STAT250 HW2

Yu Pei
SID999438479

February 13, 2014

1 Introduction

In this session, we explore the usage of pthread in C, a little higher level's OpenMP, R's parallel package, and Hadoop's JAVA API, C++ Pipe and Streaming. Since the instructor Duncan had implemented a lot of them, so the ones that are not shown in here can be found in **stat250.a1**repository.

The answers we are trying to compute here still are the median and mean of Arrival delays for all the records. Since I want to experiment with as many different paradigms as possible, I didn't update the method to work for the two different formats and NA handling is also not dealt with. But they all can be run with a consistent formatted set of files.

2 R parallel packages

This method is the most easy one. We can direatly utilize the code written in C from last assignment to parallel over different files. Code is in github repo but is also included in appendix for easy viewing.

In the ArrDelays package, main_funcC can take a vector of files names then return the Frequency tables for each files. So we load the parallel package then create cluster nodes in a single machine. Then split the files to the number of nodes we create using ClusterSplit. Then all we need to do is to use clusterApply to send out the names of files(in a list)and the function so they can calculate in parallel.

Since we use a share memory model, we don't need to worry about file locality problem and message passing. Life is good. If we time the run time versus number of nodes, we can see a linear trend between the two.

A straight line graph goes here.

3 OpenMP

In this part, I take advantage of the code in Duncan's **AirlineDelays**package. And write it as a stand-alone C executable instead of integrating with R. The basic structure is the same as the pthread code, but we just need to say pragma parallel and it will automatically create the threads for us. And we can benefit from the **for**pragma to help us set up the parallel, so we do need to explictly set up **single**,**barrier** or **critical**. In fact, we use the implict barrier at the end of the for pragma then we just do a normal for loop over the resulted freqency tables. Which is trvial to do.

Note we set the table array to global so everybody can access it. We can set the maximum number of threads to use using environment variable :

$$export\ OMP\_NUM\_THREADS = num$$

The resulting time is approximately linear in time. Although it is hard to control exactly how many nodes it will use.

4 Hadoop streaming with C code

Hadoop is a scatter/gather model parallelism. And it is easy to use(once the cluster has been set up properly!) because it has done a lot of things for you implictly. With mapper reading lines from STDIN and emitting (key value)pairs, Hadoop will sort the key implictly for us, then pass the output to reducer, where we count the number of lines for a specific delay values. Since it won't be able to tell the change point from one key values to another, we need to check that by ourselves.

5 Hadoop JAVA API

This method inherit the Mapper and Reducer classes defined in Hadoop class. it works mostly the same as Streaming mode, but since different reducer will have distinct key(maybe few different keys but all those same key will be in one reducer), We don't need to check if we have reached the end of one key like we did in Streaming mode.

6 Appendix Code

```
library(ArrDelays)
library(parallel)

files = list.files(,pattern = '(19.{2}|200[1-7])\\.csv$')
cl = makeCluster(2, 'FORK')
ff = clusterSplit(cl, files)
system.time(result.list <- clusterApply(cl, ff, main_funcC))

result = result.list[[1]]
for(i in 2:length(result.list)){
  result = result +result.list[[i]]
}
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "readRecords.h"
#include <omp.h>



/* Initialize a Table object, allocating it if necessary
 or filling in an existing instance.  */

Table *
makeTable(Table *tt)
{
    if(!tt)
        tt = (Table *) malloc(sizeof(Table));
```

```
    tt->min = - MAX_NUM_VALUES/2;
    tt->max = MAX_NUM_VALUES/2;
    tt->numValues = MAX_NUM_VALUES + 1;

    memset(tt->values, 0, sizeof(int) * (MAX_NUM_VALUES + 1));
    return(tt);
}

/* Increment the count for the specified value in the given table */
void
insertValue(int value, Table *t)
{
    if(value < t->min || value > t->max) {
        fprintf(stderr, "%d is outside of the range of the table\n", value);
    } else {
        int i;
        i = value - t->min;
        t->values[i] ++;
    }
}


/* Display a table on the console/terminal.
   This is used in the standalone version.
   So it doesn't use's R's print routines.
  */
void
showTable(Table *t)
{
    int i;
    for(i = 0; i < MAX_NUM_VALUES; i++) {
        if(t->values[i] > 0)
            fprintf(stderr, "%d: %ld\n", t->min + i, t->values[i]);
    }
}

/* process a file, line by line. */
double
readDelays(const char *filename, Table *data, int fieldNum)
{
    FILE *f;
    char line[MAX_NUM_CHARS];

    f = fopen(filename, "r");
    if(!f)
        exit(1);

    // header line
    fgets(line, MAX_NUM_CHARS, f);

    int val;
    while(fgets(line, MAX_NUM_CHARS, f)) {
        val = readRecord(line, fieldNum);
```

```
        insertValue(val, data);
    }

    return((double) val);
}


/* Read an individual record in a file,
returning the value of the ARR_DELAY variable. */

int
readRecord(char *line, int fieldNum)
{
    int i = 0, field;
    char *val;

#if 0
    char *tmp;
    for(i = 0; i < 43; i++)
        val = strtok_r(val, ",", &tmp);

#else

    for(i = 0, field = 0; i < MAX_NUM_CHARS; i++) {
        if(line[i] == ',') { // used = rather than ==
            field++;
            if(field == fieldNum) {
                val = line + i + 1;
            } else if(field == fieldNum + 1) {
                line[i] = '\0';
                break;
            }
        }
    }

#endif

    return(atoi(val));
}

/*
  Merge several tables into a single table.
This sums counts from the different tables
  for the same value.
  Could do this with threads,
but probably not worth the overhead. */

Table*
combineTables(Table *table, Table *out)
{
    int i;

        for(i = 0; i < out->numValues; i++)
```

```
            out->values[i] += table->values[i];


    return(out);
}


Table *tables[30];
int
main(int nargs, char *argv[])
{
    Table * out = makeTable(NULL);
        Table tables[0] = makeTable(NULL);
    int t;
        #pragma omp parallel
        #pragma omp for shedule(static)
    for(t = 1; t < nargs; t++){
        tables[t] = makeTable(NULL);
        readDelays(argv[t], tables[t], FIELD_NUM);
                }

    for(t = 0; t < nargs - 1; t++)
            out = combineTables(tables[t], out);
        showTable(out);
    return(0);
}
```

```
//Mapper
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define FIELD_NUM 14
#define MAX_NUM_CHARS 2000



/* Read an individual record in a file,
 returning the value of the ARR_DELAY variable. */

int
readRecord(char *line, int fieldNum)
{
    int i = 0, field;
    char *val;


    for(i = 0, field = 0; i < MAX_NUM_CHARS; i++) {
        if(line[i] == ',') { // used = rather than ==
            field++;
            if(field == fieldNum) {
                val = line + i + 1;
            } else if(field == fieldNum + 1) {
```

```
                line[i] = '\0';
                fprintf(stderr, "%s\t1\n", val);    //print out the key value pair
                break;
            }
        }
    }


    return(0);
}



int
main(int nargs, char *argv[])
{

        char line[MAX_NUM_CHARS];
        while(fgets(line, MAX_NUM_CHARS, stdin)) {
        readRecord(line, FIELD_NUM);
    }
    return(0);
}


//reducer

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define BUFFER_SIZE 50
#define DELIM   "\t"

int main(int argc, char *argv[]){
    int count =0,delaytime,currentdelay=10000;//Arbitrary large number
        char line[BUFFER_SIZE];
        while( fgets(line,BUFFER_SIZE -1, stdin)){
            delaytime = atoi(strtok(line, DELIM));
                if(delaytime == currentdelay){
                    count += 1;
                }else {
                    if(currentdelay != 10000){
                        fprintf(stdout,"%d\t%d\n",currentdelay,count);
                        }
                currentdelay = delaytime;
                        count = 1;
        }

        }
    fprintf(stdout,"%d\t%d\n",currentdelay,count);
return(0);
}
```