Optimizing Urban Environmental Simulations using Boinc

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

Aditya Vegesna

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

Dr. Peter Willemsen

August 2013

Acknowledgements

I would like to thank Dr. Peter Willemsen for his constant guidance and advice. I am indebted to my family for their support, without whom I would not have gotten this far.

Dedication

I would like to dedicate this thesis to my grandmother who brought me up against all odds and supported me with all my decisions.

Abstract

Urban cities are usually densely populated and have massive infrastructure. They consume a lot of energy and generate pollution. Urban form and structure interact with the environment in a complex way. There is transfer of energy between buildings and the ground layer. Winds flow through the urban street canyons, affecting evaporation, temperature and pollution dispersion. The effects of such complex interactions are still not widely known or understood. How well an urban space disperses pollution, or requires energy for heating or cooling is potentially impacted by many components, such as where the buildings are located with respect to each other, which materials the buildings are constructed from, or where trees or parks are placed.

The aim of the Genusis project is to provide a tool for urban planners that they can utilize to understand such impacts and to assist them in taking design decisions accordingly. Even with just a few choices in building locations or tree types the number of possible configurations is vast. Running the simulations on many thousand of these configurations is a huge problem on its own and truly not feasible for urban planners to use in their daily routines.

This thesis strives towards tackling that problem by developing a computational environment in which specifying these configurations is easy and can compute potential solutions to the problems within an acceptable time frame using multiple machines. A simple and yet powerful language is created to let urban planners control the simulations and specify the configurations. In order to reduce the computational time, Berkeley Open Infrastructure for Network Computing (BOINC) is used to harness all available computational resources. Experiments were conducted to analyze the implementation and performance of the system. The results obtained validate the implementation and indicate a significant performance gain.

Contents

List of Tables

List of Figures

1 Introduction

Urban cities are on the rise in the 21st century. They are the sites for technological advancement and financial investment in turn leading to increase in population and infrastructure. A huge portion of the world's population resides in urban cities [25]. Consequently, urban cities have become the centers for air pollution and heavy energy use. Hence people, governments and industries should start analyzing the impact of their actions on the environment and try to minimize their effect on the environment.

A sustainable city is a city designed with the environment in mind, focussing on minimizing the use of energy, water, food and emission of outputs like heat, air pollution, water pollution[24]. City planners and building designers are motivated and obligated to build sustainable cities and buildings.

Environment interacts in many complex ways with urban spaces which are still not widely understood. The sun is the main source of heat that the earth receives and radiates, wind disperses particles such as pollen and dust, water evaporates cooling the surface of the earth. These are few of the many phenomena that interact with the urban space effecting the temperatures within an urban spaces and also pollution levels in and around the urban environments.

Various design considerations and factors of a building have the potential to affect the environment (like building locations, materials used for construction, trees and parks in the area etc). For instance, if a building is in the open, it receives more energy from the sun when compared to a building surrounded by other buildings. Such a building placed in the open may waste more energy in cooling offices and homes.

Similarly if a building is planned and placed in the appropriate location with respect to existing buildings and urban structures by analyzing wind flow patterns, it might minimize the amount of pollution build up in the walking layer of atmosphere around it making it more comfortable for people. Due to pavements and buildings replacing trees and landscapes the temperature of urban landscapes can be warmer than surrounding areas (heat island effect). This temperature buildup can be reduced by planting trees, constructing parks, installing green rooftops and using cool pavements [34][16] .

The hypothesis of the Genusis project [8] is that there exist urban structures that can minimize pollution concentrations and energy use. As a means of evaluating the hypothesis and to better understand the various complex interactions between the environment and the urban form, the Genusis group has developed simulation models. QUIC Energy and GPU Plume are two such models being developed. QUIC Energy is a simulation designed to calculate and visualize transfer of energy from the sun to the earth using radiant energy modeling[29][26]. It models the interactions between buildings, vegetation, air and the ground layer. GPU Plume is a simple Lagrangian dispersion model based on Quick Urban and Industrial Complex (QUIC) Dispersion Modelling System[39] [30] [33]. The model can be used calculate and visual particle concentrations and movement of particles due to wind in urban regions. These tools when given an urban space can be used to estimate the amount of pollution in a block or the amount of energy received from the sun by a certain region. Graphic processing units were used to assist in millions of calculations of complex mathematical equations that need to be solved to estimate particle movement or energy transfer.

Imagine being an urban planner who's been given the opportunity to design a small city block or a community with a bit of flexibility on where buildings can be placed, what kind of trees could be planted, and the choice of whether to place a

park. With just a couple of choices the possible designs are huge in number and this number increases exponentially with more choices. The planner may come up with a few designs and then use these tools to determine which one would be more optimal. To do so, the simulations would have to be run once per every configuration and there is a very good chance the planner might not have even thought about the best one possible.

An experiment (Fig 1.1) was run with 4 cubical buildings of fixed dimensions and 2 particle emitters, where the buildings could be moved 4 grid cells in x & y directions within the quadrants formed by the emitters. The goal was to determine the optimal placement of the four buildings that reduces maximum pollution in the walking layer.The time taken for a single simulation with 640,000 particles was approximately 120 seconds with the visuals [14]. The number of different possible configurations for this problem were 65,536. It was estimated that a single machine would take more than 1100 hours to compute the optimal configuration for the setup. Instead 20 machines of mixed speeds were used to complete the problem over the span of 60 hours. However it involved manually setting up the problem on the 20 different machines with continuous monitoring, results gathering and analysis. Such a task on a larger urban space with more configurations would be laborious.

The aim of the Genusis group is to be able to develop an interactive tool that can assist planners in making design decisions while understanding the various effects on the environment. The goal of this thesis is to provide a computational environment for such optimization tasks. It involves addressing the problems of specifying optimization tasks in a simple and yet powerful way and distribution of these tasks across a cluster of GPUs.

A language was created that could be used to specify the optimization tasks and control various simulation parameters with ease. To provide sufficient computational

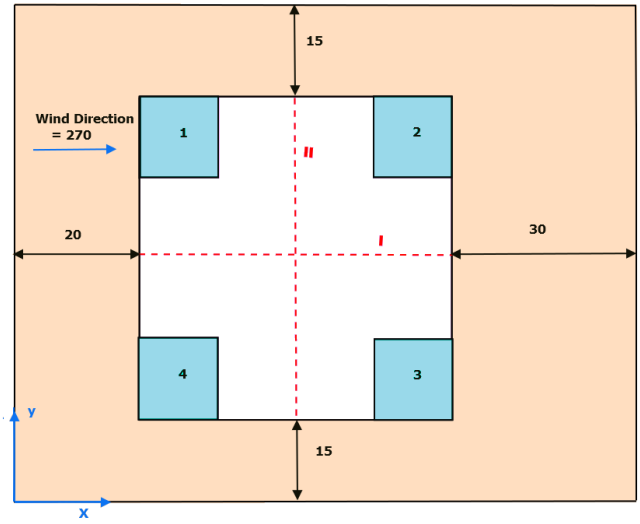

Figure 1.1: The 8D experiment setup. Red dotted lines are the emitters.

power to compute the optimizations, volunteer grid computing is used. A volunteer grid computing network was preferred to traditional grid computing because the latter requires substantial investment in hardware to form the grid. BOINC is an open source, middleware software which provides an easy and extensible framework for developing volunteer grid computing networks [1]. This framework was extended and used along with the language to support optimization of simulations. Using BOINC eliminates the need for manual work and allows willing people to volunteer their system resources to contribute to the project.

Experiments were conducted to validate the language and the computational environment and also evaluate the performance of the system. The results were promising and also helpful in analyzing aspects of the implementation that could be further improved.

The remaining chapters describe the background and related work (Chapter 2), the implementation details related to this thesis (Chapter 3), the results (Chapter 4), and a follow-up discussion and conclusion (Chapter 5).

2 Background

This chapter comprises of two sections the first 2.1 includes details about softwares used and the second 2.2 describes the work related to grid computing and implementation decisions.

2.1 Background Information

This first half of this section describes the various tools, simulations, frameworks used in the implementation. The second half 2.1.5 gives an in depth description of BOINC, its architecture, features and policies.

2.1.1 General Purpose GPU Computing

Graphics Processing Units are dedicated hardware with specialized pipelines for graphics. GPUs were increasingly developed for computer games as they used intricate graphics with high refresh rates. Rendering of such graphics more than often requires a series of single operations to be performed on millions of different data points quickly and dedicated hardware (GPU). To support this, GPUs are highly optimized to support SIMD (Single Instruction Multiple Data) model where a single instruction is run on multiple data simultaneously. Since the past decade, the computing power of GPUs is increasingly being harnessed for general purpose scientific and engineering applications as GPUs tend to have more processing pipeline yielding a good throughput and are growing at a faster rate compared to CPUs.

Nvidia corporation developed a programming model called CUDA [31] which is also a parallel computing framework to facilitate development of code that can run on their GPUs. Using CUDA, computationally intensive calculations can be offloaded to GPUs thereby increasing the speed of an application. Optix [35] is a general purpose ray tracing engine developed by Nvidia and built using CUDA. It enables developers to accelerate the tracing of rays for any application. Optix also has the ability of utilizing all the GPUs present on the machine.

2.1.2 Environmental Simulations

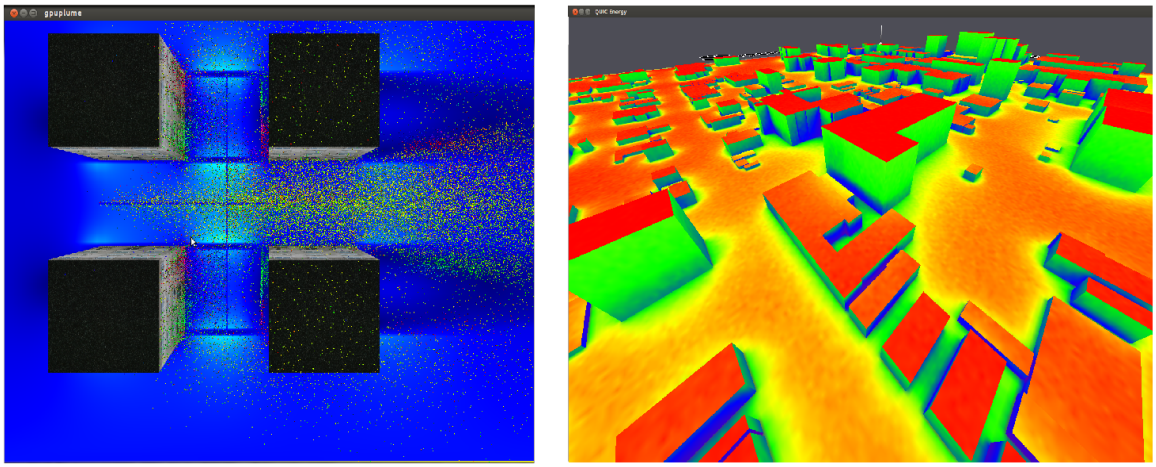

Figure 2.1: (a) A screenshot of GPU Plume running the 8D test case (b) A screen capture of QUIC Energy running a model of salt lake city

Quick Urban Industrial Complex (QUIC) Dispersion Modeling system is a Los Alamos national laboratory's initiative to create a fast response urban dispersion model. QUIC consists of a set of tools some of which are the dispersion model called QUIC-PLUME, a model to compute wind field around buildings called QUIC URB and another called QUIC-PRESSURE.

GPU Plume 2.1(a) is the GPU implementation of QUIC-PLUME with improved performance. QUIC Energy 2.1 (b) is a heat transfer model being developed to model transfer of heat due to short wave radiations from the sun and long wave radiations emitted by the ground and buildings. Vegetation is being added to the model to estimate the effect of trees on heat transfer in urban space. GPU Plume uses CUDA while QUIC Energy uses both Optix and CUDA.

2.1.3 Boost

Boost [13] is an open source collection of libraries for C++ which are free to be used to build commercial or non commercial applications. They are peer-reviewed, portable and usable across a broad spectrum of applications. A few of the libraries were accepted as a part of the latest C++11 standard. The following is the list of classes and libraries used by this thesis.

- Boost Any : A generic type class that supports copying of any values into it [4].
- Boost Regex : A library that provides powerful regular expression based functions [5].
- Boost Filesystem : A library that allows accessing and modifying files and directories [2].
- Boost PropertyTree : A library allows storing values as trees and function to store them as xml files [3].

2.1.4 CMake

Cross-Platform Make (CMake) [38] is an open-source build system. A platform and compiler independent configuration file is used to control the software compilation

process. It can support different directory hierarchies and native build environments like GNU Make, Visual Studio and Apple's Xcode. It creates a build directory structure separate from the source directory tree to facilitate deleting of the build directory while keeping the source files untouched.

2.1.5 Berkley Open Infrastructure for Network Computing

BOINC is a an open-source middleware system originally developed for SETI@HOME as a software platform for distributed computing. The goal was to overcome the lack of computing power to solve complex scientific problems by utilizing hundreds of thousands of volunteered computer resources from all over the world. BOINC is currently being used for grid or volunteer computing. A volunteer can be any individual with an internet connected computer or gaming console willing to allow projects to utilize their resources for computation or storage. Volunteer computing can produce more computational power than any other source due to the sheer number of personal computers and due to the rate at which they are increasing. BOINC has 2,595,771 active users crunching 7,298.855 TeraFLOPS as of June 2013 [21]. BOINC supports a wide variety of platforms (a combination of hardware and software) like any of Windows, Linux, Mac os running on Intel or AMD processors, Android running on ARM, Sony ps3 running Linux and many more [37].

Organizations can gain access to resources by creating their own projects on BOINC. Different projects can utilize BOINC independently by hosting their own servers and maintaining their own databases. However they share resources, in the sense that a single user can contribute his computational resources to multiple projects. Users cannot be held accountable and can choose which projects to contribute to. Most of the projects are non-profit and rely heavily on volunteers for

computation.

Design and Workflow

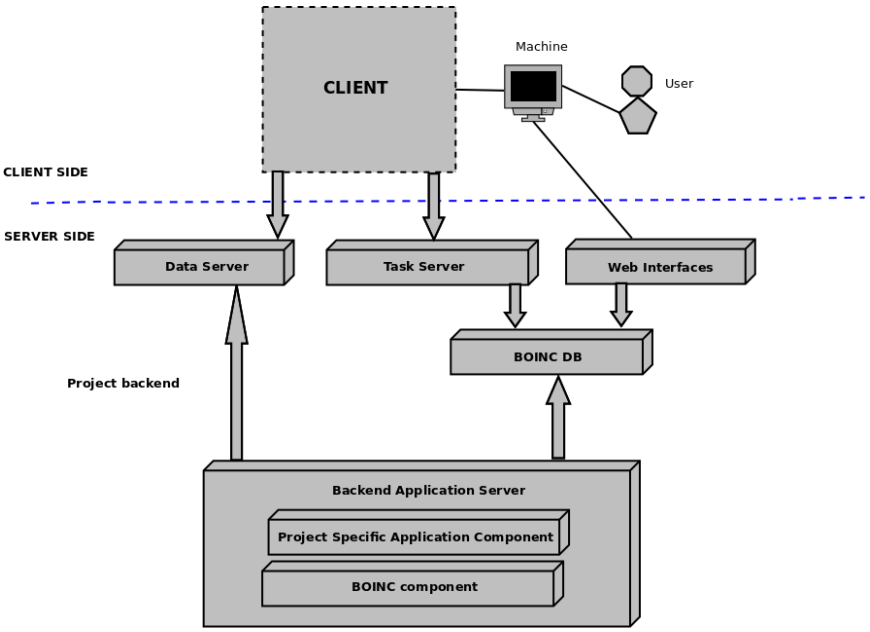

Figure 2.2: BOINC architecture: A client server model (modified version [17])

BOINC follows a client-server model for its architecture. BOINC projects are hosted on BOINC servers which are very efficient and a project's server can be hosted on more than one machine to support large number of volunteers. A BOINC client is installed on a user's machine which is responsible for communicating with various project servers, downloading and managing applications, computing the results and reporting them back. The figure (Fig 2.2) shows the basic components of BOINC and their interactions.

When the BOINC client is running low on work it sends a request to one of the projects to which it is connected to along with its information like platform, RAM available, GPU information. The project's BOINC scheduling server sends instruc-

tions to the client as a reply. Based on the instructions, the client downloads the required input files and application executables from the data server when necessary. It then runs the executable at a suitable time producing output files which are then uploaded to the data server. Based on preferences set by the user the client reports that the work is done and requests for more work eventually. The above process is repeated indefinitely.

BOINC has various useful features [17]: *a*) Redundant computing to deal with erroneous results that occur in public-resource computing either because of faulty computers or intended tampering by users. The project can choose the level of redundancy where a single work unit is sent to different clients and the results are compared and analyzed for consistency; *b*) Credit and accounting, used to assign credit to volunteers for work done by their machines since credit and relative rankings motivate users to donate resources. Credit is assigned based on validity of a result and amount of computation, storage and network used to obtain the result; *c*) Every failure is handled with a backoff strategy to ensure that there is no overload of server resources. Network congestion is avoided by handling all communications with an exponential backoff; *d*) A graphics system to allow applications to display visuals on the monitor during execution of their application. Charts, status updates and animations are some of the graphics that can be used by projects to attract users to contribute to the project; *e*) Remote diagnostics and debugging features when set, ensure the clients collect information during application execution to allow efficient debugging of application during crashes; *f*) User community feature allows users to form teams, create online profiles and utilize message boards to collaborate with each other; *g*) Checkpointing allows applications that take a long time to run, to resume at the same point since applications might have to quit and restart at a later stage. A checkpoint is created when application reaches a stage where they can save some

meaningful data, stop and resume at the same point; and *h*) Support for compound applications that require more than one program running, is available;

BOINC client

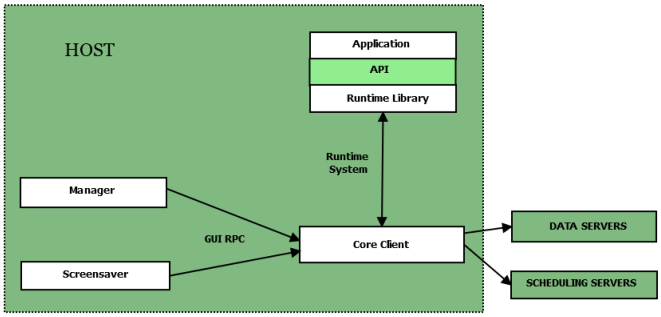

Figure 2.3: BOINC client architecture (Re-illustrated [27])

The BOINC client (see Figure 2.3) is a collection of components that communicate using BOINC's RPC (Remote Procedure Call) mechanism over a TCP connection and shared memory message passing to perform various functions. The client mainly consists of a core client, a GUI , runtime system (an API lined with applications) and a screensaver. All communications with the BOINC server are done through the core client which is also responsible for downloading and uploading input/output files and binaries for applications, controlling and executing applications, scheduling requests and computing results. Local scheduling is done by the core client to decide when to request work, which project to request work from and which tasks to execute. Locality scheduling ensures fair resource sharing across all projects connected. The core client enforces the preferences set by the user.

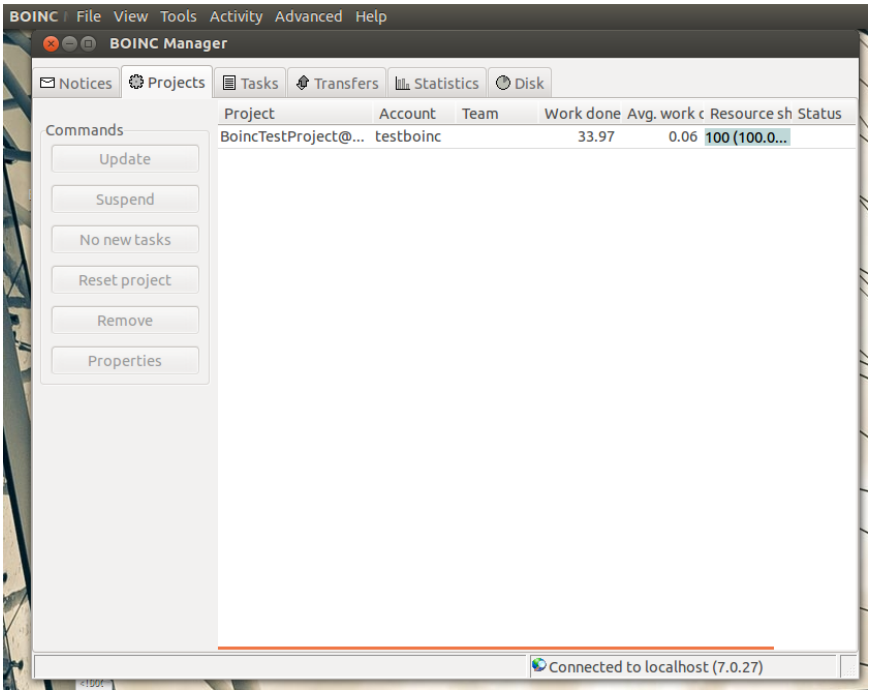

Figure 2.4: Screenshot of the BOINC manager

The API allows the core client and the application to interact with each other. For instance, the application can send a trickle message to the client or the client can suspend, resume or quit an application. The core client and runtime system (API and applications) interact through messages in the shared memory. Every application has its own shared memory that is comprised of 8 message channels used to relay task control messages, graphics, status and trickle messages in both directions. The BOINC client has a screensaver module that allows the applications to display graphics like animations, graphs, icons or current status of the work. The GUI and the screensaver components communicate with the core client using RPC. The BOINC manager (shown in 2.4) is a GUI that allows users to control and monitor the BOINC client. It is built using a cross platform library to allow uniform user experience across different platforms. It can be used to add/remove projects, update applications, set

preferences and view logs. A command line tool is also available for Unix environments apart from the manager. BOINC clients present on different machines can be controlled by a single BOINC manager remotely. Using RPC presents the security risk of intruders gaining access to the machines [23].

Users can configure the client by setting certain preferences [19]. This can be done by editing the cc_config.xml file present in the BOINC directory or by using the GUI/command line tool. Users have control over the amount of network bandwidth that BOINC can use, when BOINC should start running, whether GPUs can be used for computation.

BOINC Server

The BOINC server is essentially a set of web services, daemons and databases. The server was initially designed to be run using LAMP (Linux Apache Mysql PHP) bundle. The server's components are a web interface, a task server and a data server.

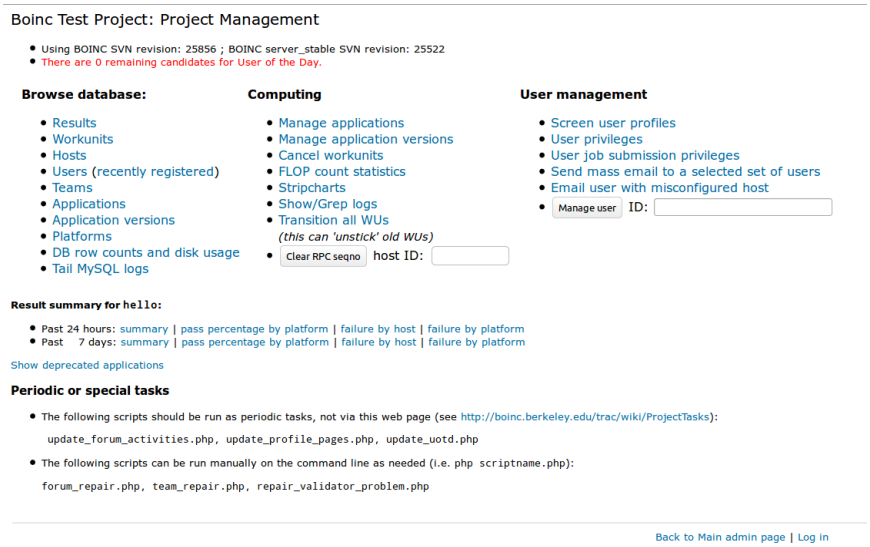

Figure 2.5: Administrators view of the web interface

The web interface (see Figure 2.5) creates a web page that users can login into to access forums, account information, server status, project description and list of applications. An administrative interface is provided for project managers to view/edit any project related information with ease. An admin can cancel work units, review results, verify the logs, manage applications and email users. The data server is intended for data transfer between the client and the server, it handles input files, executables and output files that are legitimate. The data server is implemented using a web server along with the BOINC supplied *file_upload_handler*. The task server handles creation, dispatching of tasks and processing results. Information is stored in a mysql database by BOINC. The database consists of close to 35 tables, a few important ones are : *1*) A *platform* table to maintain the information about various platforms; *2*) An *app* table to keep a track of all the applications within the project; *3*) An *app_version* table to record the information about different versions of applications and whether they are deprecated or not; *4*) A *user* table to store information about users; *5*) A *host* table to store machine information; *6*) A *workunit* table used to track and store information and state of workunit. The input template file is also stored along with the urls for the input files; and *7*) A *result* table.

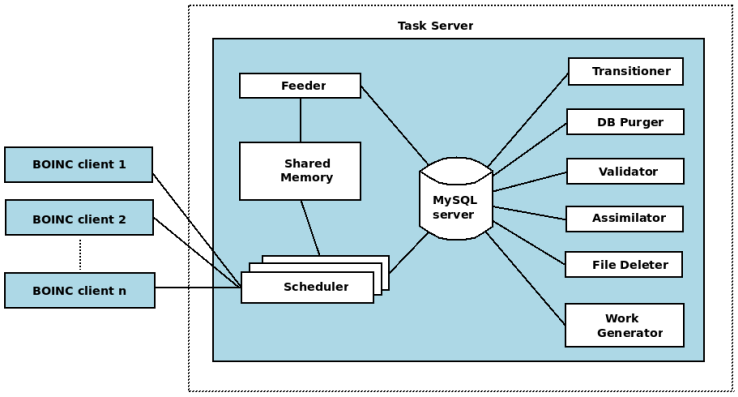

Figure 2.6: Task server (Re-illustrated [20])

The architecture of a task server [20] is shown in Figure 2.6. The feeder, transitioner, validator, assimilator, file deleter and work generator are daemons (background processes), while the scheduler and file handler (data server) are CGI scripts. These components except the file handler, access the database and complete their task if corresponding flags are set. The reason for the asynchronous setup is to ensure failure or performance of one process does not affect the others and allows components to be distributed or replicated on other servers. However, it creates a substantial load on the database server. The flags are usually indexed and when a component cannot find any entries with a flag set it sleeps for a small period of time.The Work generator, assimilator and validator are components that should be tailored to the specific needs of BOINC projects.

The *Work generator* is a program that creates work with required input files and then stores it into the BOINC database. Many projects have their own customized work generators which create work either when data is available or when there is space to create new input files. The transitioner is responsible for handling different state changes of work units and results. It changes flags in the database to facilitate

the validation or assimilation phase and also creates new result instances when there is an error in the previous results or when there is a need for redundant computing.

The file deleter deletes input files that are not being referred to by any work units as well as output files that are no longer needed.The *database pruger* is used to ensure that, the size of a workunit and result tables does not increase drastically which in turn affects the speed of indexing. The daemon stores the entries that are not required into xml files and then deletes them.

The scheduler is a CGI script that is used to handle client requests. Clients periodically request work and report uploaded results. Requests contain information about the client's system like available RAM, free disk space, GPU existence, operating system and others. The scheduler takes into account all this information before sending a list of workunits available to the client as a response. Also, the scheduler is responsible for handling homogeneous redundancy and identifying reliable hosts. Multiple instances of the scheduler are run to handle the load.

For fast access and to decrease load on the database a shared memory segment containing required data is maintained and updated by the feeder. The feeder copies database records of applications, platforms, versions and maintains a fixed number of unsent results and workunit pairs. A semaphore is used to manage concurrent access to the memory. The validator analyzes various results for a workunit and decides whether they are correct. If they are correct it selects one of the results as a canonical result for that workunit. Each application must have a validator which can either be a custom validator or one of the standard validators provided by BOINC. They are also responsible for granting credit for valid results by updating required database entries.

The assimilator is used to handle completed jobs that are either successful or have encountered an unrecoverable error. It can be used to copy output files from BOINC

upload directory to a specified location or to populate a database.

Project , Work units and flow of computation

Every BOINC project is identified using a master URL. Clients can utilize this URL to attach to the project and also view its description. Each project usually contains multiple applications with different versions (one for every platform or old versions of the application). The project can be configured by editing the config.xml file. Scheduler preferences, locality scheduling, logging, web site features, client controls, daemons and periodic tasks are some of the options that can be configured.

A job consists of a workunit and one or more results. A workunit describes the computation that needs to be done such as the application version to be run, references for the input files, command line arguments, memory and storage deadlines. Results describe an instance of work either unstarted, in progress or done. Each result is associated with a workunit.

The following steps are the typical stages in a workunits lifecycle. A workunit is created by the workgenerator. Then the transitioner examines the workunit and creates duplicates if needed. The feeder stores the workunit's data in the shared memory segment when there is free space. A client requests work, the scheduler dispatches workunits/results to the client if all the criteria are satisfied. The client, upon receiving the work, downloads the required files from the data server and performs the computation when free. The result files are uploaded by the client to the file_upload_handler. The results are reported to the scheduler the next time the client sends a scheduler request. Based on the number of results and status the transitioner either creates new results to be sent or calls the validator. The validator based on the workunit validates the results and assigns the credit. The assimilator does its job and exits. Input files are deleted by the file deleter when they are no longer referred

to.

```
<input_template>
  <file_info>
    <number>0</number>
    [ <gzip/> ]
    [ <sticky/> ]
    [ <no_delete/> ]
    [ <report_on_rpc/> ]
    [ <url>...</url> ]
    [ <url>...</url> ]
    [ <md5_cksum>...</md5_cksum> ]
    [ <nbytes>...</nbytes> ]
  </file_info>
  [ ... other files  ]
  <workunit>
    <file_ref>
      <file_number>0</file_number>
      <open_name>NAME</open_name>
      [ <copy_file/> ]
    </file_ref>
    [ ... other files ]
    [ <command_line>-flags xyz</command_line> ]
    [ <rsc_fpops_est>x</rsc_fpops_est> ]
    [ <rsc_fpops_bound>x</rsc_fpops_bound> ]
    [ <rsc_memory_bound>x</rsc_memory_bound> ]
    [ <rsc_disk_bound>x</rsc_disk_bound> ]
    [ <delay_bound>x</delay_bound> ]
    [ <min_quorum>x</min_quorum> ]
    [ <target_nresults>x</target_nresults> ]
    [ <max_error_results>x</max_error_results> ]
    [ <max_total_results>x</max_total_results> ]
    [ <max_success_results>x</max_success_results> ]
    [ <size_class>N</size_class> ]
  </workunit>
</input_template>
```

```
<output_template>
  <file_info>
    <name><OUTFILE_0/></name>
    <generated_locally/>
    <upload_when_present/>
    <max_nbytes>32768</max_nbytes>
    <url><UPLOAD_URL/></url>
  </file_info>
  <result>
    <file_ref>
      <file_name><OUTFILE_0/></file_name>
      <open_name>result.sah</open_name>
      [ <copy_file>0|1</copy_file> ]
      [ <optional>0|1</optional> ]
      [ <no_validate>0|1</no_validate> ]
    </file_ref>
    [ <report_immediately/> ]
  </result>
</output_template>
```

Figure 2.7: Input and Output Templates

Every job/workunit is associated with input and output templates (see Figure 2.7) which are separate files that are used to describe the properties of the job. The input template is mainly used to describe input files required by the workunit and contains file information (file number, md5 key, number of bytes, whether to delete on client etc) for every input file, file references (logical name for file, file number), command line arguments and job attributes (maximum number of flops, memory bound, maximum errors, maximum results). Input template file contents are stored in the workunit table of the BOINC database as a blob. The maximum size of the input template is limited by the size of a blob. Hence the number of input files per

workunit are limited (more the number of input files, more is the size of the input template).

The output template file is used to list properties of output files that would be generated by the application and the ones that have to be uploaded to the server. The file information, file references and other preferences can be set. However an issue with output template files is that the names are stored in the database hence they should not be changed or deleted until the results are returned.

2.2 Work related to Grid Computing

To support running of multiple simulations in a short period of time, an efficient cluster management system was required. Grid computing has been around for some decades and there have many grid computing middlewares developed such as ARC [28], gLite [10], Unicore [32]. To be able to solve large scale real world urban environment optimizations a large amount of computational resources is required. However it would be not cost effective to acquire the required number of computational resources to form a grid. Hence, the focus turned towards volunteer grid computing.

Volunteer grid computing was first used by the Great Internet Mersenne Prime Search [9]. Volunteer computing is a form of distributed computing. Distributed computing also has a share of middleware systems such as BOINC [17], XtremeWeb [22], Condor [36], Xgrid (currently no longer supported), Oracle Grid Engine. BOINC is very popular volunteer computing middleware that can be used to form a grid within an organization. There are many projects currently utilizing BOINC (the list can be found at [12]) including the well-known SETI@Home [18] and Einstein@Home [6].

After analyzing various platforms, BOINC was selected for this thesis since it is one of the widely used platforms thereby increasing the chance of volunteers con-

tributing to this project. Another reason is because projects such as Folding@Home [7], GPUGRID [11], Einstein@Home [6] are using GPU versions of their applications proving that GPU clusters can be harnessed which is an essential requirement.

Bhagirath Adeppali used simulated annealing and genetic algorithms to find the optimal placement of buildings by minimizing either maximum average concentration of pollution at the street level or the number of concentration values about certain levels [15]. The results of simulated annealing and genetic algorithms were compared to each other and validated against the results obtained from the brute force mechanism. Such global optimization algorithms can be applied for finding the desired configuration in the implementation.

3 Implementation

This chapter describes the infrastructure and the support software that was implemented to reach the goals of the thesis. The problems encountered during running the 8D experiment (mentioned in the introduction chapter) were mainly : *1*) Being able to let the planner vary any of the simulation parameters (building locations in that experiment) with ease; *2*) Identifying the various possible configurations of the urban space based on the planner's input; and then *3*) Distributing these various configurations to more than one machine for higher throughput

The next few sections address these problems and provide efficient solutions which when put together form a computational environment that can be used for such tasks. A language (Section 3.1) was created as a solution to the first problem. A framework (Section 3.2) was developed to facilitate creation of different urban configurations to address the second problem. To tackle the last problem a BOINC application and supportive daemons were implemented to provide the required computational power by distributing the work across various machines (Section 3.3).

3.1 The Language

The language was created to be able to specify variables and their values that determine possible configurations, change various simulation parameters and to control which factors to optimize on. Simulation paramteres might get added or removed due to new functionalities. An important design consideration was to allow for these

changes to be supported by the language with the least effort. The language being created should be simulation independent with the ability to be ported to another similar simulation without too much of effort. The simulations should be able to run on a wide range of platforms, hence the language implementation should be platform independent. The language should be intutive and easy to use.

3.1.1 Syntax and Semantics

The following subsection describes the language specification.

To keep the syntax fairly easy, the basic unit of the language is an assignment statement (3.1).

$$reference\_to\_parameter \ = \ value \tag{3.1}$$

The left hand side of the assignment is a reference to a parameter which is a name that can be used to uniquely identify the parameter. The righthand side is the value which could be a single value or a group of values.

$$reference\_to\_parameter = 12 \tag{3.2}$$

$$reference\_to\_paramter = 13.0 \tag{3.3}$$

$$reference\_to\_parameter = average \tag{3.4}$$

Single values can be integers (3.2), decimal numbers (3.3) or strings (3.4). The language's syntax was modelled off Matlab's variable and set initializations with the goal of making it easier for scientists to use, since Matlab is well known within the engineering and several science communities. A group of values can be represented either using a set or a range depending on whether the group forms an arithmetic

progression of numbers or not.

$$reference\_to\_parameter = [10\ 50\ 13] \tag{3.5}$$

$$reference\_to\_parameter = [29.0 : 1.0 : 30.0] \tag{3.6}$$

A set of numbers is represented in the language (3.5) by separating numbers by a space and enclosing them in square braces. A range of numbers is represented (3.6) by first value, step size and last value enclosed within a square brace and separated by a colon.

$$reference\_to\_parameter\_1 = reference\_to\_parameter\_2 + 10; \tag{3.7}$$

$$reference\_to\_parameter\_1 = reference\_to\_parameter\_2 - 12.2; \tag{3.8}$$

$$//comments\ are\ useful\ in\ understanding\ why\ certain\ parameters\ were\ set. \tag{3.9}$$

Apart from constant values, users can specify a dependency between parameters. The language supports parameter references to be used as values. Addition (3.7) and subtraction (3.8) are currently allowed. The input given to the program is a file containing a series of assignment statements indicating changes to simulation parameters desired. The language also supports single line comments (3.9), by using "//" to indicate the start of a comment.

$$quBuildings.buildings[0].xfo = [10.0 : 2.0 : 20.0]; \tag{3.10}$$

$$qpParams.numParticles = 65500; \tag{3.11}$$

If a user wanted to vary the building one's x coordinate from 10 to 20 in steps of two, it could be done using the equation in 3.10 . To set the number of particles released in a simulation to be 65500, the assignment statement in 3.11 would suffice.

3.1.2 Design issues of the language

C++ was the choice of technology as the simulations and BOINC's API were written in C++. The C++ standard template library and Boost::any were used to develop the language. In C++ data, is usually organized in a set of classes to take advantage of object oriented principles. Each of the data members have private or protected or public access specifiers associated with them. To gain access to data outside the scope of a class, pointers were used. Using a pointer, data present in a location can be modified or read irrespective of the access specifier and scope. However, the problem with pointers is that a pointer should be of the same datatype as the data member. Hence its not possible to store pointers to different data members using a key-value data structure like a map or an *unordered_map*. Void pointer is a generic pointer in C++ which can be used to point to any memory location. Void pointers cannot be directly dereferenced, rather they have to be typecasted into the appropriate type of pointer before dereferencing them. Unfortunately there is no way to determine the type of data pointed by them. C++ is a statically typed language and consequently typecasting cannot be done at runtime (i.e we cannot use a string to hold the type and then typecast a void pointer using that string).

The Boost library contains a class called Boost::any which is a generic type that can hold any data type. The advantage of using Boost::any is that apart from storing data it also stores its type.

$$const\ std :: type\_info\ \&\ type()\ const; \tag{3.12}$$

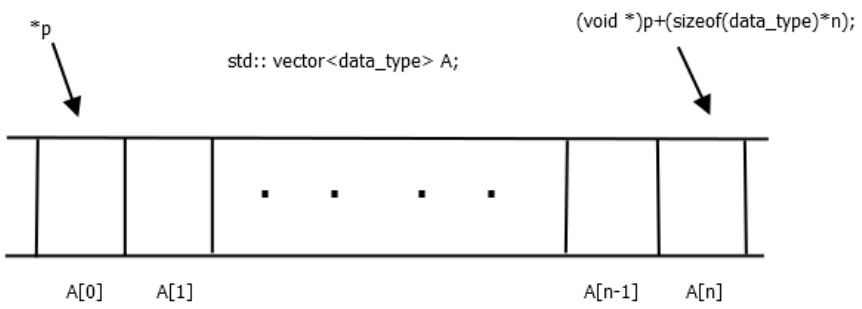

Figure 3.1: vector memory diagram

The type stored can be retrieved by using the function specified in 3.12. The *type_info* can be used to compare if two types are equivalent.

The Standard template library's map was used as a key-value data structure to hold the reference to a data member and its address. The key is a string which is a unique way to identify the data member stored and the value is the address of the data member stored as a Boost::any object (std::map $_{<}std :: string, boost :: any_{>}$).

Storing all the addresses of data members within a map would suffice. However, more than often simulations have vectors and arrays of user defined or primitive data types. Keeping track of addresses of all elements of a vector would be difficult due to their sheer number and also since new elements can be pushed into vectors. To solve this problem pointer arithmetic is used. A pointer basically stores addresses and changing the address changes the location to which the pointer is pointing to. The facts that vectors and arrays occupy contiguous memory and are homogenous in nature can be exploited. By storing only the first element and applying pointer arithmetic, any of the members of the vector can be accessed. However the size of the element needs to be known. Figure 3.1 gives a visual description of the same. When an integer pointer is added by one, the pointer points to the next integer rather than

the next address location. Internally, the size of the integer is added to the address. Hence before performing pointer arithmetic, addresses are stored temporarily in char pointers. Adding a number to a character pointer adds the same amount to the address.

The implementation of the language uses character based pointer arithmetic to modify or access the various simulation parameters. The addresses of the parameters that can be accessed through the language are stored as Boost::any objects in a map.

3.1.3 Implementation of the language

```
class languageMap
{
public:

        languageMap() {} //empty constructor
        virtual ~languageMap(){}
        virtual std::string retrieve(std::string variable_name);  //function to retrieve a value
        virtual void modify_value(std::string variable_name,std::string newvalue); //function to modify a value

        virtual void build_map(){
        std::cerr<<"build_map has not been overloaded"<<std::endl;
        };

protected:
std::map<std::string,boost::any>  var_addressMap;


};
```

Figure 3.2: languageMap class

To take advantage of OOP principles a base class called *languageMap* (see Figure 3.2) was created, which holds an std::map and virtual functions that can be used to set the value of a data member or retrieve the value of one. The advantage of having a base class with virtual functions is that these functions can be called from a base class pointer pointing to a derived class object and functions would have access to the map of the derived class. All classes that have at least one data member that could be modified through the language must inherit from the *languageMap* or from a class

that inherits it. A virtual function called *build_map()* is responsible to populate the map that was inherited. The map is filled with addresses of each of the class's data member unless the data member is a vector or an array in which case the size of the vector and the address of the first element are populated. The key in the map is just the name of a data member in the class.

```
class simParams : public languageMap    //inherit from the class
{
public:
        simParams();
                virtual void build_map(){                       //store address into the map
                   var_addressMap["timestep"]=&timestep;
                   var_addressMap["date"]=&date;
                   var_addressMap["particles[]"]=sizeof(particle);
                   var_addressMap["paticles.x"]=&sources[0].x;
                  };
         struct particle{
             int x;
           };
private:
        int timestep;
        string date;
        vector<particle> particles;

};
```

Figure 3.3: Example of a class being modified to be used with the language

Figure 3.4 gives an example of a class *simParams* inheriting from *languageMap* and overriding *build_map()* function. To allow timestep to be changed by the language, the address of the variable timestep is stored in the map with timestep as the key. To store the size of the vector element the key would be the name of the vector appended with [ ] at the end. The key for the vector's first element would be the name of the vector appended by the name of the data. A wrapper class that encapsulates all the data that would be accessed through the language is implemented to get a unique key reference to all parameters and to provide a single point of access to the data. The wrapper class will also inherit from the *languageMap* class.

When a parameter value needs to be changed, the name of the parameter and its value are passed as arguments to the *modify_value* function. The function utilizes the name of the parameter passed in to determine the address of the parameter from the series of map data structures and writes the appropriate value into the parameter's memory location.

```
modify_value(variable name, value) :

 search the name for a dot

 ///SECTION - 1
 IF no dot is found THEN  // suggests the parameter is a primitive data type of this class

    search the map for the name and retrieve the address, throw an error if not found.

 ELSE     // the parameter is datamember of a datamember of this class or a vector

    retrieve the name of the parameter before the dot

    Search for a square brace

    IF not found   //the parameter is a datamember of a datamember of this class

          retrieve the address type cast it to languageMap pointer and call modify\_value on the pointer
          by passing the remaining variable name and the value

    ELSE       //the parameter is a vector

         extract the index, vector base address and the size of the vector and determine
         the exact address of the parameter

   ///SECTION -2

   Do for all primitive types :
       match the type of a primitive data type pointer with the type of the any object
            if they are a match typecast the pointer and convert the value into the datatype
            Store the value.
            exit loop.
```

Figure 3.4: psuedocode for *modify_value* function

The *modify_value* function can be broken down into two essential steps 1) Finding the address of the parameter (refer section-1 in Figure 3.4) 2) Finding the data type of the parameter, type casting into the corresponding data type and writing the value into the memory location (see section-2 in Figure 3.4).

Finding the address of the parameter involves determining if the parameter is part of this class (as an element of a vector or a datamember) or a datamember of a datamember. The presence or absence of a dot and square brackets can be used to determine what the parameter is. If the parameter is a datamember or an element of the vector its address is retrieved/calculated. If its a datamember of the datamember of this class then the address of the datamember of this is calculated and the function *modify_value* is called by type casting the datamember into the *langaugeMap*.

Once the address is found out, the type of data stored at the memory location is retrieved using the type function of Boost.Any. The type is put through a switch case with the type being compared with all the primitive types. When a match is found, an appropriate pointer is used to set the value. Figure 3.4 gives a full account of the function.

The purpose of evaluating the various configurations of urban spaces is to find a configuration that optimizes on a certain aspects such as average temperature, minimum temperature or maximum concentration of pollution. Such values are computed using fitness functions and the planner should be able to choose which fitness functions to use.

Apart from changing the values of parameters, the language is extended to give the user the ability to control the simulation. The user can specify control commands like which fitness function to be used by the simulation or what the averaging parameter would be across all simulations.

3.2 Population generation and multiple simulation runs

QUIC Energy and GPU Plume were designed to simulate a given urban environment. The input is a set of files that would populate the required simulation parameters. However to make decisions on optimal placements of buildings, green roofs, trees, parks, many simulations need to be run for various possible urban configurations. When broken down, all the different configurations are small changes to the original input files. Hence the language is used as a means of specifying these changes as sets and ranges of values. The collection of all possible configurations is called a population, while a single configuration is called a sample. Once the file written in the language is given to the program, it generates a population. To facilitate easy generation of population, a *population_gen* class was implemented. The class can generate a population either when the set and range values are given or it can generate a random population. To efficiently handle values for the parameters, a vector is used to represent a configuration. The vector contains values for all the parameters that can be varied. The whole population is stored as a vector of vectors.

The algorithm below is used to generate the population :

```
generate_entire_population:
create a vector s that contains the minimum values for all the parameters
bool done = false;
while (!done)
{
   save s into the population
```

```
   set the dimension to zero
   bool done_incrementing = false;
    while (!done_incrementing)
  {
     if dimension being incremented is a range value
         calculate the next value
         if the new value is within the range
            set done_incrementing to true
         else
            set the value to the minimum value
            increment the dimension
     else ( the dimension being incremented is a set value)
    search for the current value
 if the value is the last value
            set the new value to the first value
    increment the dimension
 else
    set the new value to the next value
    set done_incrementing
     if dimension is the last possible parameter
        Done with the generation of population
        set done and done_incrementing to true.
  }
}
```

The population generation function returns a vector of vectors which is a vector of different configurations. The simulation needs to run for each vector and for every run the parameter values can be set by calling *modify_value* for the vector. This would facilitate running the whole set of simulations on a single machine.

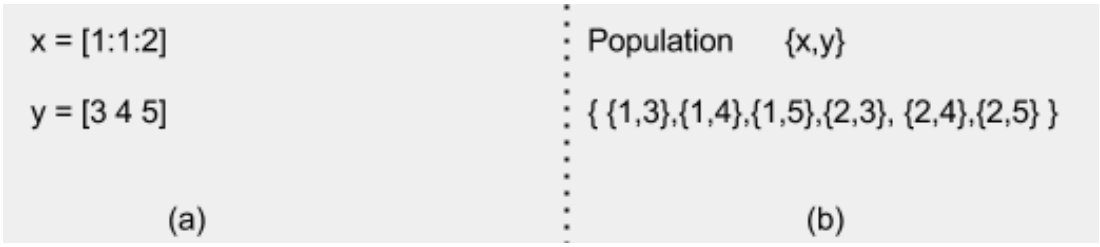

Figure 3.5: (a) A Simple optimization specification (b) Population generated based on the optimization file

For example consider a simple optimization file in Figure 3.5 (a). The file specifies that the value of x can start from 1 vary up to 2 in steps of 1 and the value of y can be either 3, 4, 5. When the file is given as input to the *population_gen* class and *generate_entire_population* is called, it returns the six possible configurations as a vector of vectors (see Figure 3.5(b)).

However the 8D test case (introduced in the introduction) is not feasible to run on a single machine and ideally should be run on many machines to complete in a timely manner. All the configurations need to be distributed to available machines equally to utilize all machines and complete the problem faster.

3.2.1 Language parser and optimization file splitter

A simple parser is written for the language that discards comments, checks for syntax errors and detects parameter names and values that need to be assigned to the respective parameters. Set values, single values and range values are stored in different

data structures. The Optimization file splitter was a program designed to generate a population using the *population_gen* class and based on the number of machines generate separate optimization files or a vector of configurations for each machine. The new file generated would have the same single value assignment statements but additionally it contains a section with vectors of the configurations it is to run. The section of vectors is headed by the number of configurations, the size of each vector and a line with the corresponding parameter name strings similar to a normal table (see Figure 3.6). This file is parsed with the same parser, however the population section is parsed differently due to its syntax. It is however handled like a stream of single value statements.

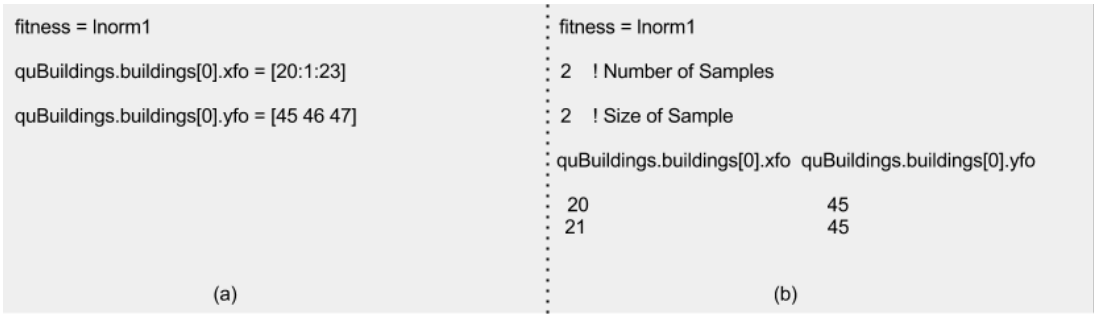

Figure 3.6: (a) An optimization specification (b) Population file with two samples

3.3 BOINC

Once the optimization file splitter had done its work of splitting up the problem into different configuration files, they are ready to be run on a cluster of machines. However manually distributing this work to a set of machines, monitoring them, gathering and processing results would take substantial effort and time. Distributed computing middlewares such as BOINC automate such tasks. BOINC was chosen for

this task because it can be used as a volunteer grid and also be used to form a fast and dedicated grid using computational resources within an organization.

The installation of a BOINC server can be done by using a virtual machine installation or downloading and installing the prerequisites (software packages) and then downloading and compiling the BOINC source code. A BOINC project can be created using the *make_project* script that is provided by BOINC. The script creates the required project directories and subdirectories, sets up mysql project database, creates the projects configuration file and also copies the required source files. To utilize the BOINC system a BOINC application, a work generator and an assimilator were developed.

3.3.1 BOINC application

Any application written in any language can be ported onto the BOINC environment. Running an existing application without any modification on the BOINC environment causes many issues. There are several reasons for this :*a*) The input files present in the directory along with the executable are actually xml files. These xml files specify the location of the original input files; *b*) The BOINC client looks a for a file called the finish file as a sign of completion of execution; *c*) BOINC client issues suspend, resume, and restart commands to the application; and *d*) Output files are uploaded to the server by the client. However they would need to have a particular name for BOINC to recognize them as outputs produced by execution of the application;

Hence the input files accessed by the application would be xml reference files rather than the original ones, application would be restarted by the core client repeatedly even after completion because of the missing finish file. The application would not

respond to BOINC clients suspend, resume, and restart commands and the results produced would never be sent to the server.

BOINC provides a wrapper application that can be used to run an existing application. The wrapper application manages all the communications between the core client unmodified application. It runs the existing application as a subprocess. Another way to port applications is to make changes to the source code by making use of the BOINC's runtime library.

BOINC provides various APIs for different purposes. The basic BOINC API is used for initialization, the diagnostics API is used to enable data collection for debugging and a graphics API for allowing the application to render graphics to the user's screen. The BOINC application's pseudocode is given below, which utilizes the BOINC API rather than the wrapper application.

```
main()
{
 Initialize BOINC diagnostics  \\to help debug
 Initialize BOINC
 Search for the optimization/population file corresponding to this workunit
 Resolve the name of the optimization file
 Resolve the name of the input zip file, unzip and load the data
 Parse the optimization file and make the required changes to the data
 Run QUICEnergy
 Generate output files and rename to match the name BOINC client expects
 Terminate BOINC \\terminate BOINC
}
```

The remainder of the section describes each step of the application in more detail.

To get access to the BOINC API and diagnostics API we have to include "boinc_api.h" and "diagnostics.h" header files. The BOINC initiation calls are different for applications that use graphics and applications that do not utilize graphics. For this version of the implementation there were no visuals displayed by the BOINC application. The BOINC diagnostics is initialized by calling *int boinc_init_diagnostics(int flags)*. There are multiple flags available but this application uses :

BOINC_DIAG_REDIRECTSTDERR Redirects standard error stream to a text file and can be viewed on the administrators web interface, which is very useful

BOINC_DIAG_MEMORYLEAKCHECKENABLED Lists all the memory leaks to stderr after the application exists

BOINC_DIAG_DUMPCALLSTACKENABLED Prints a stack trace to the stderr when the application crashes

BOINC_DIAG_TRACETOSTDERR Writes trace macros to stderr (Microsoft Windows specific).

BOINC is initiated by calling *boinc_init()*, this call must be done before calling other API functions. The BOINC client maintains a separate directory for every project and slot directories for each of the running jobs 3.7. All the required input files are located in the projects directory and links to those files are stored as xml files in the present slot directory. If an application tries to open an input file it would open up the xml rather than the original file.

$$int\ boinc\_resolve\_filename(char\ *logical\_name,\ char\ *physical\_name,\ intlen) \tag{3.13}$$

To resolve this issue BOINC provides a function that maps a logical name to the physical name called *boinc_resolve_filename* (see 3.13). When the logical file name is provided and the length given, it provides the physical name. The problem however with the simulation app is that the logical name of the file with the optimization information is unknown and also the physical name returned by the function call is a relative path to the file.

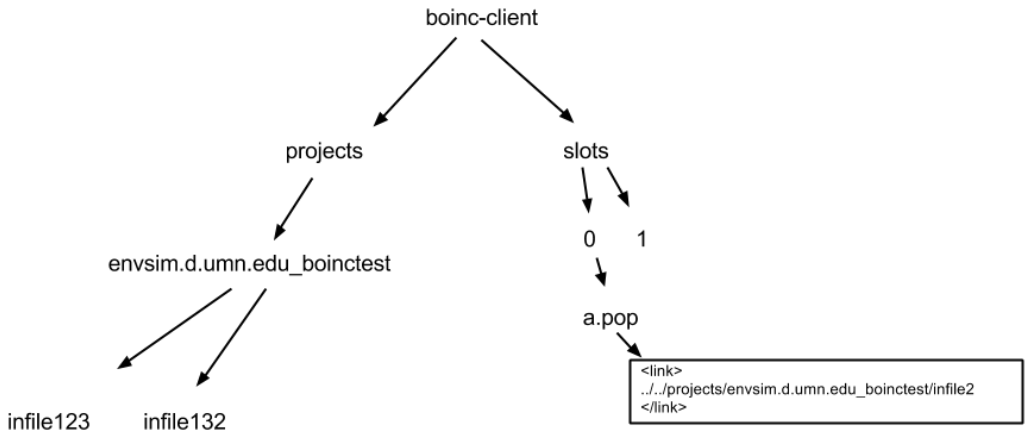

Figure 3.7: Directory structure of a BOINC client

To resolve the name of the population file, the fact that the "projects" directory is at the same level as "slots" directory in the directory tree structure with the project directory as their parent is utilized. The name of the project is always the master

url of the project appended with an underscore and the name of the project (eg : master url is : https://envsim.d.umn.edu/boinctest and project name is boinctest hence the directory name is envsim.d.umn.edu_boinctest). Boost::filesystem has a directory iterator class which was used to retrieve a list of all files within the directory. The optimization files end with a extension ".pop". Boost regex was used to match the list of the files in the directory with the regular expression ".+pop" to filter out application files and other input files. The new list would contain all the optimization files for all the workunits sent to this host. Through experimentation, one observation is that *boinc_resolve_filename* returns the same logical name of the file that was passed in when the workunit does not have access to it (i.e An application can access files through BOINC only when they are mentioned as input files to that workunit). Hence all the names of files in the list were resolved through *boinc_resolve_filename* and only one would return the relative path to the actual file. This relative path obtained was converted into absolute path by utilizing linux system call *realpath*.

Once the population file is found, the simulation data is loaded from the input files. The input files were compressed and sent as a single zip file from the BOINC server. The zip file is unzipped and the path is to the directory is used for obtaining the information from the input files and loading all the data. The population file is then parsed and the corresponding changes are made to the simulation data using the *modify_value* function.

The base configuration was loaded and changed to new configuration using the population file. QUIC Energy is run, generating the required outputs.

On successful completion of the simulation, the output files are gathered, zipped, and renamed to match the name of the output file expected by the BOINC client. The name of the output files can be obtained using the *boinc_resolve_filename* function.

The application must call *boinc_finish(int status);* to indicate the completion. This

function call creates the required finish file.

3.3.2 Work generator

The work generator is responsible for creating and submitting workunits. A job/-workunit can be submitted either by using the command line tool *create_work* or by using the C++ function in Figure 3.8.

```
int create_work(
    DB_WORKUNIT& wu,                    //workunit structure containing workunit attributes
    const char* wu_template,            // contents of the template file
    const char* result_template_filename,    // relative path of result template
    const char* result_template_filepath,    // path to current directory
    const char** infiles,                // input file names as a character array
    int ninfiles,                        // no of input files
    SCHED_CONFIG&,      //reference to sched_config
    const char* command_line = NULL,   //command line arguments
    const char* additional_xml = NULL  //contents of additional xml
);
```

Figure 3.8: Create Work function syntax

However before submitting the job, the input and output xml template files required by the workunit must be created and the input files required by the application need to be placed in the appropriate location on the server to allow the client to download them through the data servers. The input files are usually stored in a subdirectory under downloads in the project folder. The upload/download directories are not flat, rather they are hierarchical directories and their subdirectory names range from 0 to 3ff with files being hashed into them based on filenames, *stage_file* is a command line tool that manages the moving of files or by using *dir_hier_path* to retrieve the path and then copy the file into the obtained path.

```
main()
{
  // parse the optimization file and generate population
  // generate individual population files
  // check for input files status and stage them
  // generate input xml templates for each population file
  // connect to boinc database get retrieve required information
  // submit the individual workunits
}
```

Figure 3.9: Work generator Pseudo Code

The figure outlines the work generator that is being used for the project. The program is supplied with the path to the optimization file. The file is parsed, the single values are stored and the range and set values are supplied to the population generator. To ensure work is distributed evenly once the population is generated the optimization file splitter is used to split the population by putting each configuration into a different population/optimization file. The reason for generating one pop file per configuration is that each configuration can be run as a single job thereby any errors in a single job would not stall the other jobs.

The input file generated must be placed in the appropriate directory. Before proceeding, the file is checked for the existence using boost filesystem::exists function. To get the path of the directory for the input file we can utilize a C++ structure provided by BOINC called *SCHED_CONFIG*. The *sched_config* loads the server configuration data by reading the config.xml file which can be done by calling *config.parse_file()* function. Once the config file has been parsed the directory path is determined by calling *download_path* on the config and passing the name of the file and path as char arrays. The input file is then copied into the corresponding directory.

Since the input file names are different for every workunit, a separate input template must be created for each of the workunits. The boost property tree is library

that allows storing nested tree values. A property tree is created resembling the input template and then an xml file can be created by using the property tree's generator. File info and file information are the two xml elements that would change per workunit, and the rest stay the same. The names of the files and their file numbers are set accordingly and then the xml file is generated.

To submit the job the *create_work* function must be called by filling up all the required parameters. The first argument is structure that holds all the properties of a workunit. A unique name of the workunit, the application id and other properties that could also be specified in the input template. To generate a unique name for the workunit, the program accesses the BOINC database on the server and retrieves the latest workunit information from the workunit table. All the workunits have an automatically generated workunit id, hence the last workunit would be the one with the highest workunit id. All the workunit's names within this project were structured to start with a string and be followed by an underscore and a number. The new name would be the number incremented at the end. The application id is an identification number assigned by BOINC to the applications. The application id was retrieved from the BOINC database table "app" based on the name of the application. However to make it more generic, *sched_config* was used to retrieve the database information (username, password, name of database) from the config.xml file by parsing it. The information retrieved can be then used by the *boinc_db structure* and the app structure to retrieve the application id and the workunit name.

The second argument is the char array containing the input template. The fifth argument is the list of input files. To make the work generator efficient and reusable, a work generator class was created. A function was created called *pushinputfile_boincdir()* that would do everything needed to stage a file, a function called *generate_inputtemplate()* is used to generate an input template based on all the files

that were pushed into the download directory, and also a *submit_job()* function that would create a workunit by keeping track of the previous function calls. Through the use of this class, work generation for any application can be accomplished with a few function calls. However the *create_work()* function provided by BOINC has dependencies to the BOINC project folder hence cannot be run from anywhere else except the BOINC project directory or any one of its direct sub directories.

The work generator, when supplied a optimization file and an input file, would create one workunit with no redundancy setting for every configuration and push it onto the BOINC server for the feeder to process.

3.3.3 Assimilator

Workunits that are marked as completed by the transitioner are handled by an assimilator. Assimilators are usually tailored based on the application's requirements on processing valid results and errors. BOINC provides a driver program (wrapper code) that handles command line arguments, establishes a connection to the database, scans tables to determine workunits that are ready for assimilation. The driver program is called assimilator.cpp and is present in the "sched" directory within the BOINC source code. An extern function (assimilate_handler) is called by the assimilator.cpp for every workunit that can be assimilated.

```
int assimilate_handler(
    WORKUNIT& wu, vector<RESULT>& results, RESULT& canonical_result
);
```

Figure 3.10: Extern function used to link to assimilator.cpp

A custom assimilator can be created by defining a function with the prototype

presented above 3.11 and linking it with sched/assimilator.cpp. The figure below outlines the assimilator being used for this project.

```
assimilate_handler {

Create an output directory for the corresponding batch id of the workunit
If the workunit has completed with a valid result
        Retrieve output file names from the upload directory
        Copy them into a local output directory with an appropriate name
If workunit has too many errors
      write an entry into an error log for future verification
Retrieve the current time store it in the output directory

}
```

Figure 3.11: Pseudo code for the Assimilator

The workunit table has a field called *error_mask* which is set when a workunit encounters too many errors and is aborted, a field called *canonical_resultid* to store the id of a result if available. Required information is passed as arguments to the *assimilate_handler* function. A directory with its name corresponding to its batch id (retrieved from the workunit structure) is created by using a BOINC backend utility function *boinc_mkdir("path")*. If the workunit errored out, a custom error message is entered into the log using *write_error function*. The output file information of successful workunits is retrieved by calling a BOINC function *get_output_file_infos* and passing the *canonical_result* structure and a vector to store information. The returned vector contains paths to output files present in the upload directory maintained by data servers. The files are then copied into the required directory by using *boinc_copy* function. A file containing the current time is created and is stored in the results directory.

4 Results

Testing and validation was an important part of the implementation of this system. This section describes two of the main experiments that were conducted, the results obtained and their in-depth analysis. The first experiment was designed to be a proof of concept. The proof that the language works, accomplishing all its goals, and that the BOINC application utilizes available resources and performs the required computation within an acceptable time frame. The second experiment was focused on determining whether the implementation can be used to compute optimizations of large scale urban environments. The performance of the BOINC application was analyzed to determine the uses of the implementation and to gain insight into the direction that should be followed for future research.

4.1 Experiment 1 : Proof of concept

For the purpose of this experiment, QUIC Energy was selected to be the simulation that would be used for optimization because its average time for a single simulation was shorter than that of GPU Plume and since it uses both CUDA and Optix for computation while GPU Plume uses only CUDA. QUIC Energy was ported onto the BOINC platform and modified to allow access to its simulation parameters through the language. To decrease the run time and extra work required to render graphics, QUIC Energy was run in a headless mode (a mode without visuals).

A simple urban space was used as the environment in this experiment (see Figure

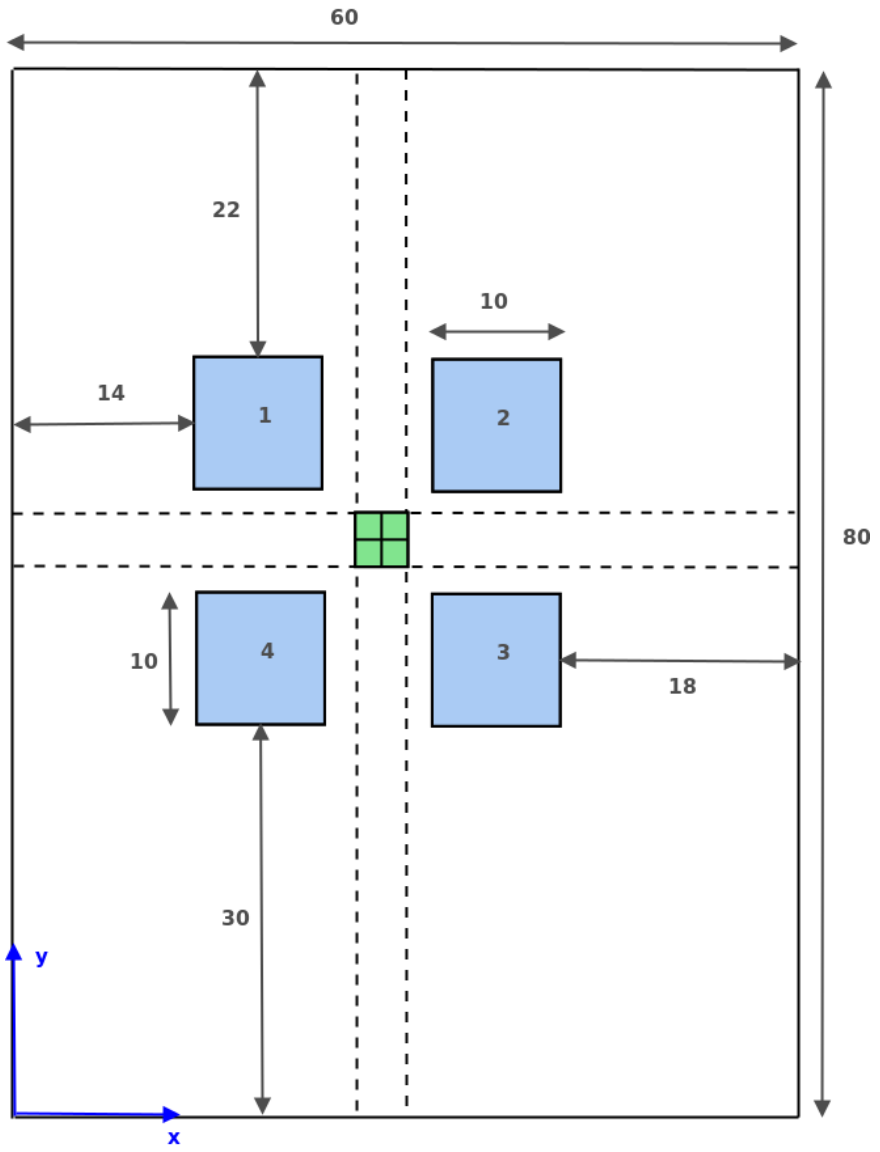

Figure 4.1: The 8D experiment setup for QUIC Energy.

4.1). It consists of four buildings with 10x10x10 dimensions, resting on a plain with a dimension of 60x80 units. The buildings could move from their initial position towards the center courtyard in both x and y directions (2 units in x and 2 in y). Average temperature of the center courtyard covered by four patches was measured for a full diurnal cycle (24 hours) and used as an optimization measure. The sun's location was

Processor	Main Memory	GPU	Machine Count
Intel i7 CPU 960 @ 3.20GHz	*12009.2 MB*	*GeForce GTX 480*	*1*
AMD Phenom(tm) II X4 955	*7986.78 MB*	*GeForce GTX 480*	*1*
Intel i5 CPU @ 3.00GHz	*7882.25 MB*	*GeForce GTX 645*	*7*

Table 4.1: Configurations of machines used for the experiments

based on time, latitude and longitude which were set to Duluth's location on July 6th, 2013. The temperatures were calculated every hour (time step). The optimization was to find configurations of buildings with minimum average temperature of the courtyard. The number of possible building configurations were 6561 as there were 8 dimensions for movement with 3 possible values each ($3^8 = 6561$).

Since the BOINC project is in a beta stage it was not released to the public. College laboratory machines were used as clients. Nine machines with GPU's were installed with BOINC clients and configured to allow computation always (see Table 4.1). The machines were attached only to this BOINC project to avoid resource sharing with other projects.

Existing limitations of the current QUIC Energy version warranted computation of values for only one configuration per execution of QUIC Energy. The BOINC server was configured to send only one workunit for every client request. No redundant computing was employed since all the clients were trusted and there was no necessity or scope for cross validation of results. Similarly, the security mechanisms of BOINC such as checksums, server certificates were turned off to reduce network traffic. The maximum number of erroneous results that were allowed for a workunit before it got aborted by BOINC was set to three. Single instances of assimilator, work generator, and validator were run.

The allowed movement of buildings and other problem specific simulation parameters were specified through an optimization file using the language (refer 4.2. The

```
//optimization file..
BaseProjectPath = SimpleDomain2x2BuildingGrid.zip

//Solver = BruteForce
//seed = 878

quBuildings.buildings[0].xfo = [15.0:1.0:17.0]
quBuildings.buildings[0].yfo = [50.0:1.0:52.0]
quBuildings.buildings[1].xfo = [29.0:1.0:31.0]
quBuildings.buildings[1].yfo = [50.0:1.0:52.0]
quBuildings.buildings[2].xfo = [29.0:1.0:31.0]
quBuildings.buildings[2].yfo = [36.0:1.0:38.0]
quBuildings.buildings[3].xfo = [15.0:1.0:17.0]
quBuildings.buildings[3].yfo = [36.0:1.0:38.0]

fitness = avg
fitness_index = [2203 2204 2284 2283]
fitness_label = patch_temprature
```

Figure 4.2: The input optimization file for expteriment one

variable BaseProjectPath indicates the name of the folder where all the required input files were stored. quBuildings.buildings[0].xfo is the datamember that stores the x coordinate of building zero (zero as building numbers start from zero). The file indicates that the x coordinate of building zero can have values between 15 and 17 with a step size of one. The last three lines specify how the fitness value should be computed (in this case its the average patch temperature of the center four patches). The simulations identify patches using unique identifiers. The date and location were not set since the simulation defaults to current date and time (the experiment was conducted on July 6th, 2013).

In order to analyze the experiment, various statistics were collected. The work generator and the assimilator had to record both start and end times of processing in a file. For each workunit BOINC recorded amount of CPU time used, number of floating point operations performed, total time taken to download, compute, and upload the result. The application stored and printed the coordinates of all buildings into a

file and the standard error stream to facilitate validation of the language and whether all the configurations possible were computed. The output file also contained values of average temperature, minimum temperature and maximum temperature across the four patches. A program (process_results) was written to analyze the result files returned by the clients. The program parsed the files and recorded the different configurations and the corresponding courtyard temperatures. The program generated a report to indicate the total number of unique configurations, configurations that had minimum average temperature, and also threw errors when it encounters duplicate configurations.

The experiment was started by providing the work generator with the optimization file and a batch number at 02:59:14 AM. The validator and assimilator were already running as daemons. Once the work generator started pushing workunits into the BOINC database, the first result was returned and assimilated by 03:01:00. The last result was assimilated at 6:1:28 AM. The total time from starting the work generator to receiving the last result was 3 hours, 2 minutes and 14 seconds. However, on observing log files the assimilator processed only 13 workunits in the last 20 minutes. On an average each run took around 6 to 7 seconds of CPU time and around 11 seconds for a client to complete execution and report results. The averages do not factor the time spent by various daemons and processes on the server.

Out of 6561 workunits, 13 of them were aborted because of 3 erroneous results without a successful result, 15 more erroneous results were returned but those workunits were able to obtain a correct result. By running the *process_results* it confirmed that the language worked as it returned 6548 unique results (6561-13 = 6548). The results are proof that values can be changed, simulation options can be set and different configurations can be run using the language. It was estimated that a single machine would take close to 11 hours to run all the different configurations assum-

ing a single simulation takes only 6 seconds. The estimate did not factor in the time required for setting up the problem manually. Taking the estimate into the consideration the implementation was indeed successful in providing an effective computational platform for optimizing simulations.

4.2 Experiment 2

The goal of this experiment was to test if the implementation can be used to compute large scale optimization problems (similar to the 8D case mentioned in the introduction) without performance degradation. The same environment 1.1 was used, however the buildings were placed further away from the center courtyard. The buildings were allowed to move towards the center court and could move 3 units in both x and y direction. The sun's position was changed by setting the date to June 21st, 2013. The number of possible configurations were 65536 ($4^8 = 65536$). The rest of the setup is similar to the first experiment.

Figure 4.3 displays the optimization file that was given as input to the work generator. The year, month and day were changed using simple assignment statements and the range values of the building coordinates have been changed to reflect the new problem.

The work generator was launched at 21:17:16. The first result was assimilated at 21:19:37. The last result was assimilated at 19:50:40 the next day. The whole problem took 22 hours, 33 minutes and 24 seconds with 9 machines. Out of the 65536 simulations none of the workunits got aborted. However there were 108 erroneous results reported, due to which BOINC re-sent work to get correct results. A single machine would have taken approximately 109 hours only for computation.

The problem took significantly less time compared to the estimated time inter-

```
BaseProjectPath = SimpleDomain2x2BuildingGrid.zip

//Solver = BruteForce
//seed = 878

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]

fitness = avg
fitness_index = [2203 2204 2284 2283]
fitness_label = patch_temprature
suntracker.year = 2013
suntracker.month = 6
suntracker.day = 21
```

Figure 4.3: The input optimization file for experiment two

polated from the first experiment. The optimization did not have any workunits that were aborted. The implementation did work effectively for a larger optimization problem with good performance.

The configuration with the highest average temperature of 336.266K for the center courtyard was the same as the intial configuration (see Figure 4.4). The configuration with the lowest average temperature of 322.748k for the courtyard was the one where all the buildings were closest to the patch (see Figure 4.5).

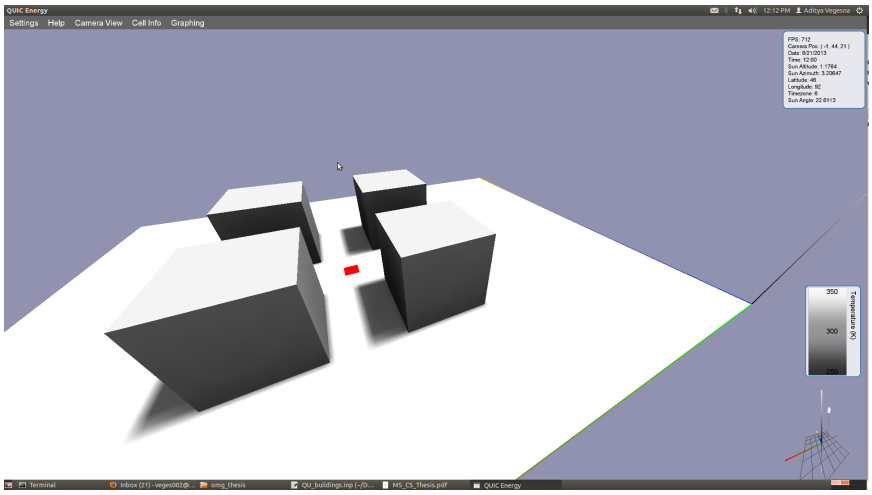

Figure 4.4: The initial configuration of the experiment and also the configuration with the highest average temperature

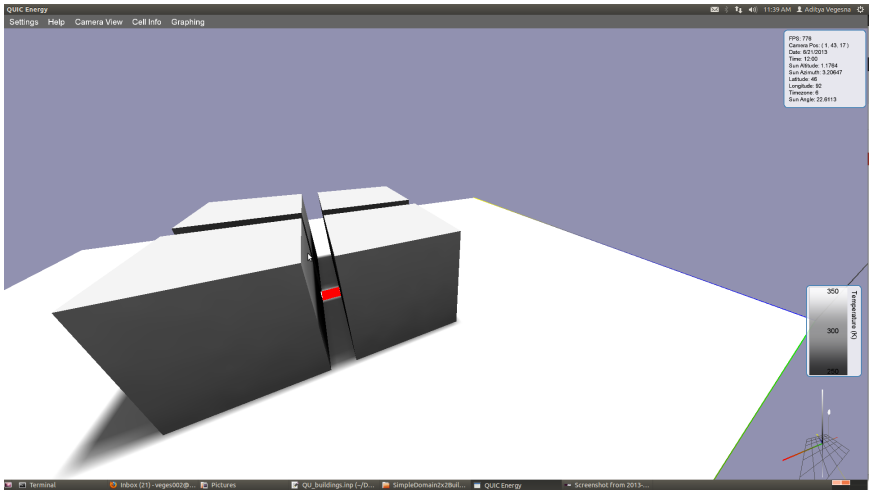

Figure 4.5: The configuration of the experiment with the lowest average temperature. The red patch is the center courtyard

4.3 Analysis

When the outputs and statistics of the two experiments were compared and analyzed interesting facts were found. Table 4.2 gives the various statistics for the two experiments.

	Experiment-1	**Experiment-2**
Number of configurations	*6561*	*65536*
Total Time (h:m:s)	3:2:14	22:33:24
Aborted WorkUnits	13	0
Execution errors	15	108
Time taken by work generator[1]	1:45:19	17:57:29
Faulty results reported as sucess	0	2

Table 4.2: Semi-Real World Condition. All values are in meters.

Overall the second experiment fared better than the first one. The number of configurations of the second experiment were almost 10 times the first experiment however the time taken was less than 8 times. There were no workunits that got aborted during the second experiment but more workunits with retries. The second experiment encountered more errors because of the sheer number of workunits that were run. However those errors did not lead to aborted workunits since the new instances were never re-sent to the same machine thereby avoiding the same failing conditions.

When *process_results* program was run on the two experiment results it detected no anomalies in the first experiment but detected two duplicate configurations for the second one. On examining various logs of the daemons, it indicated that the input files were appropriate, the workunits were correctly generated and the results assimilated. Hence the problem was not on the server side, but rather on the client side. By analyzing the standard error stream of the faulty workunits, a particular system was deemed responsible for the duplicate computations. This was caused since the application accessed an input file with the same name but belonging to the previous optimization problem rather than the current one. This error crept in because the security mechanisms were switched off. Either they can be turned back

[1]All the daemons were started simultaneously. Results were being computed by the clients concurrently with work generator.

on or instead of the trivial validator that marks all results as valid, a proper validator can be used to determine if a result is valid or not before accepting it.

There is a significant hike in the number of workunits dispatched to the clients after the completion of the work generation compared to the time while the work was being generated. This is an interesting observation and can be attributed to the fact that the work generator caused a drop in performance since work generation involves a lot of writing to and reading from the database. A scheduler can dispatch work only if the shared memory segment maintained by the feeder has workunits. A feeder can only put workunits into the shared memory once the transitioner creates the required number of results per workunit. Both the feeder and transitioner scan the database frequently for work. Another factor that could have contributed to decrease in speed is the fact that the transitioner is responsible for creating initial results and is also responsible for setting a workunit for validation, assimilation. The workload on the transitioner is a lot when operating at full capacity.

Overall the language implementation works, and a BOINC application can be used to leverage computation resources for problems that do not require immediate responses.

5 Conclusions

This work presents a computational environment to facilitate specifying and running of optimization tasks. The tasks are computed utilizing a cluster of machines. It is a step closer to the goal of designing an efficient and interactive tool that can be used by urban planners to understand complex interactions of the environment with the urban space and to assist them in making decisions for a sustainable future.

The current implementation uses BOINC as a middleware grid management system which can be improved based on its intended use. Without any modifications it is ready to be used as a low cost environment to compute results for problems that do not require immediate results. The following Future Work section gives a brief outline of improvements to the system and direction in which the research can proceed.

5.1 Future Work

The language can be improved by changing the strings used to access parameters. Currently the string is the parameter name prefixed with the class name. These names can be hard to remember and difficult to learn.

The search for optimal configuration is currently exhaustive. Instead of using brute force techniques, global optimization techniques such as simulated annealing or genetic algorithms can be used. This would decrease the number of configurations that need to be evaluated by a drastic factor. The current implementation can be modified to run a genetic algorithm on the server without much effort since there

already exists sufficient support. All that is missing is a feedback channel from the assimilator to the work generator to facilitate the genetic algorithm to generate new configurations based on the results of previous ones.

The computational environment can be used to compute solutions that do not require immediate results. However modifications need to be made to BOINC daemons to create a fast responsive cluster. As seen from the results, an improved transitioner or multiple synchronous transitioners would increase the performance drastically. Changes to QUIC Energy can be made to compute more than one configuration per execution. This would drastically reduce the network load and delays caused due to exchange of workunits.

A interactive tool can be created either for a tablet or for a virtual reality system where urban planners can design urban spaces. The planner can choose what to minimize on and the system would use the genetic algorithm to find configurations matching the user's query. The current implementation can be a system that starts computing all the configurations, while another fast responsive system would calculate what the user requested.

Bibliography

[1] Boinc. *http://boinc.berkeley.edu/.*

[2] Boost filesystem. *http://www.boost.org/doc/libs/1_54_0/libs/filesystem/doc/index.htm.*

[3] Boost propertytree. *http://www.boost.org/doc/libs/1_41_0/doc/html/property_tree.html.*

[4] Boost.any. *http://www.boost.org/doc/libs/1_54_0/doc/html/any.html.*

[5] Boost.regex. *http://www.boost.org/doc/libs/1_54_0/libs/regex/doc/html/index.html.*

[6] Einstein@home. *http://einstein.phys.uwm.edu/.*

[7] Folding@home. *http://folding.stanford.edu/.*

[8] Genusis. *http://envsim.d.umn.edu/.*

[9] Gimps. *http://www.mersenne.org/.*

[10] glite. *http://glite.cern.ch/.*

[11] Gpugrid. *http://www.gpugrid.net/.*

[12] List of boinc projects. *http://boinc.berkeley.edu/projects.php.*

[13] David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series).* Addison-Wesley Professional, 2004.

[14] E.R. Pardyjak P. Willemsen D. Johnson A. Vegesna Addepalli, B. Gpu-mcdm: A new module of the quick urban and industrial complex (quic) dispersion modeling system for urban form optimization. Dublin, Ireland., 2012. 8th International Conference on Urban Climate ICUC 8 and 10th Symposium on the Urban Environment.

[15] E. Pardyjak P. Willemsen Addepelli, B. and D. Johnson. Urban form optimization for air quality applications using simulated annealing and genetic algorithms. 2010.

[16] United States Environmental Protection Agency. Heat island effect. *http://www.epa.gov/heatisland/index.html*, January 2012.

[17] David P. Anderson. Boinc: A system for public-resource computing and storage. In *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, GRID '04, pages 4–10, Washington, DC, USA, 2004. IEEE Computer Society.

[18] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. Seti@home: an experiment in public-resource computing. *Commun. ACM*, 45(11):56–61, November 2002.

[19] David P. Anderson and Gilles Fedak. The computational and storage potential of volunteer computing. In *Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid*, CCGRID '06, pages 73–80, Washington, DC, USA, 2006. IEEE Computer Society.

[20] David P. Anderson, Eric Korpela, and Rom Walton. High-performance task distribution for volunteer computing. In *Proceedings of the First International*

Conference on e-Science and Grid Computing, E-SCIENCE '05, pages 196–203, Washington, DC, USA, 2005. IEEE Computer Society.

[21] BoincSTATS. Overall boinc stats. *http://boincstats.com/en/stats/-1/project/detail/overview*, 2013.

[22] Franck Cappello, Samir Djilali, Gilles Fedak, Thomas Herault, Frédéric Magniette, Vincent Néri, and Oleg Lodygensky. Computing on large-scale distributed systems: Xtrem web architecture, programming models, security, tests and convergence with grid. *Future Gener. Comput. Syst.*, 21(3):417–437, March 2005.

[23] CERT. Exploitation of vulnerabilities in microsoft rpc interface. *http://www.cert.org/advisories/CA-2003-19.html*.

[24] Anna Chiesura. The role of urban parks for the sustainable city. *Landscape and Urban Planning*, 68(1):129 – 138, 2004.

[25] CIA. Urbanization. *https://www.cia.gov/library/publications/the-world-factbook/fields/2212.html*, January 2012.

[26] Joshua Gordon Clark. A fast and efficient simulation framework for modeling heat transport. 2012.

[27] Ricardo M.maderia Diamantino Cruz and Rui Lopes. Volunteer computing with boinc client-server side.

[28] M. Ellert, M. Grønager, A. Konstantinov, B. Kónya, J. Lindemann, I. Livenson, J. L. Nielsen, M. Niinimäki, O. Smirnova, and A. Wäänänen. Advanced resource connector middleware for lightweight computational grids. *Future Gener. Comput. Syst.*, 23(2):219–240, February 2007.

[29] S. Halverson. Energy transfer ray tracing with optix. Master's thesis, UNIVERSITY OF MINNESOTA.

[30] Andrew P. Norgren. Gpu based particle dispersion modeling with interactive visualization support for real-time simulation. June 2008.

[31] NVIDIA. Nvidia cuda c programming guide. 2011.

[32] Mathilde Romberg. The unicore architecture: Seamless access to distributed resources. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing*, HPDC '99, pages 44–, Washington, DC, USA, 1999. IEEE Computer Society.

[33] Balwinder Singh. Development of a fast response dispersion model for virtual urban environments. 2010.

[34] William D. Solecki, Cynthia Rosenzweig, Lily Parshall, Greg Pope, Maria Clark, Jennifer Cox, and Mary Wiencke. Mitigation of the heat island effect in urban new jersey. *Global Environmental Change Part B: Environmental Hazards*, 6(1):39 – 49, 2005.

[35] James Bigler Steven G. Parker. Optix: A general purpose ray tracing engine. May 2010.

[36] Todd Tannenbaum, Derek Wright, Karen Miller, and Miron Livny. Condor – a distributed job scheduler. In Thomas Sterling, editor, *Beowulf Cluster Computing with Linux*. MIT Press, October 2001.

[37] Boinc wiki. Supported platforms. *http://boinc.berkeley.edu/trac/wiki/BoincPlatforms*.

[38] Wikipedia. Cmake. *http://en.wikipedia.org/wiki/CMake*.

[39] P Willemsen, A Norgren, B Singh, and E Pardyjak. Development of a new methodology for improving urban fast response lagrangian dispersion simulation via parallelism on the graphics processing unit. In *11th Intl Conf on Harmonisation within Atmospheric Dispersion Modeling*, pages 363–367, 2007.