Brendan Celii, Danh Nguyen

October 20, 2014

35103176

**Lab 3 Testing Analysis:**

The following methods were attempted and tested as possible solutions. Each implementation and explanation of testing will be explained below:

1. Merge Sort
2. MergeSort with length prefixed string
3. Merge Sort, Insertion Sort Hybrid
4. Quicksort
5. Quicksort without passing array between functions
6. Multiple variations of quicksort with multi-threading

First, the overall design of our program is as follows. We first read in the data from the input file by starting off with clearing all previous values in the vector we used to store the input, and then all words from the input file are stored in the vector by putting the last word encountered at the very end of the vector. The data is then prepared for sorting by copying all the words in the vector into a standard array of char\* in the same order that they were stored in the vector. At the beginning of this prepareData() function the array is cleared and any dynamically previously used for values in the array of an old test are deleted. The sortData function is then called in which the sorting method of our choice will sort our array of char pointers, and then the ordered characters are output to a file

The sorting method that we first considered was a standard Merge Sort. The way we implemented this was by following the basic merge sort procedure of dividing the data recursively until the level of division was a size of 1, and then to merge the data sets back together in the correct order, we simply replaced the regular comparison operators with a function called comparedWords(). This function received 2 char\* as arguments and first compared the char arrays using strlen(), and if the char arrays were the same size then the char arrays were compared using strcmp in order to alphabetize them, the results of these comparisons were the return value of the function. The first improvement we then made to this sorting algorithm was to prepare our data as length-prefixed char arrays so as to avoid having to call the strcmp() function, which would involve overhead and time. We could do more work upfront when preparing the data by finding the length of the string and then putting the length of the char array at the beginning so that the comparison of two char array sizes would only involve looking at the beginning elements. We first implemented this by storing the numbers that defined the length as characters (such as ‘1’,’9’ for a length of 19), but then we optimized it even more where we just arranged that the first character of every array be a character whose actual value (since char are really stored as a number) would represent the length of the char array. For example, if the first character of a char array was ‘#’ then the char array would have a length of 35. This was advantageous because we would only need to devote one character to defining the size of the char array, and to compare the sizes of each char array would only need the line str1[0] > str2[0], avoiding a function call with only one comparison. We then did some tests sorting files with the strlen() function call and the length-prefixed string method, and it was clear the length-prefixed string was more efficient.

A major way we analyzed if whether one sorting method was faster than another, was that we creating a function called algorithmTester. The code is included at the end of this document, but the function would essentially, create a random file of data size N, and as many specified sorting algorithms on this data size N 30 times, average the runtimes (in nanoseconds), and then output this average to a csv file. This analysis would continue for a random file of data size 2N, 3N…KN until KN was larger than the max size specified. This csv file could then easily be imported into excel and analyzed.

The next way we improved was by implementing a standard quicksort using the length pre-fixed strings. We then went on to improve the quicksort by instead of passing the prepared array of char arrays between quicksort functions, to just access the char\*\* as a private data member. The merge sort, standard quicksort, and quick sort with private data member access (called quicksort2) were compared using the algorithmTester and the results were as followed. We then concluded we should continue with the quicksort2, and that the quick sort method was clearly preferable to mergeSort (especially when the data set started to get larger).

|  |  |  |  |
| --- | --- | --- | --- |
| N | quickSort1 | quicksort2 | mergeSort |
| 100000 | 0.048169547 | 0.042569093 | 0.100605797 |
| 200000 | 0.11607377 | 0.116490687 | 0.5701335 |
| 300000 | 0.179576987 | 0.183843853 | 1.220370087 |
| 400000 | 0.32472194 | 0.254348093 | 2.22501498 |
| 500000 | 0.39767649 | 0.372024277 | 2.764880343 |
| 600000 | 0.469144953 | 0.45310969 | 3.707558283 |
| 700000 | 0.494695583 | 0.44705896 | 3.901025193 |
| 800000 | 0.562299403 | 0.55636572 | 4.609267843 |
| 900000 | 0.617919667 | 0.612335963 | 5.078095007 |

We then implemented a hybrid sorting method using quicksort and insertion sort. Specifically, we found that insertion sort was actually more efficient than quicksort as the data set got much smaller. So we created a function quickSortWithInsertion(int X), that would implement the basic recursive quick sort algorithm, but once the data sets were divided down to a size of X, passed as an argument, these data sets would be finished sorting by insertion sort. However, we did not know the optimal X at which to switch between insertion sort and quick sort, so we used the algorithm tester to compare our normal quicksort2 and multiple quickSortWithInsertion(int X) where the X varied in order to test which X produced consistently a faster run time. The results were as followed, so we decided to establish the pass off size as 3 (it was very close a lot of the time though and after researching online there did not appear to be a clear consensus so we just picked based on our results). The same hybrid method with merge sort was attempted by but it was quickly observed that it was slower than the quicksort approach.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| N | quicksort2 | insert (1) | insert (2) | insert (3) | insert (4) | insert (5) | insert (6) | insert (7) |
| 100000 | 0.04319541 | 0.042061023 | 0.040960423 | 0.042595253 | 0.042194877 | 0.041394303 | 0.042294737 | 0.041627643 |
| 200000 | 0.09499677 | 0.092094813 | 0.09553021 | 0.094095823 | 0.093361937 | 0.09042716 | 0.092995457 | 0.09179459 |
| 300000 | 0.144696843 | 0.144363087 | 0.144096143 | 0.14533076 | 0.144395963 | 0.14533045 | 0.144396233 | 0.143629277 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| N | quicksort2 | insert (0) | insert (10) | insert (20) | insert (30) | insert (40) | insert (50) |
| 100000 | 0.03885913 | 0.038258833 | 0.038125257 | 0.0380252 | 0.037858403 | 0.037058123 | 0.03722474 |
| 200000 | 0.086591357 | 0.087859023 | 0.085423637 | 0.086391217 | 0.085857247 | 0.086924623 | 0.08522357 |
| 300000 | 0.150300253 | 0.152468363 | 0.15036718 | 0.150100543 | 0.151000573 | 0.151301143 | 0.149766647 |

Our last optimization involved incorporating multithreading into our program to increase efficiency. We implemented this using the omp library. Our first implementation involved using multiple threads to carry out the insertionSort sorting when the sorting function switched from quick sort to insertion sort. We then compared this multi-threading technique to our previous hybrid quicksort method (with the algorithmTester() ), and we observed that incorporating multi-threading made it faster. We then formulated the idea of creating a multi-threaded program that would divide the initial data set into 4 data sets, quick sorting each data set using a separate thread, and then merging the 4 sorted arrays back together in sorted order. However, we tested this method before fully implementing it, and it was observed that the runtime for just sorting (not sorting and then merging) half of the data set with 1 thread and the other half with another thread with quicksort was consistently slower than just calling the same quicksort twice with the main thread. After continually testing it, researching online and talking with some of the TA’s, we could not generate a good solution as to why (probably just too much overhead associated or other factors) and since we did not have a complete understanding of multi-threading with this method, we decided that our final approach would be the hybrid quicksort/insertion sort that implemented multi-threading that performed the insertion sort that was explained above.

The function used for testing:

void SortingCompetition::algorithmTester(void){

int A = 1;

int stepSize = 10000;

int max = 10001;

//number of sorting methods implemented

int sortingCount = 1;

//setting number of words in RandomInput.txt

this->inputSize = A\*stepSize;

//output header to file for output

ofstream output("SortingAnalysis.txt");

output<<"N,";

for(int y = 0;y<sortingCount;y++){

switch(y){

case 0:

output<<"Bubble Sort,";

case 1:

output<<"Merge Sort,";

}

}

output<<endl;

//close output file temporarily while make the randomInput file

output.close();

while(A\*stepSize<max){

//function call to make random input file from the words collected from input.txt

makeRandomFile(A\*stepSize,"RandomInput.txt");

//processing now done on

setFileName("RandomInput.txt");

//prepare words2 with RandomInput file instead

readRandomData(stepSize);

prepareData();

//re-open output file and append to the end

output.open("SortingAnalysis.txt",std::ios\_base::app);

//output input size to output file

cout<<A\*stepSize<<", ";

//find average of all sorting methods for this specific

//input size

for(int i = 0;i<sortingCount;i++){

double sumRuntime = 0;

for(int j = 0;j<30;j++){

//declare 2 time points

std::chrono::time\_point<std::chrono::system\_clock> start, end;

//store current time (now()) in start

start = std::chrono::system\_clock::now();

//decide which sorting method to use

switch(i){

case 0:

bubbleSort();

case 1:

mergeSort(0,getWordCount()-1);

}

//store time(now()) in end

end = std::chrono::system\_clock::now();

//get No. of seconds elapsed & output duration

//need to do this multiple times & get average

std::chrono::duration<double> elapsed\_seconds = end-start;

sumRuntime += elapsed\_seconds.count();

}//end of 30 runtimes

//find average runtime

double average = sumRuntime/30;

//output to file the average run time and size

cout<<fixed<<setprecision(9)<<average<<", ";

}//processed all sorting algorithms

//increment input size

A +=1;

this->inputSize = A\*stepSize;

cout<<endl;

}

}

//making random file filled with specified length by pulling form

//the list of words collected from input.txt

void SortingCompetition::makeRandomFile(int size,char\* name){

//seed random number generator

srand(static\_cast<unsigned int>(time(0)));

//open input file

ofstream randomFile(name);

int dummy = getWordCount();

//populating input file with random words form base input.txt file

for(int i = 0;i<size;i++){

int index = rand()%(words.size());

char\* dummyWord = words.at(index);

//write random word tou randomFile file

randomFile<<words.at(index)<<" ";

}

//close file

randomFile.close();

}

//will read in words from input file

void SortingCompetition::readRandomData(int stepSize){

//free memory from older copy of data

for(size\_t i = 0; i < (inputSize - stepSize); i++)

{

delete[] words2[i];

}

delete[] words2;

/\* //store off the size of the previous wordCount

setPreviousWordCount(words.size());\*/

//clear all existing words in the vector in order that

//if readData run multiple times that words are not just concatenated

//onto the end of the existed vector from a previous run

for (size\_t i = 0; i < wordsRand.size(); i++)

{

delete[] wordsRand.at(i);

}

wordsRand.clear();

//collect and store words from input file

string buffer;

for(int i = 0;i<inputSize;i++)

{

fin >> buffer;

//store pointer to a dynamically allocated

//char\* at end of vector

wordsRand.push\_back(new char[buffer.length()+1]);

//copy word into chra\* pointer

strcpy(wordsRand.at(wordsRand.size()-1),buffer.c\_str());

}

setWordRandCount(wordsRand.size());

//close input file

fin.close();

}