CS840 Project 1: Program Run Time Measuring, Modeling, and Prediction

Samuel Gluss

Jozo Dujmovic

2-20-2016

Abstract – Performance of a program is a crucial aspect of its quality. Many performance categories of a program may be measured, but the most tangible to the end user is the program’s runtime. Much effort is spent to make software run faster, and it is the task of any engineer to find the elements of a system that weaken the performance of the whole the most. In order to address the issue of program runtime, then, we must first find a way to measure the runtime of the program elements. Furthermore, it is essential to be able to model and predict the runtimes of different program elements, so that optimal solutions can be developed. This paper will study the performance of networking and matrix math operations to elucidate the concepts of runtime analysis and modeling.

I. Introduction:

This paper will first take a look at series of ping times to university servers in various parts of the world, as near as Stanford and as far as Bangladesh. In so doing, the paper will uncover relationships between different locations, and distribution of ping time delays. By recording 600 ping requests in each session, the data sets will be large enough to give a high level of confidence in the findings. This will allow conclusions to be safely drawn regarding relationships between ping time and network size or quality.

The next subject investigated by this paper will be the quality of the clock function in C++. This is an important metric to quantify, because it effectively determines the error induced by the clock function in any time measurement. Knowing this error, it will then be possible to calculate the necessary duration of any test to minimize the error caused by the clock to an acceptable level.

Once a method for effective time measurement is established, a model for matrix multiplication will be established and tested. Because matrix multiplication can be modeled by a polynomial, it’s possible to fit such a curve to experimental data, and analyze its effectiveness.

Lastly, the steps taken for creating a model for matrix multiplication will be repeated for matrix inversion. These results will be analyzed and compared, to determine if the models created are accurate, and if there is any variation in the quality of the model for multiplication versus inversion.

Table of Contents:

Ping Times Analysis ---- page 3

Determining Resolution of Clock Function ---- page 11

Matrix Multiplication Performance and Modeling ---- page 12

Matrix Inversion Performance and Modeling ---- page 14

References ---- page 15

Ping Batch Script ---- page 16

MatLab For Ping Data ---- page 17

Resolution of <chrono> ---- page 19

Matrix Multiplication Source Code: ---- page 21

Matrix Inversion Source Code ---- page 23

MatLab Script: Clock Resolution ---- page 26

MatLab Script: Matrix Experiments ---- page 27

Data ---- page 28

II. Ping Times Analysis

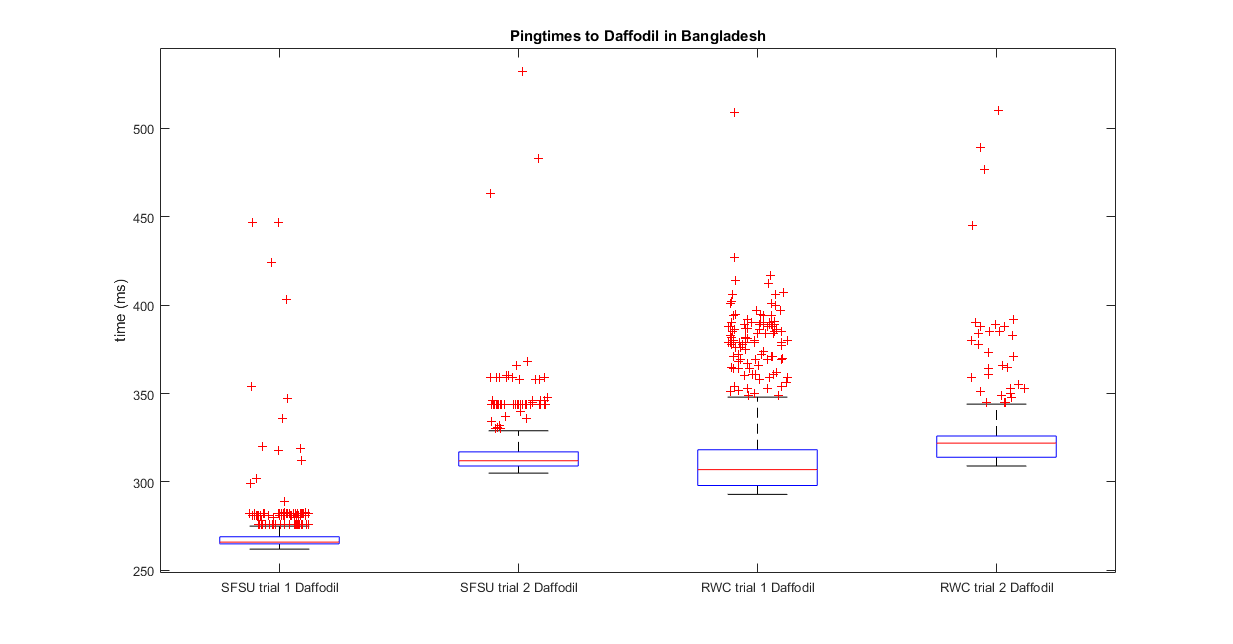
For this assignment, I produced a batch script which would allow me to define which servers I’d like to ping test, the parameters of the request, and a log file to redirect the log to, along with timestamping. The batch script launches these ping requests as new processes, and is thus able to execute them in parallel. I was concerned that the parallel processes may have affected my results, so I took some steps to address this issue. I ran the tests on an Intel i7-4710MQ 2.50 GHz quad-core processor, not in power saving mode, and with only the minimum processes running in the background. In this way, I believe it is safe to say that any processing time on my computer would be negligible compared to the ping TOF (time of flight), which is on the order of many to hundreds of milliseconds. The batch script allowed me to easily gather large amounts of data from many servers worldwide, and store it in log files which are easy to parse.

I chose to gather data from two different locations, and at two different time periods each, for 8 different universities. This much data makes it easy to find interesting phenomena, and offers plenty of opportunity for analysis based on distance, source, time of day, and other variables.

The log files are parsed using regex find-and-replace commands within notepad++. I chose this route because I already had notepad++ installed (it is my favored barebones development environment) and because it has a robust regex search feature, which enables it to do find and replace in all open files. By simply executing 3 commands, the ping data is cleaned up and ready to be inserted into a favorite data analysis program, which for me is MatLab. An alternative would have been to write a perl script or something comparable for file parsing, but my familiarity with notepad++ made it an easy choice.

Within MatLab, the data is stored in objects which give easy access to the aggregated ping data, timeout count, and informative titles for each set. I chose MatLab due to having prior experience with it in CSC872 at SFSU. I appreciate its robust data analysis feature set, thorough online documentation, and comprehensive figure production capability. Alternatively, R and Excel can be used, but I chose Matlab for its superior suite of data analysis features.

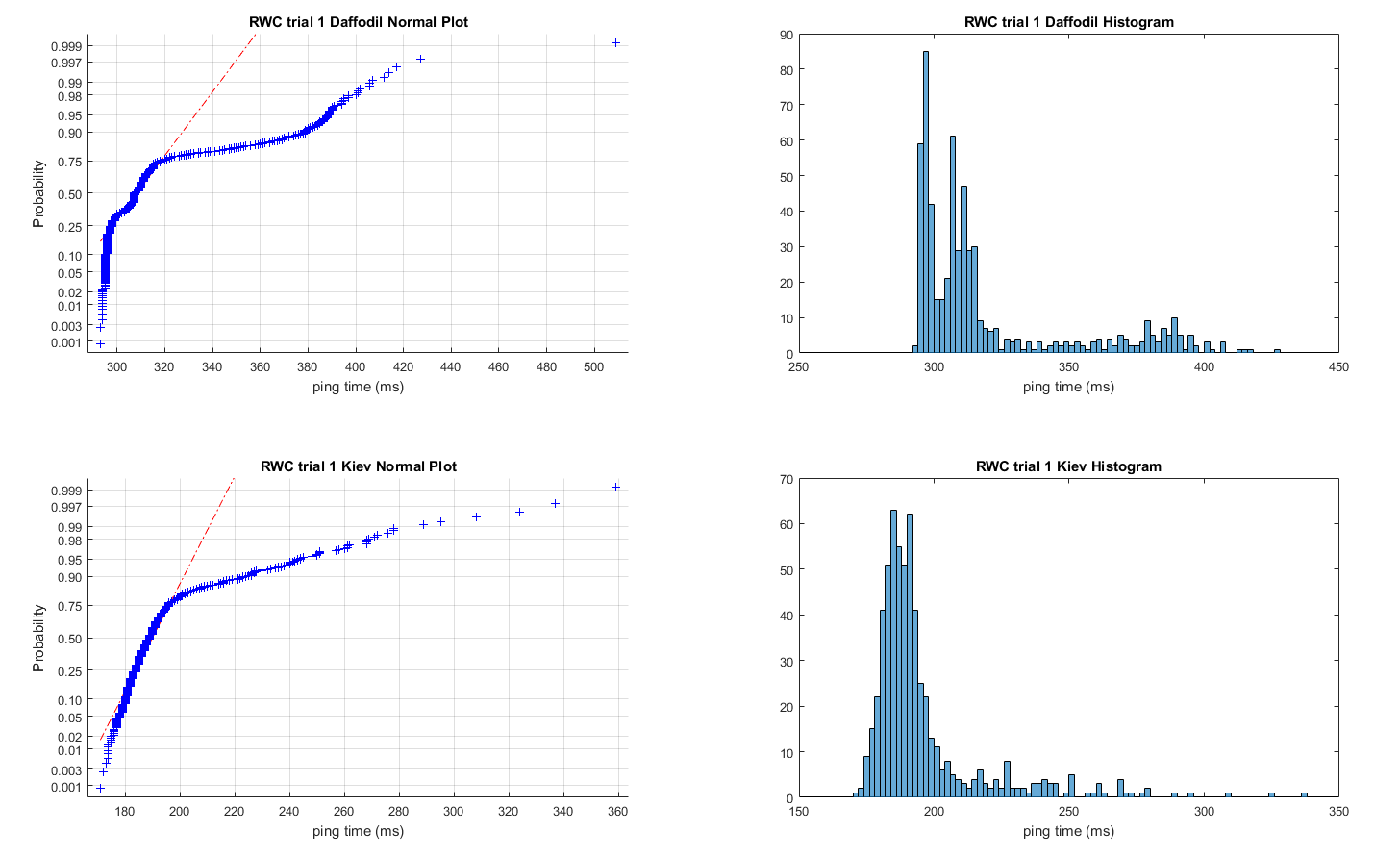
Data Analysis:



Across four trials to the same university in Bangladesh, I experimented with changing the source location, as well as the time when the experiments were conducted. The trials were conducted from either the SFSU undergrad lab, or a private residence in Redwood City with Comcast cable internet. What is immediately apparent is that the time when data was collected greatly affects the position of the distribution. In this case, The SFSU trials were taken at 8PM on 2/10/2016, and 6PM 2/15/2016, while the RWC trials were taken at 5PM and 10PM on 2/14/2016. It is apparent that the boxplot shapes are roughly the same for each dataset for one location.

Between the two locations though, we observe that the shape of the distribution changes quite dramatically. This suggests that the network from a residence is more convoluted than the universities, with more opportunities for the packets to be redirected before hitting a trunk line. A possibility for further research would be to gather traceroutes to Daffodil from both locations, and determine how many more nodes are utilized from one location, versus another.

RWC Daffodil vs Kiev Distributions

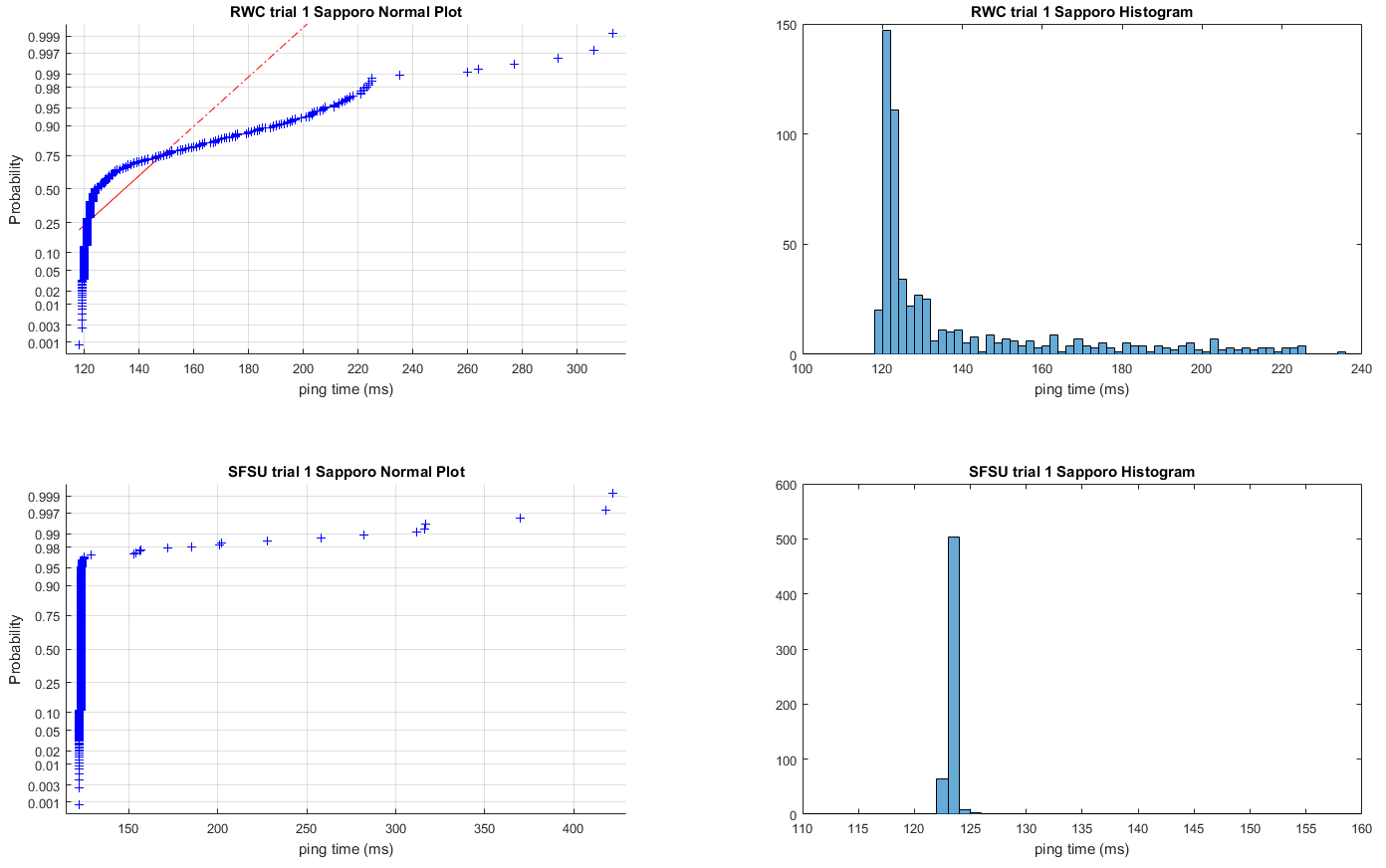


In this figure I compared the distributions of the ping data from Kiev and Bangladesh. These universities are both extremely distant, and the data was taken simultaneously from the Redwood City location. One of the questions I wanted to answer was if the data gathered would follow a normal distribution, and if not, what the possible reasons for this could be.

For far away universities, it appears that the distribution is mostly normal for the Redwood City data, but with a long tail on the right side of the distribution. In the case of the Daffodil data, there is even an observable second mode. This is most likely caused by the convoluted path that data must take to reach these Universities, so that while most data takes a reasonably direct path, some gets redirected through longer routes. In the case, it’s apparent that these other paths are utilized enough to create a second mode in the data set!

The next step in this line of analysis would be to gather traceroute data for these universities, which would provide the exact route each packet takes. This data would allow us to determine if the reason for the long tails on this data really is due to the various complex possible paths that each packet can take, or if it might be due to some other variable entirely, such as the compounded time of a large number of switching routers redirecting data.

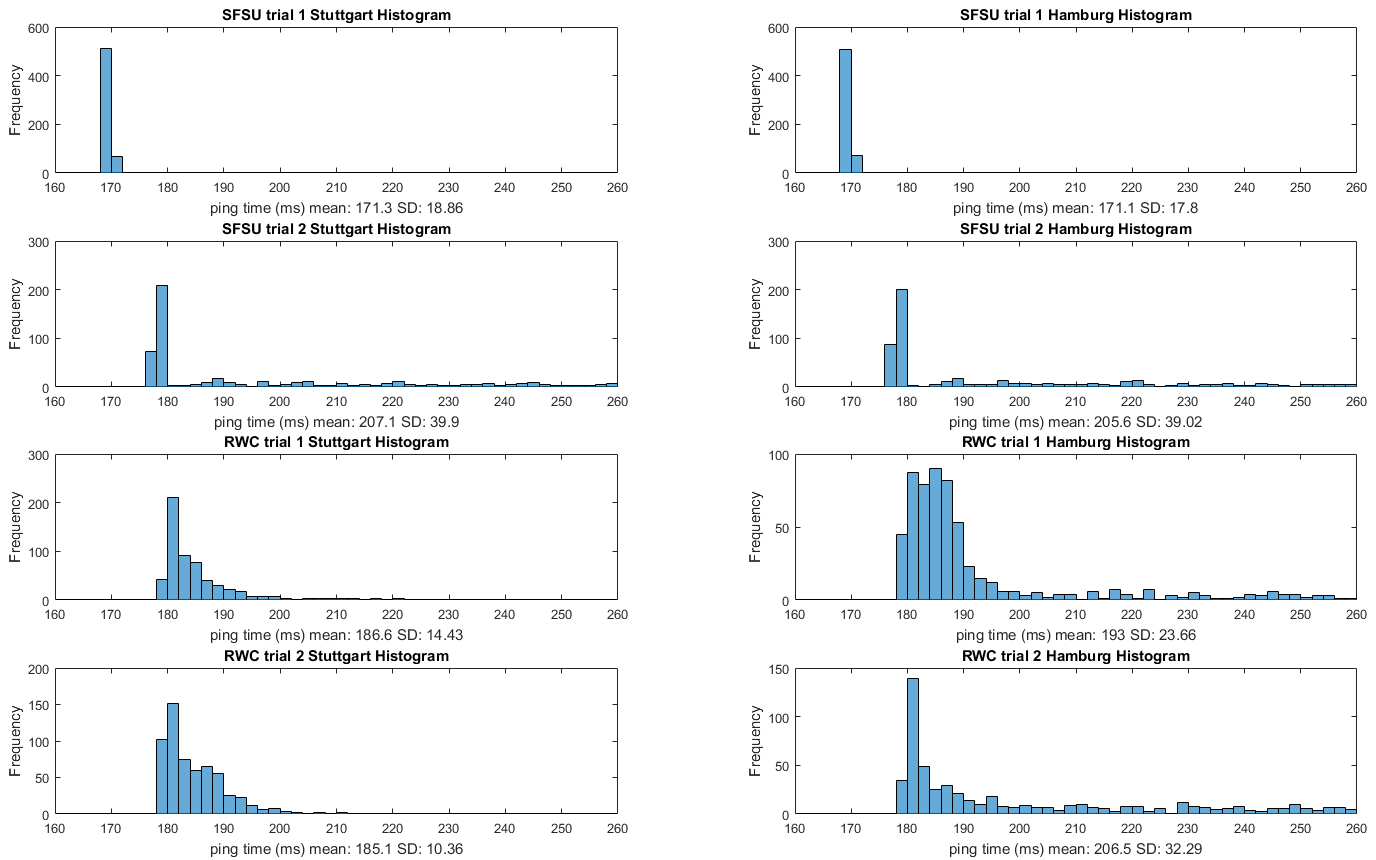
RWC vs SFSU Sapporo Distributions



In this figure I want to take a closer look at the differences in the data obtained from different sources. In this case of course, the data was not taken at the exact same time, so the subject of our analysis here is somewhat flawed. This pattern is consistent across the data sets though, so this particular case will suffice to demonstrate the difference in distribution between the SFSU and RWC datasets.

The first thing that is apparent is that the data taken from SFSU does not really appear to follow a normal distribution at all, with almost all of its data concentrated at just 1 or 2 points, compared to the RWC data which is much more spread out. With that said, the RWC data isn’t very normal either, with a significant right skew, and a large tail on the high side of the distribution. Despite these differences, the standard deviations of the data are 26.45ms for SFSU and 31.07ms for RWC, which are not very far apart, so the variation in the distributions are more superficial than one might guess.

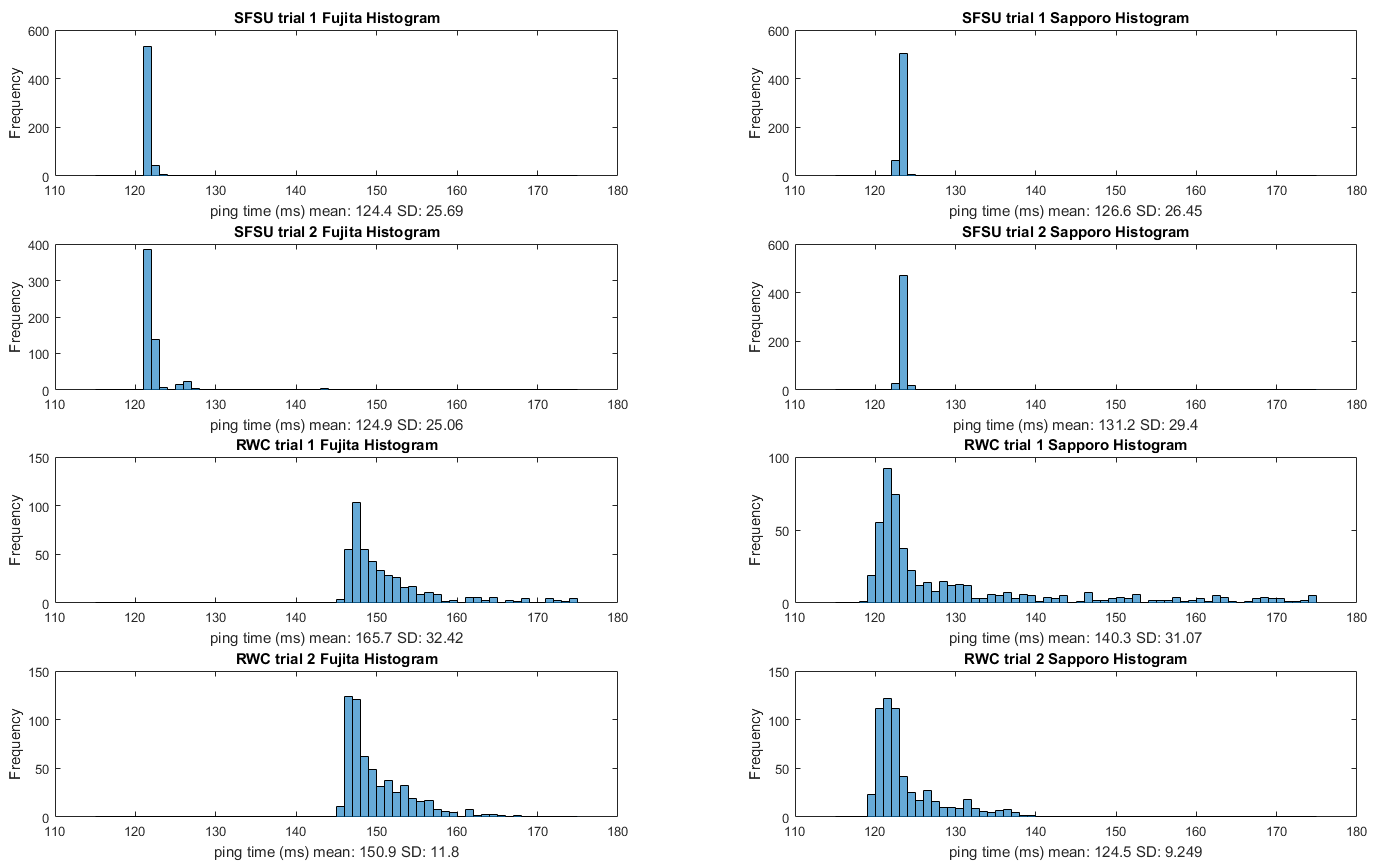
Stuttgart vs Hamburg: Geographical Ping Latency Correlation



This series of histograms is meant to show the side by side comparison of all the data to the universities of Stuttgart and Hamburg, which are both in Germany. We see, as before, the addition of random noise in the data from Redwood City, while the SFSU data almost entirely lacks outliers. Something that is a bit unusual about these charts is that the standard deviation does not always appear to correlate to the distribution, which is because of extreme outliers to the right side of the chart scale.

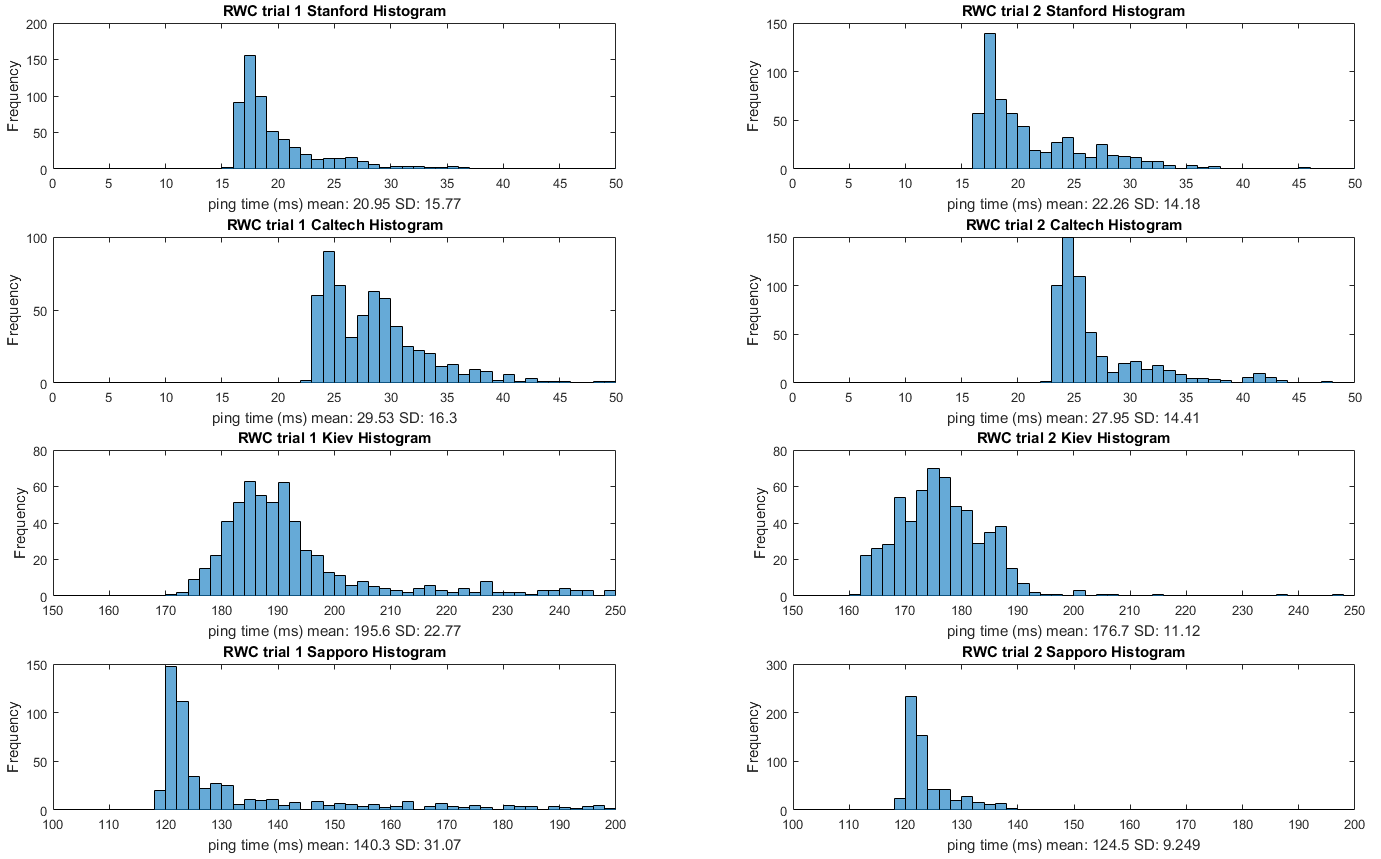
The mean of the data is extremely consistent for the SFSU data from Hamburg and Stuttgart, with the distributions being almost identical. The Redwood City data is less consistent, but we can still easily see the distributions are shaped the same and roughly centered at the same place. This correlation seems a bit tenuous, but especially compared to other dataset comparisons in this paper, the resemblance is obvious.

Fujita vs Sapporo: Geographical Ping Latency Correlation



Again we take a look at the relationship between physical distance and ping latency, this time in Japan. This test backs up the previous relationship in Germany, and behaves almost the same. One interesting difference is that the Redwood City data sets run about 15ms faster to Sapporo, while there is no substantial difference for the SFSU data. This is a very unusual difference between the two locations, and it is hard to come up with an explanation for this. My first step for gathering more information would be to get a traceroute to the two universities, and look for large differences between them. I would expect something significant, because the distributions look extremely similar, but that 15ms offset must be accounted for somewhere.

Time of Day Analysis: Redwood City Data



In this data set the focus of the experiment is to see if time of day makes a difference to the data. Controlling for location and selected servers, the only change is that the data was recorded at 5PM and 10PM respectively. It’s hard to tell from the distribution shapes, but for ever server, the mean latency as well as standard deviation are reduced in the data taken at 10PM. This suggests reduced network usage at this time, which makes sense because the current time in the continental US would be between 10PM and 1AM at this time, and most people will be asleep, especially compared to the 5PM dataset.

Conclusion: By collecting large samples of data, this experiment is able to demonstrate confident relationships between different datasets based on source and target destinations, time of day, and other variables. Some unsurprising conclusions were shown, such as internet traffic being lower when people are asleep. There were also some unusual questions that were discovered, such as why two foreign universities would have different latencies relative to each other, depending on what the source of the requests was.

For this series of experiments, there are two things that I would like to add, to learn more about some of the eccentricities of the data. First, I would like to gather data on an hourly (or every other hour) schedule, to get better control over the time-of-day variable. There are complications, of course: I would not easily be able to gather simultaneous data with my personal laptop being the control rig.

Another piece of data I’d like to gather is the trace route to each university. This came up in many experiments, where I suspected that perhaps there were multiple paths that data might take. The ping command returns only performance metrics, and nothing qualitative about the path taken, so it would be illuminating to see if there were multiple paths taken, and at what proportion was each path utilized. The Daffodil normal plot, with its unusual second mode, was one of the datasets that caused me to believe that there may be some alternate, slower path that is utilized nonetheless.

III. Determining Resolution of Clock Function

For this exercise, I tried a couple of different ways to measure the <chrono> library in C++11. At first, I utilized a while loop to check if a new time was being posted. This method used a bit more math than was strictly necessary, so I opted for a more memory wasteful technique, wherein I pre-allocated an array of 40000 doubles, then filled the array as rapidly as possible with new measurements from the <chrono>::steady\_clock::now() function, which serves as a monotonic clock.

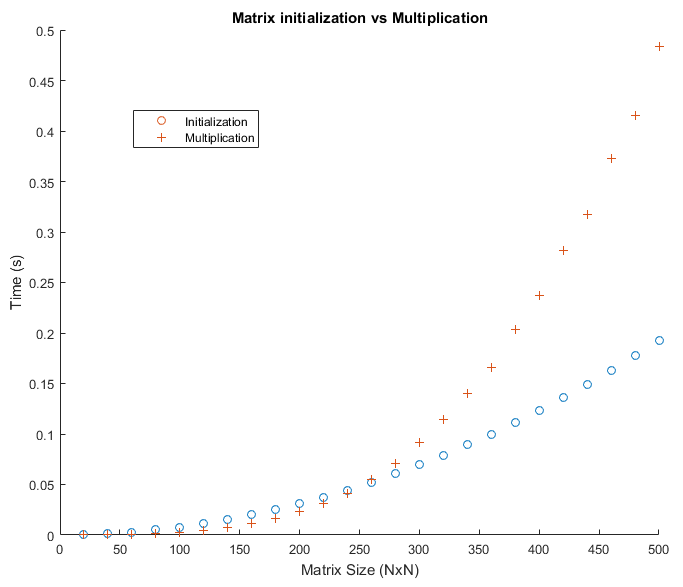
In so doing, I kept the number of calculations to an absolute minimum during the measurement, and the sequential memory access for the array is extremely fast as well. At each iteration of the for loop, I measure the time difference from the previous iteration of the loop, and store the difference in the current array index. I am satisfied with the quality of the measurement, of the 40000 array elements, 27760 were empty zeros where the clock had not updated. I wrote this output to a file, where I removed the zeros using a string match in notepad++.

I then took this large group of durations and entered it into MatLab, where I calculated the standard deviation. With the standard deviation and the number of recorded durations, I produced the standard error mean, which is calculated by dividing the standard deviation by the square root of the number of measurements. The percent error is found by dividing the standard error mean by the experimental mean, and for this dataset equals 0.58%. The mean measurement was calculated to be 2.689e-07 seconds for <chrono>::steady\_clock.

I believe this method of calculation is best because it isolates the time measurement as well as possible, with only a loop iterator, memory write and duration calculation on top of the timing methods. It doesn’t require any extra logic to check for 0 duration values, as this is done in post-processing.

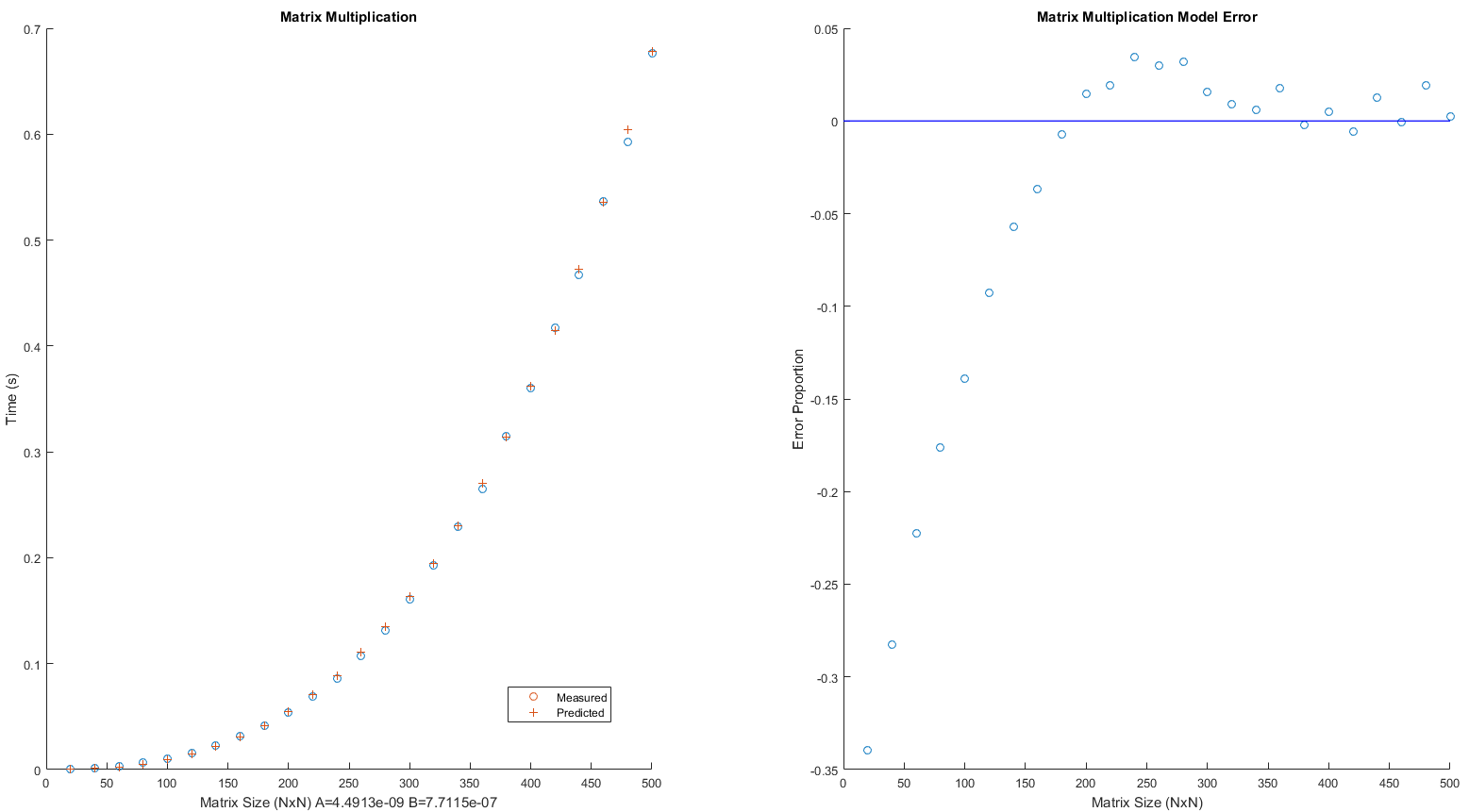
As with the previous experiment, this was conducted on my windows 7 laptop, equipped with an Intel i7-4710MQ 2.50 GHz quad-core processor, not in power saving mode. The laptop is fitted with 16GB of ddr3 memory. The test program was written and compiled with Visual Studio 2015 Community.

IV. Matrix Multiplication Performance and Modeling



In this experiment, the findings from the <chrono> library experiment are utilized to time matrix multiplications. The program conducts all experiments within loops to ensure that the errors from the timer are minimized (if there are any). At lower matrix sizes, these counts are higher to account for the shorter total runtime. I scaled the number of runtimes to scale inversely with the square root of the matrix size, which yields good accuracy for small matrices while keeping decent runtimes with larger datasets.

For matrix generation, two dynamically allocated NxN matrices are populated with random numbers between 1 and 1000. I used the C++ std library <random>, which is considered to have a sufficiently even random generator for the purposes of this experiment. I timed the code in two sections, one for the matrix initializations, and one for the actual multiplication operation. In hindsight this was unnecessary, but it gave the additional option to analyze the initialization and multiplication elements separately.



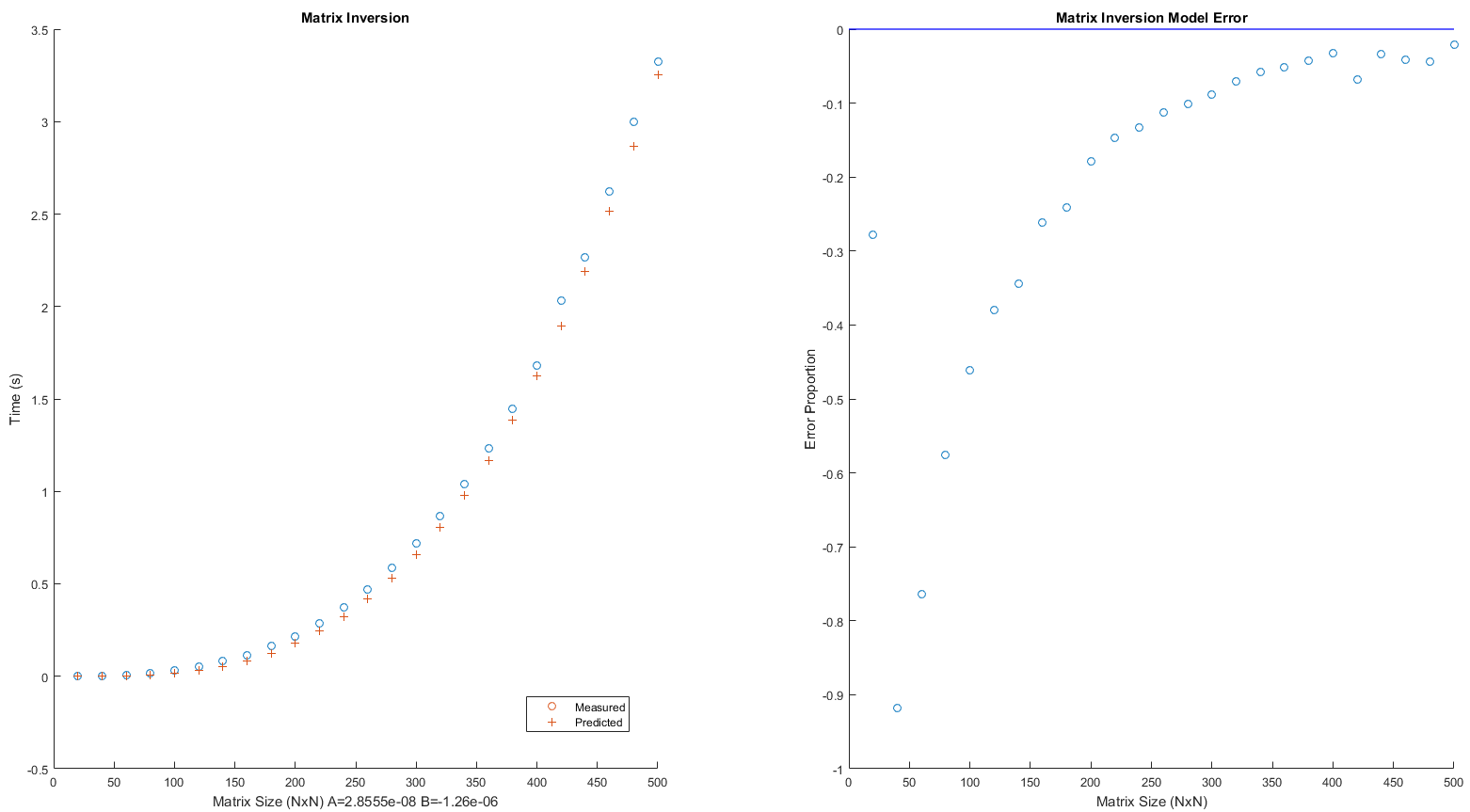
It is apparent from the data represented in the matrix initialization vs multiplication chart that the initialization follows a quadratic model while the multiplication is n^3. I used this to design a model for the combined data.

In the C++ program, I recorded the measurements in a pair of arrays, one for initialization, and one for multiplication. At program termination, these arrays are outputted to console, or to an outfile. From there, the data is put into MatLab, where a polynomial fit function is used to match coefficients. Error is calculated, and scatterplots are generated to visualize performance of the Ax^3 + bx^2 model. While the scatterplot shows that the model is a close approximation, the error plot actually shows the model becoming far more accurate as the matrix sizes increase, converging to 5% error by the time the matrix size exceeds 200x200. The model’s equation is y = (4.49E-9) x^3 + (7.71E-7) x^2.

I am quite pleased with the performance of the model in this case, as it closely matches the data for most values of n. Something I would like to explore in future experiments is to implement a non-linear least squares algorithm such as Levenberg-Marquardt, which would give much more control over the generation of the model. However, within the context of this experiment, such an excursion would be a project by itself.

This experiment was conducted with Visual Studio Community 2015, with the compiler in the default mode. Timing was done with the <chrono> steady\_clock, whose performance was characterized in the previous experiment. All tests were run on the baseline laptop.

V. Matrix Inversion Performance and Modeling



This experiment was structured very similarly to the matrix multiplication program. The MatLab program is almost identical, and the C++ code varies in that only one matrix is initialized, and the matrix multiplication is swapped out for a matrix inversion algorithm taken from Numerical Recipes in C, by William H. Press. This algorithm, like the matrix multiplication one it replaces, is also n^3 complexity, so the expectation is that this system can also be modeled by the Ax^3 + bx^2 model.

The data collection method was relatively unchanged, with many passes being run on each matrix size, to eliminate errors from the timer, or any other aberrations that might have occurred. The charts demonstrate that the polynomial fit is not quite as good as it was for the matrix multiplication experiment, with rather large errors until matrix size exceeds 300-400. The model this time worked out to y = (2.86E-8) x^3 + (1.26E-6) x^2.

The equipment and environment variables remain unchanged from the previous experiment. It seems that some more experimentation with the curve fitting algorithm could yield dramatic improvements in model accuracy, as the MatLab algorithm does not leave room for adjustment.

References:

1. Numerical Recipes in C, William H Press Cambridge University Press 2002

Appendices:

Ping Batch Script:

@ECHO off

ECHO Program started

start "stuttgart" cmd.exe /k "(echo %DATE:/=-%@%TIME::=-% & ping -n 600 www.uni-stuttgart.de.edu & echo %DATE:/=-%@%TIME::=-%) >> log4sfsu/stuttgart.txt & exit"

start "hamburg" cmd.exe /k "(echo %DATE:/=-%@%TIME::=-% & ping -n 600 www.uni-hamburg.de.edu & echo %DATE:/=-%@%TIME::=-%) >> log4sfsu/hamburg.txt & exit"

start "caltech" cmd.exe /k "(echo %DATE:/=-%@%TIME::=-% & ping -n 600 turing.caltech.edu & echo %DATE:/=-%@%TIME::=-%) >> log4sfsu/caltech.txt & exit"

start "daffodilvarsity" cmd.exe /k "(echo %DATE:/=-%@%TIME::=-% & ping -n 600 daffodilvarsity.edu.bd & echo %DATE:/=-%@%TIME::=-%) >> log4sfsu/daffodilvarsity.txt & exit"

start "kiev" cmd.exe /k "(echo %DATE:/=-%@%TIME::=-% & ping -n 600 www.imi.kiev.ua & echo %DATE:/=-%@%TIME::=-%) >> log4sfsu/kiev.txt & exit"

start "fujita" cmd.exe /k "(echo %DATE:/=-%@%TIME::=-% & ping -n 600 www.fujita-hu.ac.jp & echo %DATE:/=-%@%TIME::=-%) >> log4sfsu/fujita.txt & exit"

start "sapporo" cmd.exe /k "(echo %DATE:/=-%@%TIME::=-% & ping -n 600 www.sapporo-u.ac.jp & echo %DATE:/=-%@%TIME::=-%) >> log4sfsu/sapporo.txt & exit"

start "stanford" cmd.exe /k "(echo %DATE:/=-%@%TIME::=-% & ping -n 600 stanford.edu & echo %DATE:/=-%@%TIME::=-%) >> log4sfsu/standford.txt & exit"

ECHO Program terminated successfully

PAUSE

----Summary----

Script which spawns new cmd instances to gather simultaneous ping data from many servers

PingSession:

classdef pingSession

properties

title

pingCount

timeoutCount

latencies

end

methods

% methods, including the constructor are defined in this block

function obj = pingSession(a, b, c, d)

obj.title = a;

obj.pingCount = b;

obj.timeoutCount = c;

obj.latencies = d;

end

end

end

----Summary----

MatLab object which allows the storage of all data about a ping session

MatLab Ping Data Presentation:

boxplot comparison

figure

data0 = sfsu0data\_daffodil;

data1 = sfsu3data\_daffodil;

data2 = rwc1data\_daffodil;

data3 = rwc2data\_daffodil;

data = [data0.latencies data1.latencies data2.latencies data3.latencies];

group = [zeros(1,size(data0.latencies,2)),ones(1,size(data1.latencies,2)),2\*ones(1,size(data2.latencies,2)),3\*ones(1,size(data3.latencies,2))];

boxplot(data, group, 'labels',{data0.title,data1.title,data2.title,data3.title}, 'jitter',.5);

title('Pingtimes to Daffodil in Bangladesh');

ylabel('time (ms)');

normal distribution analysis

figure

data1 = sfsu0data\_hamburg;

data2 = sfsu0data\_stuttgart;

subplot(2,2,1);

normplot(data1.latencies);

title(strcat(data1.title, ' Normal Plot'));

xlabel('ping time (ms)');

subplot(2,2,2);

histogram(data1.latencies, 150:2:250);

title(strcat(data1.title, ' Histogram'));

xlabel('ping time (ms)');

subplot(2,2,3);

normplot(data2.latencies);

title(strcat(data2.title, ' Normal Plot'));

xlabel('ping time (ms)');

subplot(2,2,4);

histogram(data2.latencies, 150:2:250);

title(strcat(data2.title, ' Histogram'));

xlabel('ping time (ms)');

----Summary----

MatLab script which produces side-by-side and standalone figures. In this experiment, boxplots, normal distribution plots, and histograms were all used to help visualize patterns in different sets of data.

Resolution of <chrono>:

#include <iostream>

#include <chrono>

using namespace std;

int main()

{

chrono::steady\_clock::time\_point currentTime, newTime;

int counter = 0;

double timeDiffs = 0;

double duration = 0;

int reportCounter = 0;

// while loop method

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

while (counter < 10000)

{

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

duration = chrono::duration\_cast<chrono::duration<double>>(newTime - currentTime).count();

if (duration)

{

timeDiffs += duration;

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

counter++;

}

reportCounter++;

}

cout << "clock accuracy determined to be " << (timeDiffs / 10000) << " seconds\n\n";

// wasteful array filler method

double times[40000];

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

for (int i = 0; i < 40000; i++)

{

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

times[i] = chrono::duration\_cast<chrono::duration<double>>(newTime - currentTime).count();

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

}

for (int i = 0; i < 40000; i++)

{

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

}

return 0;

}

----Summary----

This code demonstrates a couple ways to gather data from the clock function. In one, a while loop is used to gather durations whenever they’re nonzero. In the other, all durations are recorded, and the nonzero ones are separated out afterwards. An even higher performance method would be to simply record timestamps as rapidly as possible, without bothering to calculate durations. These techniques progressively try to do as few function calls and comparisons as possible during program runtime, as these are easy tasks to handle after collection is completed.

Matrix Multiplication 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 multTime1;

double initTime2;

double multTime2;

double initTimes[25];

double multTimes[25];

int iteration = 0;

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

// matrix A/B initialization

int initRepCount = 250 / sqrt(rowCount);

int multRepCount = 250 / sqrt(rowCount);

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

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

int\*\* matC = 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];

matB[i] = new int[rowCount];

matC[i] = new int[rowCount];

// initialize rows to random values

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

{

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

matB[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++)

{

int product = 0;

// Populate C with appropriate values

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

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

product = 0;

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

product += matA[i][k] \* matB[k][j];

}

matC[i][j] = product;

}

}

}

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

multTime1 = chrono::duration\_cast<chrono::duration<double>>(stopTime - startTime).count() / multRepCount;

initTimes[iteration] = initTime1;

multTimes[iteration] = multTime1;

iteration++;

//end of execution for each rowcount test

}

cout << "init times:\n";

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

{

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

}

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

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

{

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

}

return 0;

}

----Summary----

This code uses a <random> generator to initialize two NxN matrices, then calculates their product. This experiment is repeated multiple times, for each N value from N=20 to N=500 at intervals of size 20.

Matrix Inversion 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;

}

----Summary----

Very similar in structure to the matrix multiplication experiment. Only real difference is that the multiplication stage is replaced with an inversion algorithm.

MatLab Script: Clock Resolution

u = mean(durations);

sd = std(durations);

standardErrorMean = sd/sqrt(size(durations,2));

errorPercent = standardErrorMean/u;

----Summary----

This simple script uses the collected durations to calculate the mean duration of one clock time for <chrono>, and the standard error of the mean.

MatLab Script: Matrix Experiments

initCoeff = polyfit(20:20:500,initTimes,2);

multCoeff = polyfit(20:20:500,multTimes,3);

% for ax^3 + bx^2 :

b = initCoeff(1);

b2 = multCoeff(2);

a = multCoeff(1);

figure

scatter(20:20:500,initTimes);

hold on;

scatter(20:20:500, multTimes, '+');

title('Matrix initialization vs Multiplication');

ylabel('Time (s)')

xlabel('Matrix Size (NxN)');

legend('Initialization','Multiplication')

figure

subplot(1,2,1);

scatter(20:20:500,initTimes + multTimes);

hold on;

scatter(20:20:500, a\*(20:20:500).^3 + (b + b2)\*(20:20:500).^2, '+');

title('Matrix Multiplication');

ylabel('Time (s)')

xlabel(['Matrix Size (NxN) A=' num2str(a) ' B=' num2str(b)]);

legend('Measured','Predicted')

subplot(1,2,2);

errors = bsxfun(@rdivide,(abs((a\*(20:20:500).^3 + (b + b2)\*(20:20:500).^2)) - (initTimes + multTimes)),(initTimes + multTimes));

scatter(20:20:500, errors);

hold on;

p1 = [0,0];

p2 = [0,500];

plot([p1(2),p2(2)],[p1(1),p2(1)],'Color','b','LineWidth',1);

title('Matrix Multiplication Model Error');

ylabel('Error Proportion')

xlabel('Matrix Size (NxN)');

----Summary----

This script determines the coefficients for the best fit curve, as well as the errors.

Data:

Example Ping Data:

rwc1data\_daffodil = pingSession('RWC trial 1 Daffodil', 600, 3, [296,320,322,297,311,307,321,296,323,322,293,321,307,371,303,334,305,294,295,326,349,371,395,298,296,361,509,345,359,379,412,427,306,294,300,299,414,297,307,379,299,323,295,369,306,306,297,296,384,407,298,307,380,295,307,296,365,389,306,320,300,380,301,299,300,369,394,314,300,317,316,364,311,350,298,397,314,309,316,389,296,312,297,329,313,305,306,372,395,307,295,295,307,296,296,295,377,401,307,295,369,315,313,312,295,313,406,327,309,314,309,304,296,361,331,294,330,353,360,339,321,303,311,388,297,298,297,379,299,298,313,369,394,309,310,299,384,381,320,300,376,397,318,316,295,389,296,294,312,379,402,310,310,372,394,310,296,296,385,296,310,354,376,401,302,311,311,392,391,307,310,382,298,309,310,309,389,307,297,307,296,380,334,311,302,295,296,348,294,295,307,307,307,386,295,337,302,318,296,295,295,309,295,295,312,307,296,295,310,319,312,307,313,406,362,385,295,295,307,294,390,296,295,307,387,295,295,294,378,389,322,307,299,296,305,308,359,383,390,315,351,374,305,417,341,364,386,308,331,300,378,297,322,345,368,389,313,335,358,381,388,326,349,304,390,317,295,364,387,390,309,309,382,388,294,344,367,304,311,295,356,379,299,324,347,370,386,315,338,361,400,305,329,353,310,302,319,301,309,388,309,331,352,375,297,319,343,366,384,308,332,354,379,296,322,347,371,385,296,338,309,385,306,296,307,307,295,310,293,302,295,296,318,295,313,313,379,297,315,312,296,326,297,298,311,309,314,302,298,298,298,299,311,314,311,299,312,306,299,310,303,305,308,296,296,295,296,306,315,300,311,299,311,301,304,312,308,295,307,310,295,296,297,307,296,294,295,297,297,295,296,295,310,296,328,295,295,307,306,307,305,297,297,297,306,307,318,298,296,294,306,307,296,296,295,296,309,307,296,315,310,296,311,295,304,296,296,296,296,297,308,308,296,307,307,294,295,296,297,296,317,315,315,314,294,296,303,306,305,303,303,312,305,304,303,309,299,307,297,297,306,314,305,315,307,296,296,317,307,304,310,300,296,307,307,313,315,304,310,307,299,315,296,310,295,315,309,310,298,312,311,315,314,298,314,298,312,309,311,310,297,310,297,310,298,315,310,299,313,299,298,299,301,312,303,299,298,301,307,296,307,305,295,311,310,296,311,314,343,296,307,308,310,295,307,296,306,295,309,315,295,307,312,317,315,303,295,298,296,307,297,297,297,306,314,311,315,310,297,306,296,307,307,296,307,315,305,304,315,312,301,312,311,295,300,316,295,296,310,295,311,296,298,312,309,309,309,311,312,299,299,313,298,299,312,298,298]);

----Summary----

This is how the ping data was entered into matlab. It was cleaned up from the output files using regex find-and-replace-in-all-files from notepad++ first. There are 32 datasets like this.

Clock Tick Duration Data:

durations = [4.11e-07,4.11e-07,4.11e-07… ….,4.11e-07,4.1e-07,4.1e-07,4.11e-07];

----Summary----

This data set ended up being quite large to obtain the desired low error level, so only a small subset will be shown here.

Matrix Multiplication Data Set:

initTimes = [0.000314809,0.00125467,0.00279878,0.00497755,0.0077674,0.0111735,0.0151968,0.0198023,0.0250046,0.0309519,0.0374806,0.0444943,0.0521777,0.0605702,0.0694805,0.0790187,0.089222,0.099837,0.11132,0.1235,0.136048,0.149398,0.163331,0.177621,0.193023];

multTimes = [2.21094e-05,0.000185637,0.000608034,0.0014362,0.00286703,0.00478017,0.00756775,0.0116817,0.016601,0.0228369,0.0315675,0.0414805,0.0550665,0.0703848,0.0912619,0.114135,0.139784,0.1654,0.20318,0.236833,0.281416,0.317488,0.372892,0.415152,0.483338];

Matrix Inversion Data Set:

initTimes = [0.000163707,0.000622905,0.00141165,0.00249452,0.00388239,0.0061674,0.00806767,0.0107595,0.0128599,0.0161829,0.0191738,0.0233933,0.026332,0.031127,0.0365778,0.0408588,0.0481683,0.0501633,0.0568939,0.0662243,0.0711,0.0767107,0.0883953,0.100316,0.10119];

invTimes = [0.000217716,0.00167448,0.00552115,0.0129649,0.0257418,0.0441518,0.0737702,0.103916,0.152754,0.2005,0.265786,0.348179,0.443567,0.556201,0.684578,0.82684,0.989027,1.18225,1.38899,1.6143,1.96093,2.18813,2.5337,2.89806,3.22259];

----Summary----

These datasets track the timing for matrix initialization and calculation separately. The values are indexed according to the size of the matrix, eg, the first element corresponds to a 20x20 matrix, and the final value represents timing for 500x500 matrices.