Osmond Gunarso
CSS 499
Spring 2015 Final Report
Fukuda

FLuTE MASS

FLuTE is an stochastic model of simulating the spread of influenza. FLuTE MASS is an agent based version of that simulation that takes advantage of the MASS library to generate it's `agent`s and `place`s. The goal of this project is to compare the programmability of the MASS library to OpenMP/MPI by implementing an `agent` based simulation.

How the simulation works

The FLuTE simulation is comprised of two primary data types `people` and `communities`.

Person / people

`people` is a representation of a person in the simulation. They carry with them information about where they live, their age, and the information needed to calculate spread of infection. The person's information is randomly generated from the population statistics from the census tract information being loaded into the simulation. Agents perform the bulk of the computation in the simulation, calculating the spread of the disease. Agents perform the basic computation as follows

```
If I am not sick
    interact with every sick person in the community
        if I get sick stop and change my state
else
    do nothing
Save current state
Migrate to my next location
```

in order for the simulation to work in parallel the data in an agent is saved in two stages. The first is the `current` state of the agent, e.g. were they sick in the last turn. This information is read in parallel by many different agents and is not modified for the duration of the turn. The second state is the `future` state of the agent, e.g. did they get sick during the current turn. Only the agent is allowed to modify its future state, and the information is saved at the end of the turn. The order of interactions was the primary difference between the MASS and OpenMP/MPI implementations of the simulation. In the OpenMP/MPI implementation of the simulation each sick agent iterated over a list of the healthy agents and removed them from the set of healthy agents as they became sick. Whereas the MASS implementation of the simulation it is the healthy individuals whom loop over the sick and are added to the set of sick individuals after the turn is done.

The set of sick individuals that the `person` will iterate over is different for each turn. During the `day` the individual will loop over the sick individuals who are present in its community if they haven't been quarantined and during the night the individual will loop over its logical `neighbors'

migration

Migration between communities by agents is a frequent operation. Most migrations are from day to night communities. Since the total number of communities is a number that is known at the start of the execution each individual can be assigned to a `day` and `night` community at the start of the simulation. So when the it comes time to move to the next location the `person` will always know which community to migrate to. The `day` and `night` locations are kept as the first two elements in the migratable Data space. For example, the moveDayTime function looks like such.

```
void *Person::moveDayTime(void *argument) {
        //move to working location
        vector<int> dest;
        int destinationtract = ((int*)migratableData)[0];
        dest.push_back(destinationtract);
        dest.push_back(0);
        migrate(dest);
}
```

When it comes time to travel the individual person will migrate to another community selected at random. Since the total number of communities is known, and the community location is the same as the community id travel can be done in a similar manner.

```
// We're going to DisneyWorld!
double r = get_rand_double;
int i;
for (i = 0; r > travel_length_cdf[i]; i++)
        ; //scan untill travel length is found

((int*)migratableData)[nTravelTimer] = i + 1; //stay for this many days
vector<int> dest; //migrate
// assign traveler to a community in tract
int destinationtract = srand(UNSIGNED_SEED) % nTotalcomm;
dest.push_back(destinationtract);
dest.push_back(0);
```

Information that is global and unchanging is kept in static variables inside of the class. In the case of nTotalcomm two copies are kept in the `person` class as well as the `community` class which are assigned inside of the class's init function.

Interactions

One of the largest parts of the simulation is the ability for agents to interact with one another through their place. Part of the `Place` class is a vector of MObjects that can be cast to their respective classes. Because the FLuTE simulation only requires one kind of agent we can safely cast all the agents in the vector to `person`. Once that is done we can inspect the data that resides in each class and then perform the necessary operations. For instance, when the `person` is interacting with sick members in its community it performs an iteration over a vector of `person` which was previously made by the `community`. At the start of the turn the `community` will iterate over its local agents and send `sick` residence into a separate set like so

```
void *Community::sortSick( void *argument ) {
  //empty the current sick vector
  sick.clear();

  for (vector< MObject* >::iterator it = agents.begin(); it != agents.end(); ++it) {
    Person* p = (Person*) *it;
    if (isInfectious(*p)) {
      sick.push_back(p);
    }
  }
}
```

Communities

A `Community` is a logical grouping that represents people who live geographically close together. The `community` is not the only logical grouping of people based off of geography. There are also 'Tracts' `neighborhoods` and `families`. A `Tract` is a census tract and is defined as "Census Tracts are small, relatively permanent statistical subdivisions of a county or equivalent entity that are updated by local participants prior to each decennial census as part of the Census Bureau's Participant Statistical Areas Program." In this simulation a tract consists of around 2000 people. `Communities` are than subdivisions of tracts that contain anywhere from 500 - 700 individuals, who live there. During the loading of a simulation a `person` is assigned two `communities` a `day` and `night` community. Depending on the age of the person the `community` can either represent where the person works, or where the person goes to school. The second `community` that the person is assigned is the `night` community. This is where the individual lives. It is possible for an individual to have the `day` and `night` community be the same. The `night` community contains a `person`'s `neighborhood`, `family cluster` (a group of four close `families) and , `family`. Each collection of people represent a group of individuals whom are more likely to infect each other than the last, so the person is more likely to be infected by a family member than by a neighbor because of the level of interaction.

The primary responsibility of the `Community` is to provide a logical grouping of individuals. However; much of the state of the simulation is kept in the `Community`. Between agent migrations the communities collectively react to the current state of the population and handle the policy decisions that are set in the simulation. Distribution of vaccines and antivirals is handled by the `Community` class as well as sorting sick and healthy individuals. The assignment of a `person` to a `family`, `family cluster`, and `neighborhood` is done by the `community` as well. The general policy enforcing algorithm goes as such

```
start vaccines
    Open schools as appropriate
    Count ascertained cases of infection
    If there is an epidemic
adjust migration policies
    Close or open schools
    Take stock of vaccines
    Distribute vaccine
```

```
        Repeat for antivirals omitting schools
```

In terms of the MASS library, each ‘community’ is a new object that inherits from the ‘place’ class. The identity of the `community` is equivalent to the location in the distributed array. So when an agent migrates to a new location the vector will always look like such [x][o] where x is the id of the target community. Initially [x][y] would represent tract, community but the issue arose where not all tracts would have the same number of communities making it difficult to ensure that the community an agent would migrate to would be an active one. Now it is the case that each community keeps track of it’s own tract id and then actions are broadcasted with a tract ID and communities discard calls unrelated to themselves.

Master node

The Master node keeps track of the entire simulation, this is not so much of a class as it is just a driver function. Loading of configuration and data files goes into this node. It can be said then that the driver function’s logic is quite simple and follows these simple rules.

```
While run count > max number of turns
    Agent interactions
    Move to night state
    Agent Interactions
    Move to day state
    Update State and Places
    increase run count by 1
```

The setup and distribution of the of communities and people is also handled

Considerations

Performance

The MASS version of the simulation has been shown to run slower than the OpenMP/MPI version of the simulation, performance results are listed at the bottom of this paper.

Programmability

The largest difference between the OpenMP/MPI version of the simulation and the MASS version is the paradigm used in development. The MASS version uses the Object Oriented paradigm for development. What were originally struts `people` and `community` became classes with methods attached. The introduction of Object Oriented programming made the simulation significantly simpler when comparing the two using Log Metrics

MPI	MASS
● 13 Files ● 6369 Significant LOC ● 6730 LOC ● Cyclomatic complexity 801	● 19 Files ● 5282 Significant LOC ● 5793 LOC ● Cyclomatic Complexity 566

However; because of the popularity of OpenMP/MPI finding resource on how to use the library and how to debug the simulation were significantly easier to do that with the MASS library, where most questions did not have documented answers and I had to ask either Professor Fukuda or another member of the distributed systems lab to find the answer. So while the code produced using the MASS library should be easier to maintain, access to the resources needed to learn the MASS library and debug the system are much harder to access.

How to run

1 set up the MASS cluster
2 first run the make massflute command
3 then run ./massflute [path to config file] [username] [password] [# threads] [#processes

StarCluster MASS installer

Starcluster is a tool developed at MIT to make deploying MPI/OpenMP clusters simple on AWS. During the development of FLuTE the UWB css labs were having stability issue so I turned towards developing on AWS.

Development

Starcluster plugins are all written in python. The behavior of the MASS installer is to first clone the most recent push of the master branch of whatever version of MASS is installed and then to modify the bash profile of root on the master node of the cluster to enable the properties required for mass to execute. There is a list of machines to execute in under the /home route as well as a function in the bash profile that will create a symbolic link between the current working directory and the MASS library.

Issues

Most of the development of the mass installer was quite simple, performing bash commands through python code. However; enabling communication between the nodes in the MASS library was difficult to establish. Starcluster sets up passwordless ssh between all the nodes in a given cluster. OpenMP/MPI, which Starcluster was developed for, requires that there be passwordless ssh between the nodes in the cluster. MASS however requires the username and password of the user running the application. Since there was no default username and password for logging in and out of all of the nodes, starting an application with root and then the root password did not work. As it turns out MASS does support ssh communication between nodes by using a public / private key pair. In order to enable MASS to communicate through public / private key the location of the rsa files needed to be modified inside of the mass library to the absolute path of the key files

Usage

To use the MASS installer you must first add the plugin to your plugin directory listed under the .starcluster folder. Simply including the plugin to the plugins directory will instruct starcluster to run the plugin when making a new cluster.

The MASS installer will modify the bashrc profile on the master node, so when running the MASS application it is important that you only do so from the master. The MASS library will be installed on the NFS server underneath the path /home/MASS. Additionally a machinefile.txt will be installed under /home/machinefile.txt. This file will contain a list of all the non-master nodes in your cluster. If you wish to add additional nodes to the cluster, it is important that this machinefile.txt folder be copied.

Exports will already be included in the bash profile of the master node, so when compiling and running the application it is unnecessary to re-export library variables. The MASS installer will only install one version of MASS, specified in the configuration, so it it unnecessary to include the version in the path like it was on the uw1-320 machines. Lastly, before starting your application be sure to run the setup command to make a symbolic link between your current working directory and the MASS library. This will make links for the killMProcess.sh file as well as MProcess, both of which are requirements to run the MASS library.

Before running your application

A modification will need to be made to the MASS library before you can begin using it. In your local copy of MASS you will need to change the Utilities.h file listed under /home/MASS/source. In the initializer list for the constructure the values ~/.ssh/id_rsa.pub ~/.ssh/id_rsa are passed to keyfile1 and keyfile2 respectively. The MASS library takes advantage of libssh2 to provide the transport between nodes, unfortunately libssh2 is not able to resolve the ~/ relative path. Therefor the values passed to the constructors should be the absolute paths to the rsa file, typically this will be /root/.ssh/id_rsa.pub and /root/.ssh/id_rsa for keyfile1 and keyfile2 respectively.

Configuration

In your configuration file you will need to add the following lines to correctly use the MASS starcluster plugin.

```
[plugin MASSInstaller]
setup_class = MASSInstaller.MASSInstaller
```

```
username = bit bucket username
password = bit bucket password
version = cpp
```

Steps to use MASS installer

1) Download the MASS Installer from https://github.com/kingozzy24/massinstaller
2) copy the MASSInstaller.py file into the plugins directory listed under your starcluster files.
3) Add the configuration text from the configuration section into the starcluster configuration file.

Metrics

L.A. Population

Tracts: 2049

Communities: 5547

Individuals: 11095039

```
========= Benchmark summary start MPI =========
Execution cpu time: 2338.04
size: 1
config: ./plos-flute-configs/config.la-1.6
========== Benchmark summary end ==========
========= Benchmark summary start MPI =========
Execution cpu time: 1085.1
size: 2
config: ./plos-flute-configs/config.la-1.6
========== Benchmark summary end ==========
========= Benchmark summary start MPI =========
Execution cpu time: 525.81
size: 4
config: ./plos-flute-configs/config.la-1.6
========== Benchmark summary end ==========
========= Benchmark summary start MASS =========
Execution cpu time: 2344.64
size: 1
config: ./plos-flute-configs/config.la-1.6
```

```
========== Benchmark summary end ==========
========= Benchmark summary start MASS =========
Execution cpu time: 1852.63
size: 2
config: ./plos-flute-configs/config.la-1.6
========== Benchmark summary end ==========
========= Benchmark summary start MASS =========
Execution cpu time: 905.16
size: 4
config: ./plos-flute-configs/config.la-1.6
========== Benchmark summary end ==========
```