CS840 Project 7:

C++ Profiler

Samuel Gluss

Jozo Dujmovic

5-23-2016

Abstract – Each line of code in a program takes a certain amount of time to execute. Therefore, the total program execution time is the sum of all the line execution times. In order to find where a program spends most of its time, knowing the time to execute a line of code is only part of the battle. The number of times each line is executed is just as important. Though this can be calculated, a profiler can be used for determining which lines in a program are run the most. Then, effort to optimize the program can be focused in these areas.

# Introduction

The profiler developed in this project leverages metaprogramming with Python to instrument and analyze C++ programs. This profiler is capable of working on any single-file C++ program, but could be extended to work with multiple file programs.

The profiler works by adding counters to all the scopes in the C++ program, which is basically any code enclosed by curly brackets. It also adds a file output to the C++ code, so that the profiler output is separated properly from the normal output of the program. This modified program code is then executed, and the recorded values from the code block counters is fed back into the script. Finally, the script prints the original C++ source code, with appropriate annotations inside code blocks to indicate how many times each logical line of code was executed.

Contents

[Introduction 1](#_Toc452014091)

[Methodology 3](#_Toc452014092)

[Results 4](#_Toc452014093)

[Conclusion 7](#_Toc452014094)

[Appendix: Profiler Output 8](#_Toc452014095)

[FizBuz Profiler Output 8](#_Toc452014096)

[Longest Consecutive String of Digits Profiler Output 9](#_Toc452014097)

[Matrix Inversion Profiler Output 11](#_Toc452014098)

[Appendix: Original Source Code 14](#_Toc452014099)

[FizBuz Original Source Code 14](#_Toc452014100)

[Longest Consecutive String of Digits Original Source Code 15](#_Toc452014101)

[Matrix Inversion Original Source Code 17](#_Toc452014102)

[Appendix: Profiler Instrumented Code 20](#_Toc452014103)

[FizBuz Instrumented Code 20](#_Toc452014104)

[Longest Consecutive String of Digits Instrumented Code 21](#_Toc452014105)

[Matrix Inversion Instrumented Code 23](#_Toc452014106)

[Appendix: Profiler Instrumented Code Output 27](#_Toc452014107)

[Appendix Profiler Code 28](#_Toc452014108)

[Appendix: Gnuplot Log Scale Histogram Generator 31](#_Toc452014109)

# Methodology

For this project, a Python program is used to instrument, build, and execute C++ code, which then outputs results from the instrumentation counters. The python script reads this output file, and outputs the original source code, annotated with the runtimes for each line of logical code.

The script first reads the C++ source code into a string, and counts the number of left curly braces in the file. It then checks to see if it has the libraries <iostream> and <fstream> already. If not, these library includes are inserted. With the number of curlies, the script then goes back through the program, inserting a counter at each brace. Once this is done, a global counter array is added at the top of the file, with the libraries. These counters are of type long long, so that they won’t be easily overflowed by code that is executed extremely frequently.

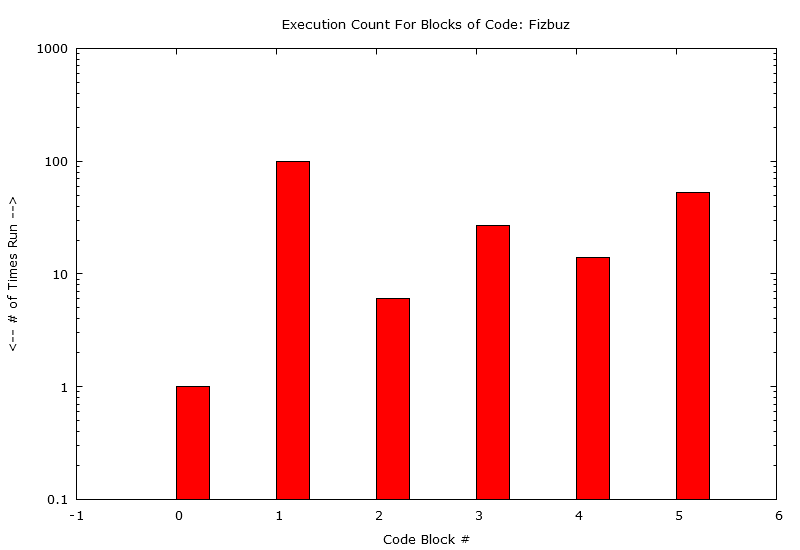
Next, the script seeks the main function using a regex. Once it finds “int main()”, it regex searchs from that point for the first occurrence of “return 0;”, then inserts code that causes the C++ program to output the counters as a newline separated string to an output file. This is done so that the counter output is completely segregated from the normal program output. The counters are delimited by newlines so that the outfile can also be read by programs such as Gnuplot for data visualization.

The C++ file is now completely instrumented, and is written to a new C++ file so as to not alter the original source code. The CL compile command is then invoked on the instrumented source code, producing the instrumented executable. This executable is then executed by the script, to produce the output file.

The python script resumes execution after the C++ program returns, and reads the produced outfile into a list, and the original unaltered C++ program in as a string. The script uses a pushdown automata to step through the source code. Every time a left curly is matched, the next counter value is popped off the list and pushed onto the stack. Every time a right curly is matched, the top counter on the stack is popped off. Each time a semicolon is read, the current line is considered an LLOC (Logical Line Of Code) and the LLOC state is set to 1. Then, if a newline character is read, and the counter stack is non-empty, the script will print the line of code, along with the counter value for that line. Any read newline char resets the LLOC state to 0.

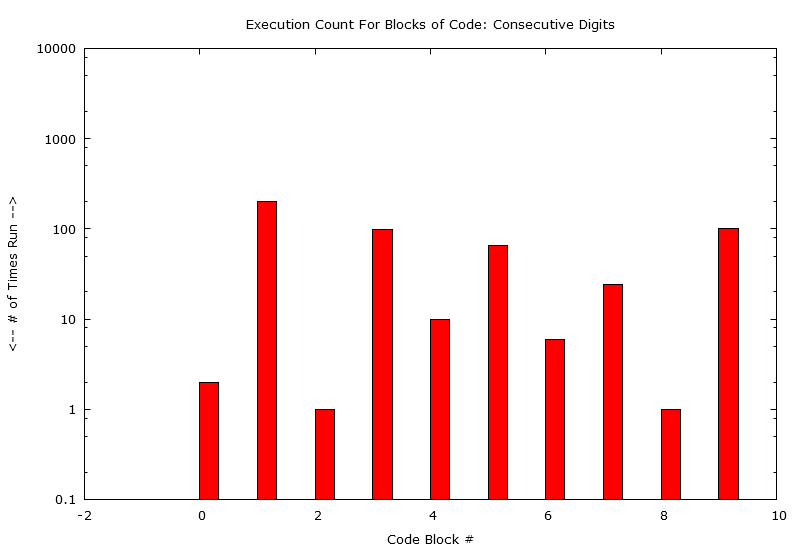
This will print the entire original source code, with all LLOC annotated with their run-count. The outfile is saved in a subdirectory where it is can then be read by a Gnuplot script, which shows the code block runcounts with a logarithmic y-scale.

# Results



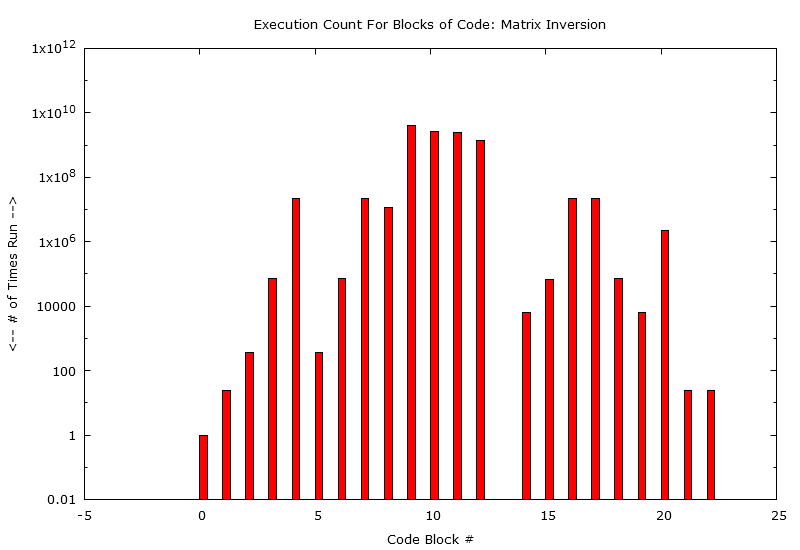
Fizbuz is a very simple program with no function calls, and was simply used as a trivial test case for the intitial profiler prototypes. This was a good program to test with because its run counts are already known, and it is easy to verify that the profilers is working properly. For example, the for loop code block executes 100 times, and the sum of the blocks inside should also be 100. Once this was working correctly, the profiler could be used on other programs confidently.

From the histogram, it is evident that certain blocks in Fizbuz are executed over an order of magnitude more than others. In future tests this disparity will become even more evident.



Consecutive Digits is a C++ homework assignment from CSC600, where a list of integers must be sorted, and then the longest string of consecutive integers found. This program is relatively similar to fizbuz in that the run counts only vary by a couple orders of magnitude, but this information would help point out which code was being run the most, in order to optimize efforts to speed execution time.

The principle purpose of this test was to determine that the profiler would work well on a real piece of code with functions, and not developed with the profiler in mind. The seeking of curly braces is a robust method of adding instrumentation, and this test was passed without any problem.



This matrix inversion benchmark is a real test of the profiler. It executes a battery of matrix inversions on matrix sizes from 20x20 up to 500x500. The algorithm is taken from Numerical Recipes, 2nd Edition. The repetitive tests on matrix ops mean some parts of the code get hit far more than others. Indeed, a spread of ten orders of magnitude is observable in the profiler output. To improve performance of the program, it is immediately evident where the time should be spent: in the code blocks that execute over 109 times.

One interesting eccentricity of this histogram is the zero reading for code block 13. This is an instrumented if() statement whose purpose is not clear, but seems meant to catch some error in the calculation. The fact that this statement never gets hit is certainly a good thing, then.

This programs most executed code segments actually overflowed INT\_MAX for C++, so it was necessary to switch the counters to long longs to prevent overflows from occurring during counting.

# Conclusion

The profiler worked quite well, but in future there are a few things to change. First off, the profiler will count and anchor on curly braces that are not part of normal code, e.g., part of a string. Using a state machine to parse the source code would eliminate this issue.

Another problem is that there is no way to profile a program that requires user input, so finding some way to pipe input and output back and forth between the python script and the C++ program. In the current rendition, the script will simply hang when the C++ program asks for user input.

The profiler does not correctly handle conditional or loop bodies that lack curly braces. The solution to this problem is to scan the C++ code for anything matching a loop or condition check, as was done in project 3 and 4 to count cyclomatic complexity. If the first character after the condition check, other spaces or newlines, is not a left curly brace, then the statement needs to have curly braces added so that the counter will be populated correctly. Simply add a left curly before the first non-space or newline, and then a closing curly after the first semicolon.

This tool would be easy to adapt to other languages as well, since it simply searches for the openings of new scopes. In C++ or Java this means checking for curly braces, whereas for Python it would mean checking for changes in indentation level. This is a very intuitive and robust method, and works very well on any program.

# Appendix: Profiler Output

## FizBuz Profiler Output

instrumentedsource.cpp

LLOC runcount for original sourcecode below:

#include <iostream>

using namespace std;

int main()

{

for (int i = 1; i <= 100; i++) // Ran 1 times

{

if (!(i % 3) && !(i % 5))

{

cout << "fizzbuzz\n"; // Ran 6 times

}

else if (!(i % 3))

{

cout << "fizz\n"; // Ran 27 times

}

else if (!(i % 5))

{

cout << "buzz\n"; // Ran 14 times

}

else

{

cout << i << "\n"; // Ran 53 times

}

}

return 0; // Ran 1 times

}

## Longest Consecutive String of Digits Profiler Output

LLOC runcount for original sourcecode below:

// hmw2part1.cpp : Defines the entry point for the console application.

// Samuel Gluss

// CS 600 Homework 2 Part 1

// 10/4/2015

#include <iostream>

#include <stdlib.h>

#include <algorithm>

using namespace std;

void printElements(int \*&array, int length)

{

//useful function we can use to print the elements in a dynamic array

cout << "array elements:" << endl; // Ran 2 times

for (int i = 0; i < length; i++) // Ran 2 times

{

cout << array[i] << " "; // Ran 200 times

}

cout << endl << endl; // Ran 2 times

}

int maxlen(int \*&testArray, int& length)

{

//find the longest string of consecutive numbers in the sorted array

int currentMax = testArray[0]; // Ran 1 times

int currentMaxCounter = 1; // Ran 1 times

int runnerUp = 0; // Ran 1 times

int runnerUpCounter = 0; // Ran 1 times

for (int i = 1; i < length; i++) // Ran 1 times

{

//if next value is the same as running current max, increment maxcounter

if (currentMax == testArray[i])

{

currentMaxCounter++; // Ran 10 times

}

//if next element is a runnerup element, increment runnerupCounter

else if (runnerUp == testArray[i])

{

runnerUpCounter++; // Ran 65 times

//if runnerup has the same count or greater than current max, change runnerup to be new longest string

if (runnerUpCounter >= currentMax)

{

currentMaxCounter = runnerUpCounter; // Ran 6 times

runnerUpCounter = 0; // Ran 6 times

currentMax = runnerUp; // Ran 6 times

runnerUp = 0; // Ran 6 times

}

}

//if next element is neither a max or a runnerup, start runnerup over with new value and counter

else

{

runnerUp = testArray[i]; // Ran 24 times

runnerUpCounter = 1; // Ran 24 times

}

}

return currentMaxCounter; // Ran 1 times

}

int main()

{

//create our initial array

int arraySize = 100; // Ran 1 times

int \*testArray = new int[arraySize]; // Ran 1 times

//populate the array with some random values from 1 to 25

for (int i = 0; i < arraySize; i++) // Ran 1 times

{

testArray[i] = rand() % 25 + 1; // Ran 100 times

}

cout << "unsorted array:" << endl; // Ran 1 times

printElements(testArray, 100); // Ran 1 times

//sort the array using algorithm library sort

sort(testArray, testArray + 100); // Ran 1 times

cout << "sorted array:" << endl; // Ran 1 times

printElements(testArray, 100); // Ran 1 times

//execute the maxlen function here, output the results

cout << "Longest consecutive digit string has length of " << maxlen(testArray, arraySize) << endl; // Ran 1 times

return 0; // Ran 1 times

}

## Matrix Inversion Profiler Output

LLOC runcount for original sourcecode below:

#include <iostream>

#include <random>

#include <chrono>

using namespace std;

int main()

{

// chrono::steady\_clock has previously been established to have an error of 2.689e-07 seconds

// for these experiments, we must assume a total error of 5.378e-07 seconds

// due to the worst case scenario for two time measurements (start/stop)

// initialize random generator, set generator type/range here

random\_device rd; // Ran 1 times

mt19937 gen(rd()); // Ran 1 times

std::uniform\_int\_distribution<> dis(1, 1000); // Ran 1 times

// initialize timers

chrono::steady\_clock::time\_point startTime, stopTime; // Ran 1 times

double initTime1; // Ran 1 times

double invTime; // Ran 1 times

double initTime2; // Ran 1 times

double multTime2; // Ran 1 times

double initTimes[25]; // Ran 1 times

double invTimes[25]; // Ran 1 times

int iteration = 0; // Ran 1 times

for (int rowCount = 20; rowCount <= 500; rowCount += 20) { // Ran 25 times

// matrix A/B initialization

int initRepCount = 200 / sqrt(rowCount); // Ran 25 times

int multRepCount = 200 / sqrt(rowCount); // Ran 25 times

int\*\* matA = new int\*[rowCount]; // Ran 25 times

startTime = chrono::steady\_clock::now(); // Ran 25 times

for (int K = 0; K < initRepCount; K++) // Ran 25 times

{

for (int i = 0; i < rowCount; i++) // Ran 373 times

{

matA[i] = new int[rowCount]; // Ran 73260 times

// initialize rows to random values

for (int j = 0; j < rowCount; j++) // Ran 73260 times

{

matA[i][j] = dis(gen); // Ran 22382800 times

}

}

}

stopTime = chrono::steady\_clock::now(); // Ran 25 times

initTime1 = chrono::duration\_cast<chrono::duration<double>>(stopTime - startTime).count() / initRepCount; // Ran 25 times

startTime = chrono::steady\_clock::now(); // Ran 25 times

for (int K = 0; K < multRepCount; K++) // Ran 25 times

{

/\*

begin inversion code from numerical recipes 2nd ed

\*/

int\* indxc = new int[rowCount]; // Ran 373 times

int\* indxr = new int[rowCount]; // Ran 373 times

int\* ipiv = new int[rowCount]; // Ran 373 times

int i, icol = 0, irow = 0, j, k, l, ll; // Ran 373 times

double big, dum, pivinv, temp; // Ran 373 times

int\* temp2; // Ran 373 times

for (j = 0; j<rowCount; ++j) // Ran 373 times

ipiv[j] = 0; // Ran 373 times

for (i = 0; i<rowCount; ++i) // Ran 373 times

{

big = 0.0; // Ran 73260 times

for (j = 0; j<rowCount; ++j) // Ran 73260 times

{

if (ipiv[j] != 1)

{

for (k = 0; k<rowCount; ++k) // Ran 11228030 times

{

if (ipiv[k] == 0)

{

if (fabs(matA[j][k]) >= big)

{

big = fabs(matA[j][k]); // Ran 2440770155 times

irow = j; // Ran 2440770155 times

icol = k; // Ran 2440770155 times

}

}

else

{

if (ipiv[k] > 1)

{

// Something bad happened?

}

}

}

}

}

++ipiv[icol]; // Ran 73260 times

if (irow != icol)

{

temp2 = matA[irow]; // Ran 6320 times

matA[irow] = matA[icol]; // Ran 6320 times

matA[icol] = temp2; // Ran 6320 times

}

indxr[i] = irow; // Ran 73260 times

indxc[i] = icol; // Ran 73260 times

if (matA[icol][icol] == 0.0)

{

// Something bad happened?

}

pivinv = 1.0 / matA[icol][icol]; // Ran 73260 times

matA[icol][icol] = 1.0; // Ran 73260 times

for (l = 0; l<rowCount; ++l) // Ran 73260 times

matA[icol][l] \*= pivinv; // Ran 73260 times

for (ll = 0; ll<rowCount; ++ll) // Ran 73260 times

{

if (ll != icol)

{

dum = matA[ll][icol]; // Ran 22309540 times

matA[ll][icol] = 0.0; // Ran 22309540 times

for (l = 0; l<rowCount; ++l) // Ran 22309540 times

matA[ll][l] -= matA[icol][l] \* dum; // Ran 22309540 times

}

}

}

for (l = rowCount - 1; l >= 0; --l) // Ran 373 times

{

if (indxr[l] != indxc[l])

{

for (k = 0; k<rowCount; ++k) // Ran 6320 times

{

temp = matA[k][indxr[l]]; // Ran 2161140 times

matA[k][indxr[l]] = matA[k][indxc[l]]; // Ran 2161140 times

matA[k][indxc[l]] = temp; // Ran 2161140 times

}

}

}

/\*

end inversion code

\*/

}

stopTime = chrono::steady\_clock::now(); // Ran 25 times

invTime = chrono::duration\_cast<chrono::duration<double>>(stopTime - startTime).count() / multRepCount; // Ran 25 times

initTimes[iteration] = initTime1; // Ran 25 times

invTimes[iteration] = invTime; // Ran 25 times

iteration++; // Ran 25 times

//end of execution for each rowcount test

}

cout << "init times:\n"; // Ran 1 times

for (int i = 0; i < 25; i++) // Ran 1 times

{

cout << initTimes[i] << ","; // Ran 25 times

}

cout << "\n\ninversion times:\n"; // Ran 1 times

for (int i = 0; i < 25; i++) // Ran 1 times

{

cout << invTimes[i] << ","; // Ran 25 times

}

return 0; // Ran 1 times

}

Process finished with exit code 0

# Appendix: Original Source Code

## FizBuz Original Source Code

#include <iostream>

using namespace std;

int main()

{

for (int i = 1; i <= 100; i++)

{

if (!(i % 3) && !(i % 5))

{

cout << "fizzbuzz\n";

}

else if (!(i % 3))

{

cout << "fizz\n";

}

else if (!(i % 5))

{

cout << "buzz\n";

}

else

{

cout << i << "\n";

}

}

return 0;

}

## Longest Consecutive String of Digits Original Source Code

// hmw2part1.cpp : Defines the entry point for the console application.

// Samuel Gluss

// CS 600 Homework 2 Part 1

// 10/4/2015

#include <iostream>

#include <stdlib.h>

#include <algorithm>

using namespace std;

void printElements(int \*&array, int length)

{

//useful function we can use to print the elements in a dynamic array

cout << "array elements:" << endl;

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

{

cout << array[i] << " ";

}

cout << endl << endl;

}

int maxlen(int \*&testArray, int& length)

{

//find the longest string of consecutive numbers in the sorted array

int currentMax = testArray[0];

int currentMaxCounter = 1;

int runnerUp = 0;

int runnerUpCounter = 0;

for (int i = 1; i < length; i++)

{

//if next value is the same as running current max, increment maxcounter

if (currentMax == testArray[i])

{

currentMaxCounter++;

}

//if next element is a runnerup element, increment runnerupCounter

else if (runnerUp == testArray[i])

{

runnerUpCounter++;

//if runnerup has the same count or greater than current max, change runnerup to be new longest string

if (runnerUpCounter >= currentMax)

{

currentMaxCounter = runnerUpCounter;

runnerUpCounter = 0;

currentMax = runnerUp;

runnerUp = 0;

}

}

//if next element is neither a max or a runnerup, start runnerup over with new value and counter

else

{

runnerUp = testArray[i];

runnerUpCounter = 1;

}

}

return currentMaxCounter;

}

int main()

{

//create our initial array

int arraySize = 100;

int \*testArray = new int[arraySize];

//populate the array with some random values from 1 to 25

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

{

testArray[i] = rand() % 25 + 1;

}

cout << "unsorted array:" << endl;

printElements(testArray, 100);

//sort the array using algorithm library sort

sort(testArray, testArray + 100);

cout << "sorted array:" << endl;

printElements(testArray, 100);

//execute the maxlen function here, output the results

cout << "Longest consecutive digit string has length of " << maxlen(testArray, arraySize) << endl;

return 0;

}

## Matrix Inversion Original Source Code

#include <iostream>

#include <random>

#include <chrono>

using namespace std;

int main()

{

// chrono::steady\_clock has previously been established to have an error of 2.689e-07 seconds

// for these experiments, we must assume a total error of 5.378e-07 seconds

// due to the worst case scenario for two time measurements (start/stop)

// initialize random generator, set generator type/range here

random\_device rd;

mt19937 gen(rd());

std::uniform\_int\_distribution<> dis(1, 1000);

// initialize timers

chrono::steady\_clock::time\_point startTime, stopTime;

double initTime1;

double invTime;

double initTime2;

double multTime2;

double initTimes[25];

double invTimes[25];

int iteration = 0;

for (int rowCount = 20; rowCount <= 500; rowCount += 20) {

// matrix A/B initialization

int initRepCount = 200 / sqrt(rowCount);

int multRepCount = 200 / sqrt(rowCount);

int\*\* matA = new int\*[rowCount];

startTime = chrono::steady\_clock::now();

for (int K = 0; K < initRepCount; K++)

{

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

{

matA[i] = new int[rowCount];

// initialize rows to random values

for (int j = 0; j < rowCount; j++)

{

matA[i][j] = dis(gen);

}

}

}

stopTime = chrono::steady\_clock::now();

initTime1 = chrono::duration\_cast<chrono::duration<double>>(stopTime - startTime).count() / initRepCount;

startTime = chrono::steady\_clock::now();

for (int K = 0; K < multRepCount; K++)

{

/\*

begin inversion code from numerical recipes 2nd ed

\*/

int\* indxc = new int[rowCount];

int\* indxr = new int[rowCount];

int\* ipiv = new int[rowCount];

int i, icol = 0, irow = 0, j, k, l, ll;

double big, dum, pivinv, temp;

int\* temp2;

for (j = 0; j<rowCount; ++j)

ipiv[j] = 0;

for (i = 0; i<rowCount; ++i)

{

big = 0.0;

for (j = 0; j<rowCount; ++j)

{

if (ipiv[j] != 1)

{

for (k = 0; k<rowCount; ++k)

{

if (ipiv[k] == 0)

{

if (fabs(matA[j][k]) >= big)

{

big = fabs(matA[j][k]);

irow = j;

icol = k;

}

}

else

{

if (ipiv[k] > 1)

{

// Something bad happened?

}

}

}

}

}

++ipiv[icol];

if (irow != icol)

{

temp2 = matA[irow];

matA[irow] = matA[icol];

matA[icol] = temp2;

}

indxr[i] = irow;

indxc[i] = icol;

if (matA[icol][icol] == 0.0)

{

// Something bad happened?

}

pivinv = 1.0 / matA[icol][icol];

matA[icol][icol] = 1.0;

for (l = 0; l<rowCount; ++l)

matA[icol][l] \*= pivinv;

for (ll = 0; ll<rowCount; ++ll)

{

if (ll != icol)

{

dum = matA[ll][icol];

matA[ll][icol] = 0.0;

for (l = 0; l<rowCount; ++l)

matA[ll][l] -= matA[icol][l] \* dum;

}

}

}

for (l = rowCount - 1; l >= 0; --l)

{

if (indxr[l] != indxc[l])

{

for (k = 0; k<rowCount; ++k)

{

temp = matA[k][indxr[l]];

matA[k][indxr[l]] = matA[k][indxc[l]];

matA[k][indxc[l]] = temp;

}

}

}

/\*

end inversion code

\*/

}

stopTime = chrono::steady\_clock::now();

invTime = chrono::duration\_cast<chrono::duration<double>>(stopTime - startTime).count() / multRepCount;

initTimes[iteration] = initTime1;

invTimes[iteration] = invTime;

iteration++;

//end of execution for each rowcount test

}

cout << "init times:\n";

for (int i = 0; i < 25; i++)

{

cout << initTimes[i] << ",";

}

cout << "\n\ninversion times:\n";

for (int i = 0; i < 25; i++)

{

cout << invTimes[i] << ",";

}

return 0;

}

# Appendix: Profiler Instrumented Code

## FizBuz Instrumented Code

#include <iostream>

#include <fstream>

long long programBodyCounters[6] = { 0 };

using namespace std;

int main()

{

programBodyCounters[0]++;

for (int i = 1; i <= 100; i++)

{

programBodyCounters[1]++;

if (!(i % 3) && !(i % 5))

{

programBodyCounters[2]++;

cout << "fizzbuzz\n";

}

else if (!(i % 3))

{

programBodyCounters[3]++;

cout << "fizz\n";

}

else if (!(i % 5))

{

programBodyCounters[4]++;

cout << "buzz\n";

}

else

{

programBodyCounters[5]++;

cout << i << "\n";

}

}

ofstream outfile;

outfile.open("../../../outfiles/fizbuzoutput.txt");

int codeExecCounterArraySize = sizeof(programBodyCounters) / sizeof(long long);

for (int i = 0; i < codeExecCounterArraySize - 1; i++)

{

outfile << programBodyCounters[i] << ",";

}

outfile << programBodyCounters[codeExecCounterArraySize - 1];

outfile.close();

return 0;

}

## Longest Consecutive String of Digits Instrumented Code

// hmw2part1.cpp : Defines the entry point for the console application.

// Samuel Gluss

// CS 600 Homework 2 Part 1

// 10/4/2015

#include <iostream>

#include <fstream>

long long programBodyCounters[10] = { 0 };

#include <stdlib.h>

#include <algorithm>

using namespace std;

void printElements(int \*&array, int length)

{

programBodyCounters[0]++;

//useful function we can use to print the elements in a dynamic array

cout << "array elements:" << endl;

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

{

programBodyCounters[1]++;

cout << array[i] << " ";

}

cout << endl << endl;

}

int maxlen(int \*&testArray, int& length)

{

programBodyCounters[2]++;

//find the longest string of consecutive numbers in the sorted array

int currentMax = testArray[0];

int currentMaxCounter = 1;

int runnerUp = 0;

int runnerUpCounter = 0;

for (int i = 1; i < length; i++)

{

programBodyCounters[3]++;

//if next value is the same as running current max, increment maxcounter

if (currentMax == testArray[i])

{

programBodyCounters[4]++;

currentMaxCounter++;

}

//if next element is a runnerup element, increment runnerupCounter

else if (runnerUp == testArray[i])

{

programBodyCounters[5]++;

runnerUpCounter++;

//if runnerup has the same count or greater than current max, change runnerup to be new longest string

if (runnerUpCounter >= currentMax)

{

programBodyCounters[6]++;

currentMaxCounter = runnerUpCounter;

runnerUpCounter = 0;

currentMax = runnerUp;

runnerUp = 0;

}

}

//if next element is neither a max or a runnerup, start runnerup over with new value and counter

else

{

programBodyCounters[7]++;

runnerUp = testArray[i];

runnerUpCounter = 1;

}

}

return currentMaxCounter;

}

int main()

{

programBodyCounters[8]++;

//create our initial array

int arraySize = 100;

int \*testArray = new int[arraySize];

//populate the array with some random values from 1 to 25

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

{

programBodyCounters[9]++;

testArray[i] = rand() % 25 + 1;

}

cout << "unsorted array:" << endl;

printElements(testArray, 100);

//sort the array using algorithm library sort

sort(testArray, testArray + 100);

cout << "sorted array:" << endl;

printElements(testArray, 100);

//execute the maxlen function here, output the results

cout << "Longest consecutive digit string has length of " << maxlen(testArray, arraySize) << endl;

ofstream outfile;

outfile.open("../../outfiles/consecutiveDigitsoutput.txt");

int codeExecCounterArraySize = sizeof(programBodyCounters) / sizeof(long long);

for (int i = 0; i < codeExecCounterArraySize - 1; i++)

{

outfile << programBodyCounters[i] << ",";

}

outfile << programBodyCounters[codeExecCounterArraySize - 1];

outfile.close();

return 0;

}

## Matrix Inversion Instrumented Code

#include <iostream>

#include <fstream>

long long programBodyCounters[23] = { 0 };

#include <random>

#include <chrono>

using namespace std;

int main()

{

programBodyCounters[0]++;

// chrono::steady\_clock has previously been established to have an error of 2.689e-07 seconds

// for these experiments, we must assume a total error of 5.378e-07 seconds

// due to the worst case scenario for two time measurements (start/stop)

// initialize random generator, set generator type/range here

random\_device rd;

mt19937 gen(rd());

std::uniform\_int\_distribution<> dis(1, 1000);

// initialize timers

chrono::steady\_clock::time\_point startTime, stopTime;

double initTime1;

double invTime;

double initTime2;

double multTime2;

double initTimes[25];

double invTimes[25];

int iteration = 0;

for (int rowCount = 20; rowCount <= 500; rowCount += 20) {

programBodyCounters[1]++;

// matrix A/B initialization

int initRepCount = 200 / sqrt(rowCount);

int multRepCount = 200 / sqrt(rowCount);

int\*\* matA = new int\*[rowCount];

startTime = chrono::steady\_clock::now();

for (int K = 0; K < initRepCount; K++)

{

programBodyCounters[2]++;

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

{

programBodyCounters[3]++;

matA[i] = new int[rowCount];

// initialize rows to random values

for (int j = 0; j < rowCount; j++)

{

programBodyCounters[4]++;

matA[i][j] = dis(gen);

}

}

}

stopTime = chrono::steady\_clock::now();

initTime1 = chrono::duration\_cast<chrono::duration<double>>(stopTime - startTime).count() / initRepCount;

startTime = chrono::steady\_clock::now();

for (int K = 0; K < multRepCount; K++)

{

programBodyCounters[5]++;

/\*

begin inversion code from numerical recipes 2nd ed

\*/

int\* indxc = new int[rowCount];

int\* indxr = new int[rowCount];

int\* ipiv = new int[rowCount];

int i, icol = 0, irow = 0, j, k, l, ll;

double big, dum, pivinv, temp;

int\* temp2;

for (j = 0; j<rowCount; ++j)

ipiv[j] = 0;

for (i = 0; i<rowCount; ++i)

{

programBodyCounters[6]++;

big = 0.0;

for (j = 0; j<rowCount; ++j)

{

programBodyCounters[7]++;

if (ipiv[j] != 1)

{

programBodyCounters[8]++;

for (k = 0; k<rowCount; ++k)

{

programBodyCounters[9]++;

if (ipiv[k] == 0)

{

programBodyCounters[10]++;

if (fabs(matA[j][k]) >= big)

{

programBodyCounters[11]++;

big = fabs(matA[j][k]);

irow = j;

icol = k;

}

}

else

{

programBodyCounters[12]++;

if (ipiv[k] > 1)

{

programBodyCounters[13]++;

// Something bad happened?

}

}

}

}

}

++ipiv[icol];

if (irow != icol)

{

programBodyCounters[14]++;

temp2 = matA[irow];

matA[irow] = matA[icol];

matA[icol] = temp2;

}

indxr[i] = irow;

indxc[i] = icol;

if (matA[icol][icol] == 0.0)

{

programBodyCounters[15]++;

// Something bad happened?

}

pivinv = 1.0 / matA[icol][icol];

matA[icol][icol] = 1.0;

for (l = 0; l<rowCount; ++l)

matA[icol][l] \*= pivinv;

for (ll = 0; ll<rowCount; ++ll)

{

programBodyCounters[16]++;

if (ll != icol)

{

programBodyCounters[17]++;

dum = matA[ll][icol];

matA[ll][icol] = 0.0;

for (l = 0; l<rowCount; ++l)

matA[ll][l] -= matA[icol][l] \* dum;

}

}

}

for (l = rowCount - 1; l >= 0; --l)

{

programBodyCounters[18]++;

if (indxr[l] != indxc[l])

{

programBodyCounters[19]++;

for (k = 0; k<rowCount; ++k)

{

programBodyCounters[20]++;

temp = matA[k][indxr[l]];

matA[k][indxr[l]] = matA[k][indxc[l]];

matA[k][indxc[l]] = temp;

}

}

}

/\*

end inversion code

\*/

}

stopTime = chrono::steady\_clock::now();

invTime = chrono::duration\_cast<chrono::duration<double>>(stopTime - startTime).count() / multRepCount;

initTimes[iteration] = initTime1;

invTimes[iteration] = invTime;

iteration++;

//end of execution for each rowcount test

}

cout << "init times:\n";

for (int i = 0; i < 25; i++)

{

programBodyCounters[21]++;

cout << initTimes[i] << ",";

}

cout << "\n\ninversion times:\n";

for (int i = 0; i < 25; i++)

{

programBodyCounters[22]++;

cout << invTimes[i] << ",";

}

ofstream outfile;

outfile.open("../../outfiles/matinvertoutput.txt");

int codeExecCounterArraySize = sizeof(programBodyCounters) / sizeof(long long);

for (int i = 0; i < codeExecCounterArraySize - 1; i++)

{

outfile << programBodyCounters[i] << ",";

}

outfile << programBodyCounters[codeExecCounterArraySize - 1];

outfile.close();

return 0;

}

# Appendix: Profiler Instrumented Code Output

**FizBuz Instrumented Code Output:**

1,100,6,27,14,53

**Longest Consecutive String of Digits Instrumented Code Output:**

2,200,1,99,10,65,6,24,1,100

**Matrix Inversion Instrumented Code Output:**

1,25,373,73260,22382800,373,73260,22382800,11228030,4079371400,2723323610,2440770155,1356047790,0,6320,66437,22382800,22309540,73260,6320,2161140,25,25

# Appendix Profiler Code

import re

from subprocess import call

import os

import sys

def buildAndRunExe(pathToSource):

# change working directory to location of source file

CWD = os.getcwd()

os.chdir(os.path.join(CWD, pathToSource[0]))

# set up the required environment for compiling and linking, then compile

call(["vcvars32.bat", "&&", "cl.exe" ,"/EHsc", "".join(["instrumented", pathToSource[1]])])

# run resulting executable, slice extension and change to exe

call(["".join(["instrumented",pathToSource[1][:-3],"exe"])])

# return to original directory

os.chdir(CWD)

def addStreamLibs(sourceData, hasIOStream, hasFStream):

# make sure required libraries

if not hasIOStream:

sourceData = "".join(["#include <iostream>\n", sourceData])

if not hasFStream:

iostrLoc = re.search('\#include[ ]+\<iostream\>', sourceData)

sourceData = "".join([sourceData[:iostrLoc.end()],

'\n#include <fstream>',

sourceData[iostrLoc.end():]])

return sourceData

def addCounters(sourceData):

# Instrument code by adding counters after every left curly brace

counter, mark = 0, 0

temp = ""

for i in range(0, len(sourceData)):

if sourceData[i] == "{":

# build counter string

countStr = "".join(['programBodyCounters[', str(counter), ']++;'])

# Add counter string after curly brace

temp = "\n".join([temp, sourceData[mark:i + 1], countStr])

mark = i + 1

counter += 1

# Add remainder of program

temp = "".join([temp, sourceData[mark:]])

return temp

def addCounterArr(sourceData, numLeftCurlies):

# Add profiler counter array to wherever fstream directive is

# this is done after the code is instrumented to prevent the initializer here

# from being instrumented as well!

fstrLoc = re.search('\#include[ ]+\<fstream\>', sourceData)

counterStr = "".join(['long long programBodyCounters[', str(numLeftCurlies), '] = { 0 };'])

sourceData = "".join([sourceData[:fstrLoc.end()],

'\n',

counterStr,

sourceData[fstrLoc.end():]])

return sourceData

def addOutfilePrint(sourceData, pathToSource, outfileName):

# find the exit point of the main function.

# Output to file will go right before this

mainFuncPos = re.search('int main\([^\(]\*\)[ \n]\*\{', sourceData).end()

exitPos = re.search('[ \t]\*return 0;', sourceData[mainFuncPos:]).start() + mainFuncPos - 1

# figure out how many directories to go up to get to outfiles

upDirs = pathToSource[0].count('/') - 1

upDirStr = "".join(["../" for i in range(0,upDirs)])

# build outfile print string

ofsStr = "\n".join(['ofstream outfile;',

'outfile.open("' + upDirStr + 'outfiles/' + outfileName + '");',

'int codeExecCounterArraySize = sizeof(programBodyCounters) / sizeof(long long);',

'for (int i = 0; i < codeExecCounterArraySize - 1; i++)',

'{',

'outfile << programBodyCounters[i] << "\\n";',

'}',

'outfile << programBodyCounters[codeExecCounterArraySize - 1];',

'outfile.close();'])

sourceData = "\n".join([sourceData[:exitPos],

ofsStr,

sourceData[exitPos:]])

return sourceData

def displayProfilerOutput(sourceData, runCounts):

print "\nLLOC runcount for original sourcecode below:\n"

printBuf = ""

# only print runcount if isLLOC is 1

isLLOC = 0

runCountStack = []

for i in sourceData:

if i == '{':

# Add new scope run count to stack

runCountStack.append(runCounts[0])

runCounts.pop(0)

elif i == '}':

runCountStack.pop()

elif i == ';':

# if we read a LLOC, print the run count

isLLOC = 1

elif i == '\n':

if isLLOC == 1 and runCountStack:

printBuf += "".join(['\t', "// Ran ", runCountStack[-1], ' times'])

sys.stdout.write(printBuf)

isLLOC = 0

printBuf = ""

else:

sys.stdout.write(printBuf)

printBuf = ""

isLLOC = 0

printBuf += i

# output remainder of program

print printBuf

def profilerOutput(pathToSource, outfileName):

# load block run counts

with open("".join(["../outfiles/", outfileName]), 'r') as file:

# Read outfile into string, split to get counters

outfile = file.read()

runCounts = outfile.split("\n")

# Print source code, annotated with run counts when applicable

with open("".join([pathToSource[0], pathToSource[1]]), 'r') as file:

# read source code into string

sourceData = file.read()

displayProfilerOutput(sourceData, runCounts)

def runProfiler(pathToSource, outfileName):

with open("".join([pathToSource[0], pathToSource[1]]), 'r') as file:

# read source code into string

sourceData = file.read()

numLeftCurlies = sourceData.count('{')

hasFStream = re.search('\#include[ ]+\<fstream\>',sourceData)

hasIOStream = re.search('\#include[ ]+\<iostream\>',sourceData)

# make sure source code has required fstream and iostream libraries

sourceData = addStreamLibs(sourceData, hasIOStream, hasFStream)

# instrument code with counters

sourceData = addCounters(sourceData)

# add array of counters to beginning of source code

sourceData = addCounterArr(sourceData, numLeftCurlies)

# Add code to print to appropriate outfile

sourceData = addOutfilePrint(sourceData, pathToSource, outfileName)

# Write Instrumented source code to new file

with open("".join([pathToSource[0], "instrumented", pathToSource[1]]), 'w') as file:

# write back to gnuplot file

file.writelines(sourceData)

buildAndRunExe(pathToSource)

# Generate and display output for profiler run

profilerOutput(pathToSource, outfileName)

runProfiler(["../cpp/fizbuz/fizbuz/","source.cpp"],"fizbuzoutput.txt")

runProfiler(["../cpp/consecutiveDigits/","source.cpp"],"consecutiveDigitsoutput.txt")

runProfiler(["../cpp/matinvert/","matinvert.cpp"],"matinvertoutput.txt")

# Appendix: Gnuplot Log Scale Histogram Generator

# Samuel Gluss

# 5-26-2016

# Log Scale Histogram Generator

reset

# define output

set terminal pngcairo size 800,550 enhanced font 'Verdana,10'

set output "pngout/matinvertoutputhist.png"

# set delimiter

set datafile separator ","

# set title, labels, key position

set title "Execution Count For Blocks of Code: Matrix Inversion"

set ylabel "<-- # of Times Run -->"

set xlabel "Code Block #"

set key off

set logscale y

set offset 0,0,1,1

set style data histograms

set style fill solid 1.0 border -1

plot "../outfiles/matinvertoutput.txt" using 1 lc rgb "red"