A MPI-based Distributed Computation for Supporting Optimization of Urban Designs with QUIC EnvSim

A THESIS
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Viswanadh Kumar Reddy Vuggumudi

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

Dr. Peter Willemsen

July, 2015

Acknowledgements

The thesis would not have been possible without the help of many others especially without my advisor Dr. Peter Willemsen. I am greatly indebted to him for his continuous support and suggestions throughout the work. I am also thankful to Matthew Overby for his help.

I am also grateful to Lori Lucia, Clare Ford, Jim Luttinen and International Student Services who have been helpful all along. I would also like to thank the department of Computer Science at University of Minnesota Duluth for funding the thesis work. I would also like to extend my appreciation to Dr. Richard Maclin and Dr. Marshall Hampton for being a part of my graduate committee.

Last, but most importantly, I would like to thank all my friends in the class of UMD Computer Science-2015 for their encouragement and support.

Dedication

I would like to dedicate this thesis to my parents, Surendra Reddy Vuggumudi and Sailaja Vuggumudi and my sister, Keerthi Vuggumudi for their unconditional and endless love. I would also like to dedicate this thesis to my uncle, Kamalakar Vuggumudi for his continual guidance and with out whom I probably never would have done my masters.

Abstract

In the present day of urbanization, rise in urban infrastructure is causing an increase in air temperatures and pollution concentrations. This leads to an increase in the energy required to cool buildings and more focused efforts to mitigate pollution. An effective way to mitigate these problems is by carefully designing cityscapes i.e., by placing the buildings, vegetation optimally and choosing energy efficient building materials. Researchers have been building computational models to understand the effects of urban infrastructure on microclimate. Simulating these models is a computationally expensive task. QUIC EnvSim (QES)[11] is a dynamic, scalable and high performance framework that has provided a platform for building and simulating these models. QUIC EnvSim uses Graphics Processing Units (GPUs) to run each individual simulation faster than previous simulation codes. Though each individual simulation takes a short time, it is often required to perform large numbers of simulations and it can take a long time to complete them. This thesis introduces MPI QUIC, a scalable and extendable framework for running these simulations across a cluster of machines, effectively reducing the time required to run all simulations. Various tests on the framework have shown that the framework is capable of running large numbers of simulations in a relatively less amount of time. A test running 65536 simulation was performed. The estimated time for running the test on a single computer is approximately 11.37 days, with each simulation taking approximately 15 seconds to complete. The framework was able to finish running all the simulations in 19 hours, 0 minutes and 25 seconds showing a tremendous speed up of 92.5%. Thus urban planners can use this framework along with QUIC EnvSim to understand the effects of urban forms on microclimate and take informed design decision relatively quickly for building environment friendly urban landscapes. Besides providing a distributed computational environment, the other goal of the MPI QUIC project is to provide an user friendly interface for specifying optimization problems. The

current work provides the ground work for the successors of the current work to provide a programmable interface for end users for specifying optimization problems. The framework is also designed so that future implementers can incorporate optimization algorithms that can optimize on multiple fitness functions.

Contents

List of Figures

1 Introduction

Urban planners who want to build environment friendly cities need to understand the complex interactions between urban infrastructure and the environment to take informed design decision before building any infrastructure. They cannot afford to make changes to the cities after they are built. An inexpensive alternative is building simulations for various design decisions and choosing the optimal designs that meet the requirements. QUIC EnvSim [11] provides a framework capable of building and simulating various climatic models. Even though QUIC EnvSim performs simulations more quickly than previous simulation codes, environmental simulations are still taxing on compute resources as slight changes to infrastructure can potentially exponentially increase the total number of simulations to be performed. For example, with 2 buildings each with 8 possible positions requires performing 64 simulations to compute all possible combinations. Adding another building with the same freedom for movement increases the number of simulations to be performed to 512. Assuming an individual simulation takes 15 seconds time and imagining many more combinations, performing a million of such simulations on a single computer could take 173.6 days. That amount of wait time is not feasible for urban planning time scale. Supposedly, if the simulations can be run on several machines, say 100 machines, the total simulation time can be brought down to 1.73 days. This idea of decreasing the total simulation time by using multiple machines forms the hypothesis of the current work. As a proof of the hypothesis, the current work presents a framework that is capable of running QUIC EnvSim simulations across a cluster of machines. Besides providing a distributed computational environment for QUIC EnvSim, the current work also forms the ground work

for the long term goal of providing a simple programmatic interface for end users (urban planners, engineers or scientists) for specifying optimization problems. The main objective of the programmatic interface is to provide users with a simple Matlab like language for specifying various complex constraints of the infrastructure and defining fitness functions for optimizations. Albeit the current work does not provide implementations for advanced optimization algorithms like genetic algorithm based optimizers that can optimize multiple fitness values it was designed with those in mind. In the current implementation users of the framework can tap into the results of various simulations and can execute multiple fitness functions whose fitness values are passed onto optimization module irrespective of the optimization algorithm employed. A test running 65536 simulations was performed on 19 machines with a performance gain of 92.5% demonstrating the capability of the system to run large numbers of simulations. Recommendations were also made for fine tuning the performance of the system according to the cluster where the system will be deployed. Current

2 Background

2.1 Background

2.1.1 Distributed Computing

Problems of scale like Grand Challenge Problems [7] require huge amounts of computing power and memory but the capacity of a single or a multicore processor is finite due to constraints imposed by physical limits such as speed of light. In practice a processor's capacity is further limited by factors like power wall (increase in clock frequencies require exponential increase in power consumption) and memory wall (increase in difference between processor speeds and main memory). Thus a single computer even with a multi-core processor cannot deliver the computational resources needed for sophisticated computer algorithms. **Distributed Computing** provides a way to amass the required huge computational resources for solving complex problems. In a distributed computing system many computers (nodes) are connected with a network to obtain the required computing resources. Many issues arise when implementing distributed systems, such as communication between nodes, fault-tolerance, distributing work among nodes and dealing with heterogeneity of nodes etc. Communication forms the core of a distributed system. Though many network issues are handled by lower level protocols like TCP and UDP etc., communication can be further simplified by choosing a communication model. Communication models provide higher-level abstractions (e.g. location transparency, portability) for processes running on various nodes to communicate than dealing with the raw packets themselves.

Popular communication models used in distributed systems are message passing, Remote Procedure Calls (RPC) and Distributed Shared Memory (DSM). While DSM and RPC are easier to use than message passing, they also incur more overhead [20]. The current work uses message passing as it provides better performance among the three.

2.1.2 Message Passing Interface

In message passing, processes on different nodes communicate with each other by sending messages. Message Passing Interface (MPI) [5] is the widely accepted standard for message passing with performance, scalability and portability as its goals. MPI provides facilities like point-to-point communication, collective communication such as broadcasting etc. Programs written using MPI follow Single Program Multiple Data (SPMD) model i.e., the same copy of the program is executed on all nodes in the distributed system. MPI allows controlling parts of the program that will be executed based on the node on which the program is executing. In this way the program can take on different roles such as a producer or consumer of data. The MPI standard provides bindings for C and Fortran. Unofficial bindings are also available for other languages. Apart from performance, availability of MPI in most supercomputing environments lead to choosing MPI for the current work.

2.1.3 Boost.MPI

Abstractions provided by MPI are low level and make it difficult to program communication of C++ Standard Template Library (STL) containers (e.g. vector, map etc.) and user defined data types. **Boost.MPI** is a C++ interface for MPI [6] with support for communicating STL containers and user-defined data types. It uses the **Boost.Serialization** library for converting user-defined data types into MPI data types, which MPI provides for portability. Boost.MPI is a thin layer of data abstraction only and must be used with an actual

Listing 2.1: Sample program to illustrate point-to-point communication with Boost.MPI

```
1  const int RESULTS = 1;
2  const int MASTER = 0;
3  const int ARGS = 0;
4  int main(int argc, char* argv[]){
5    //Initialize MPI environment
6    boost::mpi::environment env(argc, argv);
7    //Create the default communicator
8    boost::mpi::communicator world;
9    //Consider process with rank 0 is master and the rest as slaves
10   if(world.rank() == MASTER){ //Master work
11     for(int i=1;i<world.size();i++){
12       cout<<"Master node sending data to worker"<<i<<endl;
13       world.send(i, ARGS, i);
14     }
15     for(int i=1;i<world.size();i++){
16      vector<int> result;
17      boost::mpi::status msg =
18                 world.probe(boost::mpi::any_source, RESULTS);
19      world.recv(msg.source(), RESULTS, result);
20      cout<<"received "<<result[1]<< " from "<<msg.source()<<"\n";
21     }
22   }else{ //Slaves work
23       cout<<"executing process "<<world.rank()<<endl;
24       int temp;
25       world.recv(MASTER, ARGS, temp);
26       vector<int> result;
27       result.push_back(temp);
28       result.push_back(temp*temp);
29       world.send(MASTER, RESULTS, result);
30   }
31 }
```

MPI implementation like OpenMPI [24]. An example MPI program can nicely illustrate the MPI model. In listing 2.1 an MPI program is provided that uses Boost.MPI. The example illustrates how the same program takes on different roles and performs point-to-point transfer of data between two or more nodes.

MPI code must first start by initializing the distributed system. In Boost.MPI, it is done by creating an instance of `Boost::mpi::environment` as shown in line-6.

A **Communicator** in MPI is an abstraction representing a collection of processes that can communicate. Each process in a communicator is uniquely identified by a number referred to as **Rank**. MPI provides a default communicator, which contains all the processes running in the system. Line-8 shows the creation of default communicator. In a communicator, a process with *rank 0* is usually considered as the master process and the rest as slave processes. As mentioned earlier, since MPI is SPMD, the same program is run on all nodes. The rank of a node can be used to control the parts of the program that are run on that node. In the above program lines 1-10 are executed by all nodes, lines 11-21 are executed in the master process and lines 22-30 are executed in the rest of the slave processes.

In the above program, the master process sends an integer (`int`) to each slave. Each slave then computes the square of that number and sends back both the number and its square in a `vector`.

Lines 11-14 show the master sending a number to each slave (all ranks except 0). In line-11, the `size()` call gives the total number of processes in a communicator, which in this case is the default communicator containing all processes. Line-13 shows the `send()` method with which a process can send a message to any other process in the communicator. The `send()` function accepts 3 arguments. The first argument represents destination process rank, the second argument represents a tag to identify the message type and the third argument represents data to be sent. Messages sent by the master are received by each slave in line-25 using the `recv()` call which needs the source from which message is to be received, type tag of the message to be received, a buffer space where the data in the message is to be stored. Since an `int` was sent, an `int` is used to receive the data. Each slave then computes the square for that number and stores both the number and its square in a `vector`. In line-29, the resulting `vector` is then sent to the master process with the message tag `RESULT` to differentiate the messages sent by slave processes from the messages sent by the master process which have the tag `ARGS`. Line-18 shows the `probe()` call which checks

for the availability of specific messages without receiving them. Since the master does not know the order in which slaves send their results, the master uses `probe()` to check for any `RESULT` messages from the slaves. Whenever such a message is available, the master receives the message using the `recv()` call explained before. The full version of the program along with instruction on compiling and running it are presented in appendix-A.

2.1.4 Plugin Support

Plugins are pieces of code that will help customize the functionality of a system. Host applications provide an API for plugins to interact with the core system. Plugins in C++ can be implemented as shared libraries. Some parts of the framework which could change often (e.g. fitness function definitions) are implemented as plugins. This is to avoid the long compilation times of the framework and the client code using the framework. Another advantage is that users of the framework will be able to switch the default-shared libraries with their own implementations without having to compile the code at all.

Plugin implementation in C++ is both platform as well as compiler dependent as C++ does not mandate Application Binary Interface (ABI) compatibility and does not support dynamic loading. ABI in the context of C++ is a standard for function calling conventions and data layouts in memory etc. Since C++ does have a portable ABI, object code generated by two different compilers may not be compatible. This requires compiling both host and plugin code with the same compiler. A solution for the lack of support for dynamic loading is to use C linkages. C functions can used to construct and return pointers for the plugin's C++ class objects, which can then be used transparently.

2.1.5 ANTLR

Specifying computations for energy simulations is already possible with QUIC EnvSim. However, this requires deep programming knowledge of the underlying framework. One of the contributions of this thesis is to provide a more general and accessible way for engineers and scientists to create their own optimization systems with QUIC EnvSim using the MPI distributed system on a cluster of machines.

Users can provide various constraints on infrastructure in a structured way in an **optimization (OPT) file**. These OPT files are parsed using a parser generated by **ANother Tool for Language Recognition (ANTLR)**[18]. ANTLR is a parser generator that generates a lexer and a parser for recognizing and processing a formal language. A formal language is constructed from an alphabet and rules combining symbols from the alphabet. ANTLR takes a description for a formal language to be recognized called a *grammar* and generates a parser to recognize and process the language. ANTLR 3.0 can generate parsers in various languages e.g. Java, C++ and Python etc. ANTLR grammars are easy to develop, as it resembles regular expressions.

An ANTLR grammar is a collection of rules of the form *rule_name : rule*. Grammars contain rules for both a lexer and a parser. Lexer rules tokenize a stream of characters i.e., recognizing numbers, white spaces, identifiers etc. Parser rules recognize phrases in languages like expressions, declarations, loop statements, conditionals etc. In an ANTLR grammar lexer rules always begin with an upper case character. Listing 2.2 is a sample ANTLR grammar for recognizing simple addition and subtraction expressions. The rule *expn* instructs the generated parser that we would like to match text that satisfies the rule *addSubExpn*. The rule *addSubExpn* is a recursive rule which matches either a number or a number followed by a + or - and another *addSubExpn*. The lexer rule *Number* finds an integer or floating point number. The lexer rule *WS* finds white spaces and the instruction

Listing 2.2: Sample ANTLR grammar for recognizing addition and subtraction expressions

```
grammar sample;
/* Try to match a addSub expressions*/
expn  : addSubExp ;
/* An add sub expression */
addSubExp
    :    Number
    |  Number ('+'|'-') addSubExp
    ;
/* A number: can be an integer value, or a decimal value */
Number
    :    ('0'..'9')+ ('.' ('0'..'9')+)?
    ;
/* We're going to ignore all white space characters */
WS
    :   (' ' | '\t' | '\r'| '\n') {$channel=HIDDEN;}
    ;
```

{$channel=HIDDEN} instructs ANTLR to ignore white spaces. This is a contrived example for illustrative purposes. The grammar recognizes only simple expressions like 2+3, 2+3-5 etc. The grammar developed for this thesis is more complex and is presented in chapter 3.

2.1.6 Database

MPI does not support fault tolerance by default and leaves the choice to applications. A single process failure in MPI results in the failure of the entire system. The current framework approaches the problem using a checkpoint based solution. The framework stores the results of simulations performed in a database. When a failure results in a system crash and the system is restarted only simulations with results not stored in the database are performed making the system more fault tolerant. The database also serves as a permanent store from which the results of simulations that are already performed can be retrieved any time in the future without having to perform them again.

MongoDB

The framework currently uses MongoDB as its database. MongoDB is a document based NoSQL database. NoSQL databases tend to be fast when compared to the traditional relational databases as they do not require a schema and can be scaled seamlessly with increasing storage requirements. Since the requirements of the framework are quick querying and scalability, MongoDB, a popular NoSQL database is chosen as the database.

2.1.7 QUIC

The Quick Urban & Industrial Complex (QUIC)[4] system was initially conceived to quickly compute the dispersion of airborne contaminants (e.g. pollution particles) that are released in urban spaces. QUIC is a collection of tools that includes simulation softwares (QUIC-URB, QUIC-PLUME, QUIC-PRESSURE) and Graphical User Interfaces (QUIC-GUI). Simulations are performed on virtual urban landscapes (domains) that are constructed using the city builder tool in QUIC-GUI. The city builder tool can be used to specify the dimensions of a domain, define building geometries and positions, create vegetative canopies, and specify location and time etc.

QUIC Project

The city builder tool stores the information about an urban domain in a collection of files called **QUIC Project**. In addition, a QUIC Project may also contain files with data specific to QUIC simulations and meteorological information such as wind angles etc. QUIC EnvSim also uses QUIC Projects for performing various simulations.

An urban landscape can be designed in a lot of different ways. Searching for an optimal design often requires performing simulations on many different designs. Making design decisions on a QUIC domain involves changing different parameters of various entities

in the QUIC domain. Making changes to a QUIC domain using the city builder tool or manually in the QUIC Project files each time, before a simulation is performed is time consuming, if not infeasible. The current framework provides an easy interface specified in an **Optimization (OPT) file** for users to manipulate data in a QUIC Project and perform simulations on the manipulated data. The following text discusses important files in a QUIC Project and other information on QUIC domains users need to know for using the OPT file. The OPT file currently supports making changes to the configuration parameters stored in the following files:

- QU_simparams.inp - contains world dimensions, UTM coordinates etc.
- QU_buildings.inp - contains buildings positions, dimensions.
- QU_metparams.inp - contains meteorological information.

Vegesna [23] provided C++ class representations for each of the these files. The OPT file interface exposes these classes as global variables: quSimParams (QU_simparams.inp), quBuildings (QU_buildings.inp) and quMetParams (QU_metparams.inp). Among the three files, QU_buildings.inp is important for energy simulations as buildings positions and dimensions can highly affect the microclimate. Listing 2.3 shows the important parts of the class structure for QU_buildings.inp.

The class structure shows various attributes that can be changed for buildings, e.g. height, width, xfo (x-offset) etc. Also, note that the various buildings in a QUIC domain are represented as the vector `buildings` (line-29). The index of a building is the same as the order of its appearance in the QU_buildings.inp file. The indexing starts with number 0. For example, a change to make building 0's height to be 100 can be specified as the following line in the OPT file.

```
1 quBuildings.buildings[0].height=100
```

Listing 2.3: C++ class for QU_buildings.inp file

```
class quBuildings : public quicDataFile
{
  //some code above//
  struct buildingData
  {
    int bldNum;
    int group;          //**********TYPE**********
    int type;           //REGULAR     = 1,
    float height;       //CYLINDRICAL = 2,
    float width;        //PENTAGONAL  = 3,
    float length;       //VEGETATION  = 9
    float xfo;
    float yfo;
    float zfo;
    float gamma;
    float supplementalData;

    int geometry;
    float centroidX;
    float centroidY;
    float rotation;
    float attenuationCoef;
    int numPolys;
  };

  float wallRoughnessLength;
  int numPolygonBuildingNodes;

  std::vector<buildingData> buildings;

  //some more code below
};
```

Changes to QU_simparams.inp and QU_metparams.inp are specified similarly in the OPT file. Section 3.1.1 discusses more on using the OPT file.

Specifying buildings position in the OPT file requires knowledge of building layouts in QUIC domains. The QUIC domains are discretized as 3D grids. Buildings are grid aligned 3D boxes and their positions are specified as offsets from the grid's axes. Offsets

for buildings in QUIC world are measured as shown in fig 2.1. X-offset (xfo) of a building is the shortest distance between Y-axis to the left edge of the building. Y-offset (yfo) of a building is taken as the shortest distance from the mid point along the breadth of the building to the X-axis.

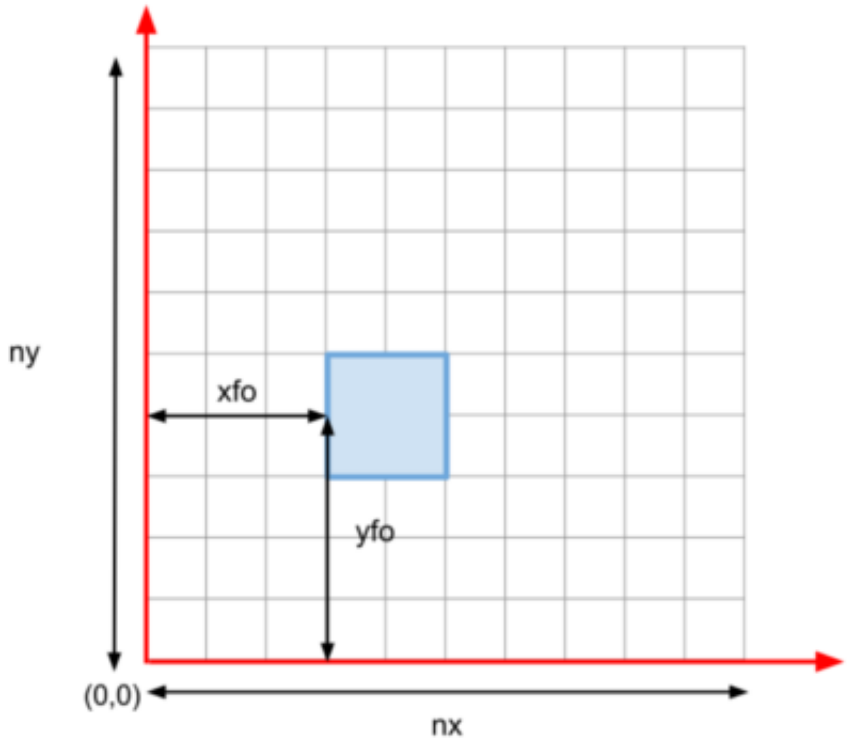

Figure 2.1: Building offsets in QUIC

QUIC EnvSim considers each of the buildings faces and the ground to be covered by several tiles called **Patches**. Patches can be thought of as the skin or the outermost layer. Simulation results like temperature, sky view factor, moisture etc. are often calculated and stored for each patch. Most of the time users are interested in collecting simulation results from only a small area in a QUIC domain. This requires knowing the exact patches in that area of interest. The framework provides an abstraction called a **collection box** in the OPT file, which users can use to specify an area of interest (for instance all patches within some cuboid). The framework will then provide a list of patches in that area which can be used

for collecting simulation results like temperature, longwave radiation, shortwave radiation etc. on each patch. The OPT file and its use will be discussed in chap-3 and chap-4.

2.2 Previous Work

Climate and weather modeling has always been one of the huge computational challenges. Much work has investigated the use of MPI for large scale computing in weather and climate related systems. Tobis et al. [22] have used MPI for building Fast Ocean Atmospheric Model (FOAM). FOAM is a coupled ocean-atmosphere model for studying phenomena of interest on decadal and century time scales. Using novel model formulations and MPI based distributed computing, FOAM achieved a simulation speed of 6000 times wall clock i.e., the system can simulate 6000 seconds of deep oceanic currents in 1 second of wall clock time. Ostromsky et al. [10] demonstrated an increase in the performance of solving large numbers of numerical equations involved in large scale air pollution modeling by using parallel programming frameworks like MPI. Yang et al. [25] have developed a hybrid algorithm for Global Atmospheric Simulations using both GPU's and CPU's. The system also uses MPI for node connectivity. Their system was able to scale up to 0.8Pflops using 3,750 nodes (45,000 cores and 3,750 GPUs). Apart from the few works mentioned above MPI has been widely used (Randall et al. [19], Hill et al. [8]) for building weather and climate modeling systems. Yang et al. [25] shows the current trend of using heterogeneous clusters with both CPU's and GPU's for obtaining much higher computational resources than traditional CPU only clusters. The current framework in combination with QUIC EnvSim also falls into the category of systems using heterogeneous clusters.

Green Environmental Urban Simulations for Sustainability (GEnUSiS) is a project that investigates the effects of green infrastructure like parks, roof gardens, green building materials like asphalt etc. on large scale landscapes like cities in terms of energy use and

microclimate (Bailey et al. [2], Overby et al. [16]). It is the computation system that is used in this thesis.

The following text outlines various related work aimed at increasing the performance of environmental simulations in Green Environmental Urban Simulations for Sustainability (GEnUSiS) and QUIC PLUME projects.

Singh et al. [21] and Norgren [9] proposed GPU PLUME, a system that uses inexpensive Graphic Processing Units (GPUs) to perform real time and interactive dispersion simulations. Though GPUs were initially used in rendering games they are now being increasingly used in scientific and engineering applications as they are optimized for performing arithmetic operations and provide large scale parallel computing resources. Usage of GPUs has increased the performance of GPU PLUME more than two fold compared to a traditional CPU based implementation.

Vegesna [23] provided a distributed computational environment for the GEnUSiS project using volunteer computing. In volunteer computing, users from around the globe donate their spare computational resources (storage and processor time) to an application. The number of computers in the world is ever increasing, but most of the computational resources are not fully utilized. Volunteer computing originated with the idea of making use of these unused resources. Given the sheer number of computers in the world, volunteer computing is capable of providing huge amounts of computational resources, but in reality the supply is inconsistent making it unfit for applications that need to produce results immediately. In volunteer computing it is not possible to hold volunteers accountable, this gives the possibility of misbehaving volunteers who might return tampered results.

Vegesna's system [23] used the Berkeley Open Infrastructure for Network Computing (BOINC) [1] for providing distributed computing. BOINC provides a platform for applications to use volunteer computing. It was initially developed to manage one of the major computing projects, SETI@home that analyzes radio signals in search of extraterrestrial

intelligence. Although BOINC provides a distributed computing platform for the GEnUSiS project, given the nature of volunteer computing, BOINC does not guarantee resource availability. This instability in resource availability is not acceptable in situation when the GEnUSiS simulation softwares need to return results immediately. Setting up and maintaining a BOINC system is also complex. The BOINC solution also suffered from some performance issues due to BOINC's heavy database dependency and unnecessary work replication. BOINC replicates work to make sure results returned by volunteers are not tampered.

Overby and colleagues ([11], [16], [3], [2], [17], [15], [14], [13], [12]) developed QUIC EnvSim (QES) as generic scalable framework for developing urban microclimate models. The framework also employs GPUs to accelerate simulations using Nvidia's Cuda and Optix frameworks. Models for computing view factors, surface temperatures and radiative interactions between buildings have been implemented using the framework.

The current work provides a distributed computing environment for QUIC EnvSim. It uses MPI-based dedicated cluster environments where resource availability is guaranteed. MPI-based distributed systems are also easier to setup and maintain. The current work also strives to be customizable and easy to use. It provides urban planners the ability to perform user-defined computations (such as fitness function) over the results obtained by performing the simulations. Users also can specify an area of interest in real world coordinates from which they can collect the results of simulations without having to worry about finding exact patch numbers.

2.2.1 Optimization & Fitness Functions

The main motto of the end users of this framework is to find optimal designs for building urban landscapes. They can optimize their designs for any/combinations of the simulation

results (e.g. temperatures, moisture etc.) that can be obtained using QUIC EnvSim simulations. To perform optimization simulations must be quantifiable for comparison. The fitness functions quantize simulations by producing a set of numbers (fitness values) for each simulation. The fitness values for a simulation are produced by evaluating various fitness functions on the results obtained by performing the simulation. Clients (programmers porting QUICEnvsim models to use MPI QUIC) of the framework implement various optimization algorithms using the fitness values for finding the designs that produce optimal results. Examples of fitness values can be minimum temperature, average temperature, longwave radiation etc. An example optimization could be to find an urban setting with minimum average temperature and maximum wind flow.

2.2.2 Outline

Chapter-3 discusses the implementation details of the framework along with information on using the framework. Chapter-4 presents the tests performed on the scalability and performance of the framework and chapter-5 presents conclusion and future work.

3 Implementation

The goal of the current thesis is to provide a system capable of running QUIC EnvSim simulations across a dedicated cluster and an easy to use interface for users to specify simulation parameters for building optimization systems. Figure 3.1 shows various components of the framework that interact with each other to form a comprehensive system that meets this goal. The following points give a high level overview of the framework's algorithm and Figures 3.2 and 3.3 present flow charts for the same.

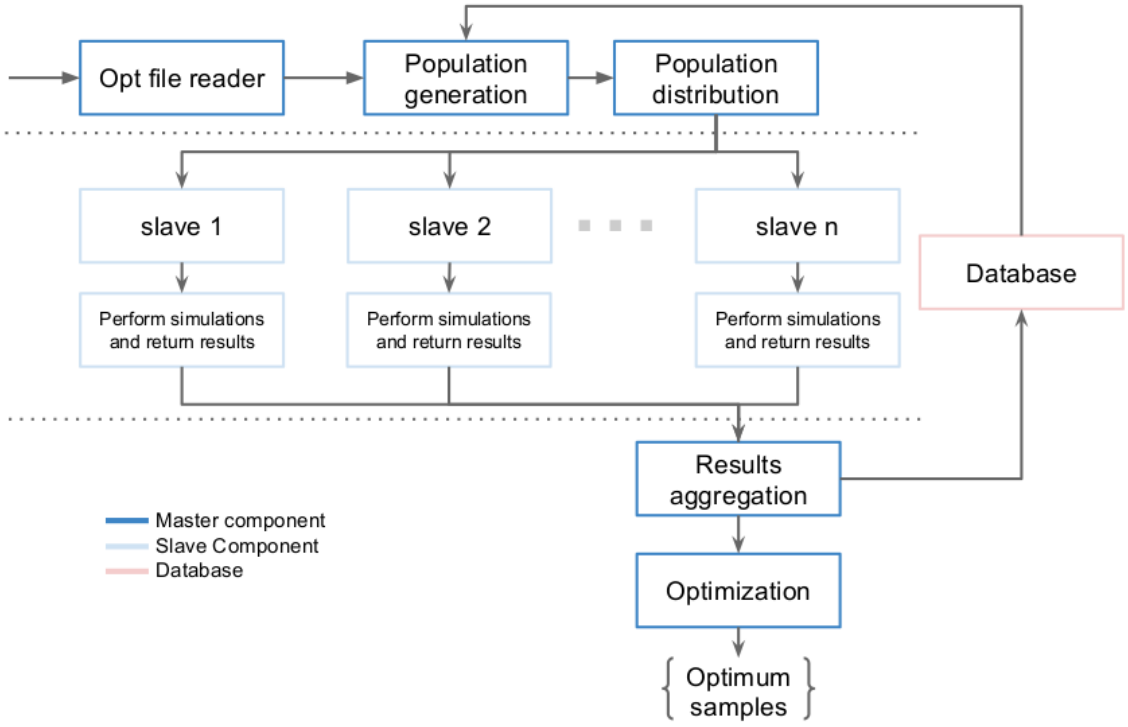

Figure 3.1: Overview of framework architecture.

The framework follows a client-server architecture and communication between pro-

cesses on different nodes is achieved through Boost.MPI. A process with rank 0 is considered as a **Master** process and the rest of them are considered **Slave Processes**. In a traditional networking sense, a master process acts as server and slaves as clients.

1. Users submit an optimization (OPT) file that specifies various constraints on buildings properties such as dimensions and positions.
2. A master process parses the OPT file and stores the constraints specified in the OPT file.
3. The master generates different combinations (**samples**) of buildings properties using the information stored after parsing the OPT file. A collection of samples is called a **Population**.
4. The master divides the population into chunks.
5. A population chunk is then sent to each slave.
6. For each sample in a chunk received by each slave process, an in-memory representation of QUIC project files is updated with the buildings properties stored in the sample and the required simulation is performed. After each simulation is performed, user defined fitness functions are executed on the results obtained by performing the simulation.
7. Each slave then collects the fitness values for the simulations it performed and sends them to the master.
8. Repeat step 4 if there are more chunks to be sent, otherwise continue to next step.
9. The master performs optimization on the fitness values received for all simulations.

The next few sections describe the implementation details of various components in the master and slave processes shown in figure 3.1.

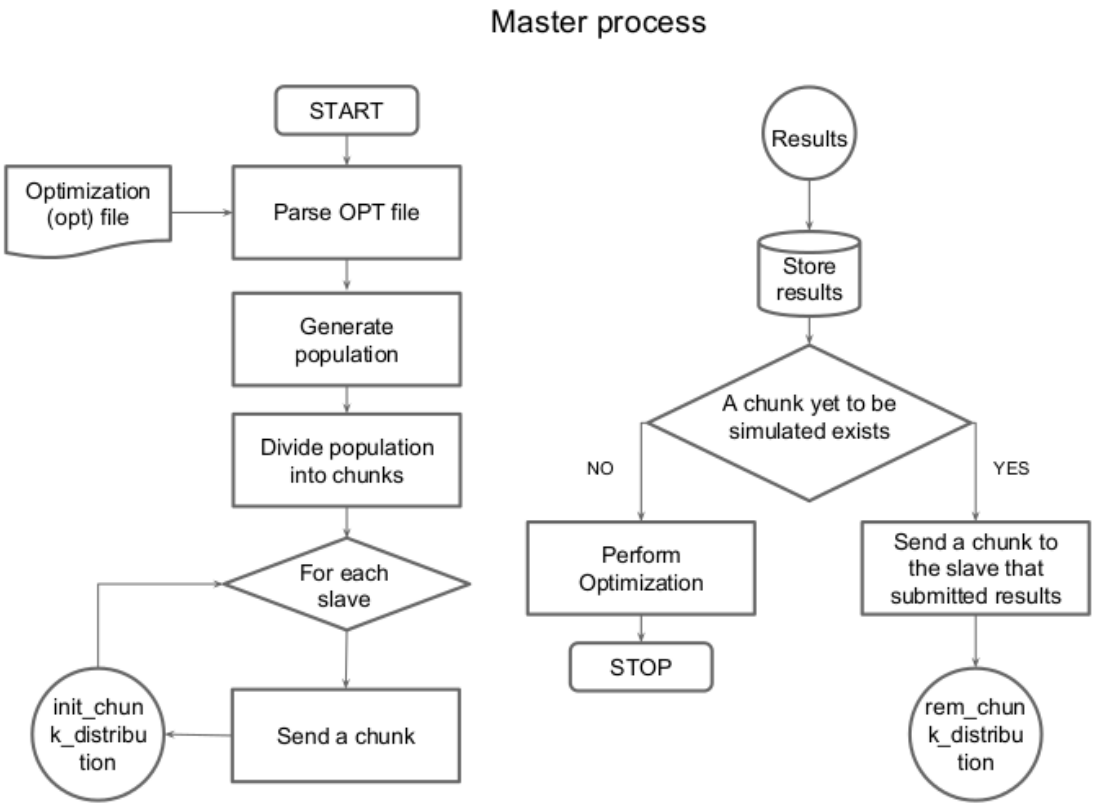

Figure 3.2: Flow chart for the master process in the framework's algorithm.

3.1 Master Process

The master process handles most of the heavy lifting done by the framework like population generation, work distribution, results assimilation, check pointing for fault tolerance and optimization. Figure 3.4 shows various components of the master process and their interactions. Users of the framework (urban planners etc.) interact only with the master process. Users give input to the master process and receive optimization results from the same. In the following subsections, each of the components in the pipeline are explained in detail.

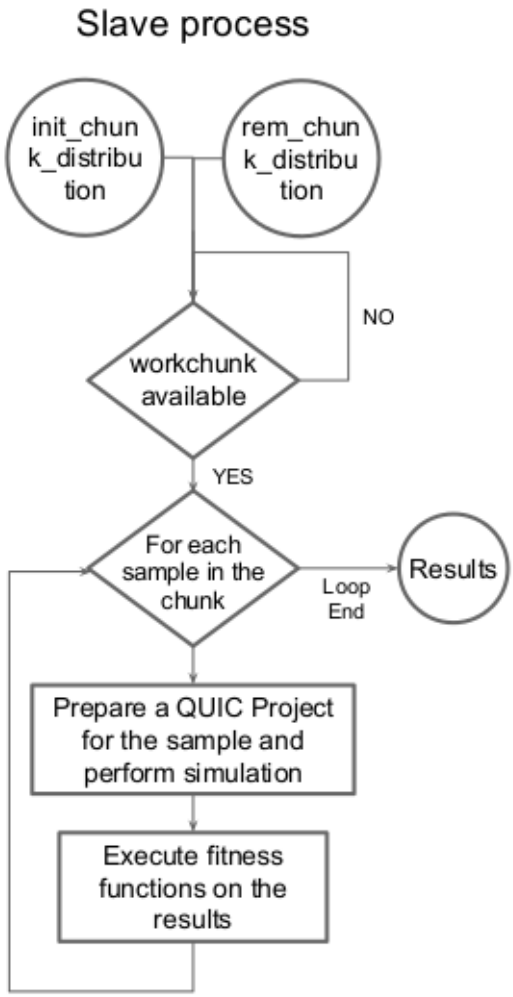

Figure 3.3: Flow chart for the slave processes in the framework's algorithm.

3.1.1 OPT File Reader

The first component in the pipeline of the master process shown in figure 3.4 is the OPT file reader. The master process receives an optimization (OPT) file as the input. The OPT file reader parses the input OPT file and stores relevant information, which will be used by the subsequent components in the pipeline.

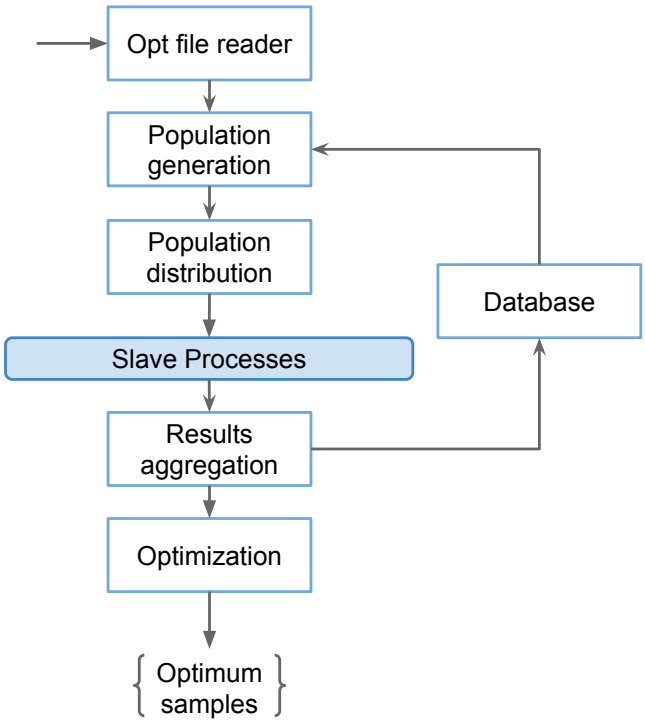

Figure 3.4: Master Process control flow.

Optimization (OPT) File

A brief introduction to the Optimization (OPT) file interface has been provided in 2.1.7. The following text presents the syntax of the OPT file as well as some of its important contents.

Optimization File Syntax

The OPT file reader is a parser generated using ANTLR. The ANTLR grammar used for parsing the OPT file is presented in appendix-B.1. The Optimization file supports two constructs: single line comments and assignments (lines 39-46 in appendix-B.1) . A comment is a sequence of characters that begin with '//' and end with a new line (lines 53-56 in appendix-B.1).

An assignment is of the form *lvalue* = *rvalue* (lines 48-51 in appendix-B.1), where

- lvalue (lines 61-66 in appendix-B.1) can be one of the following:
  - **Control variable**: They affect all the simulations, e.g. base project path, job type etc. They begin with keyword *const* signifying that they are constant across all simulations. Control variables are parsed using the rule in lines 90-93 in appendix-B.1. For example the line, *const JOBTYPE = 'lsm'*, represents a control variable JOBTYPE.
  - **Constraint variable**: They are primarily used for imposing constraints on various entities in a QUIC project and forming the basis for optimizations. They are of the form classname followed by member name that can be either an array member or a name but the last dot should always be followed by a name. The syntax follows the naming and hierarchy of data in QUIC project files as mentioned in section 2.1.7. Constraint variables are parsed using the rule in lines 104-107 in appendix-B.1.

    Examples of constraint variables:

    * quBuildings.buildings[0].xfo

      classname: quBuildings

      array-member: buildings[0]

      name: xfo

    * quSimParams.nx

      classname: quBuildings

      name: nx

    * quBuildings.buildings[0]

      The above assignment is invalid as it ends with an array member but not a name.

Control variables and constraint variables are together called "**OPT parameters**".

- rvalue (lines 71-78 in appendix-B.1) can be one of the following:
  - **number** - an integer or a decimal number. Internally they are stored as C++ double data type. e.g. 1, 2.8. The rules for parsing numbers are presented in lines 6-9 in appendix-B.1.
  - **string** - a sequence of characters enclosed in single quotes, e.g. 'a simple string'. The rules for parsing numbers are presented in lines 124-126 in appendix-B.1.
  - **array** - a set of numbers or strings enclosed in square braces separated by white spaces. e.g. [1 2 3]. The rules for parsing numbers are presented in lines 120-123 in appendix-B.1. Array type can be used for explicitly specifying a set of values that an entity in QUIC Projects can take. For example, the line *quBuildings.buildings[0].xfo=[1 20 100]* says that the first building in a QUIC domain can be at positions 1, 20 and 100 along the X-axis.
  - **range** - a range of numbers represented as [min:step:max]. The notation represents the series min, min+step, min+2*step, ... max (inclusive). e.g. [1:2:10]. The rules for parsing numbers are presented in lines 116-119 in appendix-B.1. The range data type needs a special mention as it is the mostly used datatype. It allows specifying a range of values with a specific interval for an entity in a QUIC Project without having to explicitly mention each value the entity can take. For example, it is possible to vary the height of a building from say, 10 to 100 in steps of 10 using the range [10:10:100].
    e.g. *quBuildings.buildings[0].height=[10:10:100]*.

The parser returns each of the OPT parameters defined in the OPT file as a **Named OPT parameter (NOP)**. Listing-3.1 shows the class structure of named OPT parameters. Clients

Listing 3.1: Named OPT parameter class structure

```
class named_opt_param{
    private:
     std::string value;
    public:
        std::string name;
        std::string value_type;// single || range ||
                     //numeric_array || string_array
        std::string var_type;//control || constraint
        void set_value(string str);
        void get_value(double& num);
        void get_value(string& str);
        void get_value(vector<double>& array_num);
        void get_value(vector<string>& array_str);
        void get_value(unordered_map<string, double>& range);
};
```

of the OPT file reader module must use the `var_type` and one of the `get_value` methods to get the value of an OPT parameter in the required format. All the NOP objects are further organized into a symbol table. The symbol table is a C++ map of map of NOP's (map<string, map<string, vector<named_opt_param>>> symbol_table). NOP's in the symbol table are organized into a hierarchy as shown figure 3.5, so that NOP's for a particular category can be retrieved easily. Symbol table will be discussed in future sections as it is used in population generation(section-3.1.2), preparing QUIC Project for simulations (section-3.2) and Fitness functions (section-3.2.1).

Optimization File Contents

An OPT file must contain a path to a directory containing QUIC Project files and constraints on various entities in the chosen QUIC Project. It may also contain options for controlling the way simulations are performed and results are collected. Listing-3.2 shows the contents of a sample OPT file.

Following are the required contents of an OPT file:

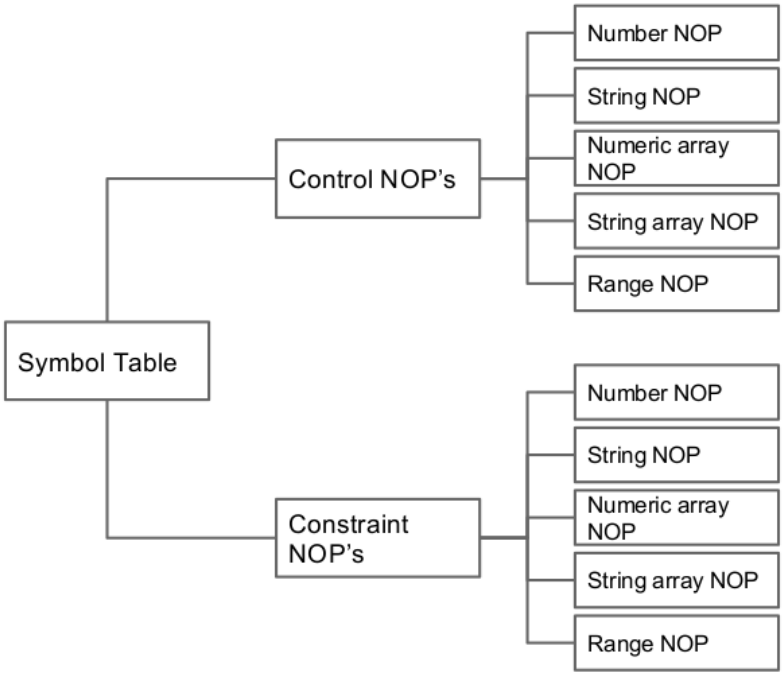

Figure 3.5: Structure of the symbol table generated by OPT FILE Reader.

Listing 3.2: Sample OPT file

```
1  const JOBTYPE = 'lsm'
2  const BASEPROJECTPATH = '2by2_q572_270/2by2_q572_270_inner'
3  const  SOLVER = 'BruteForce'
4  //Building 0 x-offset is fixed at 24
5  quBuildings.buildings[0].xfo = 24
6  //Building 0 y-offset can be either 37 or 40
7  quBuildings.buildings[0].yfo = [37 40.0]
8  //Building 1 x-offset can be any where between 37 and 40
9  // in steps of 2 i.e., 37, 39
10 quBuildings.buildings[1].xfo = [37.0:2.0:40.0]
```

- JOBTYPE - It is a control variable of type string. It is used to represent the type of simulation to be performed (e.g. svf(Sky View Factor), lsm(Land Surface Modeling) etc).

- BASEPROJECTPATH - It is also a control variable of type string. It represents the

location of the QUIC project files on which simulations are to be performed. Since process on different machines perform the simulations, each of the processes requires access to a copy of the base QUIC Project. The current implementation assumes the existence of the specified QUIC Project on a shared file system which is accessible by all the processes.

- Simulation constraints - Multiple lines representing constraints on different simulation parameters (entities in a QUIC domain whose properties are changed across different simulations) in the QUIC Project specified through BASEPROJECTPATH. These are constraint variables and can take numeric datatypes: number, array of numbers and range as their values. The framework currently does fully support specifying constraints on all entities in QUIC Projects. Section-2.1.7 describes the currently supported entities for which constraints can be specified. In listing-3.2 lines 5,7 and 10 represent the simulation constraints of the shown OPT file.

Following are some optional contents of an OPT file:

- SOLVER - The type of solver to use (e.g. brute force, gradient decent etc). This is used by the master process in deciding how optimization is computed. The current implementation of the framework uses a *bruteforce* solver. This variable is provided as a place holder for the future when other sophisticated solvers, like genetic algorithms, are available. SOLVER is a control variable and takes a string value.

- COLLECTION_BOX - As mentioned in 2.1.7, it is difficult for users to know the exact patches in an area of interest (collection box) to collect a simulation's results (e.g. the average temperature on a set of patches). Users can specify a collection box (a cuboid) in the QUIC domain by using the control variables collectionbox_min_X, collectionbox_min_Y, collectionbox_min_Z, collectionbox_max_X,

collectionbox_max_Y and collectionbox_max_Z which represent the min and max grid aligned coordinates as shown in figure 3.6. Coordinates not specified are defaulted to 0. The framework provides a utility method (shown in appendix-D.1) for clients to retrieve patches in the specified collection box.

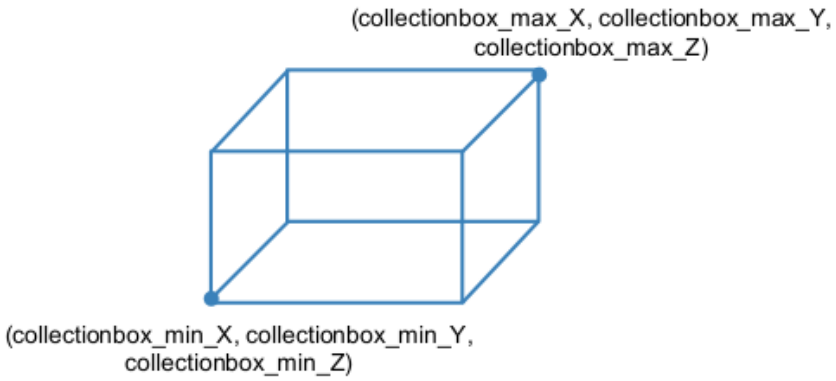

Figure 3.6: Coordinate positions of a collection box.

3.1.2 Population Generator

The population generator module generates different combinations for the constraints specified for various simulation parameters (entities in a QUIC domain across simulations) in the input OPT file. Each individual combination is called a **Sample** and a collection of samples is called a **Population**. Since it is only possible to generate combinations with multiple values for a parameter, only those parameters whose constraints are specified as arrays or ranges are only used for population generation. Each sample internally is a vector of numbers in the combination it represents and a set of fitness values for the sample. The numbers in the vector are ordered such that parameters whose constraints are specified as ranges come first and in the order they are defined in the input OPT file followed by those represented as arrays in the order specified in the opt file. The population generator used

depends on the *SOLVER* option mentioned in 3.1.1. For example, the order of parameter for all combinations generated using the OPT file in 3.2 is as follows:

$(quBuildings.buildings[1].xfo,$

$quBuildings.buildings[0].xfo,$

$quBuildings.buildings[0].yfo)$

It is useful to know this ordering as it is used to uniquely identifying samples in database. Samples generated for the constraints in the OPT file in listing-3.2 using a brute force approach are as follows[1]:

- *(37, 24, 37)*
- *(39, 24, 37)*
- *(37, 24, 40)*
- *(39, 24, 40)*

By default, a brute force approach is used and it generates a population with all possible combinations for the constraints specified in OPT file. The brute force population generation algorithm used is the same as in Vegesna [23]. The brute force generator fetches constraints specified as arrays and ranges from the symbol table created in the OPT file reader phase (3.5) to generate the combinations. The brute force approach does not handle the case where single numeric values are given for all the simulation parameters. This cases is handled by the framework by using a new population generator that generates a population of 1 sample and marks it with a boolean flag to distinguish it from the samples generated by the brute force algorithm. The sample vector is represented by the single numeric values of the parameters in the order the parameters are specified in the OPT file. Listing-3.3 shows an OPT file in which all simulation constraints are given single numeric

[1] fitness values for the samples are omitted as they are not present until a sample is simulated

values. The sample generated from the OPT file in listing-3.3 is represented as (14.0, 50.0, 29.0, 50.0, 29.0, 35.0, 14.0, 35.0).

Listing 3.3: OPT file representing the case where all simulation parameters are given single numeric values.

```
1  const JOBTYPE = 'lsm'
2  const BASEPROJECTPATH = '2by2_q572_270/2by2_q572_270_inner'
3  const SOLVER = 'BruteForce'
4  quBuildings.buildings[0].xfo = 14.0
5  quBuildings.buildings[0].yfo = 50.0
6  quBuildings.buildings[1].xfo = 29.0
7  quBuildings.buildings[1].yfo = 50.0
8  quBuildings.buildings[2].xfo = 29.0
9  quBuildings.buildings[2].yfo = 35.0
10 quBuildings.buildings[3].xfo = 14.0
11 quBuildings.buildings[3].yfo = 35.0
```

Changing the second constraint in listing-3.2 to quBuildings.buildings[0].yfo=[37.0:1.0:40.0] results in 8 samples. A small increase in the constraints potentially results in an exponential increase in the number of samples. Clients of the framework (programmers porting QUICEnvSim to MPI QUIC) can also implement their own population generators such as, genetic algorithm based generators (which will be supported in the future) as long as they implement the following interface shown in listing-3.4. As seen in listing-3.4 the method *generate_population* can take a C++ lambda (requires C++11) predicate that acts as a filter function. The filter function can be used to filter samples during the generation based on a predicate. An example predicate could be to check if the sample being generated is already simulated or not. Listing-3.6 shows a filter function currently being used in the framework. The parameter `vector<population>* seed` is useful for population generator that require a seed population like genetic algorithm based generators.

Listing 3.4: Interface to be implemented by population generators.

```
1  class popgen{
2    public:
3      virtual unique_ptr<population> generate_population
4      (vector<population>* seed,
5         std::function<bool(sample& s)>* filter_fn=nullptr)=0;
6      virtual ~popgen(){}
7  };
```

3.1.3 Population Distribution & Results Aggregation

The Population generated in the population generator (3.1.2) is to be shared among the slaves. Dividing the population equally among the slaves might lead to poor load balancing in the cluster as all the nodes in the cluster may not have equal computational resources. The population is divided into small chunks and a chunk is distributed to each slave initially. After the initial distribution, remaining chunks are given to the slaves as soon as they submit fitness values for the samples in the chunk previously received. This way a powerful node submitting results quickly gets more chunks and a slow node will not slow the whole system while other finish their work. Thus chunk size plays a key role in load balancing the cluster. Choosing a small chunk size incurs network overhead while choosing large chunk size leads to imbalanced load across the cluster. Choosing an optimal chunk size depends on the nodes in the cluster and on its network topology. Section-4.1 discusses an experiment for choosing an optimal chunk size for a given cluster. The chunk size to be used can be provided to the framework through the command line argument `--tilesize`.

3.1.4 Database

As mentioned in section 2.1.6 the framework achieves fault-tolerance using a database. The database layer is implemented using the data access object (DAO) pattern. This separates the framework from the specifics of the database code making it easier to switch

Listing 3.5: Virtual methods in DAO interface

```
1   virtual string insert_optFile(std::string& opfile_path)=0;
2   virtual string get_optFile(string& SHA1)=0;
3   virtual void insert_sample(string& SHA1, sample& s)=0;
4   virtual void insert_population(string& SHA1, population& pop)=0;
5   virtual bool sample_exists(string& SHA1, sample& s)=0;
```

Listing 3.6: Filter function currently being used in population generation

```
1 std::function<bool(sample& s)> pop_db_filter =
2   [&opt_SHA1, &db_instance, &population_in_db](sample &s) -> bool{
3     auto exists = db_instance->sample_exists(opt_SHA1, s);
4       return exists;
5 };
```

the database or improve the efficiency of database code without the framework getting affected. Listing-3.5 shows the interface that every implementation of the DAO layer must implement.

The method `insert_optFile` takes a OPT file and stores it in the database. It returns the SHA-1 hash of the file. The hash returned is used to distinguish samples from different OPT files. The function `sample_exists` checks if a sample is stored in the database. If the sample exists in database its fitness values are retrieved. An implementation using MongoDB is currently being used. The current framework uses a filter function (shown in listing-3.6) during the population generation (3.1.2) that checks if the given sample exists in the database i.e, the sample is already simulated and its fitness values are stored in the database.

3.1.5 Optimization

After simulations are performed and fitness values are assigned to all the simulations i.e. fitness values for the simulations are stored in the corresponding samples representing

the simulations, the total population is passed to the optimization module. The optimization module is implemented as a plugin and can be switched with different plugins. Some examples of optimizations include finding out simulations giving maximum, minimum temperatures and minimum average temperature in a given patch area etc. The framework supports assigning multiple fitness values to a simulation and hence it is possible to optimize on multiple criterion. The multiple fitness values for a simulation are stored as key-value pairs in the sample corresponding to the simulation. An example of such multi-objective optimization could be to find an urban setting where temperatures are minimum and wind flow is maximum. In the case of genetic algorithm based population generators, it is possible to send the optimum samples back to population generation phase as a seed for generating next generation population. The current implementation of the framework does not standardize optimization library hence clients of the framework (programmers porting QUICEnvSim to MPI QUIC) must provide ad-hoc implementations for various optimizations algorithms and fitness values. Listing-3.7 shows the optimization function used for testing the framework. The optimization function uses two fitness values: minimum average temperature (represented by the key *min_avg_temp*) and minimum temperature (represented by the key *min_temp*). The optimization function first sorts the samples based on minimum average temperatures and then sorts the resulting top 20 samples based on the minimum temperature. This way the top 20 urban settings with minimum average temperature and minimum temperatures are found.

3.2 Slave Process

The main role of the slave processes in the framework is to run simulations corresponding to various samples and return fitness values to the master as shown in the flow chart 3.3. This section presents detailed notes on the workings the slave processes. The master

Listing 3.7: Optimization function used for testing the framework

```
void Optimization::optimize(population& results,
                 population& optimizedResults){
    population min_samples;
    std::sort(results.begin(), results.end(),
     [](const sample& a, const sample& b) -> bool{
        return a.fitness_values.at("min_avg_temp") <
              b.fitness_values.at("min_avg_temp");
    });
  for(int i=0; i<results.size() && i<20 ;i++){
        optimizedResults.push_back(results[i]);
    }
    std::sort(optimizedResults.begin(), optimizedResults.end(),
     [](const sample& a, const sample& b) -> bool{
        return a.fitness_values.at("min_temp") <
                 b.fitness_values.at("min_temp");
    });
}
```

sends the required information for running simulations to the slaves in the form of various MPI messages. Each slave runs in a continuous loop waiting for different messages from the master. The following are the different kinds of messages a master process can send to a slave:

- STOP - Instruct a slave to stop with an exit code of EXIT_SUCCESS.
- ERR_STOP - Instruct a slave to stop with an exit code of EXIT_FAILURE. This is used in case of error conditions.
- OPT_PARAMS - Used by master to send the OPT params symbol table to a slave.
- POPULATION_CHUNK - Used by master to send a chunk of samples to a slave.

Before sending any *POPULATION_CHUNK* message the master will send an OPT_PARAMS message. The *OPT_PARAMS* message includes an object of class `opt_params` that carries the symbol table that was generated during the OPT file generation phase (3.1.1). Each

Listing 3.8: Virtual methods in job class

```
1 virtual bool job::setup();
2 virtual bool job::teardown();
3 virtual bool job::eval(sample& s)=0;
```

Listing 3.9: Code snippet showing slave processes execute the life cycle methods of the job class.

```
1  while(true){
2  /*slave initialization and error messages handling*/
3  else if(status.tag() == OPT_PARAMS){
4    /* store opt params into variable optParams */
5    job = new JOB_CLASS(optParams);
6    if(!job->setup()){
7      exit(EXIT_FAILURE);
8    }
9  }
10 else if(status.tag() == POPULATION_CHUNK){
11   /* store population received into variable pop */
12   for(sample& s :pop){
13     job->update_quicdata_with_sample(s);
14     if(!job->eval(s)){
15       exit(EXIT_FAILURE);
16     }
17   }
18   /*Send population with fitness back to the master process*/
19 }
20 /*received exit signal from master*/
21 job->teardown();
```

slave then creates an object of class `job` that represents the type of simulations (e.g. sky view factor, land surface modeling etc). The `job` class is a polymorphic class for which clients of the framework (programmers porting QUICEnvSim to MPI QUIC) must provide concrete implementations. Virtual methods of the `job` class shown in listing-3.8 represent the life cycle of a job.

Listing-3.9 is a code snippet showing slave processes execute the life cycle methods of the job class. The `setup` method (line-6 in listing-3.9) is called before any simulation is

Listing 3.10: Default implementation of setup() in job class

```
1  bool job::setup(){
2    //baseproject_inner_path is created from BASEPROJECTPATH variable
3    // in the input OPT file
4      bool environment_ready;
5      if(baseproject_inner_path.compare("")!=0){
6          job::load_quicdata_from_quic_project_files(
7            baseproject_inner_path, quqpData);
8          environment_ready = true;
9      }
10     else
11     {
12         environment_ready = false;
13     }
14     return environment_ready;
15 }
```

performed. It is called after receiving the OPT_PARAMS message since each job needs an object of class `opt_params` for its initialization. The method `teardown` (line-21 in listing-3.9) is executed just before the slave quits. This gives an opportunity for the concrete implementations of the `job` class to perform simulation dependent setup operations before any simulation is performed and cleanup operations before exiting. The default implementation of `setup` as shown in listing-3.10. The utility method `job::load_quicdata_from_quic_project_files` (line-6 in listing-3.10) reads in the QUIC Project files from the path represented by the argument `baseproject_inner_path` (represented by *BASEPROJECTPATH* in the symbol table received in *OPT_PARAMS* message) and creates an in-memory representation in the argument `quqpData`.

Implementations must specify the concrete class name and its include file in the root CMakeLists.txt file with variable names JOB_CLASS and JOB_CLASS_INCLUDE. The class name supplied using the variable JOB_CLASS is instantiated (line-5 in listing-3.9) and its implementations of the life cycle methods are used. After receiving the *OPT_PARAMS* message, each slave receives zero or more *POPULATION_CHUNK* messages. Each *POP-*

Listing 3.11: A sample implementation of eval() method

```
1  bool qes_lsm_job::eval(sample &s){
2    qes::QESContext context;
3    qes::QESSurface lsm;
4    if( !context.joinModel( &lsm ) ){ return false; }
5    if( !loadScene( &context ) ){ return false; }
6    if( !context.initialize() ){ return false; }
7    context.getSunTracker()->setTimeLocal( 14, 0, 0 );
8    qes::SunTracker *g_sunTracker = context.getSunTracker();
9      if ( !context.runSimulation() )
10     {
11       cout<<"Bad simulation"<<endl;
12         return false;
13     }
14     return fitness->eval_fitness(s, &context);
15 }
16 bool qes_lsm_job::loadScene( qes::QESContext *context ){
17    qes::SceneTracker *g_sceneTracker =
18                    context->getSceneTracker();
19    qes::VariableTracker *g_varTracker =
20                  context->getVariableTracker();
21    g_sceneTracker->setUseAircells( false ); // no aircells
22    if( !g_sceneTracker->initScene( &quqpData, "" ) ){
23     std::cout << "\n**Error building scene!\n" << std::endl;
24     return false;
25    }
26    return true;
27 }
```

ULATION_CHUNK message contains a chunk generated during the population distribution phase (3.1.3). For every sample in the population chunk received, the in-memory representation of QUIC Project is updated (line-13 in listing-3.9) with the values of various simulation parameters present in the sample and `eval` function is invoked (lines-14 in listing-3.9). The `eval` function contains implementation dependent simulation code (mostly code running a single QUIC EnvSim simulation) that runs simulations on the in-memory representation of QUIC project. Listing-3.11 shows a sample implementation to run the Simple Land Surface Model simulations that is modified from one of the test cases of QUICEnvSim.

3.2.1 Fitness Functions

Fitness values are used for quantifying samples for optimization. They provide a way to compare the simulation results obtained after simulating the samples. The current framework supports assigning multiple fitness values to a sample to allow for multi-objective optimizations which will be supported in the future. The fitness function module is implemented as a shared library. As mentioned in 2.1.4, using shared libraries it is possible to avoid long compilation time and possible to switch libraries with out having to compile the program again. The current implementation of the framework requires users to call the user defined fitness functions in their *eval* method (listing-3.8) implementations. To facilitate this, the framework provides an instance of the fitness function library as the instance variable `fitness`. The framework also requires the implementations to specify the location of the shared fitness function library and it's include file through the CMake variables *FITNESS_FUNCTION_LIBRARY* and *FITNESS_FUNCTION_LIBRARY_INCLUDES*.
This, along with the ability to call fitness functions by themselves, gives the flexibility for implementations to choose specific computations for the fitness function library. A default interface shown in listing-3.12 has been provided.

Listing 3.12: Default fitness function library interface

```
1 class Fitness{
2   public:
3   Fitness(QUICProject& _quqpData, opt_params& _optParams);
4   virtual bool eval_fitness(sample& s, qes::QESContext* context);
5   virtual void fetchBuffers(qes::QESContext* context);
6   virtual void fetchPatchIds(qes::QESContext* context);
7 };
```

The implementation of the method `fetchPatchIds` provides patches in the collection box specified by users using the input OPT file. The method also takes care of the fact

the grids in QUIC domain may be scaled, thus facilitating users to specify the coordinates of the collection box in real world coordinates. The implementation of the method `fetchPatchIds` is shown in appendix-D.1.

Users are required to fill in definitions for the other 2 methods, `fetchBuffers` and `eval_fitness`. Sample implementations of the methods for the simple LSM simulations are shown in listing-3.13. QUICEnvSim stores the results of a simulation in various GPU buffers indexed by patch id's and provides API for fetching the buffers from the GPU based on the buffer names. For example, in order to retrieve temperatures computed for various patches, the buffer name *patch_temperature* must be used. Currently, knowledge of various buffer names is required in order to fetch the appropriate buffers for appropriate simulations. The users of the fitness function library can be relieved of this responsibility by defining an interface once all QUIC EnvSim models are ported to MPI QUIC. The method `fetchBuffers` is used for fetching various buffers for the patches in the collection box, which are retrieved by calling the method `fetchPatchIds`. Users with the knowledge of other buffer names can retrieve the buffers as shown in lines 6-9 in listing-3.13. The fetched buffers are stored in the instance variable `buffers` and is accessible to other methods of the `Fitness` class. The `eval_fitness` method shown in lines 35-41 in listing-3.13 shows fetching buffers for the patches in the collection box, evaluating two fitness functions (C++11 lambdas) `min_temp_fitness` (lines 12-23 in listing-3.13) and `avg_temp_fitness` (lines 25-33 in listing-3.13) and storing the resulting fitness values in the sample that is simulated which is passed as a parameter to the `eval_fitness` method. Multiple fitness values can be stored in a sample as key-value pairs. These fitness values will used for comparing samples during optimization (section-3.1.5). Users can use the code in listing-3.13 as a template for defining various fitness functions, storing fitness values and fetching various buffers etc.

Listing 3.13: Sample implementations of the default fitness function library

```
1
2  void Fitness::fetchBuffers(qes::QESContext* context){
3    qes::BufferTracker *g_buffTracker = context->getBufferTracker();
4    qes::SceneTracker *g_sceneTracker = context->getSceneTracker();
5    PatchMap *g_patchData = g_sceneTracker->getPatchData();
6    std::vector<float> temperature;
7    string buffer_name = "patch_temperature";
8    g_buffTracker->getBuffer<float>( buffer_name, &temperature );
9    buffers[buffer_name] = temperature;
10 }
11
12 std::function<double(Fitness& f, sample& s)> min_temp_fitness =
13  [](Fitness& f, sample& s) -> double{
14   float minTemperature = 9999;
15   //int minPatchID=-1;
16   string query = "patch_temperature";
17   for(auto patchID: f.patchIDs){
18     float temp = f.buffers[query][patchID];
19     if(minTemperature > temp)
20       minTemperature = temp;
21   }
22   return minTemperature;
23 };
24
25 std::function<double(Fitness& f, sample& s)> avg_temp_fitness =
26 [](Fitness& f, sample& s) -> double{
27   float avgTemperature = 0;
28   string query = "patch_temperature";
29   for(auto patchID: f.patchIDs){
30     avgTemperature += f.buffers[query][patchID];
31   }
32   return avgTemperature/f.patchIDs.size();
33 };
34
35 bool Fitness::eval_fitness(sample& s, qes::QESContext* context){
36     fetchPatchIds(context);
37     fetchBuffers(context);
38     s.fitness_values["min_temp"] = min_temp_fitness(*this, s);
39     s.fitness_values["min_avg_temp"] = avg_temp_fitness(*this, s);
40   return true;
41 }
```

4 Results

This section presents the various experiments that were performed to test the functionality of the framework. Tests were conducted to check the scalability of the framework and the capability of the framework for running large numbers of simulations. An experiment was also performed to see the effects of chunk size on the performance of the framework. For all the experiments conducted, an implementation running QUICEnvSim's simple Land Surface Model (simple LSM) simulations was used. The simple LSM model is a climatic model that helps in understanding the heat absorption and reflection in a QUIC domain. The model can be used to calculate the temperature on various patches of buildings. The simulation are performed on the *2by2_q572_270* QUIC domain. It is a simple domain with 4 buildings. A small domain was chosen to run the experiments as it is easier to reason about small domains. Figure 4.1 shows a rendering of the *2by2_q572_270* domain.

4.1 Experiment 1: Finding Optimal Chunk Size

As mentioned in section 3.1.3, all the samples that are to be simulated are divided into small chunks for load balancing the system. Tiny chunks result in more network traffic potentially resulting in more time to be spent in network communication than simulating the samples. The optimal values for chunk size depends on the number of nodes in the cluster as well as the network topology of the cluster. Hence, it is recommended to run a small number simulations with varying chunk sizes to find an optimal chunk size for a given cluster. An experiment was conducted with 23 nodes and a load of 2200 simulations. It must

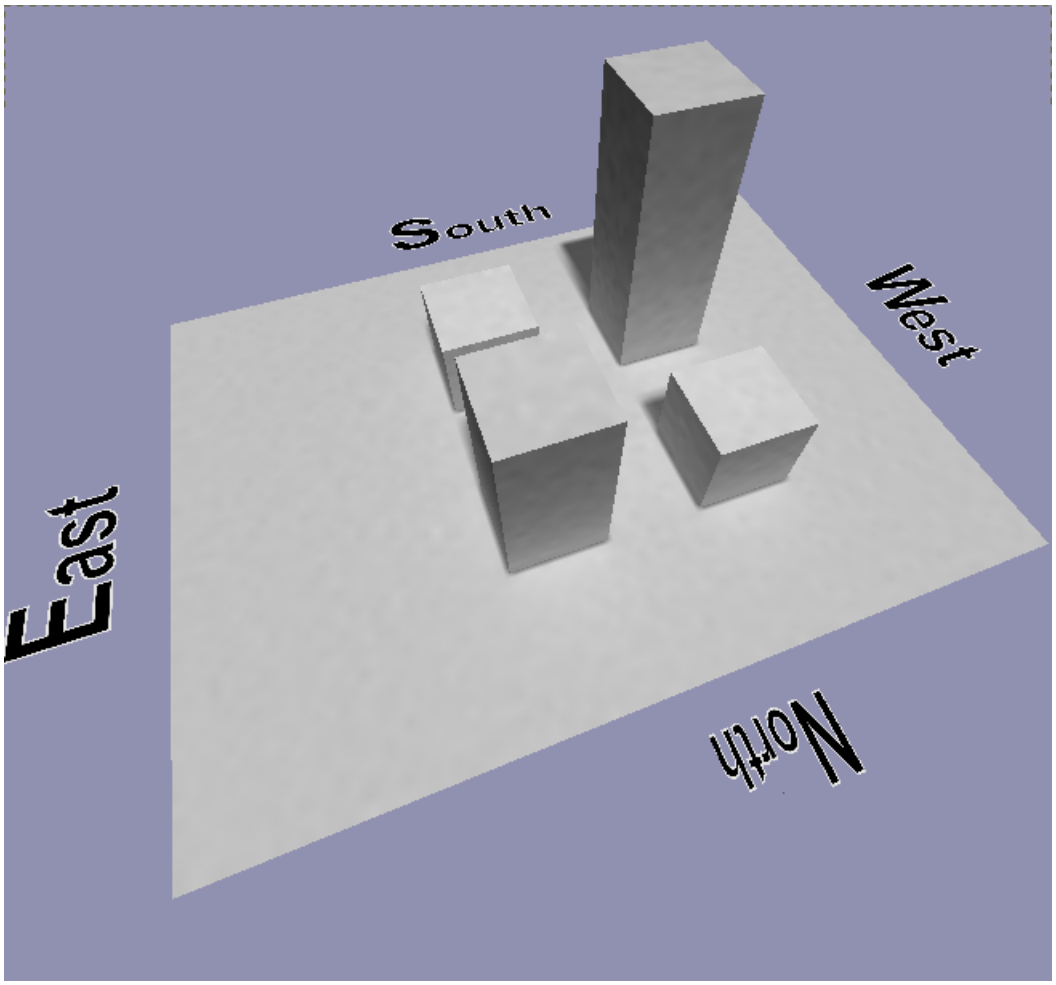

Figure 4.1: Domain represented by the QUIC Project 2by2_q572_270

be noted that of the 23 nodes, only 22 nodes perform the actual simulations, the remaining one node runs the master process which coordinates the other 22 processes. Chunk size is given the values 1, 2, 5, 10, 15, 20, 25 and 50. The results of the experiment as shown in figure 4.2 were quite surprising. The performance decreased with increase in chunk size for the cluster on which the experiment is performed. This could be because the network communication is not very taxing and the nodes in the cluster are heterogeneous. The cluster has different machines with different capabilities as shown in table in appendix-C. Since the network communication is not very demanding, for small chunk sizes machines with

high capabilities fetch and complete more work but for large chunk sizes they are idle after simulating the chunks received while machines that are slow take more time. This results in the decrease in performance of the whole system. But, as this varies with each individual cluster, it could still be worthwhile to run this small experiment before deciding on chunk size.

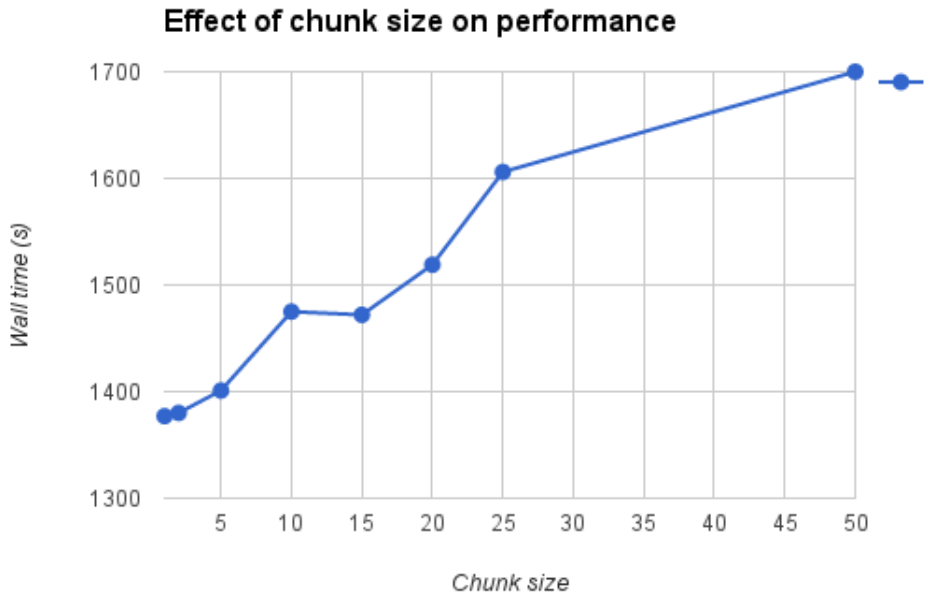

Figure 4.2: Chart showing the results of finding optimal chunk size experiment

4.2 Experiment 2: Testing Scalability

Scalability can be defined as the ability of a system to increase its performance with the addition on computing resources. The scalability test is performed on both homogeneous and heterogeneous cluster setups. To make a fair comparison between the results of the two experiments a random but powerful node (ahti in table-C) is fixed as the master node for both the cluster setups. The initial test is performed on a homogeneous cluster as it is easier

Listing 4.1: OPT file used for running the scalability experiment.

```
1 const JOBTYPE = 'lsm'
2 const BASEPROJECTPATH = '2by2_q572_270/2by2_q572_270_inner'
3 const SOLVER = 'BruteForce'
4 //30 simulations case
5 quBuildings.buildings[0].height = [14.0:1.0:43.0]
```

to reason about. For the homogeneous cluster experiment 10 slow but equally competent nodes (csdev10-19 in table-C) are used as slaves. The reason for choosing a cluster with slow but homogeneous nodes is that the results can be used as a baseline to measure the system performance in a heterogeneous cluster with few powerful nodes.

The first experiment was performed by keeping the number of simulations to be constant at 30 and increasing the number of slave nodes from 1 to 10 in steps of one. The OPT file in listing 4.1 was used for the experiment. For the OPT file in listing 4.1, a population of 30 is generated. A chunk size of 1 was chosen for both the experiments as it is the optimal value for the current cluster as obtained from experiment-4.1. The chart-4.3 shows the

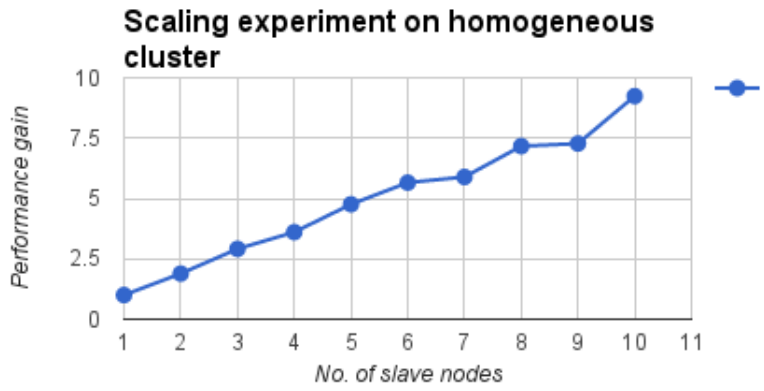

Figure 4.3: Chart showing the results of scalability experiment on homogeneous cluster

results of the experiment. The scalability of the system is measured in terms of performance gain. Performance gain is measured as the ratio of the total time taken with 1 slave node to the total time taken with n slave nodes. The system shows a linear performance gain

for the cases when the total number of simulations is a multiple of the number of slaves in the system as shown in figure-4.4. For understanding the cases when the total number of simulation is not a multiple of the number of slaves, let us consider the case when the number of slave machines in the system is 9. Since the total number of samples is 30, all nodes being homogeneous simulate 3 samples completing 27 simulations. Now for the remaining three simulations, albeit 10 slave nodes are available only three nodes get to simulate the remaining three samples resulting in a decrease in the performance gain. Thus for the cases when the total number of simulations is not a multiple of the number of slaves the performance gain is not linear.

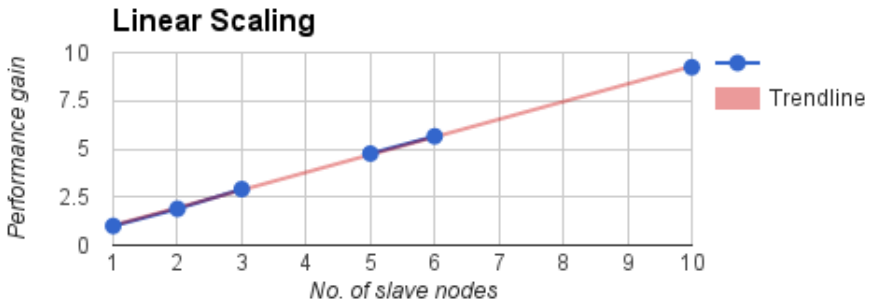

Figure 4.4: Chart showing the results of scalability experiment on homogeneous cluster

For the second experiment 3 nodes from the homogeneous cluster are swapped with 3 powerful (approximately 1.5 powerful) nodes (csdev01, csdev05 and tapio in table-C) to form a heterogeneous cluster. The second experiment is conducted with a load of 300 simulations. The load for the second experiment is increased to make sure work is always available for the faster nodes. Multiple runs are performed by increasing the number of nodes from 1 to 10 but the initial runs are performed on faster nodes. The chart-4.5 shows the results of the experiment.

The red line shows the estimated run times of the homogeneous cluster from the first experiment for the load of 300 simulations. As seen in the chart, as the number of slaves are increased the performance difference between heterogeneous cluster and homogeneous

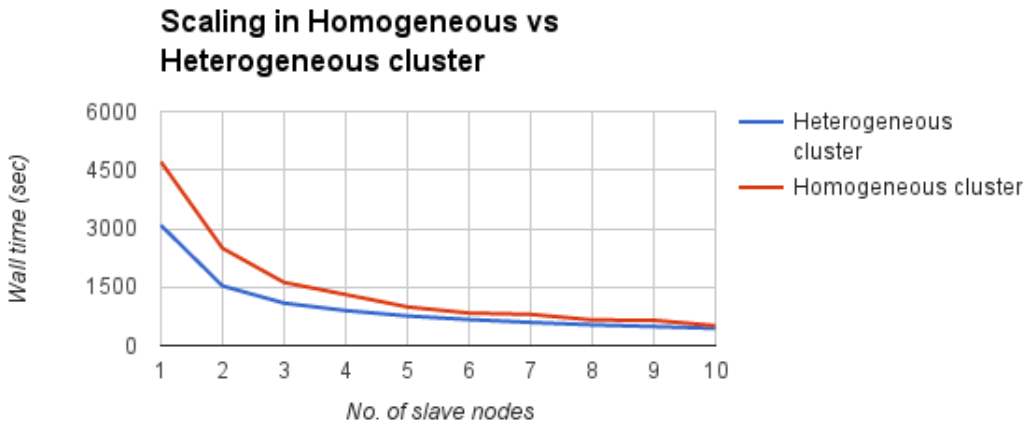

Figure 4.5: Chart showing the results of scalability experiment

cluster decrease. To understand this behavior let us consider the case when all the 10 slave nodes are performing simulations. The number of slow nodes (7) currently in the heterogeneous cluster is 2.333 times more than the number of fast nodes (3) and while the faster nodes are just 1.5 times powerful than the slower nodes. Hence as more number of slower nodes are added they collectively receive more work than the faster nodes. The initial increase in performance is because the number of powerful nodes in the cluster is more than the number of slower nodes. Thus the heterogeneity of a cluster matters only when there are more powerful machines than slow machines.

4.3 Experiment 3: Large Test Cases

This experiment was done to see whether the program is capable of running large numbers of simulations. For the experiment a hypothetical case was considered for the *2by2_q572_270* domain. Imagine a builder who wants to build an apartment complex with a play ground in the middle. People desire to have some shade during a sunny day for kids to play. To meet this requirement the urban planner has to optimize the building positions

such that the temperature on the play ground are minimum. The experiment tries to place buildings optimally so that maximum shade is received in the play ground resulting in cooler temperatures. The experimental layout is as shown in figure 4.6.

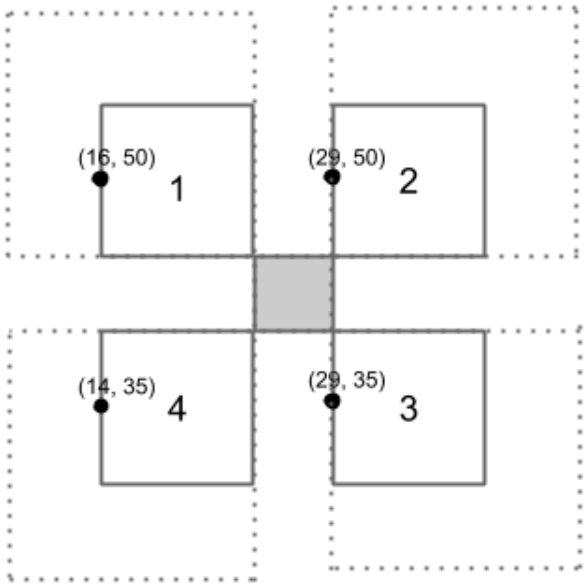

Figure 4.6: Experimental layout for large test cases experiment

The figure 4.6 shows four buildings in their initial positions. The dotted lines represent 4x4 grids in which area building can be placed at any position. For each building, changes can be made to its X-offset(xfo) and Y-offset (yfo) making it a 8D problem with 65536 possible combinations for building positions. The optimization criteria chosen for the experiment is minimum average temperature and minimum temperature with priority for minimum average temperature. Thus for each simulation the fitness functions, `min_temp_fitness` (lines 12-23 in listing-3.13) and `avg_temp_fitness` (lines 25-32 in listing-3.13) are evaluated and stored in the sample corresponding to the simulation being performed. These fitness values are used for optimization as shown in listing-3.7. The simulations when performed with MPI QUIC took 19 hours, 0 minutes and 25 seconds to complete on a cluster of 19 machines.

Listing 4.2: OPT file used for Large test cases experiment

```
const JOBTYPE = 'lsm'
const BASEPROJECTPATH = '2by2_q572_270/2by2_q572_270_inner'
const SOLVER = 'BruteForce'
const collectionbox_min_X = 27
const collectionbox_min_Y = 43
const collectionbox_max_X = 29
const collectionbox_max_Y = 45

quBuildings.buildings[0].xfo = [14.0:1.0:17.0]
quBuildings.buildings[0].yfo = [50.0:1.0:53.0]
quBuildings.buildings[1].xfo = [29.0:1.0:32.0]
quBuildings.buildings[1].yfo = [50.0:1.0:53.0]
quBuildings.buildings[2].xfo = [29.0:1.0:32.0]
quBuildings.buildings[2].yfo = [35.0:1.0:38.0]
quBuildings.buildings[3].xfo = [14.0:1.0:17.0]
quBuildings.buildings[3].yfo = [35.0:1.0:38.0]
```

The Opt file for this experiment is shown in listing 4.2. The collection box represented by (27, 43, 0) and (29, 45, 0) is the play ground in the middle of the apartment complex shown in as a gray area in the figure 4.6. The constraints represent the ability of each building to move in its 4x4 grid.

Figure 4.7 shows the top simulation that maximizes shadow in the play ground. The building placements are (15, 52) for building 0, (32, 53) for building 1, (32, 35) for building 2 and (14,37) for building 3. The red patch in the figure shows the play ground. Albeit, the system was capable of running the 65,536 case the amount of memory being used increased over time, suggesting a memory leak in the system. Manual inspection of code has not revealed any memory leaks in MPI QUIC, further investigations must be made to identify whether the memory leak is in MPI QUIC or QUIC EnvSim. MPI QUIC begin a distributed application and QUIC EnvSim being a complex program makes it difficult to use memory checkers like Valgrind to identify the leaks.

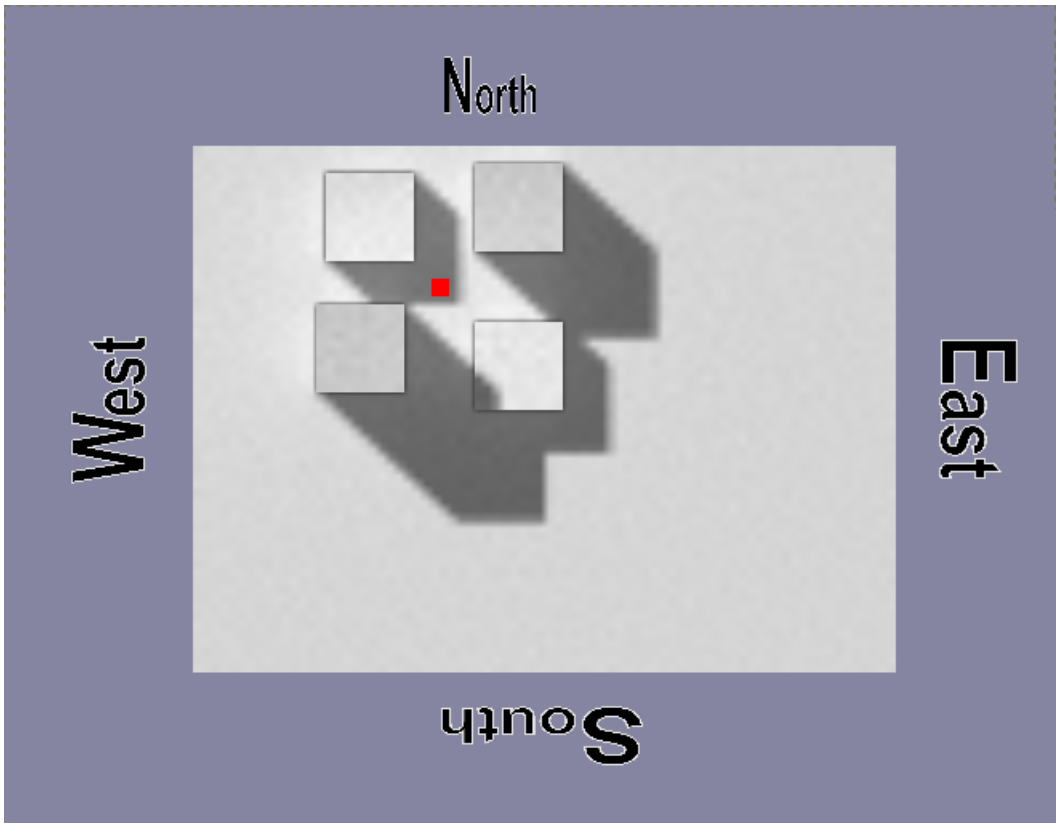

Figure 4.7: Best simulation case

4.4 Experiment 4: Speeding Up Small Size Simulations

The simulations on small domains can be sped up by running multiple copies of the slave programs on the multi core machines in the cluster. For small domains, it is possible to run multiple QUIC EnvSim simulations in the GPU as the GPU resources are not fully utilized. The experiment is performed by running multiple slave processes on a single machine performing simulations on the *2by2_q572_270* which is a small QUIC domain. The result of the experiment are shown on figure 4.8. The experiment is performed by running 10 simulations each time increasing the number of slave processes. As seen from the figure, the performance increased up to 6 processes and then started decreasing. This is because the machine (ahti in table-C) on which the experiment was run has 6 CPU cores

and hence upto 6 processes can be run simultaneously after which point the performance decreases due to increased context switching.

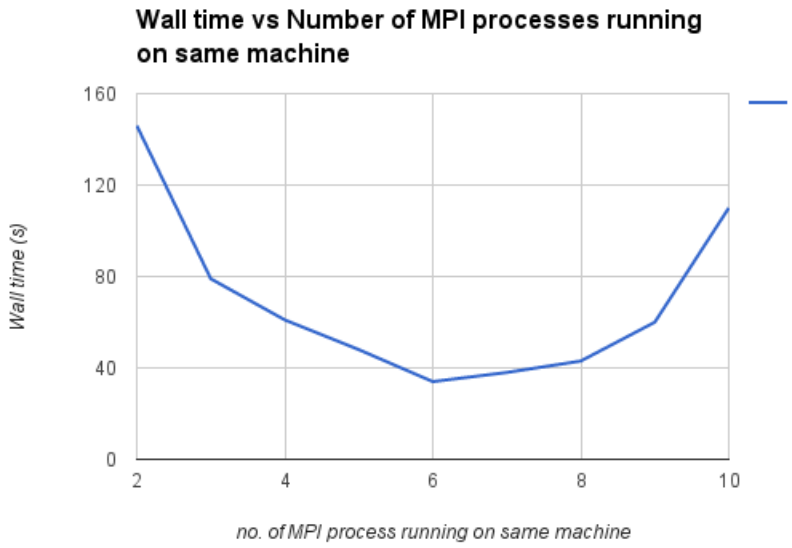

Figure 4.8: Chart showing the results of speeding up small size simulations experiment.

5 Conclusions

A framework capable of running QUIC EnvSim simulations in a distributed setting is presented and evaluated. The framework makes it possible to run large numbers of simulation which would otherwise take days to complete in a matter of hours or less. It has also provided an easy interface for the end users to specify optimization problems. The thesis also includes experiments for fine tuning the performance of the system like choosing optimal chunk sizes, leveraging multi core machines for small QUIC domains etc. Thus, this framework along with QUIC EnvSim can be used by urban planners to take informed design decisions relatively quickly for building environment friendly urban landscapes.

5.1 Future Work

The following are a few points where future work can be made:

- A big boost to the system performance can be obtained by making the chunk size adaptable to the computational ability of slaves. That involves the master process favoring faster nodes than slower nodes.
- The OPT grammar can be made to support simple expressions to make buildings and the collection box move relative to other buildings etc. ANTLR support for C++ is restrictive, alternatives could be investigated.
- The possibility of using OPT file for specifying fitness functions is investigated but this approach is abandoned as it is difficult to translate Matlab code to C++. This

could be revisited using embeddable scripting languages like chai script.

- Currently, it is needed to have a knowledge of how data is organized in QUIC project files to specify constraints on the infrastructure in the opt file. A GUI tool can be made to select buildings and the perimeter within which they can be moved.
- The MongoDB C++ drivers currently in use are developer version. The MongoDB layer code must be appropriately updated after stable versions are released.
- An exciting possibility is to make the framework capable of taking high priority OPT files over the network and return their results immediately.

A Appendix A

A.1 Compiling and Running Boost.MPI Using CMake

Following is the full working version of the program presented in listing 2.1.

```
1  #include <boost/mpi/environment.hpp>
2  #include <boost/mpi/communicator.hpp>
3  #include <boost/serialization/vector.hpp>
4  #include <iostream>
5  using namespace std;
6  const int RESULTS = 1;
7  const int MASTER = 0;
8  const int ARGS = 0;
9  int main(int argc, char* argv[]){
10   //Initialize MPI environment
11   boost::mpi::environment env(argc, argv);
12   //Create the default communicator
13   boost::mpi::communicator world;
14   //Consider process with rank 0 is master and the rest as slaves
15   if(world.rank() == MASTER){ //Master work
16     for(int i=1;i<world.size();i++){
17       cout<<"Master node sending data to worker"<<i<<endl;
18       world.send(i, ARGS, i);
19     }
20     for(int i=1;i<world.size();i++){
21       vector<int> result;
```

```
22        boost::mpi::status msg =
23                        world.probe(boost::mpi::any_source, RESULTS);
24        world.recv(msg.source(), RESULTS, result);
25        cout<<"received "<<result[1]<< " from "<< msg.source()<<"\n";
26      }
27    }else{ //Slaves work
28        cout<<"executing process "<<world.rank()<<endl;
29        int temp;
30        world.recv(MASTER, ARGS, temp);
31        vector<int> result;
32        result.push_back(temp);
33        result.push_back(temp*temp);
34        world.send(MASTER, RESULTS, result);
35    }
36  }
```

The *CMakeLists.txt* file used for compiling the program is shown below. CMake is a build generation tool that is capable of generation different builds like make, eclipse, XCode etc giving portability and ease of use.

```
1
2  cmake_minimum_required (VERSION 2.8)
3  project (mpi_demo)
4
5  FIND_PACKAGE(MPI REQUIRED)
6
7  #BOOST
8  SET(Boost_USE_STATIC_LIBS ON)
9  SET(Boost_USE_MULTITHREADED OFF)
10 FIND_PACKAGE(Boost COMPONENTS mpi serialization REQUIRED)
11
```

```
12 #include_directories
13 include_directories (${Boost_INCLUDE_DIRS})
14 include_directories (${MPI_CXX_INCLUDE_PATH})
15 add_executable(mpidemo mpi.cpp)
16
17 #libraries
18 target_link_libraries (mpidemo ${MPI_LIBRARIES})
19 target_link_libraries (mpidemo ${Boost_LIBRARIES})
```

Assuming the program is saved in a file called `mpi.cpp`, compiling the program using the above CMakeLists.txt results in an executable called `mpidemo`. Before compiling build files required for compiling the program can be generated by issuing the command `cmake .` and then the program can be compiled on a *nix machine by issuing the `make` command.

Programs using mpi must be run using a special program `mpirun`. The syntax of `mpirun` is as follows: `mpirun <options for mpirun> <mpi_executable> <command line parameters for the executable>`.

Some of the important options for mpirun are as follows:

- np - denotes the number of copies of program that must be run
- machinefile - denotes a file which contains the names of the machines one per line, on which the copies of the program must be run on. MPI tries to run all copies on the machines leading to the possibility that a single machine can run multiple copies of a program. This can be restricted by giving the option `pernode` which restricts MPI to run only a single copy of the program on a node.
- x - exports any specific environment variable to all machines before running the program. This is useful for exporting environment variables like `LD_LIBRARY_PATH`.

The above program can be run on 5 machines as follows: `mpirun -np 5 -machinefile <machine-file> mpidemo`.

B Appendix B

B.1 OPT ANTLR grammar

```
/* A number: can be an integer value, or a decimal value */
Number
  :
  ('0'..'9')+ ('.' ('0'..'9')+)?
  ;
num
  :
  Number
  ;
/* We're going to ignore all white space characters */
WS
  :
  (
    ' '
    | '\t'
  )
  {
   //we need white spaces to differentiate separate const keyword from
   //the variable name
  }
  ;
Identifier
```

```
23    :
24    (
25      'a'..'z'
26      | 'A'..'Z'
27      | '_'
28    )
29    (
30      'a'..'z'
31      | 'A'..'Z'
32      | '_'
33      | '0'..'9'
34    )*
35    ;
36  /*
37    Basic unit in opt file language is assignment
38  */
39  unit
40    :
41    (
42      assignment
43      | comment
44    )
45    EOF
46    ;
47
48  assignment
49    :
50    WS* lvalue WS* '=' WS* rvalue WS*
51    ;
52
53  comment
```

```
54    :
55    '//' .*
56    ;
57  /*
58   lvalue can be a control variable (respresented as constants) or local
59   variable or a member of a class
60  */
61  lvalue
62    :
63    constant
64    | local_variable
65    | member_variable
66    ;
67  /*
68   rvalue can be a number (int, float), string, set, range or a simple
69   expression
70  */
71  rvalue
72    :
73    str
74    | numarray
75    | stringarray
76    | range
77    | num
78    ;
79  /*
80    local variable is just an identifier
81  */
82  local_variable
83    :
84    Identifier
```

```
85    ;
86  /*
87    constant variables are control variables that specify various things
88    like solvers etc.
89  */
90  constant
91    :
92    'const' WS+ Identifier
93    ;
94  /*
95    member variable are of the form classname followed by member names
96    which can be either array member (array[2]) or a name but the last
97    dot should always be followed by a name
98    eg: quBuildings.buildings[0].xfo = 23
99              ^           ^          ^
100             |           |          |
101         classname   array_member   |
102                                   name
103 */
104 member_variable
105   :
106   Identifier ('.' vector)* '.' Identifier
107   ;
108 numarray
109   :
110   '[' (Number WS*)+ ']'
111   ;
112 stringarray
113   :
114   '[' (String WS*)+ ']'
115   ;
```

```
116 range
117   :
118   '[' Number ':' Number ':' Number ']'
119   ;
120 vector
121   :
122   Identifier '[' Number ']'
123   ;
124 str
125   :
126   String;
127 String:
128   '\''
129   ~(
130     '\r'
131     | '\n'
132     | '\''
133    )*
134   '\''
135   ;
```

C Appendix C

Node name	Memory (KB)	Processor	GPU
ahti	12294964	Intel® Xeon(R) CPU X5690 @ 3.47GHz × 6	GeForce GTX TITAN X/PCIe/SSE2
csdev01	32847732	Intel(R) Core(TM) i7-5820K CPU @ 3.30GHz x 12	GeForce GTX TITAN X
csdev05	32847816	Intel(R) Core(TM) i7-5820K CPU @ 3.30GHz x 12	GeForce GTX TITAN X
ether	32847716	Intel(R) Core(TM) i7-5820K CPU @ 3.30GHz x 12	GeForce GTX TITAN X
erlik	12295832	Intel(R) Core(TM) i7 CPU 960 @ 3.20GHz x 8	Tesla C2075 x 2
tapio	8090088	Intel(R) Core(TM) i5-4460 CPU @ 3.20GHz x 4	GeForce GTX 750 Ti
csdev02-04, csdev06-09	8066172	Intel(R) Core(TM) i5-4430 CPU @ 3.00GHz x 4	GeForce GTX 660
csdev10-19	8069864	Intel(R) Core(TM) i5-3330 CPU @ 3.00GHz x 4	GeForce GTX 645

Table C.1: Configurations of the machines used for testing the framework.

D Appendix D

Listing D.1: Implementation of the utility function that converts collection box coordinates to patches.

```
1  void Fitness::fetchPatchIds(qes::QESContext* context){
2  //make a list of patches not inside buildings
3    std::unordered_set<unsigned int> patches_not_inside_buildings;
4    qes::SharedResources sr = context->getSharedResources();
5    float3 patchDim = sr.sceneTracker->patchDimensions();
6    ulong3 worldDim = sr.sceneTracker->worldDimensions();
7    BuildingBuilder buildingBuilder( sr.patchData,
8                            sr.buildingData, patchDim, worldDim );
9
10   PatchMap::iterator patch = sr.patchData->begin();
11   for( patch; patch != sr.patchData->end(); ++patch ){
12     if( !buildingBuilder.insideBuilding( patch->second ) ){
13      patches_not_inside_buildings.insert( patch->first );
14     }
15   }
16   //end make a list of patches inside buildings
17
18   //clear patches in the collection box from previous iteration
19   patchIDs.clear();
20
21   //get dimensions of QUIC world
22   float nx = quqpData.nx;
```

```
23    float ny = quqpData.ny;
24    float nz = quqpData.nz;
25    //get the scaling factor between patches in real world vs patches
26    //in QUIC worlds
27    //e.g. 2 units of length in real world might correspond to 1 unit
28    //of length in QUIC world
29    float dx = quqpData.dx;
30    float dy = quqpData.dy;
31    float dz = quqpData.dz;
32
33    if(dx<=0 || dy<=0 || dz<=0)
34      throw "dx or dy or dz negative";
35    //Change real world coordinates to quic world coordinates
36    double y2_temp = y2/dy;
37    double y1_temp = y1/dy;
38    double x1_temp = x1/dx;
39    double x2_temp = x2/dx;
40    double z2_temp = z2/dz;
41    double z1_temp = z1/dz;
42    if(y2<y1 || x2<x1 || y1<0 || x1<0 || y2>ny ||
43                                x2>nx ||z2<z1 ||z1<0 || z2>nz)
44      throw "Collection box out of bounds" ;
45
46    //pull the points to the middle of each cell/patch
47    //this is to gaurentee that the point is on a patch not at
48    // the borders of patches
49    x1_temp = 0.5 + floor(x1);
50    x2_temp = 0.5 + floor(x2);
51    y2_temp  = 0.5 + floor(y2);
52    y1_temp = 0.5 + floor(y1);
53    z1_temp = 0.5 + floor(z1);
```

```
54    z2_temp = 0.5 + floor(z2);
55
56    if(x1_temp==x2_temp){
57      x1_temp-=0.1;
58    }
59    if(y1_temp==y2_temp){
60      y1_temp-=0.1;
61    }
62    if(z1_temp==z2_temp){
63      z1_temp-=0.1;
64    }
65
66    if(x1_temp==0.5&&x2_temp==0.5){
67      x1_temp=0;
68    }
69    if(y1_temp==0.5&&y2_temp==0.5){
70      y1_temp=0;
71    }
72    if(z1_temp==0.5&&z2_temp==0.5){
73      z1_temp=0;
74    }
75
76    for(float i=x1_temp; i<x2_temp; i++){
77      for(float j=y1_temp; j<y2_temp; j++){
78        for(float k=z1_temp; k<z2_temp; k++){
79          float3 point = make_float3( i, j, k );
80          unsigned int patch_id =
81          qes::QESUtils::getNearestPatch(
82                                        point, context->getSceneTracker());
83          if(patches_not_inside_buildings.count(patch_id)>0){
84            patchIDs.insert(patch_id);
```

```
85          }
86          else{
87            cout<<"Found patch inside a building at point x1: "
88                                                  <<i<<" y1: "<<j<<endl;
89            cout<<"Check the Collection box bounds"<<endl;
90            //patches should never be inside buildings
91            exit(EXIT_FAILURE);
92          }
93        }
94      }
95    }
```

Bibliography

[1] D. P. Anderson. ``Boinc: A system for public-resource computing and storage". In: *Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on*. IEEE. 2004, pp. 4–10 (cit. on p. 15).

[2] B. Bailey, M. Overby, P. Willemsen, E. Pardyjak, W. Mahaffee, and R. Stoll. ``A scalable plant-resolving radiative transfer model based on optimized GPU ray tracing". In: *Agricultural and Forest Meteorology* 198 (2014), pp. 192–208 (cit. on pp. 15, 16).

[3] K. A. Briggs, M. Overby, R. Stoll, P. Willemsen, and E. Pardyjak. ``Evaluation of moisture and heat transport in the building-resolving urban transport code quic envsim". In: (In preparation for submission) (cit. on p. 16).

[4] M. J. Brown, M. Nelson, M. Williams, A. Gowardhan, and E. Pardyjak. *The Quick Urban &; Industrial Complex(QUIC) chemical, biological, and radiological agent dispersion modeling system*. Tech. rep. American Meteorological Society, 2010 (cit. on p. 10).

[5] M. P. Forum. *MPI: A Message-Passing Interface Standard*. Tech. rep. Knoxville, TN, USA, 1994 (cit. on p. 4).

[6] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, et al. ``Open MPI: Goals, concept, and

design of a next generation MPI implementation". In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer, 2004, pp. 97–104 (cit. on p. 4).

[7] . *Grand Challenges*. http://en.wikipedia.org/wiki/Grand_Challenges/. [Online; accessed 5-March-2015] (cit. on p. 3).

[8] C. Hill, C. DeLuca, M. Suarez, A. Da Silva, et al. ``The architecture of the earth system modeling framework". In: *Computing in Science & Engineering* 6.1 (2004), pp. 18–28 (cit. on p. 14).

[9] A. Norgren. ``GPU Based Particle Dispersion Modeling with Interactive Visualization Support for Real□time Simulation". PhD thesis. UNIVERSITY OF MINNESOTA, 2008 (cit. on p. 15).

[10] T. Ostromsky, W. Owczarz, and Z. Zlatev. ``Computational Challenges in Large-scale Air Pollution Modelling". In: *Proceedings of the 15th International Conference on Supercomputing*. ICS '01. Sorrento, Italy: ACM, 2001, pp. 407–418. ISBN: 1-58113-410-X. DOI: 10.1145/377792.377898. URL: http://doi.acm.org/10.1145/377792.377898 (cit. on p. 14).

[11] M. C. Overby. ``A High Performance Framework for Coupled Urban Microclimate Models". PhD thesis. UNIVERSITY OF MINNESOTA, 2014 (cit. on pp. iii, 1, 16).

[12] M. Overby, B. Bailey, K. Briggs, A. Vegesna, E. Pardyjak, R. Stoll, and P. Willemsen. *Modeling vegetative heat transfer in urban environments with optix*. Poster GPU Technology Conference 2013. 2013 (cit. on p. 16).

[13] M. Overby, B. Bailey, R. Stoll, P. Willemsen, and E. Pardyjak. ``A highly scalable modeling framework based on gpu technology for simulating radiative transport in

complex urban and plant canopies". In: ESA 2013, Sustainability: Urban Systems, 2013 (cit. on p. 16).

[14] M. Overby, B. Bailey, R. Stoll, P. Willemsen, and E. Pardyjak. ``Simulating radiative transport for vegetation in complex urban environments with green infrastructure". In: AMS 2014, Symposium on the Urban Environment, 2014 (cit. on p. 16).

[15] M. Overby, S. Halverson, B. Bailey, P. Willemsen, R. Stoll, and E. Pardyjak. ``QUIC EnvSim: Radiative heat transfer in vegetative and urban environments with nvidia optix". In: GPU Technology Conference 2014, 2014 (cit. on p. 16).

[16] M. Overby, P. Willemsen, B. Bailey, S. Halverson, and E. Pardyjak. ``A rapid and scalable radiation transfer model for complex urban domains". In: (Submitted for review to Urban Climate, 2015) (cit. on pp. 15, 16).

[17] M. Overby, P. Willemsen, E. Pardyjak, and R. Stoll. ``GPU accelerated surface energy balance computations for urban environment simulation". In: AMS 2015, Symposium on High Performance Computing for Weather, Water, and Climate, 2015 (cit. on p. 16).

[18] T. Parr. ``The definitive ANTLR reference: building domain-specific languages". In: (2007) (cit. on p. 8).

[19] D. A. Randall, T. D. Ringler, R. P. Heikes, P. Jones, J. Baumgardner, et al. ``Climate modeling with spherical geodesic grids". In: *Computing in Science and Engineering* 4.5 (2002), pp. 32–41 (cit. on p. 14).

[20] J. Silcock. *Message Passing, Remote Procedure Calls and Distributed Shared Memory as Communication Paradigms for Distributed Systems* (cit. on p. 4).

[21] B. Singh, E. Pardyjak, A. Norgren, and P. Willemsen. ``Accelerating urban fast response Lagrangian dispersion simulations using inexpensive graphics processor parallelism". In: *Environmental Modelling & Software* 26.6 (2011), pp. 739–750 (cit. on p. 15).

[22] M. Tobis, C. Schafer, I. Foster, R. Jacob, and J. Anderson. ``FOAM: Expanding the Horizons of Climate Modeling". In: *Proceedings of the 1997 ACM/IEEE Conference on Supercomputing*. SC '97. San Jose, CA: ACM, 1997, pp. 1–15. ISBN: 0-89791-985-8. DOI: 10.1145/509593.509620. URL: http://doi.acm.org/10.1145/509593.509620 (cit. on p. 14).

[23] A. Vegesna. ``Optimizing urban environmental simulations using boinc". PhD thesis. UNIVERSITY OF MINNESOTA, 2013 (cit. on pp. 11, 15, 29).

[24] D. W. Walker and J. J. Dongarra. ``MPI: a standard message passing interface". In: *Supercomputer* 12 (1996), pp. 56–68 (cit. on p. 5).

[25] C. Yang, W. Xue, H. Fu, L. Gan, L. Li, Y. Xu, Y. Lu, J. Sun, G. Yang, and W. Zheng. ``A peta-scalable CPU-GPU algorithm for global atmospheric simulations". In: *ACM SIGPLAN Notices*. Vol. 48. 8. ACM. 2013, pp. 1–12 (cit. on p. 14).