5-2013

Designing Mobile Applications Around Load-Balancing Principles to Improve Performance.

Chris Eaton
East Tennessee State University

College of Business and Technology
Honors Thesis Signature Approval Form

Designing Mobile Applications Around Load-Balancing Principles to Improve Performance

April 8th, 2013

The members of the Thesis Committee approve the Senior Honors Thesis for
Chris Eaton _____

Thesis Committee Chair
Dr. Martin Barrett _____

CBAT Thesis Committee Member
Dr. Phil Pfeiffer _____

External Thesis Committee Member
Dr. Jeff Knisley _____

CBAT Honors Director
Dr. Tom W. Moore _____

Contents

Abstract

Developers of mobile applications commonly delegate computations to networked resources, which have considerably more processing capacity than contemporary mobile devices. The work described here investigates an alternative approach to managing these computations, which uses a dynamic load-balancing algorithm to divide processing work between a mobile device and a back-end server.

Introduction

In spite of major improvements in mobile technologies, the performance of mobile hardware continues to lag that of stationary devices. Due to this performance gap, mobile applications commonly dispatch large processing tasks to back-end servers. This research investigated an alternative approach for managing large processing jobs in a mobile environment. This approach used dynamic load-balancing to divide an iterative workload between a mobile device and a back-end server, in the hope of using the device's processing power to speed processing. An adaptive algorithm was implemented that allocated the work of computing a function $f()$ between a client and a server device. This algorithm was designed to respond to observed client and server throughput and network latency to formulate an efficient division of workload between the client and server. It was hypothesized that this split would yield a completion time $T_m < min(T_c, T_s + T_{net})$, where

- T_c is the time that the client requires to evaluate $f()$
- T_s is the time that the server requires to evaluate $f()$
- T_{net} is the average time to send $f()'s$ result from the server to the client.

It was further hypothesized that this adaptive division of effort would yield a faster completion time T_m relative to a variable workload than any predetermined, static division of effort, in situations where

- $f()$ takes a reasonably long time to compute (i.e., $f() \gg T_{net}$); and
- the mobile client is slow, relative to the server: i.e., $T_c \gg T_s$

History

There has been extensive research using networked resources in wireless networks to improve mobile application performance (Jiang, Yang, & Athale, 2008) (Petrou, Amiri, Ganger, & Gibson, 2000) (Pinho & Nogueira, 2012). This includes the exploitation of a client-server model as a secondary source of computational power (Scarpa, Villari, Zaia, & Puliafito, 2002), with contingencies for redistributing work to the mobile device if network conditions become unfavorable (Jing, Helal, & Elmagarmid, 1999). This fluidity with which work can be distributed over a network suggests the use of non-local parallel processing, and with it, load balancing to improve performance. Due to the volatility of mobile network connections, persistence-based and diffusive load balancing strategies are natural candidates for performance optimization (Corradi, Leonardi, & Zambonelli, 1999) (Elsässer & Sauerwald, 2010) (Hui & Chanson, 1999) (Lifflander, Krishnamoorthy, & Kale, 2012).

Methodology

The problem that was selected for this research, the Sieve of Eratosthenes, is easy to implement and iterative in nature. The Sieve first constructs the sequence of all integers

between 2 and some ceiling value, an integer under which to find prime numbers. It then iteratively reduces the sequence's size by removing the sequence's least term, placing it into a sequence of known primes, and then removing all multiples of this term from the original sequence. Because each successive iteration uses the previous iteration's output as its input, one iteration must complete before the next can begin. In order to distribute the workload within an individual iteration, the sequence to sieve were split in two subsequences, with the client's percentage of the overall sequence for iteration n given by $ClientSplit_n$. Within a given iteration, the client first removes the sequence's first item, which is guaranteed to be a prime, and reports this integer to the server. Both machines can then remove all multiples of this integer from their respective sequence in parallel.

The relative performance of the sequential and split-based versions of the Eratosthenes algorithm depends on the relative performance of the client and server systems and supporting network. In situations where the performance of these resources can't be predicted with a reasonable degree of accuracy, static load balancing approaches could easy perform poorly. The approach for load balancing explored here uses dynamic calculations of resource performance to dynamically distribute load as a function of the devices' measured performance. In this way, the system can be said to adhere to the principle of persistence, in that it maintains computational balance over iterations with gradual changes (Lifflander, Krishnamoorthy, & Kale, 2012).

Additionally, the implementation aims to dynamically diffuse work: i.e., shift work from the more heavily loaded to the more lightly loaded node, in proportion to the difference in

workload between the systems. This transfer takes the form of minimum-sized units, due to an inability to create a perfect division of work by transferring partial terms. This is the basis of the unit-size token model. In this model, which was adapted for this research, the amount of work to transfer is a discrete sequence of a fundamental transfer unit, or token (Elsässer & Sauerwald, 2010) n.

Persistence-Based Load Balancing

Persistence-based load balancing uses historical runtime data to redistribute load. Some metric is used to initially split a workload into pieces perceived to require the same processing time on a sequence of target machines. After each iteration, the time that each machine required to complete its work is reported to the load balancer. The balancer then distributes a next round of work in a way that attempts to balance run-times across the target machines, based on an assessment of the machines' relative performance over previous iterations (Lifflander, Krishnamoorthy, & Kale, 2012).

Diffusive Load Balancing

Diffusive load balancing schemes move work from highly concentrated nodes to neighboring nodes with low concentrations of work. These schemes vary according to their strategies for determining when to evaluate the need for balancing, whether to shift workload, which nodes to shed and assume load, and how much load to transfer.

Evaluating when to balance

Strategies for triggering load balancing are typically periodic or reactive, initiating load balance as a function of elapsed time or after some event. For instances of continuous,

predictable work, periodic triggering effectively ensures that load balancing occurs neither too frequently nor infrequently. However, an iterative work is more suited to reactive load balancing, such that work concentration is assessed and balanced between iterations.

Evaluating whether to balance

Strategies for determining whether to shift work must account for the difference in load between nodes. If this difference is slight, the network overhead required to reallocate work can be greater than the benefit of doing so, indicating the need for a balance threshold. Furthermore, if the unit-size token is relatively large, moving a single token may fail to improve the overall balance. To this end, this phase of a load-balancing algorithm should identify a computation's nodes as overloaded, underloaded, balanced, or balanced under threshold. If a node is balanced or balanced under threshold, no work reallocation should take place.

Evaluating source and target nodes for shifts

Balancing can be initiated in one of three ways. Overloaded nodes can issue requests to send work. Underloaded nodes can issue requests to receive work. Systems can also allow for symmetric load balancing, where both types of requests are supported. After nodes have been identified, workload-transfer partners are established. This step is especially important in a network topology with more than two nodes.

Evaluating how much work to transfer

In the unit-token model, the algorithm computes the amount of work to transfer as a multiple of a unit-size token. This multiple should be the value that best offsets the imbalance between the nodes. If both partners are equal in processing speed, the work to transfer is

computed by balancing the number of tokens. Otherwise, this computation must account for the difference in the workload-transfer partners' processing speeds.

Overdecomposition

Overdecomposition refers to a high-level approach to load balancing, implemented at the application level. In this approach, work is parallelized by partitioning it into medium-sized tasks that can be migrated to another machine (Lifflander, Krishnamoorthy, & Kale, 2012).

Application

The Sieve of Eratosthenes is an algorithm that identifies prime numbers between 2 and some user-specified integer upper bound, $SearchCeiling$. The Sieve operates by repeatedly removing (sieving) all multiples of the smallest known prime in that sequence. On every pass through the sequence, this smallest known prime will also be the sequence's smallest value. In this way, all primes in the sequence will be found when the sequence is exhausted.

```
n = []
primes = []
for i in range(2, top):
        n.append(i)

while(n.size > 0):
        prime = n.pop(0)
        primes.append(prime)

        for p in range(prime*2, top, prime):
                n.remove(p)
```

Figure 1: Simple Sieve of Eratosthenes

The Sieve of Eratosthenes generates a large, repetitive workload suitable for optimization through overdecomposition. The distributed version of the algorithm explored in this work is a three-step algorithm that involves four processes. The first, UI process receives data from the other processes and displays data to the user. It also spawns the Worker and

Networker threads. The second, Worker process sieves integers and does load redistribution calculations. The third, Networker process maintains a socket connection to the fourth, Server process. The Worker uses this socket to communicate with the Server. Additionally, the Networker records the time that elapses between Worker messages and corresponding Server responses, given as $RoundtripMessageTime$. The Server process accepts connections from the Networker, sieves proffered data, generates runtime statistics and performs load redistribution on request.

Initialization

The algorithm's first, initialization step begins with a request to compute primes less than some user-specified $SearchCeiling$. The UI fields this request; spawns a Networker process, which opens a connection with the Server; then spawns the Worker, passing it $SearchCeiling$ and a reference to the Networker.

During its initialization phase, the Worker first calculates $ClientSplit_0$, an initial workload split ratio for the client and server devices. While $ClientSplit_0$ could be calculated based on reported processor speeds or benchmarked processing times, for this research, a static value for $ClientSplit_0$ was established as approximately twice the equilibrium point found through experimentation, so as to allow the system to settle on its own.

$$ClientSplit_0 = 0.15$$

Using $ClientSplit_0$, the Worker calculates $ClientCeiling$, the upper boundary of the sequence for which it will be initially responsible. The Worker extracts the first prime number on which to sieve, given by $Prime$, from this sequence and computes the lower and upper

boundaries of the Server's sequence, given by $ServerFloor$ and $SearchCeiling$, respectively. It passes $Prime$, $ServerFloor$ and $SearchCeiling$ to the Networker, then creates its sequence, ending its initialization step,

In parallel with the Worker's computing it sequence, the Networker packages $ServerFloor$, $SearchCeiling$, and $Prime$ in JSON and forwards them to the Server. Upon receiving these descriptors, the Server creates its sequence, ending its initialization step.

$$ClientCeiling = ceil(SearchCeiling * ClientSplit_0)$$

$$ServerFloor = ClientCeiling + 1$$

$$WorkerSequence_0 = \{2, 3, 4, \dots, ClientCeiling\}$$

$$Prime = ClientSet.first()$$

$$ServerSequence_0 = \begin{Bmatrix} ServerFloor, ServerFloor + 1, \\ ServerFloor + 2, \dots, SearchCeiling \end{Bmatrix}$$

Calculation

In the calculation step, which follows the initialization step, the Worker and Server first record their initial timestamps, given as $WTStart$ and $STStart$, respectively. The Worker begins sieving immediately, removing all multiples of $Prime$ from its sequence. The Server first calculates the least multiple of $Prime$ greater than the least value in its sequence, given as LPM_n, then removes all multiples of $Prime$ from its sequence. After this, the calculation step of the Worker and Server are identical; after the Worker and Server finish sieving, they record

the difference between the current time and $WTStart$ or $STStart$, as $WorkerRuntime$ or $ServerRuntime$, respectively, ending the calculation step for that device.

Redistribution

Following the calculation step, the Worker determines how much work to offload to the Server in the next round of computing, if one is needed. The Worker begins by requesting $ServerRuntime_n$, $ServerSequence.Size()$, and $RoundtripMessageTime_n$ from the Networker. If the Networker cannot immediately satisfy this request, the Worker waits until the Server has sent the first two values and the Networker has computed the third. Upon receiving these values, the Worker calculates the relative amount of time the devices used, given as $TimeRatio$.

$$TimeRatio_n = {RoundtripMessageTime_n}/{WorkerRuntime_n}$$

A $TimeRatio_n$ close to 1 indicates that the Networker received data from the Server at approximately the same time as the Worker finished the calculation step. Otherwise, one or the other device was forced to wait for the other for a significant period of time and a rebalancing of the load may be in order. The algorithm implemented here uses a threshold-based strategy to determine when load balancing may be indicated. This strategy seeks to ensure that network overhead incurred from load balancing is not more costly than the potential gains from splitting the workload (Hui & Chanson, 1999).

To determine if rebalancing is warranted, the Worker performs two checks. For the first, the Worker retrieves the next $Prime_n$ as the first item in $WorkerSequence_{n-1}$. If

$Prime_n$ is significantly large compared to the total original workload, $SearchCeiling$, too little work remains to warrant a split. In this first case, the Worker prompts the Networker to create a JSON $RedistributionMessage$ containing $RequestVals = ServerSequence_n.size()$ and $Stop$, which it forwards to the Server. For the second, if $TimeRatio_n$ is much larger or smaller than 1, the amount of work offloaded to the Server was too small or too large, respectively. In either of these cases, $TimeRatio_n$ indicates the degree to which $ClientSplit_n$ should be altered in order to redirect the work.

$$ClientSplit_n = ClientSplit_{n-1} * \begin{cases} 1, & 0.8 < TimeRatio_{n-1} < 1.2 \\ TimeRatio_{n-1}, & otherwise \end{cases}$$

Due to the nature of the Eratosthenes algorithm, the ratio of the sizes of $WorkerSequence$and $ServerSequence$ changes after each calculation step. The Worker calculates the number of tokens to shift in order to rebalance the workload as follows:

$$TotalVals_n = WorkerSequence_n.size() + ServerSequence_n.size()$$

$$WorkerTargetSize_n = TotalVals_n * ClientSplit_n$$

$$WorkerValsNeeded_n = WorkerTargetSize_n - WorkerSequence_n.size()$$

In order to assure that load balancing is warranted, the algorithm checks that the magnitude of $WorkerValsNeeded_n$ is large compared to $TotalVals_n$:

$$WorkerValsNeeded_n = \begin{cases} 0, |WorkerValsNeeded_n| < \frac{TotalVals_n}{1000} \\ WorkerValsNeeded_n, otherwise \end{cases}$$

The worker then sends $Prime$ to the Server, as part of a JSON message that indicates whether redistribution is needed and, if so, how it should be managed:

- If $WorkerValsNeeded_n$is negative, work should be shifted to the Server. The Worker removes the last $|WorkerValsNeeded_n|$ from $WorkerSequence_n$, storing them as $Payload$. This value, $Payload$, is added to the message before being sent.
- If $WorkerValsNeeded_n$ is zero, no redistribution is appropriate.
- If $WorkerValsNeeded_n$ is positive, work should be shifted to the Worker. The value $RequestVals = WorkerValsNeeded_n$ is added to the message before being sent.

If $WorkerValsNeeded_n$ was zero or negative, the Worker's redistribution step ends and it begins the calculation step once again. If $WorkerValsNeeded_n$was negative, the Worker waits for the Networker to signal that it has received a response from the Server.

The Server, upon receiving the $RedistributionMessage$ from the Networker, responds in one of three ways.

- If $RedistributionMessage$ contains only $Prime$, no redistribution will be performed.
- If $RedistributionMessage$ contains $Payload$, the Server adds $Payload$ to $ServerSequence_n$.
- If $RedistributionMessage$ contains $RequestVals$, the Server removes the first $RequestVals$ values from $ServerSequence_n$, storing them as $Payload$ in $RedistributionData$, then sends $RedistributionData$ to the Networker.

In each case, the Server retrieves $Prime$ from $RedistributionMessage$, ending its redistribution phase, and beginning the calculation step once again.

In order to signal that the computation is complete, the Worker may send a $RedistributionMessage$ to the Server that contains a contains $Stop$ indication. When this happens, $WorkerSequence_{\mathrm{n}}$ will have been emptied and the connection with the Networker is terminated, as it is no longer beneficial to perform calculations in parallel.

When the Worker receives $RedistributionData$ from the Networker, it adds $Payload$ to $WorkerSequence_{\mathrm{n}}$, which ends the Worker's redistribution step and allows it to begin the calculation step. However, if the connection between the Networker and Server was terminated, the Worker performs the calculation step serially without redistribution until $WorkerSequence_{\mathrm{n}}$ is exhausted.

System Architecture

The distributed Eratosthenes algorithm was tested using an Android client application with a back-end Java server application, shown in Figure 1. The Android application was run on

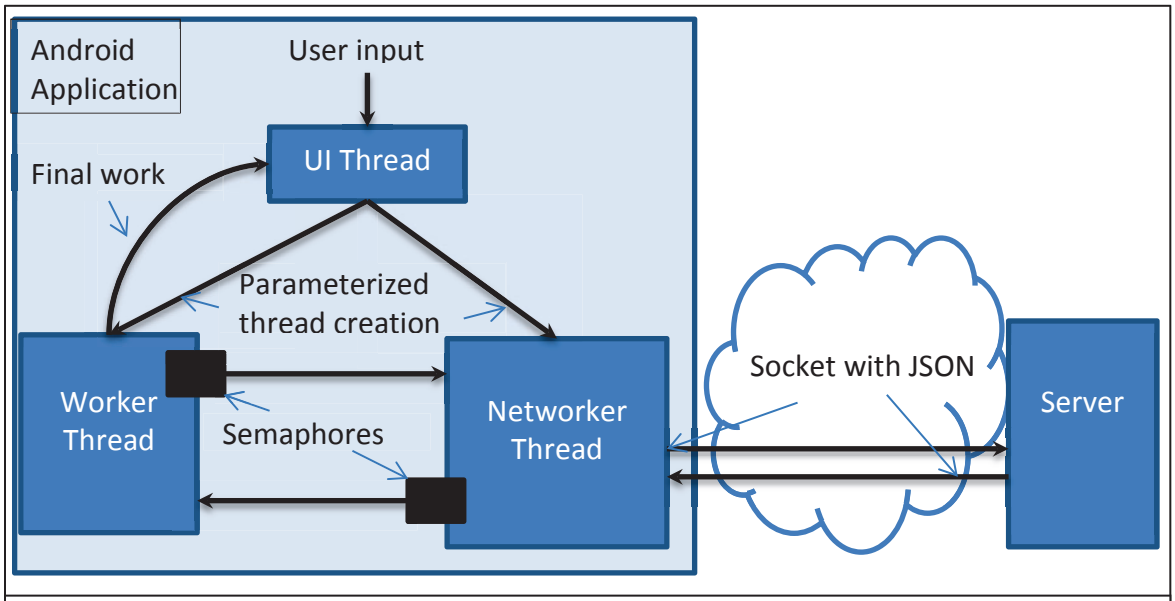

Figure 2: System Architecture with communication channels

an HTC Thunderbolt (1GHz Qualcomm Snapdragon CPU, 768MB RAM) and the server was run on a PC with an Quad-core 3.4GHz Intel i7-2600K CPU and 8.00GB RAM. The client-side's Worker and Networker were implemented as separate threads, in order to support concurrent calculation and communication. The Networker maintains a socket connection with the Server over which it transfers messages between the Worker and the remote server application. Interthread communication between the Worker and Networker used blocking semaphores initialized to "locked". These semaphores control access to private shared memory locations through coupled routines that support read-once access to messages written to this shared memory. This routine blocks the calling thread until the owner writes a new message. This technique ensures that the Worker and Networker remain synchronized during load balancing calculations.

The server-side logic was implemented as a single thread. This thread accepts a socket connection, does the fundamental Eratosthenes algorithm calculations, and reports calculation results over this socket.

Analysis

Preliminary tests of the system demonstrated that the algorithm was finding proper equilibrium, shown in Figures 3 and 4, but had worrisome total time costs to finish calculation. Specifically, a sieve with $SearchCeiling = 100000$ took on average 15 minutes to complete, while the same operation running serially on the server completed in about 6 seconds.

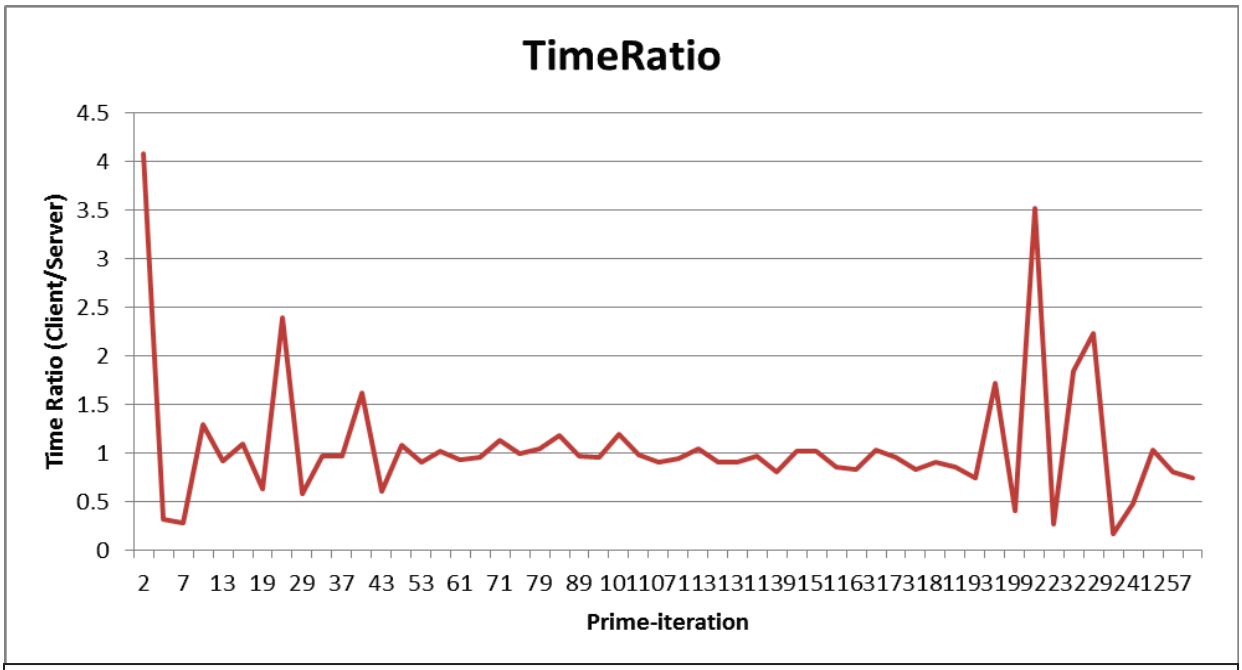

Figure 3: Graph of Client/Server computational time ratio per iteration, $TimeRatio_n$. After some preliminary imbalance, the load balancer reaches approximate equilibrium. Concurrent calculation ends as the time ratios become more chaotic.

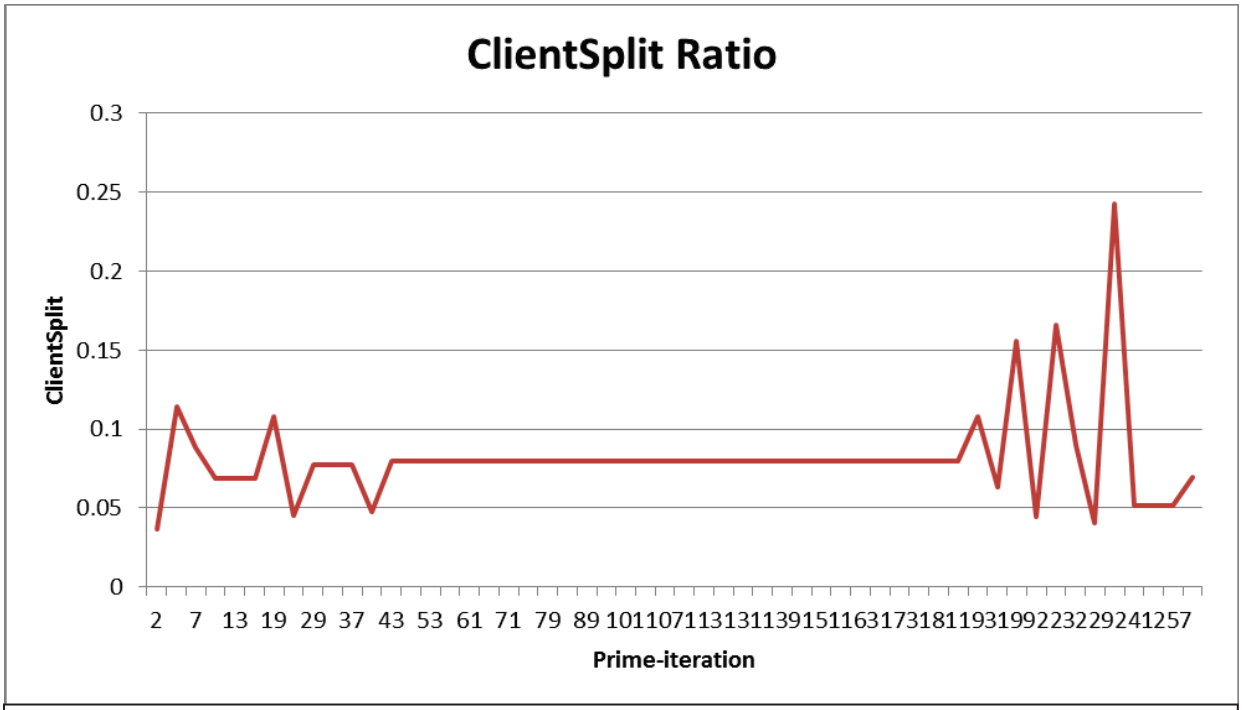

Figure 4: Graph of $ClientSplit_n$ ratio resulting from each iteration. After some preliminary imbalance, the effects of the threshold-based approach can be seen, translating approximate $TimeRatio_n$ equilibrium into a steady $ClientSplit_n \approx 0.08$. Concurrent calculation ends as $ClientSplit_n$ becomes more chaotic.

Further analysis revealed that after the 25th iteration with $Prime = 97$, prime-multiples were rarely being found in the set, so load shifting was failing to yield a performance benefit. To improve the algorithm's performance, the algorithm was revised so as to send the $stop$ message after computing with a $Prime$ sufficiently large relative to $SearchCeiling$. The result was a well-balanced concurrent calculation phase, as seen in Figures 3 and 4, with an average concurrent pass time of 15ms, average network latency of 6ms, and average $ClientSplit$ of 0.08. This was consistent with repeated test runs. However, the processing took 30 seconds to complete which was still much too long.

The degree to which the serial, server-side implementation of Eratosthenes outperformed the parallel implementation dictated a more careful analysis of the overheads required by the parallel approach. The time needed to perform some iterative calculation on a mobile device can be simply expressed as follows:

$$CalculationTime \approx n * ClientAvgIterationTime$$

The time needed for a mobile device to request that a server perform some iterative calculation serially and return the result to the mobile device can be expressed as follows:

$$CalculationTime \approx (n * ServerAvgIterationTime_{serial}) + NetworkOverhead$$

If $ServerAvgIterationTime \ll ClientAvgIterationTime$ and network latency and message transmission time are small relative to calculation time, the latter equation defines a lower bound for receiving results on a mobile device, despite requiring network overhead constituting one message and response.

If the parallel operation is relatively stable, the time needed to compute with the parallel approach can be roughly expressed as follows:

$$ClientAvgIterationTime_{parallel} \approx ServerAvgIterationTime_{parallel}$$

$$ParallelAvgIterationTime \approx ClientAvgIterationTime_{parallel} * ClientSplit$$

$$CalculationTime \approx n * (ParallelAvgIterationTime + NetworkOverhead + RedistributionTime)$$

This updated model only provides a rough estimate, since it assumes an approximately constant runtime, rather than one that decreases at each iteration. It does, however, expose the primary difference in the server-serial approach and parallel approach: server-serial requires one instance of $NetworkOverhead$, while the parallel approach requires n instances of $NetworkOverhead + RedistributionTime$. In order for parallel calculation to be of benefit, the time saved per iteration must outweigh this cost: i.e.,

$$TimeSavedPerIter = ServerAvgIterationTime - ParallelAvgIterationTime$$

$$n * TimeSavedPerIter > (n-1)NetworkOverhead + n * RedistributionTime$$

If n is sufficiently large, this requirement can be simplified to

$$TimeSavedPerIter > NetworkOverhead + RedistributionTime$$

While this is feasible, the average $ClientSplit$ of 0.08 observed from experimentation implies that $ServerAvgIterationTime_{parallel}$ was not significantly smaller than the potential $ServerAvgIterationTime_{serial}$ where computation was done solely on the server machine.

Conclusion

The parallel implementation of the Sieve of Eratosthenes investigated in this research was unable to out-perform the server-serial approach due to several points of failure. First, the Sieve was found not to be an optimal candidate for this approach; it creates a diminishing amount of work after each iteration, where the largest workload is the first. Additionally, the workload eventually diminishes to the point where parallel computation is no longer required or beneficial. Future research may attempt to implement this approach around an algorithm that creates a steadily increasing workload. Second, the volume of network traffic required by the algorithm to maintain workload balance and initialize each iteration was too great for the small performance gain created by parallel processing. Future research may attempt to implement an algorithm that does not require network operations after each short iteration. Third, the parallel approach may not have been well suited to the hardware used. The disparity of processing power between the Android device and server PC was too large to allow the Android device to partake in parallel communication without becoming a burden. Future research may look to reduce this disparity with a more powerful Android device, more occupied server hardware, or a network of similarly-capable Android devices.

Appendix

UI Thread

```
public class UIThread extends Activity {

    private TextView textView;

    protected String serverMessage;
    synchronized String getServerMessage() {
        return serverMessage;
    }
    synchronized void setServerMessage(String msg) {
        serverMessage = msg;
    }

    protected void onCreate(Bundle savedInstanceState) {
        ...

        Intent intent = getIntent();
        String message = intent.getStringExtra(MainActivity.EXTRA_MESSAGE);
        Integer ceil = Integer.parseInt(message);
        String displayString = "";

        //Manages socket connection and message passing
        NetWorker INetWorker = new NetWorker();

        //Performs local work and communicates using passed NetWorker
        Worker LocalWorker = new Worker(ceil, INetWorker);

        //Set network initialization parameters
        INetWorker.start = LocalWorker.calcServerFloor();
        INetWorker.ceiling = ceil;

        //Launch both threads
        INetWorker.executeOnExecutor(AsyncTask.THREAD_POOL_EXECUTOR);
        LocalWorker.executeOnExecutor(AsyncTask.THREAD_POOL_EXECUTOR);

        //Create list of primes
        displayString += LocalWorker.getWorkerMessage() + ", ";

        //Display final message
        new UpdateDisplayTask().execute(displayString);
    }

}
```

Worker Thread

```
public class Worker extends AsyncTask<Integer, Integer, String> {

    private String workerMessage;
    private final Semaphore messageLock = new Semaphore(0);

    public synchronized String getWorkerMessage() {
        messageLock.acquire();
        return workerMessage;
    }

    protected ArrayList<Integer> sequence;
    protected ArrayList<Integer> primes;
    protected float splitPercent = 0.15f;
    protected int clientCeiling;
    protected int totalCeiling;
    protected NetWorker INetWorker;

    public Worker(int ceiling, NetWorker workPartner) {
        totalCeiling = ceiling;
        clientCeiling = calcClientCeiling(ceiling, splitPercent);
        INetWorker = workPartner;
        primes = new ArrayList<Integer>();
    }

    protected String doInBackground(Integer... ceiling) {

        long totalts = System.nanoTime();

        sequence = buildsequence(clientCeiling);

        int p = sequence.get(0);
        sequence.remove(0);
        primes.add(p);

        while(sequence.size() > 0) {

            passData pData = eSievePass(sequence, p);
            sequence = pData.sequence;

            long clientDTS = pData.dts;

            String serverMessage = INetWorker.getServerMessage();

            p = sequence.get(0);
            sequence.remove(0);
            primes.add(p);

            JSONObject serverJSON = new JSONObject(serverMessage);
            long serverDTS = serverJSON.getLong("dts");
            int serverSequenceSize = serverJSON.getInt("sequenceSize");

            double sequenceTimeRatio = calcTimeRatio(clientDTS, serverDTS);
```

```
            splitPercent = calcNewsequenceSplit(splitPercent,
                           sequenceTimeRatio);


            JSONObject redistJSON = buildRedistJSON(splitPercent,
                                                    pData.sequence.size(),
                                                    serverSequenceSize,
                                                    totalCeiling,
                                                    p);

            redistributeLoad(redistJSON);

            if(redistJSON.has("stop")) {
                break;
            }
        }
        messageLock.release();
    }

    private static Integer calcClientCeiling(int ceiling, float split) {
        int algSplit = (int)(Math.floor(ceiling*split));
        return algSplit;
    }

    private static double calcTimeRatio(long clientDTS, long serverDTS) {
        return (double)clientDTS / (double)serverDTS;
    }

    private static double calcSequenceSizeRatio(int clientSize, int
     serverSize) {
        return (double)clientSize / (double)serverSize;
    }

    private float calcNewsequenceSplit(float oldSplit, double timeRatio) {
        float newSplit = oldSplit;
        if(timeRatio < 0.8 || timeRatio > 1.2) {
            newSplit /= timeRatio;
        }
        return newSplit;
    }

    private JSONObject buildRedistJSON(float targetSplit, int
    clientSequenceSize, int serverSequenceSize, int startCeil, int nextP) {

        int totalSize = clientSequenceSize + serverSequenceSize;
        int clientTargetNumVals = calcClientCeiling(totalSize, targetSplit);

        //Stop if total remaining workload is small
        if(totalSize < 100) {

            JSONObject stopRequest = buildStopJSON(serverSequenceSize);
            return stopRequest;

        } else if(clientTargetNumVals > clientSequenceSize){
            //client needs more vals
```

```
            JSONObject dataRequest = buildDataRequestJSON(
                                        clientTargetNumVals,
                                        clientSequenceSize,
                                        serverSequenceSize,
                                        nextP);
            return dataRequest;

        } else if (clientTargetNumVals < clientSequenceSize) {

            JSONObject dataSend = buildDataSendJSON(
                                    clientSequenceSize-clientTargetNumVals,
                                    clientSequenceSize,
                                    nextP);
            return dataSend;

        } else {

            JSONObject continueCommand = buildContinueJSON(nextP);
            return continueCommand;
        }
    }

    private void redistributeLoad(JSONObject redistJSON) {
        if (redistJSON.has("payload")) {
            INetWorker.setClientMessage(redistJSON);

        } else if (redistJSON.has("requestVals")) {
            INetWorker.setClientMessage(redistJSON);
            JSONObject payloadJSON = new JSONObject(
                                            INetWorker.getServerMessage());
            ArrayList<Integer> payload = JSONArrayPayloadToArrayList(
                                        payloadJSON.getJSONArray("payload"));

            sequence.addAll(sequence.size(), payload);

        } else if (redistJSON.has("p")) {
            INetWorker.setClientMessage(redistJSON);

        } else if (redistJSON.has("stop")) {
            INetWorker.setClientMessage(redistJSON);
        }

    }

    private JSONObject buildDataRequestJSON(int clientTarget, int clientSize,
                                            int serverSize, int nextP) {
        JSONObject dataRequest = new JSONObject();
        int basicRequest = clientTarget-clientSize;

        dataRequest.put("p", nextP);
        if(serverSize > basicRequest) {
            dataRequest.put("requestVals", basicRequest);
        }
        return dataRequest;
    }
```

```
private JSONObject buildDataSendJSON(int numValsToSend,
                                     int clientSequenceSize,
                                     int nextP) {
    ArrayList<Integer> payload = new ArrayList<Integer>(numValsToSend);
    //Remove requested number of values from end of sequence
    while(payload.size() < numValsToSend) {
        payload.add(sequence.get(sequence.size()-1));
        sequence.remove(sequence.size()-1);
    }
    //Put in ascending order
    Collections.reverse(payload);
    JSONArray JSONpayload = new JSONArray(payload);
    JSONObject dataPayload = new JSONObject();

    dataPayload.put("p", nextP);
    dataPayload.put("payload", JSONpayload);

    return dataPayload;
}

private JSONObject buildContinueJSON(int nextP) throws JSONException {
    JSONObject continueJSON = new JSONObject();
    continueJSON.put("p", nextP);
    return continueJSON;
}

private JSONObject buildStopJSON(int remainingVals) throws JSONException
{
    JSONObject stopJSON = new JSONObject();
    stopJSON.put("requestVals", remainingVals);
    stopJSON.put("stop", "stop");
    return stopJSON;
}

protected ArrayList<Integer> buildsequence(int clientCeiling)
{
    ArrayList<Integer> sequence = new ArrayList<Integer>(clientCeiling);
    for(int i = 2; i <= clientCeiling; i++) {
        sequence.add(i);
    }
    return sequence;
}

passData eSievePass(ArrayList<Integer> sequence, int p)
{
    long ts = System.nanoTime();
    int endVal = sequence.get(sequence.size()-1);

    for (int i = p*2; i <= endVal; i += p) {
        int index = sequence.indexOf(i);
        if (index != -1) {
            sequence.remove(index);
        }
    }
    long dts = System.nanoTime() - ts;
    return new passData(sequence, dts);
}
```

```
    static class passData
    {
        public ArrayList<Integer> sequence;
        public long dts; //delta time stamp in nanoseconds

        public passData(ArrayList<Integer> sequence, long dts) {
            this.sequence = sequence;
            this.dts = dts;
        }
    }
}
```

Networker Thread

```
public class NetWorker extends AsyncTask<Integer, Integer, void> {

    public NetWorker() {

    }

    public int start;
    public int ceiling;

    private Socket clientSocket;
    DataInputStream inFromServer;
    DataOutputStream outToServer;

    private JSONObject clientMessage;;
    private final Semaphore clientMessageLock = new Semaphore(0);
    private String serverMessage;
    private final Semaphore serverMessageLock = new Semaphore(0);

    public String getServerMessage() {
        serverMessageLock.acquire();
        return serverMessage;
    }

    public void setClientMessage(JSONObject clientJSON) {
        clientMessage = clientJSON;
        clientMessageLock.release();
    }

    public JSONObject getClientMessage() {
        clientMessageLock.acquire();
        return clientMessage;
    }

    protected void doInBackground(Integer... initVals) {
        clientSocket = new Socket("192.168.0.100", 6789);
        DataInputStream inFromServer = new DataInputStream(
                                            clientSocket.getInputStream());
        DataOutputStream outToServer = new DataOutputStream(
                                            clientSocket.getOutputStream());
        //Prompt initialization
        int serverStart = start;
        int serverCeiling = ceiling;
        String startupJSON = buildFirstJSON(serverStart, serverCeiling);

        sendJSON(startupJSON);

        boolean stop;
        do {
            serverMessage = readJSON();

            //Blocking call: Will wait for Worker thread to send message
            JSONObject clientRedistMessage = getClientMessage();
            sendJSON(clientRedistMessage);
```

```
            if(clientRedistMessage.has("requestVals")) {
                serverMessage = readJSON();
            }

            stop = clientRedistMessage.has("stop");
        }while (!stop);
        clientSocket.close();
    }

    private String buildFirstJSON(Integer serverStart, Integer ceiling){
        JSONObject firstPack = new JSONObject();
        firstPack.put("p", 2);
        firstPack.put("start", serverStart);
        firstPack.put("ceiling", ceiling);
        return firstPack.toString();
    }

    private void sendJSON(JSONObject redistJSON) throws IOException{
        outToServer.writeUTF(redistJSON.toString());
    }

    private JSONObject readJSON() throws IOException{
        JSONObject json = inFromServer.readUTF();
        serverMessageLock.release();
        return json;
    }
}
```

Server

```
public class SieveServer {

    private static ArrayList<Integer> sequence;
    private static Socket connectionSocket;
    private static DataInputStream inFromClient;
    private static DataOutputStream  outToClient;

    public static void main(String[] args) throws IOException {
          ServerSocket welcomeSocket = new ServerSocket(6789);
          while(true)
          {
            //accept new connection
            connectionSocket = welcomeSocket.accept();

            //Accepts a user's message to the server and makes it readable
            inFromClient = new DataInputStream(
                                    connectionSocket.getInputStream());
            //Allows server to send a response to the client
            outToClient = new DataOutputStream(
                                    connectionSocket.getOutputStream());

            JSONObject receivedJSON;
            JSONObject sendingJSON;
            while(!connectionSocket.isClosed()) {
                int p;
                String clientData = inFromClient.readUTF();
                receivedJSON = new JSONObject(clientData);

                int remainingTokens;

                sequence = buildsequence(receivedJSON.getInt("start"),
                                         receivedJSON.getInt("ceiling"));
                do {
                    p = receivedJSON.getInt("p");
                    passData pData = eSievePass(sequence, p);
                    sequence = pData.sequence;

                    sendingJSON = new JSONObject();

                    sendingJSON.put("dts", pData.dts);
                    sendingJSON.put("sequenceSize", pData.sequence.size());

                    outToClient.writeUTF(sendingJSON.toString());

                    //Get load redistribution info from client
                    clientData = inFromClient.readUTF();
                    receivedJSON = new JSONObject(clientData);
                    remainingTokens = redistributeLoad(receivedJSON);
                }while(remainingTokens > 0);

                connectionSocket.close();
            }
          }
        }
```

```
private static int redistributeLoad(JSONObject redistJSON)
                        throws JSONException, IOException {
    if (redistJSON.has("payload")) {

        JSONArray JSONpayload = redistJSON.getJSONArray("payload");
        ArrayList<Integer> payload =
                        JSONArrayPayloadToArrayList(JSONpayload);
        //add given values to the beginning of list
        sequence.addAll(0, payload);
        return redistJSON.getInt("p");

    } else if (redistJSON.has("requestVals")) {

        int numVals = redistJSON.getInt("requestVals");
        ArrayList<Integer> payload = new ArrayList<Integer>(numVals);

        for(int i = 0; i < numVals; i++) {
            //grab subsequent elements from beginning
            //of list to send to client
            payload.add(sequence.remove(0));
        }
        JSONArray JSONpayload = new JSONArray(payload);
        JSONObject payloadJSON = new JSONObject();
        payloadJSON.put("payload", JSONpayload);

        outToClient.writeUTF(payloadJSON.toString());
        if(redistJSON.has("p")) {
            return redistJSON.getInt("p");
        } else
            return 0;

    } else if (redistJSON.has("p")) {
        return redistJSON.getInt("p");

    } else if (redistJSON.has("stop")) {
        return 0;
    } else {
        return -1;
    }
}

static ArrayList<Integer> buildSequence(int start, int ceiling) {
    int sequenceNumItems = ceiling-start+1;
    ArrayList<Integer> sequence =
                        new ArrayList<Integer>(sequenceNumItems);
    for(int i = start; i <= ceiling; i++) {
        sequence.add(i);
    }
    return sequence;
}
```

```
    //Calculates the least multiple of a given P
    //that is greater than the first item in the sequence
      static int firstPMultiple(int p, int firstInsequence) {
          int fpm = firstInsequence + (p - firstInsequence % p);
          return fpm;
      }

      static passData eSievePass(ArrayList<Integer> sequence, int p) {
          long ts = System.nanoTime();
          int startVal = firstPMultiple(p, sequence.get(0));
          int endVal = sequence.get(sequence.size()-1);

          for (int i = startVal; i <= endVal; i += p) {
              int index = sequence.indexOf(i);
              if (index != -1) {
                  System.out.print(sequence.get(index)+",");
                  sequence.remove(index);
              }
          }

          long dts = System.nanoTime() - ts;
          return new passData(sequence, dts);
      }

      static class passData
      {
          public ArrayList<Integer> sequence;
          public long dts; //delta time stamp in nanoseconds

          public passData(ArrayList<Integer> sequence, long dts) {
              this.sequence = sequence;
              this.dts = dts;
          }
      }
  }
```