform the same. Before the call to the function `MPI_FINALIZE` is made the processes in the communicator should complete all the initiated but pending communication as the `MPI_FINALIZE` call will shut down the MPI communication layer by shutting down all the network connections, release the held resources and clean the process space. To ensure that the pending send and receive calls are completed before the `MPI_FINALIZE` call, the processes need to synchronize by calling the `MPI_BARRIER`. The `MPI_BARRIER` synchronizes all the processes within the communicator and any process calling it is blocked until all the processes in the communicator have executed the statement. The syntax of `MPI_BARRIER` and `MPI_FINALIZE` can be seen in the Listing 5.9 and 5.10 respectively.

**Listing 5.9: Explicit Synchronization via Barrier**

1. `#include <mpi.h>`
2. `void MPI::Comm::Barrier() const = 0`

**Listing 5.10: Shutting down the MPI environment**

1. `#include <mpi.h>`
2. `void Finalize()`

### 5.5.2 Communication Abstraction Layer

It is also possible to use some other message passing communication mechanism other than MPI. The distributed system is connected through the enterprise network and messages could be exchanged through TCP/UDP sockets or REST protocol. To make it possible to have various flavors of communication mechanism easily implementable, an interface exposing the necessary methods for message passing is provided. This interface acts as an abstraction layer such that the underlying communication mechanism is determined at run-time and the appropriate implementation is invoked. The abstraction layer allows for cleaner replacement of different implementations of underlying communication mechanism. Developer writing an implementation for a new communication mechanism does not need to be necessarily aware of existing implementation or modify existing solution which allows for easy extension of different methodologies. The interface provided is called P3DNetworkComm which exposes the methods for initialization of the communication group, getter methods for information regarding the communication group members as well as the group size, role of each node (if any) in the group, send/ receive methods for different types of data and to shut down the set up. The class `P3DNetworkCommMPI` is the MPI specific implementation for the methods exposed by the interface. The user can select the communication mechanism by setting the value in the main configuration file for the tag ‘CommunicatorType’ to the name of the implementation. The Figure 5.2 enlists the methods of the interface.
5.5.3 How does MS-MPI work internally?

MPI, as mentioned in the earlier sections, provides the methods which allow for communication via message passing in a point-to-point communication or collective communication paradigm. MPI specifies a standard on what a message passing interface is supposed to do and not how it is done. It is the specific implementation which decides how it is done. Varying implementations can provide the same MPI methods using different underlying protocols. The overhead of the communication depends on the implementation, the underlying network and type of data communicated. MPI is message oriented protocol meaning that data is sent in distinct packets which are assembled at the receiver whereas a stream oriented protocol will have data sent byte-by-byte i.e. stream of bytes. The size of the data sent determines the way in which the message is sent or received. The underlying message protocol can be ‘eager’ or ‘rendezvous’ protocol

**Eager Message Passing Protocol:** The responsibility of storing the message is at the receiver’s end if no matching receive has been posted by the receiver. The sending process sends the message which is received by the receiving process if the matching receive has been already posted other wise the message is buffered by the receiving process. The sending process sends the message irrespective of whether the matching receive has been posted or not. It is allows for easy implementation of MPI_Send and better performance as the send call does not need to block to get an acknowledgment from the receive operation. The problem is that this solution is not scalable and may lead to memory exhaustion if there are large number of messages which need to be buffered at the receiver. Another problem is the extra copy of the buffered message is required at the receiver from the buffer allocated in the kernel space to the receiver's buffer i.e. the buffer allocated in the receiver process's address space. This also leads to investing CPU cycles for the extra copy to be made.

**Rendezvous Message Passing Protocol:** This protocol can be used on top of the eager protocol i.e. when the buffering limit has been reached. There is no underlying assumption made by the sending process that the receiver can allocate the memory for buffering the incoming message and hence it needs to first signal the receiver process about the incoming message. This signaling is called as ‘handshake’. The format of the message for MPI was introduced in the section 5.5.1. The Figure 5.3 shows the message format specified by MPI standard [29]. During the handshake the following actions are performed in the specified order:

1. The receiver process first receives the envelope of the message to be sent from the sender.
2. The received envelope is stored by the receiver process if there is no buffer space.
3. As the buffer space becomes available, receiver process sends an acknowledgment to the sender, indicating that it can receive the message.
4. The sender after receiving the acknowledgment from the receiver, sends the actual payload.
5. The receiver then receives the message.

The protocols fixes the buffer exhaustion issue by not overwhelming the receiver’s buffer by actual data rather the receiver needs to just store the envelope of the message which is relatively smaller in size. Therefore it makes the receiver process more robust as the receiver does not hang because of buffer overflow. As the messages are received after the acknowledgment by the sender, CPU cycles could be saved if the data can directly be copied to the buffer allocated in process’s address space. The problem with this protocol is extra messages exchanged during the handshake.
The receive event of a message can either be notified through an interrupt or the receiving process can poll at regular intervals to check if a message has arrived [30]. The interrupt technique works better when the send/receives are asynchronous i.e. they are non-blocking where as polling works well in case of synchronous send/receives i.e. they are blocking. In the polling mode, the receiver process busy waits using the CPU and queries the hardware frequently and manages the messaging by bypassing the OS where as in interrupt mode, the receiving process interacts with the OS which might decide to suspend the process while it is waiting for the message. As the time wasted in the context switch can be significant if the receive event is frequently done as well as OS involvement is generally costly, for better performance the polling method is used by most of the implementations of the MPI specification.

MPI can use TCP/IP as the underlying protocol for message passing in a cluster connected via Infiniband i.e. TCP/IP over Infiniband (IPoIB) or it can use lower level protocols like the OpenFabric protocol or Qlogic Performance Scaled Messaging Protocol or it could be as simple as TCP/IP over Ethernet. Microsoft's implementation of MPI-2 specification varies from others such that it uses mechanisms by which the use of TCP/IP can be bypassed. MS-MPI uses Microsoft WinSock Direct Protocol for compatibility over different kinds of Ethernet interconnect and for CPU efficiency. The WinSock direct protocol [31] uses remote direct memory access i.e. RDMA and thereby bypasses TCP/IP stack as it goes directly to the network interface card allowing for high-speed communication of data. When the TCP/IP stack is bypassed, the overhead occurred due to use of extra CPU cycles and memory bandwidth could be avoided. Winsock Direct Protocol reduces the library overhead and application distribution can scale better even over traditional interconnect networks. Figure 5.4 depicts how the WinSock protocol works [32].

### 5.6 Comparison of MPI to traditional socket programming

MPI is designed to allow communication amongst HPC (high performance cluster) and preferred choice of programmers over traditional socket programming [33]. The goal of programming applications to run on HPC is to perform huge and complex computation in a distributed manner. To achieve this goal, programs generally divide the huge computation into smaller parts and compute in parallel. The logic to perform splitting of tasks, accumulation of results, synchronization and communication if needed among the cluster nodes requires significant programming effort and leads to complicated code. So, it is justified that the developers use libraries which abstracts from the underlying complexities of communication but allow to write simple, clean and understandable code. MS-MPI is one such implementation of MPI-2 specification which not only provides decent performance but also lets developers not be bothered about managing the IP addresses, sockets/connection - opening and closing them, and other idiosyncrasies of the underlying network. Simple MPI SEND and MPI_RECV calls do the magic necessary for successful communication. MPI allows to use customized proprietary networks leading to high-throughput and low latency communication. These networks provide abstractions which might not be suitable for socket programming and in-turn lead to lower performance of the application when sockets are used.

The problem with MPI-2 specification based implementation is lack of robustness when the communication fails amongst the network. There is no standardization of error management and reporting which leads to vague error messages being shown in an event of failure. This is where socket programming might win against MPI as level of robustness provided is pretty standard. Another issue with MS-MPI is that the spawning of the processes and group management is done when the application is launched and dynamic scaling up/down of cluster resources cannot be done at run-time. The cases where the number of sub-tasks are lowers than the configured cluster size needs to be handled by the implementation. So in the implementation specific to the thesis under discussion, when the number of print object files in the main configuration is less than the number slave nodes in the cluster, the master needs to drop some of the slaves from the cluster and only wait on ‘working nodes’ for partial output.
5.7 External Libraries

Libraries which are developed by external sources are referred to as ‘External Libraries’ in the current documentation. The external libraries offer specific functionality which can be used by the applications. Generally the use of such libraries is encouraged as they allow the developer to use the already implemented solution and focus on the problem in-hand instead of reinventing the wheel. These external libraries generally are high-quality as they are mostly rigorously tested and used in different kinds of target applications. The algorithms implemented to perform the task by the libraries provide good efficiency at a low-cost. The probable problem of using an external library is that if there is a bug in the library then all the applications using the library duplicate the bug.

In the implementation of the thesis, the following external libraries are used:

1. **Little Color Management Engine (lcms) Version 2.0**: It is an open source small-footprint color management engine which uses ICC standard (International Color Consortium). It provides numerous APIs which can be used to access (parse, read, write) the ICC profiles, access (read/write) tags and profile structures, create formatters which describe the organization of the bitmap buffers and performing color transformation. In the implementation of prototype II, the master reads the ICC profile and sends the byte stream to the slave nodes. The master node uses `cmsOpenProfileFromFile`, `cmsSaveProfileToMem` APIs where as slave nodes use `cmsOpenProfileFromMem` API. The header ‘lcms2.h’ should be included to use the mentioned APIs. The syntax and the use of the APIs is described below:
• **cmsOpenProfileFromFile** - API used to open the profile from the file on the disk. The syntax of the API is shown in the Listing 5.11.

• **cmsSaveProfileToMem** - API used to save the profile to the memory block. The syntax of the API is shown in the Listing 5.12.

• **cmsOpenProfileFromMem** - API used to open the profile already in the stored in a memory block. The syntax of the API is shown in the Listing 5.13.

2. **RapidJSON** - It is an open-source JSON parser/generator for C++ supporting both SAX/DOM style API. It is not only fast, small and header-only but also is unicode and memory friendly. It provides APIs to parse, read, write/modify and query value from the JSON document. The APIs are used within the `rapid::json` namespace. The following RapidJSON library APIs are used:

• **IsObject** - Normally a type ‘Value’ stores the JSON value and ‘Document’ holds the root Value of the DOM tree. IsObject is an API to check if the root is a DOM object.

• **ParseStream** - API used to parse the JSON stream from the input.

• **HasMember and AddMember** - APIs used to check if the specified member is present in the object and to add the member to the object respectively.

• **Setters and Getters** - APIs used to set {SetString} and get a value {GetInt, GetUint, GetString, GetObject}.

• **Queries for value type**: IsBool, IsNull, IsNumber, IsString are the APIs which allow to check the type of the value.

3. **Zlib** - It is a general purpose compression library providing functions to perform compression and decompression of data. The functions provided use in-memory buffers which allows for compressing the whole data at once by using deflation compression methodology. The library also offers functions to perform integrity check on the compressed data and functions to read/write files with gzip format. The following APIs are used for compression/decompression in the implementation of prototype II where slices are sent and received as stream of bytes.

• **compress2** - Compresses the source buffer into the destination buffer. The syntax of the API is shown in the Listing 5.14. The last argument of the API is used to set the compression level which can be `Z_BEST_SPEED` for best speed, `Z_BEST_COMPRESSION` to get the smallest size of compression or `Z_DEFAULT_COMPRESSION` for default compression.

• **uncompress** - Decompresses the source buffer into the destination buffer. The syntax of the API is shown in the Listing 5.15. The original size of the data should be communicated to the site where the decompression is performed so that the buffer with appropriate size could be allocated.

---

**Listing 5.11**: Open profile from File
```c
1 cmsBool cmsOpenProfileFromFile(cmsPROFFILE hProfile, void *MemPtr, cmsUINT32Number* BytesNeeded);
```

**Listing 5.12**: Save profile to Memory
```c
1 cmsBool cmsSaveProfileToMem(cmsPROFFILE hProfile, void *MemPtr, cmsUINT32Number* BytesNeeded);
```

**Listing 5.13**: Open profile from Memory
```c
1 cmsPROFFILE cmsOpenProfileFromMem(const void *MemPtr, cmsUINT32Number dwSize);
```

**Listing 5.14**: API to compress the data
```c
1 int ZEXPORT compress2(Bytef *dest, uLongf *destLen, const Bytef *source, uLong sourceLen, int level);
```

**Listing 5.15**: API to uncompress the data
```c
1 int ZEXPORT uncompress(Bytef *dest, uLongf *destLen, const Bytef *source, uLong sourceLen);
```
5.8 Debugging

During implementation, there will be many occasions on which it would not be possible to understand the reason for the wrong output of the program by just observing the output or the log file. Hence, developers perform a multi-step process of identification and isolation of the problem followed by correcting the problem, which altogether is called as the debugging process. Microsoft Visual Studio 12 provides an inbuilt debugging tool which comes very handy to debug the application. As the application modified through the thesis runs on the cluster, a new challenge to be addressed was which and how should the instance of the application be debugged. There were two possible solutions- using a remote debugger or use a local debugger by changing the role/rank of the machine in the cluster.

Debugging a parallel application using remote debugger tool can be performed by searching for the process representing the instance of the application running on the node i.e. the remote machine and attaching the process to the visual studio process on the local machine. The debugging process done in this manner is quite slow and might be tricky to debug applications which crash at the very start giving the developer very small time interval to find and attach the process. The advantage is that debugging can be done on the local machine with the local visual studio instance running and hence installation of visual studio on the remote location is not required.

Debugging done by changing the rank of the local machine in the cluster is simple, faster and same as performing local debugging. Depending on the rank, a node acts as the master or a slave and so changing the role of the local machine helps to debug the application behavior for master or slave node. The rank of the local machine can be 0 i.e. master if it is the first value for the -hostname argument of the mpiexec command or > 0 i.e. slave node, if it is not the first value for the -hostname argument. As this method was faster, debugging of the distributed cuttlefish application during the implementation was done by using this method.

5.9 Logging

When the software is made of thousands of lines of code, debugging can be consuming procedure. It would be tricky to know where to put the debug point when there could possibly be many sources of the error. So generally, the execution trace file is examined which could give a pointer towards the possible location of error which could later be probed through debugging. The execution trace file content is application specific and generally contains at least the input given to each component and the output from the component in a component-based pipeline architecture. This helps to ensure that the input and output is correct for the component and in case of error helps to trace the possible source.

Generating log files which provide the execution trace is helpful not only during failures but also to analyze the performance of each component. The performance of the component could be analyzed by logging the time each component takes for a given task, for the current software, it is the time taken by the component per chunk or print object size. To analyze the performance of distributed cuttlefish, the time required to perform additional tasks should be measured which could help to get insights about the cost of ‘distributed computing’. The components introduced for distributed cuttlefish are measured as a whole i.e. the total time taken by the component and the number of times the component ran. Along with this, the cost of performing serialization/deserialization, compression/decompression as well as communication overhead is interesting information to log. The Table 5.7 summarizes the distributed computing components and the information logged for each component. For some components, the logging gets a bit tricky as the components are multi-threaded. If there is a single log generated for methods which are executed by multiple threads, then the log becomes a shared resource among the threads and might lead to race condition if the access to the log is not synchronized. To avoid such a race condition, instead per thread logs were generated for these methods.

5.10 Project Management

The various stages of the thesis involved steps similar to a software development life cycle (SDLC). We started with drawing up the problem statement to be addressed in the thesis, similar to system concept development step of SDLC, followed by planning which involved deciding the duration, milestones and goals to be achieved through the thesis. The outcome of the planning step was a project plan created using Microsoft Project tool containing detailed information about the project. The next step involved requirement analysis where the problem statement was broken down into precise requirements defining the expected behavior of the system. Through the analysis of the system behavior, a possible design of the solution was sketched and a task list to implement the design was created. The first basic design is referred throughout the thesis as the ‘Prototype I’. During the development and implementation of the prototype I, the shortcomings and possible improvements were taken note of. During the testing phase of the prototype I, some scenarios where the behavior of the system deviated from the expected system behavior were observed. A few bugs which lead to
### Table 5.7: Component-wise Information Logged

<table>
<thead>
<tr>
<th>Component</th>
<th>Logged Profiles</th>
<th>Purpose</th>
</tr>
</thead>
</table>
| MasterDistributor       | 1. Component Profile  
2. Communication overhead for sending the work-item to slaves |
| MasterDistributorPJ     | 1. Component Profile  
2. Serialization of print jobs  
3. Sending print job, texture data                                                  | 1. Component profile provides summed cost of print job distribution  
2. Cost of serialization of the print jobs  
3. Communication overhead for sending print jobs and texture information to slaves |
| SlaveCollectorPJ        | 1. Component Profile  
2. Receiving the print job and texture data  
3. De-serialization of print job, texture data                                     | 1. Component profile provides cost of collection of work item from the master  
2. Communication overhead in receiving the print job and texture information  
3. De-serialization cost for print jobs and texture data                           |
| SlaveReporter-Prototype I | 1. Component Profile  
2. Serialization of slices  
3. Sending serialized slice location                                                | 1. Component profile provides cost of sending partial output to the master.  
2. Cost of serialization of slices  
3. Communication overhead of sending partial slices                                 |
| SlaveReporter-Prototype II | 1. Component Profiles  
2. Serialization of slices  
3. Compression of serialized slices  
4. Sending compressed data                                                         | 1. Component profile provides cost of sending partial output to the master.  
2. Cost of serialization of slices  
3. Cost of performing compression.  
4. Communication overhead for sending partial output.                               |
| MasterMerger-Prototype I | 1. Component Profile  
2. Receiving metadata information  
3. Receiving the serialized slice path  
2. Cost of receiving metadata information.  
3. Communication overhead for receiving partial output.  
4. Cost of performing deserialization.                                              |
| MasterMerger-Prototype II | 1. Component Profile  
2. Receiving metadata information  
3. Receiving compressed stream of bytes  
4. Decompression of the received data  
2. Cost of receiving metadata information.  
3. Communication overhead for receiving partial output.  
4. Cost of decompression of data.  
5. Cost of performing deserialization.                                              |
the deviation could be fixed as they were mainly due to coding errors but there was one scenario which was encountered which could not be fixed as it was limitation by design for prototype I.

After the prototype I design shortcomings were brought to light, modification to the design of prototype I lead to a new design which helped to overcome the faced issues and is referred throughout the thesis as ‘Prototype II’. As prototype II was developed over the prototype I design, the implementation required lesser time in comparison to prototype I. The testing of prototype II lead to defining some corner cases and making modifications to the code to handle the corner cases. Throughout the different stages of thesis, a weekly meeting with the supervisors was conducted to provide an update on the progress of thesis as well as to seek guidance.

5.11 Summary

This Chapter details the various implementation specifics, the libraries used to perform parsing of JSON files, compression and decompression. It also gives an overview about the various APIs, interface structure and newly implemented components. It describes the MPI methods used as well as their behavior in different scenarios. It provides a brief comparison between different communication protocols which could used for communication between the nodes and justifies the use of MPI to certain extent. The implementation is tested and the results are presented in the Chapter 6.
6 Results and Analysis

After the implementation phase of the design discussed in the previous chapter, the next phase involved performing tests to check if the implementation worked. To recapitulate the problem to be solved through the thesis, to increase the performance of the cuttlefish application for multiple models by adding computational capacity and distributing the prints objects to utilize the added computational power. The first part of the goal was achieved by creating a distributed system of heterogeneous systems as compute nodes connected through the enterprise network. For utilizing the additional computational capacity, the application had to be modified to perform distribution of the work items amongst the compute nodes, collect their partial results and use the partial results to generate the final output. The desired result to be achieved through the thesis primarily was less computational time for multiple models as compared to the computational time needed by current cuttlefish application and increase resource utilization by making use of the available machines.

6.1 Overview of SDLC Testing Phase

In the software development life-cycle, testing phase follows the implementation phase wherein various tests are conducted to verify and validate the implemented software. The result of the testing phase depends highly upon the type of test cases which are run. Testing can be classified into various categories depending on criteria chosen for classification, for example: the methods chosen to conduct tests, levels of testing etc. On the basis of methods used for testing, test cases can be executed manually or by automating them. In manual testing, the tester uses the software as an end-user and makes note of the deviations (if any) seen in the behavior of the system as from the expected. In automated testing, another software or scripts are used to run the various test-cases for the software under consideration. Each of these broadly classified types of testing techniques can consists of unit testing, integration testing, system testing, performance testing, regression testing etc.

Manual testing as stated earlier is done by the tester by running the various use-cases in which an end user might use the system. Mostly, manual testing is done to verify the system’s behavior with respect to the specification which involves unit testing, integration testing. The validation of the implementation is done with respect to satisfying the needs of the customer which generally involves user acceptance tests and usability tests. Automated testing is generally done to perform regression testing where in part of the software under consideration is being fixed for some bug found by verification process, by re-writing a piece of code or introducing the fix by a new piece of code. The goal of regression tests is to ensure that the fixed bug is not leading to newer bugs in the system and thus is done quite frequently i.e. each time a discovered bug is fixed. Automated testing is also used to conduct stress test or performance tests where in the system is subjected to different loads and the behavior is observed with the primary goal to measure the performance or efficiency of the system under varying loads. As automated testing involves using other software or scripts to perform the tests, it is cheaper and efficient.

On the basis of level of testing, testing can be done to check the functional or non-functional requirements. Functional requirements cover what the software is intended to do i.e. for a given input, the systems performs the intended task and the results generated by the system are correct. Non-functional requirements from the system involve checking the performance of the system, the resource utilization etc. Functional testing can broken down further into discrete levels as follows:

- **Unit Testing**: Generally, it is performed by the developer who is aware about the code and the new modules which are implemented. The unit which is tested is generally doing a particular task / dedicated task and is a smaller portion of the whole code. The goal of the test is to ensure that each unit is performing the functionality it is implemented for. Unit tests generally have quite limited scope as it is not possible to cover all the execution paths of a code and the test data used by the developer is not the same as the test data to be used by the quality assurance team.

- **Integration Testing**: As the name suggests, integration testing is done to check if the individual modules tested via unit testing work correctly in unison. The unit testing of lower-level modules can be done first followed by integration tests by combining one or more modules which is called as bottom-up approach. If all the system modules together are tested i.e. the highest-level of module is tested first followed by testing each module individually later, it is called top-down approach.
• **System Testing**: System testing is generally done by a specialized quality assurance team where in the system is tested as a whole to check if the system meets the quality standards agreed upon. The system is tested in an environment which is very similar to the production environment i.e. the environment where the developed application is going to be deployed.

• **Acceptance Testing**: One of the most crucial steps of testing, the QA team verifies if the application works correctly for test scenarios which are discussed prior to the implementation by the customer and the development team. These test cases are drawn on the basis of the acceptance criteria for the software.

**Non-Functional testing** as mentioned earlier includes testing the non-functional requirements of the software such as the performance of the system, security, ease of use of the system etc. Non-Functional testing can be broken down into following levels:

• **Performance Testing**: Performance testing involves exposing the system under consideration to various levels of stress and loads to monitor the behavior of the system. The goal of this type of testing is to identify the performance issues or bottlenecks which could be caused by network delay, large data input size, hitting peak processing capacity etc.

• **Usability Testing**: This type of testing involves the checking the ease of use for the software where the possible improvements in the software are noted by observing the users use the software. It involves checking the efficiency of use, learn-ability, error messages seen by the user and satisfaction of the user.

• **Portability Testing**: If the software is built with multi-platform support, then this type of testing checks the functioning of the software on different platforms. The diversity in the platforms covers checking different operating systems, different underlying architecture or type of network.

• **Security Testing**: The intention of security testing is to find any vulnerabilities in the software which would compromise the security guarantees that should be provided by the software. The checks are done to ensure confidentiality, integrity, authentication, availability, authorization, non-repudiation, buffer over-flow vulnerabilities etc.

### 6.2 Implementation Assessment

Amongst the types of testing discussed in the Section 6.1, the implementation of the thesis was tested by using manual testing during the development and automated testing after the development was done. Manual testing involved running the scenarios to check individual components as a whole or parts of the components. Following the unit tests, were the tests which involved checking how the components worked together for different scenarios. Some of the manual tests performed are described in Section 6.2.1. The performance of the distributed cuttlefish is measured through automated testing. The performance tests and results of the tests are discussed in the Section 6.3.

#### 6.2.1 Manual Tests

Out of the various test cases which were conducted, the important cases have been documented in this Section which highlight the scenarios testing the newly introduced components for distributed computing.

1. Test-cases specific for Prototype I: The prototype I tests included testing the *MasterDistributor*, *SlaveReporter* and *MasterMerger* components.
   
   • **Test-Case for MasterDistributor Component:**
     
     – **Scenario**: To test if the *MasterDistributor* component generates correct distribution for a given set of input objects and cluster size. The goal is to test the correct calculation of the threshold function, creation of distributed print object list and using the generated list correctly to write the main configuration file for each slave node.
     
     – **Method**: Distributed cuttlefish was executed multiple times with varying inputs. The test data for the simplest scenario consisted of 2 head models to be distributed among 3 node cluster with 2 slaves and one master node. Each head was 8cms and the expected balanced distribution of the input would to be generate load of one head model per slave node. The same test case was run repeatedly with varying the number of input objects, different types of input models i.e. varying the test-data and cluster size.
Results - As in prototype I, the MasterDistributor component generated main configuration file per slave and the output of the current test was verified by checking the number of print objects in the main configuration file for each slave node. Another way to check the threshold calculation was to use simple standard output statements or logging the information in a trace file.

• Test-Case for SlaveReporter Component:
  - Scenario - The SlaveReporter Component generates the output with partial slices, which are written to the disk after serialization and chunk-wise sending the path to the master. To test if the SlaveReporter component generates correct partial output and the path sent to the master is the correct location.
  - Method - Distributed cuttlefish was executed on the slave node. To check if the serialization of the partial slices happens correctly, the slices are deserialized at the slave and the partial output is written to the disk as images. The part of code implementing the test i.e. deserialization of the slices and generating images of the partial slices is not part of the final code which is used to build the distributed cuttlefish binary but is rather just code to test the scenario.
  - Results - The images of the partial slices generated by the SlaveReporter component were thoroughly checked to ensure correct material assignment. Ensuring that the partial slices are correctly serialized helped to narrow down the errors which were encountered in generation of the merged slices.

• Test-Case for MasterMerger Component:
  - Scenario - The MasterMerger Component generates the final slices by merging the partial slices, which are written to the disk in a chunk-wise fashion after de-serialization. To test if the MasterMerger component generates correct merged output.
  - Method - Distributed cuttlefish was executed in the cluster to check if the deserialization of the partial slices happens correctly at the master, the deserialized partial slices are written to the disk as images before writing to the merged full slices. The part of code implementing the test i.e. deserialization of the slices and generating images of the partial slices is not part of the final code which is used to build the distributed cuttlefish binary but is rather just code to test the scenario.
  - Results - The images of the partial slices generated by the MasterMerger component were thoroughly checked to ensure correct material assignment. The final output of the MasterMerger component i.e. the merged full slices were verified by checking the output of the SliceDebugger component output. Through this test a bug in the code was detected wherein the empty full slices were not initialized with EMPTY_VOXEL value during the creation i.e. no material assigned these voxels, which lead to garbage value assignment to the co-ordinates whose value was not overwritten during merging. This introduced random patches in the full slices which lead to incorrect results. This bug was fixed by initializing the full empty slices during the creation of the slices.

2. Test-cases specific for Prototype II: The testing of prototype II involved testing the components - MasterPJDistributor, SlavePJCollector, SlaveReporter and MasterMerger.

• Test-Case for MasterPJDistributor Component:
  - Scenario - To test if the MasterPJDistributor component generates correct distribution for a given set of input objects and cluster size. The goal is to test the correct calculation of the threshold function, creation of distributed print object list and using the generated list correctly to generate the print job per slave node.
  - Method - Distributed cuttlefish was executed multiple times with varying inputs. The test data for the simplest scenario consisted of 2 head models to be distributed among 3 node cluster with 2 slaves and one master node. Each head was 8cms and the expected balanced distribution of the input would to be generate load of one head model per slave node. The same test case was run repeatedly with varying the number of input objects, different types of input models i.e. varying the test-data and cluster size.
  - Results - As in prototype II, the MasterPJDistributor component generated print job per slave and the output of the current test was verified by debugging the number of print objects in the print job for each slave node. Another way to check the threshold calculation was to use simple standard output statements or logging the information in a trace file. Through this test, a bug in the code which led to incorrect threshold calculation was caught. The incorrect threshold calculation lead to incorrect number of print objects in the print jobs leading to one of the computes nodes being overloaded. Another bug related to placement of the print objects in the print bed was discovered as well, where in the print jobs sent to the slaves had print objects with the transformation created by the PrintJobOrganizer at the
master. This lead to generation of partial slices with the print objects placed with some displacement from the origin at the slave. This bug was later fix by executing the PrintJobOrganizer component at the slave node.

**Test-Case for SlavePJCollector Component:**

- **Scenario** - To test if the SlavePJCollector component receives correct input from the master. The goal is to test if the SlavePJCollector component correctly receives the serialized print job, deserializes the print job and loads the texture information to the GPU memory after receiving it from the master and deserializing the received texture data.

- **Method** - Distributed cuttlefish was executed multiple times with varying number of input models at the master.

- **Results** - In the prototype II, the SlavePJCollector component receives the serialized print job from the master. The size of the serialized data was cross-examined with the size of data sent at the master node. After deserializing the print job, the number of print objects contained in the print job was checked through debugging the component. To check if the texture data was received and deserialized correctly, the deserialized data was written to an image on the disk. Through this test, a bug in the format in which the texture data was uploaded to the GPU memory at the slave node was found. The texture data is in RGBA format and is linearized at the master node before uploading the data to master node GPU. When the texture data is communicated to the slave, the load operation should not perform the linearization again and instead load the data directly to the GPU memory.

3. **Test-Cases Common for Prototype I and Prototype II:** There are some corner cases which had to be checked for both the prototypes. These scenarios and the tests performed to check how the system handles these scenarios are described as follows:

- **Corner Case I** - Size of input objects < number of slave nodes

  - **Scenario** - To test the behavior of distributed cuttlefish when the number of input objects submitted to the master node is smaller than the cluster size i.e. number of print objects in the main configuration file are less than the number of slave nodes.

  - **Method** - Distributed cuttlefish was executed with the main configuration file consisting of less number print objects in comparison to the number of nodes given to mpiexec command, i.e. cluster size is number of print objects plus 2 (as master node does not act as a slave node).

  - **Results** - When the number of print objects is less than the number of available slave nodes, some of the slave nodes get empty input. At the master, the MasterMerger component is notifies the number of slaves nodes doing the actual work to the MasterMerger component so as to ensure that the MasterMerger component waits to receive the partial slices only from the specified number of slave nodes. For prototype I, the empty work item to the ‘dropped’ slaves is given by writing a main configuration file without any print object files. Through this test case, it was discovered that the cuttlefish application did not handle empty input case and crashed. So to handle the empty input, there were some modifications introduced in the application. For Prototype II, the empty input is given the ‘dropped’ slaves by sending a print job without any print objects. The ‘dropped’ slaves do execute the whole pipeline once with null input and wait for the other slaves doing the work to finish while they execute MPI_Barrier call.

- **Corner Case II** - Single input object

  - **Scenario** - To test the behavior of distributed cuttlefish when only one print object is submitted to the master node, i.e. the main configuration file contains only one print object file.

  - **Method** - Distributed cuttlefish was executed with the main configuration file consisting of 1 print object.

  - **Results** - Irrespective of the number of slave nodes in the cluster, as there is only one print object provided as input to the distributed cuttlefish, there is no need to calculate the threshold to perform the distribution. So, the slave node with rank 1 i.e. the first slave node in the cluster gets the print object and rest of the slave nodes (if any depending on the number of hosts set in the mpiexec command) are ‘dropped’ and the behavior of the system is similar to the behavior discussed in the above test-case.

- **Corner Case III** - No input given to the distributed cuttlefish

  - **Scenario** - To test the behavior of distributed cuttlefish when input of size is 0.

  - **Method** - Distributed cuttlefish was executed with the main configuration file consisting of 0 print objects.
- **Results**: When the input size is 0, for prototype II, the MasterPJDistributor component gets an empty print job from the PrintJobOrganizer component. The MasterPJDistributor component informs the MasterMerger component that all the slave nodes are dropped and it should not wait for any partial output from the slave nodes. For prototype I, all the slave nodes receive empty configuration files and the prototype II.

### 6.3 Results

The performance testing is done through automation of the test-cases. For testing the distributed cuttlefish, batch scripts were written and executed. For each test case, the test-data was created along with the script and the task was scheduled using the windows task scheduler. The performance tests were conducted to check the computation speed-up and efficiency achieved by distributed computing. The Section 6.3.1 explains the terms computational speed-up and efficiency in parallel computing followed by the test case used to find the speed-up and observed result. The Section 6.3.2 details the comparison of the two prototypes developed in terms of computation speed-up, communication overhead and how well the prototypes scale for a given work load.

#### 6.3.1 Comparison of distributed vs non-distributed cuttlefish performance

Speed-up is one of the most common metric used to measure performance of a parallel processing. *Absolute speed-up* calculation can be done by considering how fast the computation can be performed by using N processors in comparison to best sequential algorithm performance on a single processor [34]. *Relative speed-up* is defined as the ratio of performance of parallel algorithm on one processor to the performance of same on N processors. Relative speed-up is used to compare how well the algorithm performs when the number of processors is scaled up [34]. To do performance evaluation for distributed cuttlefish, we use the absolute speed-up definition and Equation 6.1 to calculate the speed-up, where N is number of processors, \( T_S \) - is the execution time of non-distributed cuttlefish for the given work load, \( T_N \) - is the execution time of distributed cuttlefish for the same work load.

\[
S_N = \frac{T_S}{T_N}
\]  

(6.1)

Using the Equation 6.1, the speed-up can be classified as either linear speed-up, sub-linear speed-up or super-linear speed-up. Linear speed-up is observed when \( S_N = N \) i.e. speed-up is equal to the number of processors. Sub-linear speed-up is when the speed-up is lesser than number of processors whereas super-linear speed up is when the speed-up gained is larger than the number of processors. Some of the reasons why super-linear speed-up is possible are higher cache hit rate, better resource utilization, lower network latency for small-sized messages communication. The Figure 6.1 summarizes the three types of speed-up.

Efficiency is defined as a ratio of speed-up gained for \( N \) processors to the number of processors. The Equation 6.2 is used to calculate the efficiency.

\[
E_N = \frac{S_N}{N}
\]  

(6.2)

---

**Distributed Cuttlefish Prototype I vs Non-Distributed Cuttlefish**

To check the speed-up gained by Distributed Cuttlefish Prototype I, the application was run with different work-loads and comparison between the execution times for both Distributed Cuttlefish Prototype I and Non-Distributed Cuttlefish application was drawn.

1. **Scenario**: To test the computational speed-up gained by distributed cuttlefish prototype I in comparison to non-distributed cuttlefish for input consisting of same models.

   • **Test-Case and Test-Data Description**: The cluster was composed of 3 nodes where in one node was the master and remaining two were compute nodes. The cluster nodes were heterogeneous in terms of the number of cores, CPU speed and RAM capacity. The test-data included multiple head models of size 8cms. The number of head models was increased by 2 for each following test-case, i.e. starting with 2 head models the number was increased to 10 head models for the 5 test-cases, with each test-case executed 5 times to enable collection of the average execution times. The speed-up for each test case was calculated using the average execution time for distributed cuttlefish and non-distributed cuttlefish for a given test-data i.e. given work-load. The non-distributed cuttlefish application was executed on the slowest node of the cluster as the slowest node in the cluster would impact the execution time of the distributed version.
Figure 6.1: Speed-Up Types

- **Results**: To elaborate on the amount of work to be done, each 8 cm head model consists of 777 slices with each slice having the width-946 voxels along X-axis (8.00 cm) and height-303 voxels along Y-axis (5.12 cm). The total number of voxels for each model is calculated using the Equation 6.3 where, \( T_{\text{NumVoxels}} \) is the total number of voxels for a given model, width (W), height (H) and total number of slices - NumSlices. For each head model used in the test data, the total number of voxels to be processed is \( 2.2 \times 10^8 \). Each voxel can be assigned values denoting the materials and a null material value depending upon the printer specification provided. For current test case, the materials provided in the printer specification are VeroCyan, VeroWhite, VeroMagenta, VeroYellow, VeroBlack and EMPTY_VOXEL as null material. For each material there is a bitmap generated denoting the voxels which are assigned that particular material in the slice. Figure 6.2 shows the material assignment generated for 108th slice of a single head model. The Table 6.1 summarizes the execution times for distributed cuttlefish with prototype I and non-distributed cuttlefish along with the speed-up.

- **Observation**: The number of slave nodes is constant throughout the test-cases i.e. 2 slave nodes and the work-load provided to the system is increased for every test-case. The speed-up is calculated as per the Equation 6.1 and it increases with the increase in work-load in comparison to the non-distributed cuttlefish performance. The red line in Figure 6.3 depicts the scaling of the speed-up with increase in work load i.e number of models. This is proved by calculating the efficiency using the Equation 6.2 and seen in the Figure 6.4.

2. **Scenario**: To test the computational speed-up gained by distributed cuttlefish prototype I in comparison to non-distributed cuttlefish for input consisting of different models.

- **Test-Case and Test-Data Description**: The test case was executed 5 times for each test-data. The cluster was composed of 3 nodes where in one node was the master and remaining two were compute nodes. The cluster nodes were heterogeneous in terms of the number of cores, CPU speed and RAM capacity. The test-data included multiple head and dame models of size 8 cm. The number of models was increased by 2 for each following test-case, i.e. starting with 1 head model and 1 dame model, the number was increased to 5 head and dame models each, for the 5 test-cases, with each test-case executed 5 times to enable collection of the average execution times. The speed-up for each test case was calculated using the average execution time for distributed cuttlefish and non-distributed cuttlefish for a given test-data i.e. given work-load. The
Table 6.1: Speed-Up and Efficiency: Distributed Cuttlefish Prototype I vs Non-Distributed Cuttlefish For Same Models

<table>
<thead>
<tr>
<th>Number of Head Models 8cms (Resolution: 300 X 150 X 470)</th>
<th>Total Execution Time (in Secs) for Distributed Cuttlefish Prototype I (Cluster Size- 3)</th>
<th>Total Execution Time (in Secs) for Non-Distributed Cuttlefish</th>
<th>Speed-Up</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>173.35</td>
<td>357.35</td>
<td>2.06</td>
<td>0.68</td>
</tr>
<tr>
<td>4</td>
<td>363.39</td>
<td>920.56</td>
<td>2.33</td>
<td>0.84</td>
</tr>
<tr>
<td>6</td>
<td>585.32</td>
<td>1651.17</td>
<td>2.82</td>
<td>0.94</td>
</tr>
<tr>
<td>8</td>
<td>836.23</td>
<td>4071.57</td>
<td>4.86</td>
<td>1.62</td>
</tr>
<tr>
<td>10</td>
<td>1130.51</td>
<td>4790.19</td>
<td>4.23</td>
<td>1.41</td>
</tr>
</tbody>
</table>

Table 6.2: Speed-Up and Efficiency: Distributed Cuttlefish Prototype I vs Non-Distributed Cuttlefish For Different Models

<table>
<thead>
<tr>
<th>Number of Head &amp; Dame Models - 8 cm (Resolution: 300 X 150 X 470)</th>
<th>Total Execution Time (in secs) for Distributed Cuttlefish Prototype I (Cluster Size- 3)</th>
<th>Total Execution Time (in secs) for Non-Distributed Cuttlefish</th>
<th>Speed-Up</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>184.25</td>
<td>243.94</td>
<td>1.32</td>
<td>0.44</td>
</tr>
<tr>
<td>4</td>
<td>315.34</td>
<td>635.52</td>
<td>2.01</td>
<td>0.67</td>
</tr>
<tr>
<td>6</td>
<td>563.3</td>
<td>1171.32</td>
<td>2.07</td>
<td>0.69</td>
</tr>
<tr>
<td>8</td>
<td>685.21</td>
<td>1890.70</td>
<td>2.75</td>
<td>0.91</td>
</tr>
<tr>
<td>10</td>
<td>988.70</td>
<td>4793.47</td>
<td>4.84</td>
<td>1.61</td>
</tr>
</tbody>
</table>

The non-distributed cuttlefish application was executed on the slowest node of the cluster as the slowest node in the cluster would impact the execution time of the distributed version.

**Results** - To elaborate on the amount of work to be done, each 8cms head model consists of 777 slices, with each slice having the width-946 voxels along X-axis (8.00 cm) and height-303 voxels along Y-axis (5.12 cm). The total number of voxels for each model is calculated using the following Equation \( T_{\text{NumVoxels}} = W \times H \times \text{NumSlices} \) where, \( T_{\text{NumVoxels}} \) is the total number of voxels for a given model, width (W), height (H) and total number of slices - NumSlices. For each head model used in the test data, the total number of voxels to be processed is \( 2.2 \times 10^8 \). Each 8cm dame model consists of 605 slices, with each slice having the width-946 voxels along X-axis (8.00 cm) and height-194 voxels along Y-axis (3.27035 cm). The total number of voxels for the dame model calculated using the Equation 6.3 is \( 1.1 \times 10^8 \). Each voxel can be assigned values denoting the materials and a null material value depending upon the printer specification provided. For current test case, the materials provided in the printer specification are VeroCyan, VeroWhite, VeroMagenta, VeroYellow, VeroBlack and EMPTY_VOXEL as null material. For each material there is a bitmap generated denoting the voxels which are assigned that particular material in the slice. The Table 6.2 summarizes the execution times for distributed cuttlefish with prototype I and non-distributed cuttlefish along with the speed-up for input consisting of different models.

**Observation** - The number of slave nodes is constant throughout the test-cases i.e. 2 slave nodes and the work-load provided to the system is increased for every test-case. The speed-up is calculated as per the Equation 6.1 and increases with the increase in work-load in comparison to the non-distributed cuttlefish performance. The red line in the Figure 6.5, depicts that Prototype I speed-up increases with increase in work load i.e number of models. This is proved by calculating the efficiency using the Equation 6.2 and plot seen in the Figure 6.6.

\[ T_{\text{NumVoxels}} = W \times H \times \text{NumSlices} \]  

**Distributed Cuttlefish Prototype II vs Non-Distributed Cuttlefish**

To check the speed-up gained by using the Distributed Cuttlefish Prototype II, the application was run with different work-loads and comparison between the execution times for both Distributed Cuttlefish Prototype I and Non-Distributed Cuttlefish application was drawn.
Table 6.3: Speed-Up and Efficiency: Distributed Cuttlefish Prototype II vs Non-Distributed Cuttlefish for Same Models

<table>
<thead>
<tr>
<th>Number of Head -8 cm (Resolution: 300 X 150 X 470)</th>
<th>Total Execution Time (in secs) for Distributed Cuttlefish Prototype II (Cluster Size- 3)</th>
<th>Total Execution Time (in secs) for Non-Distributed Cuttlefish</th>
<th>Speed-Up</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>102.55</td>
<td>357.35</td>
<td>3.48</td>
<td>1.16</td>
</tr>
<tr>
<td>4</td>
<td>245.08</td>
<td>920.56</td>
<td>3.75</td>
<td>1.25</td>
</tr>
<tr>
<td>6</td>
<td>440.41</td>
<td>1651.17</td>
<td>3.74</td>
<td>1.24</td>
</tr>
<tr>
<td>8</td>
<td>708.70</td>
<td>4071.56</td>
<td>5.74</td>
<td>1.91</td>
</tr>
<tr>
<td>10</td>
<td>1014.98</td>
<td>4790.19</td>
<td>4.71</td>
<td>1.57</td>
</tr>
</tbody>
</table>

1. **Scenario**: To test the computational speed-up gained by distributed cuttlefish prototype II in comparison to non-distributed cuttlefish for input consisting of same models.
   - **Test-Case and Test-Data Description**: The test-case and the test-data used here is exactly same as the test-case and test-data of Scenario 1 of Prototype I.
   - **Results**: The amount of work done by the nodes remains same as described in Scenario 1 of Prototype I testing. The Table 6.3 summarizes the execution time for the test case for prototype II.
   - **Observation**: The speed-up for the prototype II was calculated using the Equation 6.1 and efficiency was calculated using the Equation 6.2. The blue line in the Figure 6.3 and Figure 6.4 depict the trend seen for the speed-up and efficiency for prototype II. As the number of models increases the speed-up and efficiency increases for a given cluster size until a certain point (i.e. until the number of models reaches 8 models, at which there is peak performance seen) after which with increase in number of models the speed-up decreases and so does the efficiency.

2. **Scenario**: To test the computational speed-up gained by distributed cuttlefish prototype II in comparison to non-distributed cuttlefish for input consisting of different models.
   - **Test-Case and Test-Data Description**: The test-case and the test-data used here is exactly same as the test-case and test-data of Scenario 2 of Prototype I.
   - **Results**: The amount of work done by the nodes remains same as described in Scenario 2 of Prototype I testing. The Table 6.4 summarizes the execution time for the test case for prototype II.
   - **Observation**: The speed-up for the prototype II was calculated using the Equation 6.1 and efficiency was calculated using the Equation 6.2. The blue line in the Figure 6.5 and Figure 6.6 depict the trend seen for the speed-up and efficiency for prototype II for different models as input set. As the number of models increases the speed-up and efficiency increases.
Figure 6.3: Speed-up Vs Number of Models For Same Models Input

Figure 6.4: Efficiency Vs Number of Models For Same Models Input
Figure 6.5: Speed-up Vs Number of Models For Different Models Input

Figure 6.6: Efficiency Vs Number of Models For Different Models Input
Table 6.4: Speed-Up and Efficiency: Distributed Cuttlefish Prototype II vs Non-Distributed Cuttlefish for Different Models

<table>
<thead>
<tr>
<th>Number of Head &amp; Dame, Models - 8 cm (Resolution: 300 X 150 X 470)</th>
<th>Total Execution Time (in secs) for Distributed Cuttlefish Prototype II (Cluster Size- 3)</th>
<th>Total Execution Time (in secs) for Non-Distributed Cuttlefish</th>
<th>Speed-up</th>
<th>Efficiency-ProtoII</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>94.52</td>
<td>243.94</td>
<td>2.582</td>
<td>0.86</td>
</tr>
<tr>
<td>4</td>
<td>181.95</td>
<td>635.52</td>
<td>3.50</td>
<td>1.16</td>
</tr>
<tr>
<td>6</td>
<td>357.61</td>
<td>1171.32</td>
<td>3.27</td>
<td>1.09</td>
</tr>
<tr>
<td>8</td>
<td>492.65</td>
<td>1890.70</td>
<td>3.83</td>
<td>1.27</td>
</tr>
<tr>
<td>10</td>
<td>792.68</td>
<td>4793.47</td>
<td>6.04</td>
<td>2.01</td>
</tr>
</tbody>
</table>

Performance Analysis

Distributed cuttlefish has super-linear speed-up in most of the tested cases in comparison to non-distributed cuttlefish. One of the most fundamental reasons is availability of more resources in terms of computational capacity and memory. The time required to perform same the task in distributed fashion is lesser than the non-distributed cuttlefish because the sub-tasks are computed in parallel allowing more work to be done in less time. The overhead of distribution of sub-tasks and collection of the partial results is negligible in comparison to the time to taken perform the same task on single machine with non-distributed cuttlefish. The division of one single task to sub-tasks decreases the amount of work-load i.e. less number of print objects per compute node. Lesser number of print objects per compute node leads to smaller amount of data processing per node which helps to accelerate the performance due higher cache hit rates with smaller data. Smaller amount of data can be kept in main memory throughout the computation without having a significant impact of paging on the performance. With non-distributed cuttlefish, it has been observed that running the application with single models multiple times leads to better performance than running the application once with multiple models perComm thus supporting the hypothesis of smaller data leading to higher cache hits resulting in better performance.

The implementation of the components handling the distributed computing specific tasks has been done in such a way that the processing of these tasks happens in parallel with the main executing thread and the worker threads block only when they await for some input. At the slave node, the SlaveReporter component has two threads of execution wherein one thread is the main thread which performs chunk-wise processing of the input i.e. runs the pipeline components as the streaming architecture and the worker thread which is dedicated to perform the distributed computing specific tasks such as serialization, communication of data and meta-data and additionally compression in case of Prototype II. The worker thread blocks only when the buffer of holding the partial slices is empty. The design of SlaveReporter component is discussed in detail in the Section 4.1.2. This design allows for minimum blocking time between the main thread and worker thread leading parallel computation and masks the latency that might be induced due to the communication between master and slave nodes. At the master node, the MasterMerger component main thread creates a worker thread per slave which interacts with the worker thread from the slave node. This allows for communication between master node and slave nodes in parallel without the slaves blocking for one another. Also at the master nodes, the main thread can proceed with rest of the pipeline components when the container is ready with a minimum number of merged slices. The design of the MasterMerger component is discussed in detail in the Section 4.1.3. Due to this design, the overhead related to communication, deserialization, decompression is comparatively low.

6.3.2 Comparison of Distributed Cuttlefish Prototypes

After comparing the performance of distributed cuttlefish vs non-distributed cuttlefish, to understand the performance of the prototypes in comparison with each other some tests were done. The aim of the tests was to understand how well the prototypes scale by varying cluster size for a given workload. The underlying assumption is that for a given workload, if the number of slave nodes in the cluster increases, the speed-up should also increase thus increasing the efficiency of the prototype.

1. Test-Scenario: To understand how well the prototypes perform with increase in the cluster size, the distributed cuttlefish prototype I and prototype II were executed with constant load by adding a slave node for each test. There were 5 runs per test with the distributed system having 3, 4, 5, 6 cluster size respectively.
Figure 6.7: Speed-up of Prototype I and II Vs Cluster Size For Same Models Input Set

Figure 6.8: Percentage increase in Efficiency of Prototype II Vs Cluster Size For Same Models Input Set
Figure 6.9: Speed-up of Prototype I and II Vs Cluster Size For Different Models Input Set

Figure 6.10: Percentage increase in Efficiency of Prototype II Vs Cluster Size For Different Models Input Set
Figure 6.11: Distributed Computing Cost for Prototype I vs Model Size

Figure 6.12: Distributed Computing Cost for Prototype II vs Model Size
Table 6.5: Prototype I vs Prototype II Measurements for 10 Head Models

<table>
<thead>
<tr>
<th>Cluster Size</th>
<th>Total Execution Time for Prototype I</th>
<th>Total Execution Time for Prototype II</th>
<th>Total Execution Time by Non-Distributed Cuttlefish</th>
<th>Prototype I Speed-up</th>
<th>Prototype II Speed-up</th>
<th>Efficiency of Prototype I</th>
<th>Efficiency of Prototype II</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1130.51</td>
<td>1014.98</td>
<td>4790.19</td>
<td>4.23</td>
<td>4.71</td>
<td>1.41</td>
<td>1.57</td>
</tr>
<tr>
<td>4</td>
<td>917.45</td>
<td>702.24</td>
<td>4790.19</td>
<td>5.22</td>
<td>6.82</td>
<td>1.30</td>
<td>1.70</td>
</tr>
<tr>
<td>5</td>
<td>675.97</td>
<td>488.21</td>
<td>4790.19</td>
<td>7.08</td>
<td>9.81</td>
<td>1.41</td>
<td>1.96</td>
</tr>
<tr>
<td>6</td>
<td>555.87</td>
<td>416.29</td>
<td>4790.19</td>
<td>8.61</td>
<td>11.50</td>
<td>1.43</td>
<td>1.91</td>
</tr>
</tbody>
</table>

- **Tests-Data**: The whole test-case was executed with two input sets, one input set consisted of 10 head models and the second set consisted of the 5 head and 5 dame models. Each model was 8cm and resolution used was $300 \times 150 \times 470$.

- **Results**: The Table 6.5 and Table 6.6 summarizes the total execution time for both the prototypes for varying cluster size, speed-up gained by the prototypes in comparison to the non-distributed cuttlefish execution and efficiency of the prototypes for varying cluster size for same and different models input set. The Figure 6.7 depicts the speed-up gained by each of the prototypes with increase in cluster size for a given workload for same model input and Figure 6.8 shows that increase in the efficiency of the prototype II in comparison to efficiency of prototype I. The Figure 6.9 depicts the speed-up gained by each the prototypes with increase in cluster size for a given workload for different models input and and Figure 6.10 shows the increase in the efficiency of the prototype II in comparison to efficiency of prototype I.

- **Observation**: The difference between the speed-up gained by Prototype II and I increases with the increase in the cluster size, showing that Prototype II scales well with increase in cluster size for given workload in comparison to the Prototype I. The percentage increase in efficiency in the Prototype II increases with cluster size even though not at a constant rate.

2. **Test-Scenario**: To understand how well the prototypes perform with increase in the size of the models, the distributed cuttlefish prototype I and prototype II were executed with constant number but varying size of models and cluster size.

- **Tests-Data**: There were 5 runs per test with 5 head models with size 4cm, 8cm, 12cm, and 16cm executed in a distributed systems of 5 nodes and $300 \times 150 \times 470$ resolution.

- **Results**: The Table 6.7 summarizes the total execution time, time taken to perform serialization by the slowest slave node, time taken by the master node slowest worker thread to perform deserialization and time taken to receive partial output for Prototype I. The Table 6.8 summarizes the total execution time, time taken to perform serialization and compression by the slowest slave node, time taken by the master node slowest worker thread to perform deserialization, decompression and time taken to receive partial output for Prototype II.

- **Observation**: The Figure 6.13 shows that the execution time for both the prototypes increases with increase in model size. The model size determines the number of slices and size of the slices, therefore increase in model size leads to more workload per slave node in the cluster. As previously discussed Prototype II performs better than Prototype I. The cost of distributed computing is the sum of time periods spent in performing actions specific to distributed computing i.e. communication amongst the cluster nodes, serialization and deserialization of slices, and compression and decompression of slices (specific to Prototype II). In Prototype I, Figure 6.11, the time spent in partial output communication is lesser than Prototype I, Figure 6.12, where as the serialization cost is higher in Prototype I as the serialized data is written to the network file system i.e. disk I/O.

### 6.3.3 Distributed Cuttlefish Prototype II Performance Evaluation

While implementing the distributed cuttlefish prototype II, two designs were possible for distribution of print jobs and merging of the partial slices. In each of the designs, the component configuration executed at the slave node is different. This difference in the component configuration of the pipeline executed on the slave node resulted in varying performance. As there was a significant difference in the performance, this Section is dedicated to explain how the decision of the component configuration impacts the overall performance of the application. It helped to gain insights about the
### Table 6.6: Prototype I vs Prototype II Measurements for Different Models Input

<table>
<thead>
<tr>
<th>Cluster Size</th>
<th>Execution Time for Prototype I</th>
<th>Execution Time for Prototype II</th>
<th>Execution Time for Non-Distributed Cuttlefish</th>
<th>Speed-up gained by Prototype I</th>
<th>Speed-up gained by Prototype II</th>
<th>Efficiency of Prototype I</th>
<th>Efficiency of Prototype II</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>988.708</td>
<td>792.68</td>
<td>4793.47</td>
<td>4.84</td>
<td>6.04</td>
<td>1.61</td>
<td>2.01</td>
</tr>
<tr>
<td>4</td>
<td>735.64</td>
<td>514.93</td>
<td>4793.47</td>
<td>6.51</td>
<td>9.30</td>
<td>1.62</td>
<td>2.32</td>
</tr>
<tr>
<td>5</td>
<td>678.26</td>
<td>465.14</td>
<td>4793.47</td>
<td>7.06</td>
<td>10.30</td>
<td>1.41</td>
<td>2.061</td>
</tr>
<tr>
<td>6</td>
<td>513.14</td>
<td>316.07</td>
<td>4793.47</td>
<td>9.34</td>
<td>15.16</td>
<td>1.55</td>
<td>2.52</td>
</tr>
</tbody>
</table>

### Table 6.7: Prototype I Performance For Varying Model Size

<table>
<thead>
<tr>
<th>Model Size</th>
<th>Total Execution time (in secs) for Prototype I</th>
<th>Time taken for Serialization (in secs)</th>
<th>Time taken for Receiving Partial Output (in secs)</th>
<th>Time taken for Deserialization (in secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>83.64</td>
<td>44.25</td>
<td>63.38</td>
<td>1.76</td>
</tr>
<tr>
<td>8</td>
<td>390.95</td>
<td>327.45</td>
<td>118.12</td>
<td>6.68</td>
</tr>
<tr>
<td>12</td>
<td>1241.23</td>
<td>1117.58</td>
<td>935.93</td>
<td>17.58</td>
</tr>
<tr>
<td>16</td>
<td>3373.74</td>
<td>2950.68</td>
<td>1211.64</td>
<td>2000.62</td>
</tr>
</tbody>
</table>

### Table 6.8: Prototype II Performance For Varying Model Size

<table>
<thead>
<tr>
<th>Model Size</th>
<th>Total Execution time (in secs) for Prototype II</th>
<th>Time (in secs) taken to perform serialization in Prototype II</th>
<th>Time (in secs) taken to receive partial output in Prototype II</th>
<th>Time (in secs) taken to perform deserialization in Prototype II</th>
<th>Time (in secs) taken to perform compression in Prototype II</th>
<th>Time (in secs) taken to perform decompression in Prototype II</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>69.17</td>
<td>0.22</td>
<td>32.56</td>
<td>0.10</td>
<td>1.24</td>
<td>0.76</td>
</tr>
<tr>
<td>8</td>
<td>246.19</td>
<td>2.65</td>
<td>183.32</td>
<td>1.54</td>
<td>9.56</td>
<td>4.89</td>
</tr>
<tr>
<td>12</td>
<td>701.09</td>
<td>10.23</td>
<td>565.80</td>
<td>5.70</td>
<td>31.76</td>
<td>17.72</td>
</tr>
<tr>
<td>16</td>
<td>1908.44</td>
<td>24.04</td>
<td>1308.95</td>
<td>16.17</td>
<td>76.91</td>
<td>46.31</td>
</tr>
</tbody>
</table>
In the first design, the transformation of the print objects in the print bed calculated by the *PrintJobOrganizer* at the master is sent without modification to the slaves and the slaves generate the partial slices with print objects placed with the same transformation as sent by the master. This results in each slave generating partial slices with full slice height and width. At the master during merging process, the assigned material if not empty, is written from the partial slices to the full slices. After running the test-cases, it was seen that the performance of prototype II was comparatively bad with respect to prototype I. The major difference between the two prototypes is the communication of information between the master and slaves nodes. As the master nodes sent the print jobs having print objects with the transformation, each of the slaves had to do processing of slices with the full slice height and width. Also, as each slave sent back the partial slices in form of stream of bytes, the size of the message to be sent increased considerably given that each partial slice had the full slice dimension with material EMPTY_VOXEL assigned to rest of the slice where there was no object placed. To fix this performance issue, a new design of prototype II was implemented where in the slave nodes rearranged the print objects again by running the *PrintJobOrganizer* component after receiving the print objects from the master. Due to this, new transformation for the print objects was applied which resulted in the partial slices being much smaller as the print objects were placed close to the origin and hence the messages (containing the serialized compressed slices) communicated by the slave nodes were smaller.

**Test-Scenario:** To check the performance of two designs: The prototype II design I and design II were executed with a cluster of 3 nodes. The comparison of the two designs was done by comparing the speed-up achieved when compared with the non-distributed cuttlefish and the efficiency of the prototype II design II with varying workload.

- **Test-Data**: The input set for the test-cases consisted of head models, 8cm each in the first set and second set consisted of head and dame model 8cm each. The resolution used was $300 \times 150 \times 470$.

- **Results**: The Table 6.9 summarizes the execution times, speed-up and efficiency calculated for the two designs of prototype II for input set consisting of different models. The Table 6.10 summarizes the execution times, speed-up and efficiency calculated for the two designs of prototype II for input set consisting of same models. The Figure 6.16 shows the speed-up gained by each design in comparison to non-distributed cuttlefish and Figure 6.17 shows the efficiency of the prototype II for the two designs for input of different models. The Figure 6.14 shows the...
Figure 6.14: Speed-up of Design I and II Vs Number of Models For Same Models Input

Figure 6.15: Efficiency of Design I and II Vs Number of Models For Same Models Input
**Figure 6.16:** Speed-up of Design I and II Vs Number of Models For Different Models Input

**Figure 6.17:** Efficiency of Design I and II Vs Number of Models For Different Models Input
Table 6.9: Prototype II: Design I vs Design II Different Models Input Set

<table>
<thead>
<tr>
<th># Models Head &amp; Dame 8 cm</th>
<th>Total Execution Time (in secs) for Distributed Cuttlefish Prototype II Design II- print objects with transformation done by master node (Cluster Size- 3)</th>
<th>Total Execution Time, (in secs) for Distributed Cuttlefish Prototype II Design I- print objects with transformation done by slave node (Cluster Size- 3)</th>
<th>Speed-up in comparison to the non-distributed cuttlefish-Design II</th>
<th>Speed-up in comparison to the non-distributed cuttlefish-Design I</th>
<th>Efficiency Prototype II - Design II</th>
<th>Efficiency Prototype II - Design I</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>94.52</td>
<td>94.52</td>
<td>2.58</td>
<td>2.58</td>
<td>0.86</td>
<td>0.86</td>
</tr>
<tr>
<td>4</td>
<td>181.95</td>
<td>270.92</td>
<td>3.49</td>
<td>2.34</td>
<td>1.16</td>
<td>0.78</td>
</tr>
<tr>
<td>6</td>
<td>357.61</td>
<td>470.92</td>
<td>3.37</td>
<td>2.48</td>
<td>1.09</td>
<td>0.82</td>
</tr>
<tr>
<td>8</td>
<td>492.65</td>
<td>778.43</td>
<td>3.84</td>
<td>2.42</td>
<td>1.27</td>
<td>0.80</td>
</tr>
<tr>
<td>10</td>
<td>792.68</td>
<td>1951.24</td>
<td>3.43</td>
<td>2.45</td>
<td>2.01</td>
<td>0.81</td>
</tr>
</tbody>
</table>

Table 6.10: Prototype II: Design I vs Design II Same Models Input Set

<table>
<thead>
<tr>
<th># Models Head 8 cm</th>
<th>Total Execution Time (in secs) for Distributed Cuttlefish Prototype II Design II- print objects with transformation done by master node (Cluster Size- 3)</th>
<th>Total Execution Time, (in secs) for Distributed Cuttlefish Prototype II Design I- print objects with transformation done by slave node (Cluster Size- 3)</th>
<th>Speed-up in comparison to the non-distributed cuttlefish-Design II</th>
<th>Speed-up in comparison to the non-distributed cuttlefish-Design I</th>
<th>Efficiency Prototype II - Design II</th>
<th>Efficiency Prototype II - Design I</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>102.55</td>
<td>135.49</td>
<td>3.48</td>
<td>2.63</td>
<td>1.16</td>
<td>0.87</td>
</tr>
<tr>
<td>4</td>
<td>245.08</td>
<td>345.342</td>
<td>3.75</td>
<td>2.66</td>
<td>1.25</td>
<td>0.88</td>
</tr>
<tr>
<td>6</td>
<td>440.41</td>
<td>594.30</td>
<td>3.74</td>
<td>2.77</td>
<td>1.24</td>
<td>0.92</td>
</tr>
<tr>
<td>8</td>
<td>708.70</td>
<td>1534.65</td>
<td>5.74</td>
<td>2.65</td>
<td>1.91</td>
<td>0.88</td>
</tr>
<tr>
<td>10</td>
<td>1014.98</td>
<td>1832.35</td>
<td>4.71</td>
<td>2.61</td>
<td>1.57</td>
<td>0.87</td>
</tr>
</tbody>
</table>

speed-up gained by each design in comparison to non-distributed cuttlefish and Figure 6.15 shows the efficiency of the prototype II for the two designs for input of same models.

- **Observation**: The speed-up gained by prototype II design II (range: 2.5 to 6.0) is much larger than the speed-up gained by prototype II design I (range:2.35 to 2.55) for input of the different models. The efficiency of the prototype II design II increases with increase in workload for input of the different models. The speed-up gained by prototype II design II (range:3.5 to 5.55) is much larger than the speed-up gained by prototype II design I (range:2.5 to 2.8) for input of the same models. The efficiency of the same models of the prototype II design II increases with increase in workload for a given cluster size until a certain point after which the efficiency starts to drop with increase in the workload.

6.4 Summary

Through this Chapter, the various testing procedures which were followed are described as well as the results seen from the test-cases. The observed results from the test-cases are analyzed to gain insights about implemented solution.
7 Conclusion

With research fields like 3d Printing reaching new horizons, it is possible to print high-fidelity print objects due to the features offered by the new generation 3D Printers for high-resolution multi-material prints. One of the known drawbacks of 3D Printing is that it is slow and mass production of small objects takes too much time making it is costly, as the speed at which the printing can happen is strongly influenced by the material used for printing. Printing these objects economically to look as close as possible to the real-world objects requires a lot processing to be done on the digital input given to the software. Currently Cuttlefish can print multiple objects at the same time reducing the over-all production time for the prints. Although, cuttlefish is capable of performing the computation to enable multiple prints at the same time, it is limited by the machine capacity i.e. the resources offered by the single machine on which the application runs.

To remove system resources as the limitation to perform large scale computation, distributed version of the cuttlefish printer driver was developed through the thesis to allow the use of available resources by running the application on the cluster of machines. Two prototypes of the distributed cuttlefish version were implemented using the classic master-slave parallel processing pattern. The major difference between the two implemented prototypes was the form in which the input was given to the slaves and partial output communicated by the slaves to the master. In Prototype I, the communication of the workload from the master node to the slave nodes was done by writing a configuration file per slave node and communicating the file path to the slave node. In Prototype II, the communication of the workload from the master to slaves was in form of stream of bytes consisting of serialized print job per slave node. The partial output communicated by the slaves to the master was by writing the serialized partial slices to the disk in Prototype I whereas in Prototype II the serialized slices were compressed and then sent as stream of bytes to the master. The goal of implementing Prototype II was to enable distributed cuttlefish to execute even without a networked file system.

The distributed system consisted of cluster of heterogeneous processors connected via enterprise network with access to a networked file system. MPI was used to implement the communication between the cluster nodes. Manual testing of the two prototypes was done to check if the application ran as expected for discussed scenarios and automated testing of the two prototypes was done to measure the performance of the application for varying workload and cluster size. Both distributed cuttlefish prototype I and prototype II performed better than non-distributed cuttlefish for varying number and types of models. The speed-up gained by both the prototypes was at-least linear and in some cases super-linear. One of the possible reasons for super-linear speed up is the cache hit rate being higher for smaller data size in multi-core processors. When the models are distributed among the processors, the total amount of data per processor is reduced in comparison to single processor performing computation on all the models at the same time, leading to smaller size of data to be cached. This behavior, i.e smaller data size decreases the computation time due to more cache hits, is also seen when cuttlefish is executed multiple times with each run having single model rather than one time with more number of models. Another probable reason for high speed-up in distributed cuttlefish is that even though the RAM is limited, it is sufficient to keep all the necessary data in RAM unlike in non-distributed cuttlefish where higher number of models lead to data being swapped in/out of the RAM leading to disk read/writes which are costly. This justifies use of distributed computing as a solution to perform large scale computation.

Prototype II performed better than Prototype I in the various scenarios tested and the percentage increase in efficiency of Prototype II in comparison to the Prototype I scaled with the increase in cluster size. The Prototype II was implemented with two designs which lead to varying performance of the application. In Design I, the print objects transformation set by the master is used by the slaves to generate the partial slices whereas in Design II, the print objects are re-arranged at the slave leading to much smaller partial slices. The comparison of the two designs revealed that the performance of prototype II is highly affected by the amount of data to be communicated by the slaves as well as the size of the partial slices, i.e. bigger slices lead to higher computation by certain components of the pipeline.

The performance of prototypes for varying size of the models was tested with constant number of models and increasing the model size and the observation was that the speed-up gained by the prototypes decreases with increase in models size. This behavior could be justified as increase in model size increases the workload. The performance of distributed cuttlefish Prototype I is highly impacted by the serialization of the slices as it requires disk I/O whereas in Prototype II, receiving the partial slices as stream of bytes impacts strongly. For Prototype I, when the size of the models increases the time taken to deserialize the slices increases. One probable reason is that deserialization of the slices leads to allocation of large memory buffers for slices of bigger model sizes in RAM. As the system has limited RAM, the use of large amount buffers exhausts the available memory leading to frequent paging of the least used pages out of
the memory (the master node used for the test-cases had 32 GiB RAM and the pagefile.sys was 24.5 GiB while running deserialization method for 5 16cm head model, showing that the processor was swapping the pages at a high rate.)
8 Future Work

Currently, the cost function for distribution of sub-tasks among the slave nodes is calculated using the size of the bounding box to fit the print object i.e. the volume of the bounding box for the object. This results in a balanced distributed load such that none of the slave nodes are over-worked. Although, load balancing is one of the criteria for distribution, there are other criteria used as well. One possible criterion would be using the surface area of the print object as a measure for cost calculation. Complex surfaces are difficult to express with less number of triangles in the mesh representation making the beginning steps of the pipeline costly i.e. the process of converting the mesh to voxels. The more the surface voxels, higher is the cost of half-toning as cost of half-toning is proportional to the surface voxels. The complexity of the texture with high resolution leads to higher cost of rasterization process. The computation time can be expressed as a function $f(s,o)$ where $s$ is the ratio of total number of surface voxels to total number of voxels and $o$ is the ratio of total number of occupied voxels to total number of voxels. The ratios can be calculated using the Equation 8.1 and Equation 8.2 where $Num_o$ is the number of occupied voxels, $TNum_v$ is total number of voxels, $Num_s$ is the number of surface voxels. Determining the correct form of the function $f$ opens up new avenue for future work. One possible approach for determining the form of the function $f$, could be a data-drive approach wherein the information related to computation time for each print object, ratio of surface voxels to total number of voxels ($s$) and ratio of occupied voxel to total number of voxels ($o$) can be recorded on each slave node. Logging this information and then use of statistical analysis to find the correct form the function could be done. Using this function, the computation time for each print object can be predicted which could be used to implement a new cost function used for distributing the objects between the compute nodes.

$$s = \frac{Num_s}{TNum_v} \quad (8.1)$$

$$o = \frac{Num_o}{TNum_v} \quad (8.2)$$

The placement of the print objects on the print bed also strongly influences the appearance properties. This is because of the variance in time for which the slices are exposed to the UV light in the curing process which in turn affects the appearance properties. Companies in animation industry, for example Laika, use 3D printing to print the animation characters with the series of expressions. It is important that the 3D prints of the closest (in the series) facial expressions have the same appearance properties. If these print objects are placed some distance apart, worst being in the extreme corners of the print bed, then it would lead to the print having noteworthy difference in appearance. So locality of the print objects could be one of the other possibilities which could be used for distribution of the print objects among the slave nodes.

Another possibility for future work is changing the underlying communication protocol used i.e. replace MPI with TCP/IP or REST protocol. As seen in the section 6.3.2, the receive function makes the major impact on the performance of Prototype II, it would be interesting to know if the performance improves on using another communication protocol. Logging functionality can be used to understand the behavior of the distributed cuttlefish by generating the information of the run times of different nodes for different models with varying textures. This logged information along with the node configuration i.e. system features like the CPU speed, cores, memory etc can be used to make informed decisions about work load allocation if the cluster nodes remain unchanged.

Changes to the underlying distributed system architecture could impact the performance of the distributed cuttlefish application, for example as all the computers in the enterprise use the common network, there can be contention among nodes. Generally within the cluster, the compute nodes and the master nodes are connected via dedicated private network and only the master node is connected to the public network. As the cluster nodes which communicate with the master node have their own private network, there could be lower contention which might increase the performance for Prototype II as it is heavily impacted by the Receive function. For Prototype I, the master node can become the bottleneck for performing deserialization for large number of slave nodes. If there is possibility of using a faster hard drive on the master node, may be the cost of deserialization can be reduced for large number of slave nodes. Replacing cluster computing with cloud computing would be big leap, where in the distributed cuttlefish is offered as a service on the cloud.
Bibliography


