DISTRIBUTED COMPUTING ON ANDROID

TABLE OF CONTENTS

0. Compile and run 2

0.1 Operating system 2

0.2 Software 2

0.3 Compiling and running server 2

0.4 Compiling and running client 2

1. Problem statement 3

2. Decomposition techniques 4

3. Code structure 5

3.1 Server description 6

3.2 Client description 9

4. Correctness 11

5. Performance 11

6. Results 12

7. Future and on-going work 14

7.1 Create customized linkedlists 14

7.2 Dynamic time-out periods 14

7.3 Increasing coherency 14

7.4 Varying window size 15

7.5 Testing scalability 15

7.6 On site caching 15

8. References 16

9. Appendix 17

Appendix I : LU decomposition 17

Appendix II : Selecting window sizes 17

Appendix III : Graphical user interface 19

TABLE OF FIGURES

Figure 1 Server Description 5

Figure 2 Job queue example 9

Figure 3 Client-Server interaction 10

Figure 4 Matrix size compared to the time of execution 12

Figure 5 Line graph comparing 3 phones' execution time 12

Figure 6 Speed up for each matrix 13

0. COMPILE AND RUN

0.1 OPERATING SYSTEM

1. This code must be used with a UNIX-like operating system (the system was developed on MacOSX 10.8.5 and Ubuntu 12.04).

0.2 SOFTWARE

1. Install Java 7 (http://www.oracle.com/technetwork/java/javase/downloads/index.html)
2. Install Eclipse (http://www.eclipse.org/downloads/)
3. Install the Android Development Tools for Eclipse (http://developer.android.com/sdk/installing/installing-adt.html)

0.3 COMPILING AND RUNNING SERVER

1. Unzip folders
2. Open terminal and navigate to the `src` folder of "LUDecompositionServer" :

   ```
   $ cd ~/Downloads/LUDecompositionServer/src)
   ```
3. Open ServerMain.java in the `LUServer` folder
4. Change the value of *MIN_NUMBER_THREADS* to the number of phones that will run the client application (i.e. set to 1 if using one phone, set to 2 for two phones, etc.) and save.
5. In terminal, compile code:

   ```
   $ javac LUServer/*.java
   ```
6. In terminal, execute code (N.B. Matrix30.csv may be replaced with any matrix file in the folder) :

   ```
   $ java LUServer/ServerMain Matrix30.csv
   ```
7. The terminal should output the input matrix (Matrix30.csv in this example) and display:

   ```
   Looking for connections…
   ```
8. The server is now ready and waiting for connections.

0.4 COMPILING AND RUNNING CLIENT

1. Import the android project "LUServerDecomposition" into Eclipse.
2. In ClientThread.java, change the value of *SERVER_IP* to your current IP address.
3. Plug in Android device and run the code on the device.
4. Once the application is running on the phone, press "Connect". The client will display job information and the server will display connection information. NOTE: If using many phones, unplug plugged in phone and repeat step 3 and 4. Also, the server will not execute until the *MIN_NUMBER_THREADS* has been connected.

1. PROBLEM STATEMENT

From this project, the team sought to create a distributed framework for performing a simple computation, on Android devices. A central server should take a large computation and decompose it. Android devices will then be allowed to connect to the server. Upon connecting, the devices will be assigned tasks to complete, and upon completing the tasks, the results will be sent back to the server to be mapped. To test the efficiency of the framework, the team chose to perform a distributed LU Decomposition.

When designing a framework to perform a distributed LU decomposition, the team identified several key design problems, not present in local parallel computing. These problems are as follows:

1. Android devices should be allowed to connect and disconnect to the server, as they please. Because of this, the number of resources available is constantly in flux; an Android device may connect or disconnect mid-way through a series of computations. This is contrary to local parallel computing, which traditionally has a fixed number of resources (E.g. a four core machine, always has exactly four cores).
2. As an extension of the previous problem, there is no guarantee that any device will complete an allocated computation, as a device may disconnect mid-computation. Thus, traditional software devices such as barriers and locks cannot be used by the Android devices. As an example, if a remote device acquires a lock, and disconnects, never freeing the lock, deadlock may occur. Likewise, if all devices wait for a single device to finish a computation, and that device disconnects, no devices will move past the barrier, and the computation will not complete.
3. In distributed computing, communication time is considerably more burdensome. There is no upper bound to the possible communication time, as there is no guarantee where the devices are connecting from relative to our server.
4. Another fundamental problem is in ensuring all Android devices are coherent. It would be wasteful for one device to do work that is out of date. Likewise it could be problematic, if these devices overwrite results of recent work with the results out of date work.
5. Connecting Android devices may have different hardware, and different capabilities. Thus, parallelization schemes and distribution methods should be independent of underlying hardware, however they should be biased in who they allocate work too. That is, a slow phone should not bottleneck a computation.

In creating our distributed LU decomposition framework, all the above problems, should be addressed.

2. DECOMPOSITION TECHNIQUES

In LU Decomposition, the matrix is decomposed in iterations (See Appendix for description of LU Decomposition). Within each iteration, the pivot row (at iteration *i*, this represents row *i*) remains constant, while the calculation of all subsequent rows is dependent on the pivot row, and the pivot column. As such, a natural parallelisation scheme is to calculate all rows in parallel. Further parallelisation may be achieved, by parallelising not only the rows, but the columns in each row as well, however this level of granularity was detrimental. As described in class, the communication time may be modelled as:

Communication Time = $T_o + n/B$ [2]

Where,

T_o = the time to send an empty message.

n = the message size

B = the link bandwidth

In distributed computing, T_o is relatively large, while B is relatively small. As such, it is desirable to reduce the number of messages required. Which in this case, requires increasing the message size, and not using the full parallelization scheme.

Thus, it was decided that each row in the matrix, be a single task. All processes must wait for all rows to be computed, before continuing to the next iteration.

3. CODE STRUCTURE

As detailed in the Problem Statement, the program should follow a typical client server architecture. All code is written in Java.

The class ServerMain contains the main method to start the server. When first starting, the server immediately reads in a matrix, specified in the command line arguments, and starts the ConnectionManager class. ServerMain then continues to decompose the matrix into Jobs. Each Job corresponds to a single row in the matrix, and has a sequence number and an iteration number associated with it. Sequence numbers are used for piecing the results back together, they indicate where the results of the Job should be mapped to. Iteration numbers are used to indicate which iteration of the computation is currently been complete. Iteration numbers are useful in ensuring Android devices are coherent, and already completed work, is not being complete again.

Where the ServerMain class was responsible for orchestrating the computation, ConnectionManager is responsible for listening for incoming Android devices, allocating resources for them, and monitoring the connections. Upon receiving a TCP/IP connection request from an Android device, ConnectionManager creates a ServerThread for the connection, passes the new ServerThread the socket to communicate on, and then continues to listen to the front door for incoming connections. The ConnectionManager is also responsible for receiving completed Jobs from Android clients. A high-level diagram of the architecture can be seen below.

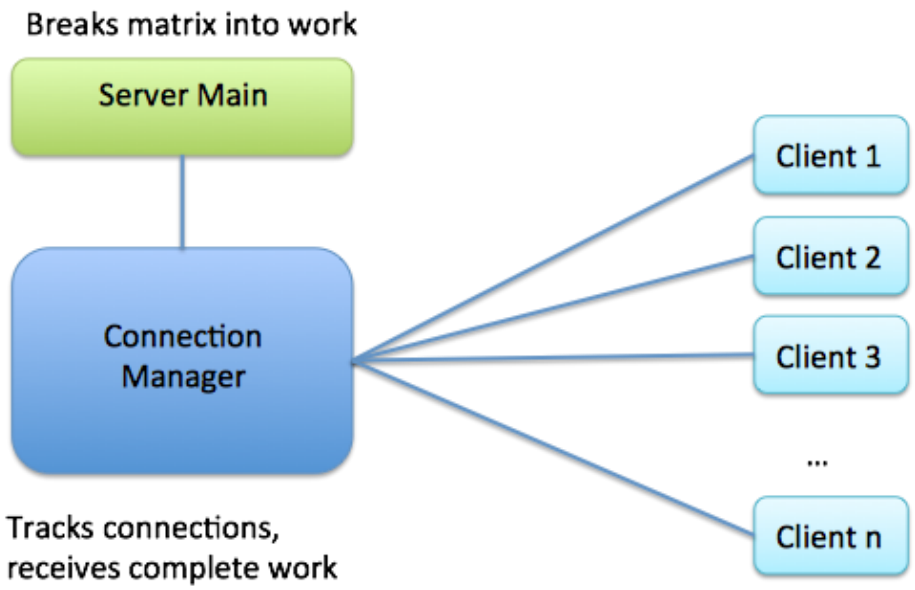


Figure 1 Server Description

3.1 SERVER DESCRIPTION

Once a ServerThread has been created to handle a connection, the ServerThread creates three new threads to help with the connection; Boss, Sender and Heart. The ServerThread then goes into a listening mode, where it simply listens for incoming messages from the Android client, and relays them as necessary. The ServerThread relays incoming messages by first translating the message into a ClientResponse, then determining where the message is bound, either Heart or Boss, and placing the ClientResponse in the Heart or Boss' message buffer respectively. If the message is bound for the Heart, an extra step of interrupting the Heart thread is performed at the end.

The Heart thread is responsible for monitoring connections to make sure they are still alive. The Heart thread does so by periodically sending heartbeats to the Android client. A heartbeat is a special message. Upon receiving the heartbeat, the Android client responds with a heartbeat. If the response heartbeat arrives before the Heart has timed-out, the connection stays alive, otherwise, the connection is closed and the resources allocated are freed. This is accomplished by sending the heartbeat, and sleeping the Heart thread for the timeout specified. If the Heart is interrupted before it stops sleeping, an InterruptedException will be thrown, and caught, and the Heart will wait for the timeout period to complete, then send another message. If the Heart is not interrupted before it stops sleeping, the 'catch' block will not be executed, and instead, the Heart thread will notify the ServerThread the connection should be closed, and proceed to die.

The Boss thread is responsible for consuming work from ServerMain, tracking the Jobs sent to the client, and writing the results of the complete Job to the appropriate location. The Boss thread, has two main queues; `jobsToSend`, and `sentJobs`. The queue `jobsToSend` tracks the Jobs consumed from ServerMain, that have yet to be sent to the Android client. The queue `sentJobs` tracks all jobs sent to the Android client, the results of which have not yet been received. When ServerThread adds a ClientResponse to the Boss' response buffer, the Boss thread iterates through the `sentJobs` queue and removes all Jobs corresponding to the same iteration and sequence number of the client response. The Boss then acquires a lock for the matrix row from the ClientResponse, and writes the results back to the matrix. The lock corresponding to the relevant matrix row, is simply the Job removed from the queue. Using the specific Job as the row lock elegantly removes contention, as little to no other clients will be contending for the same row.

The Boss thread also has a window size associated with it. The window size represents the maximum number of Jobs that the Boss can have at one time. If the sum of the number of entries in the `jobsToSend` and `sentJobs` queue falls below the window size, the Boss will consume a new Job, and add it to the `jobsToSend` queue. Upon getting the chance, it will then send entries from `jobsToSend` to the Android client. Within the Boss thread, all events are prioritized differently. Processing a ClientResponse is most important, followed by consuming more work (adding work to `jobsToSend`), followed by sending new jobs.

The window size addresses the third problem listed in the Problem Statement, referring to cumbersome communication times. Setting a window of jobs sent allows for the communication to be pipelined. This is beneficial, since one Job may be computed by the Android client, while the next job is being communicated to it. Making the

computations large enough allows us to mask the communication time of jobs in the window, thus reducing the total communication time. This shows substantial time benefits over a stop and wait protocol (where window size is 1).

Another special feature of the two queues in the Boss thread, is that they are self-purging. When the Boss thread consumes a Job it compares the new Job's iteration number, to the highest iteration number it has seen so far. If the new Job's iteration number is larger, then a new iteration has begun. To remain coherent, the Boss sends the new Job, then iterates through all Jobs in its two queues, and removes any Jobs with out of date iteration numbers, thus ensuring old jobs are not sent to the Android client. Likewise, when receiving work complete from an Android client, the iteration number of the ClientResponse is also compared to the current iteration number. If the ClientResponse is out of date, the results are not written back to the main matrix, ensuring we do not overwrite new results with old results. This partially addresses the fourth problem identified in the Problem Statement, regarding keeping the results consistent, and keeping the clients coherent. Other coherency schemes will have to be implemented on the client devices to fully address this problem.

For the Boss and Heart threads to send messages to the Android client, the Sender thread is used. Sender is responsible for taking Jobs converting them to bytes messages, and sending them to the Android client. As the I/O is blocking, is has been designated its own thread.

In the event that a client connection is closed, the ServerThread is responsible for freeing all resources. This requires closing the connection and setting all threads to die, at the next convenient time. This is accomplished by including a `die()` method within each thread. This simply sets a flag, `killSelf`, to true. Periodically throughout each thread, the thread will check its `killself` flag. If true, it frees all local resources, and returns from the `run()` method, thus killing the thread. If any thread decides the connection should be closed, it interrupts the ServerThread, indicating all resources should be freed.

If client connections should be closed for natural events (E.g. the computation has complete), before setting the children threads to die, the ConnectionManager will first add a disconnect message to all the Sender threads, and sleep a predetermined amount of time, proportional to the number of connections. This sleep should provide enough time for the disconnect messages to be sent, it then interrupts each ServerThread, signaling the start of the shutdown.

To address the first and second problems discussed in the Problem Statement, regarding the dynamic nature of distributed computing, a special Job queue (holding all Jobs to be compute) is created within ServerMain. In order to solve the second problem, regarding the unreliability of Android clients, a circular queue was used. Essentially, when a Job is consumed by a Boss, the job is popped from the head of the queue, given to the Boss, and appended to the tail. Thus, even if the Android client disconnects before completing the Job, the Job is still in the queue, and can be complete by another client. Only when a ClientResponse is received, is a Job permanently removed from the main Job queue.

Using a circular queue as so, may however, allow different Android clients to compute the same Jobs, making the computation redundant. This is highly unlikely in the early stages of the computation when there are

many Jobs remaining in the Job queue, because Jobs are taken sequentially. However, during the later stages, computations may be redundant. This is entirely acceptable, as it ensures all computations get done, and can even speed up computations in the final stages. As an example, say a very slow Android client has been working on a single computation for some time, and a faster Android client gets the slower Android client's Job from the main Job queue. Regardless, of which Android client actually completes the Job, that Job will have been completed as quickly as possible, from the greedy perspective. This actually addresses the fifth problem identified in the Problem Statement, referring to slower devices acting as a bottleneck.

To further optimize the central Job queue, the queue was actually extended to model the array-based lock seen in class. Instead of having a single Job queue in shared memory, there is an array of Job queues. This drastically decreases contention. At the start of each iteration, the matrix is decomposed into Jobs and added to the array of Job queues. The number of queues in the array is equal to the number of Android clients currently connected. The number of Jobs assigned to each queue is proportional to the number of Jobs that the Android client completed in the previous iteration. When a Boss consumes a new Job, it consumes the Job from the queue it is assigned to, and it only has to acquire the lock for its own queue. Upon completing its queue of work, the Boss is then assigned a new queue number to start consuming Jobs from. This is in the spirit of task stealing.

In the case where none of the current Android clients were present during the last iteration (this really only happens during the first iteration), the number of queues created is equal to a pre-set constant. Likewise, the Jobs are distributed evenly among the queues.

In this scheme, there is only contention for consuming Jobs when one Boss finishes its queue and moves to the next, or when a new Android client joins the computation, and gets assigned to a queue, that already has a Boss (unless that Boss has disconnected). Because the queue sizes are proportional to the number of Jobs that client computed in the last iteration, the first case of contention will rarely happen. It is likely that the Android client will consume Jobs at a very similar rate to that of the previous iteration. The second form of contention, is uncontrollable, we have no idea when a new Android client will connect.

To track the queues that remain to be compute, another circular queue is used. When a Boss needs a new queue (either because it has exhausted its own, or it is a new client), it pops a queue number from the head of the linked list, sets it's queue number to the number popped, and appends the number to the tail. The queue number is only removed from the list when the queue has been exhausted. Since the assignment queue is circular, even when contention does occur, it will be evenly distributed among the queues. That is, assuming no clients disconnect, all queues will have the same number of clients, plus or minus one, contending for work. The array based queues, can be seen in the diagram below.

This main Job queue structure effectively solves the first, second, and fifth problem posed in the Problem Statement.

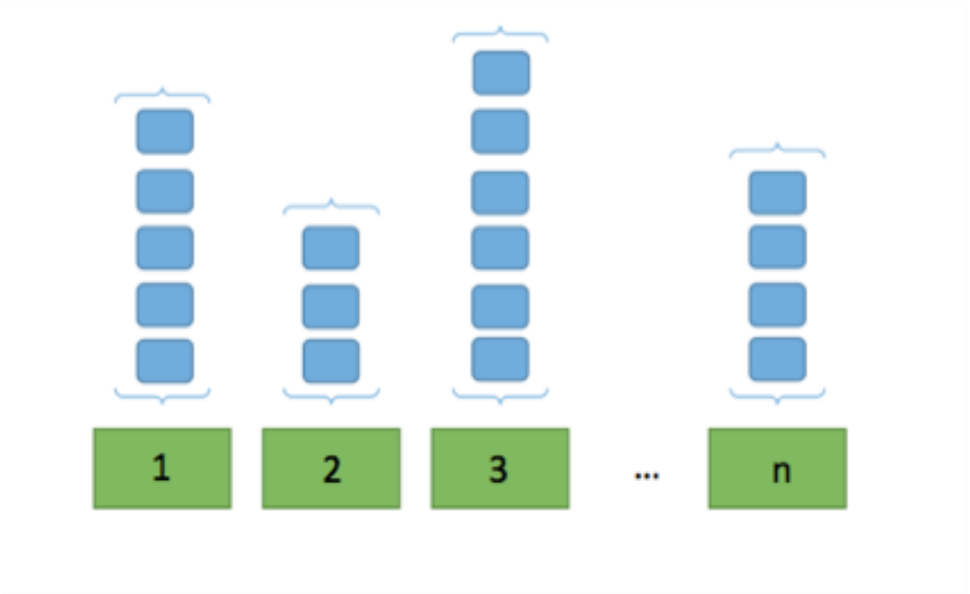


Figure 2 Job queue example

3.2 CLIENT DESCRIPTION

Within the Android client, running the class MainActivity starts the program. The MainActivity class is the main thread responsible for creating helper threads, displaying the GUI and listening to user input. As a general rule, no computationally intensive tasks, or blocking I/O should be performed in the MainActivity, as it will hinder the user experience. In fact, if Android senses that MainActivity is being overly delayed, the application will crash. As such, upon running the MainActivity, the first task is creating and running a new ClientThread.

Like its brother on the server-side (ServerThread), ClientThread is responsible for first creating any helpers necessary for the computation. It then listens for messages from the server, and relays the messages as necessary. Similarly to the ServerThread, ClientThread spawns three helper threads; Worker, Sender and Heart.

The Heart and Sender threads are analogous to their counterparts on the server-side. The Heart thread, is responsible for responding to server heartbeats, with their own heartbeat. Indicating the client is still connected. Likewise, the Sender is responsible for receiving Jobs from other threads, translating the Jobs (either work or a heartbeat) into a response, and sending them.

The Worker thread is complementary to its counterpart on the server side, the Boss thread. The Worker is responsible for actually computing the LU Decomposition. The Worker thread has a single queue of Jobs, `toDo`. The Worker spins in an infinite loop (until it is indicated to die), and pops Jobs from `toDo`. Once it has a Job, it calls the `decompose()` method on the Job. The method `decompose()` performs the actual LU Decomposition on the Job, and passes the result to Sender, to send back to the server.

In the same way the Boss queues were self-purging, the Worker's `toDo` queue is also self-purging. This completely addresses the fourth problem outlined in the problem statement, ensuring coherency. Upon adding a

Job to the `toDo` queue, the Job's iteration number is checked against the last iteration number. If the new iteration number is greater, a new iteration has begun, and all historical work should be removed. Likewise, if a Job has been completed, and the Sender notices the iteration number for that Job is out of date, the Job results will not be sent.

The Android client dispatches its threads similarly to its ServerThread counterpart. Upon receiving a message to disconnect, or being interrupted by one of the threads, ClientThread calls the `die()` method of each thread, changing the `killSelf` flag to true. Each thread will periodically check their `killSelf` flag, and respond accordingly. If any thread decides the Android client should disconnect (perhaps on receiving malformed input), the thread will interrupt the ClientThread, indicating the shutdown should commence.

For a high-level depiction of the interconnections, please see the following diagram. Note that the red lines represent virtual connections; the Sender communicates with the ClientThread or ServerThread, which relays the information as shown by the red lines.

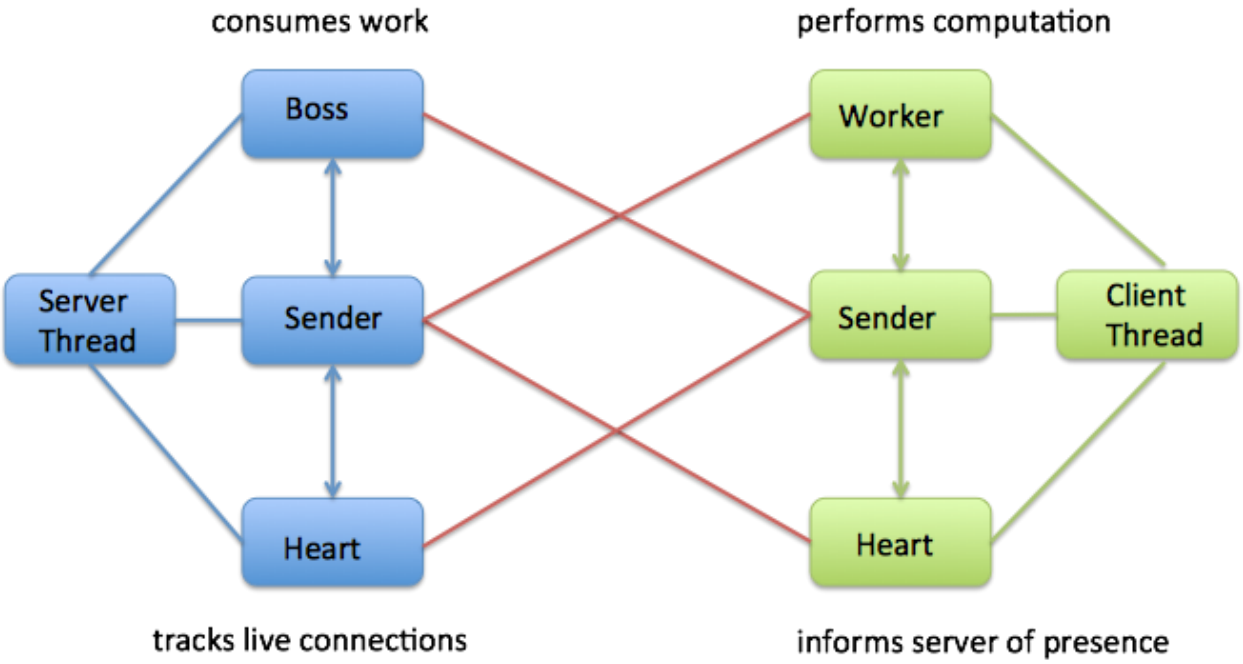


Figure 3 Client-Server interaction

4. CORRECTNESS

To test for correctness, two steps were performed. First, a serial version of the algorithm was developed and run on the server machine. The results of the decomposition are saved in a file. Next, the distributed algorithm is run on the same matrix, and similarly, the results are output to a file. The two files are then compared, equality ensures the consistency of the algorithm.

To test that the correct result was actually output (that is, testing that the serial algorithm is functional), the two matrices produced by the algorithm: the lower and the upper matrix, are first checked that they are indeed lower and upper matrices. The two matrices are then multiplied together ([L][U]). If the result of the multiplication is a matrix with the same rows as the original matrix (although the rows can be shuffled), then the algorithm is correct.

Thus, to test for correctness, the serial algorithm was first tested to ensure the outputs satisfy the axioms of an LU Decomposition, and the distributed algorithm was checked by comparing its results to the serial version.

5. PERFORMANCE

To test for performance, several steps were performed to ensure the testing conditions were identical for each run. To begin, all of the tests were performed after the cache of each phone was cleared and the power saving mode was disabled. This ensures no background processes, or power saving procedures consume processor time, thus obscuring the results. The tests were conducted on the 4th floor of the Trottier building using the same MacBookPro9.2 laptop to host the server. The server's window size was held constant for all tests. Likewise, for each of the tests the same matrices were used. Each test was performed ten times and the average time for each test was recorded. Further tests were not run, as separate trials yielded little to no deviations from the mean.

The tests were performed using two Google Nexus 5 phones and a single Samsung Galaxy S3 phone. The serial portion of the testing was performed by having each phone decompose the same matrix individually and sequentially. When using two phones, all of the possible permutations of phones decomposed the same matrix in parallel. Likewise, in the last case, all three phones were used to perform the LU decomposition.

6. RESULTS

The two graphs below illustrate the amount of time the LU decomposition took with respect to the size of the matrix and the number of phones used.

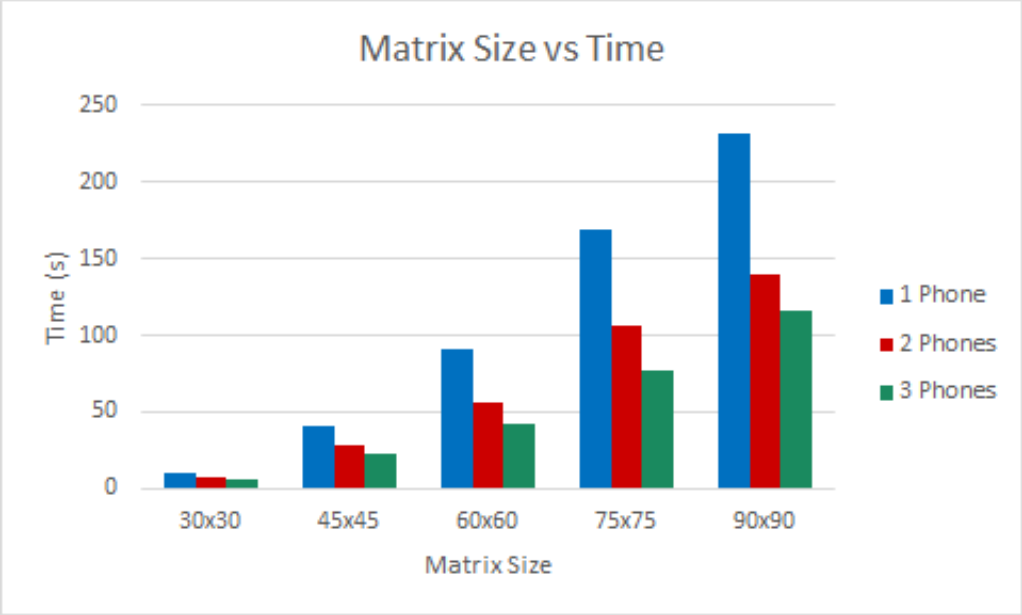
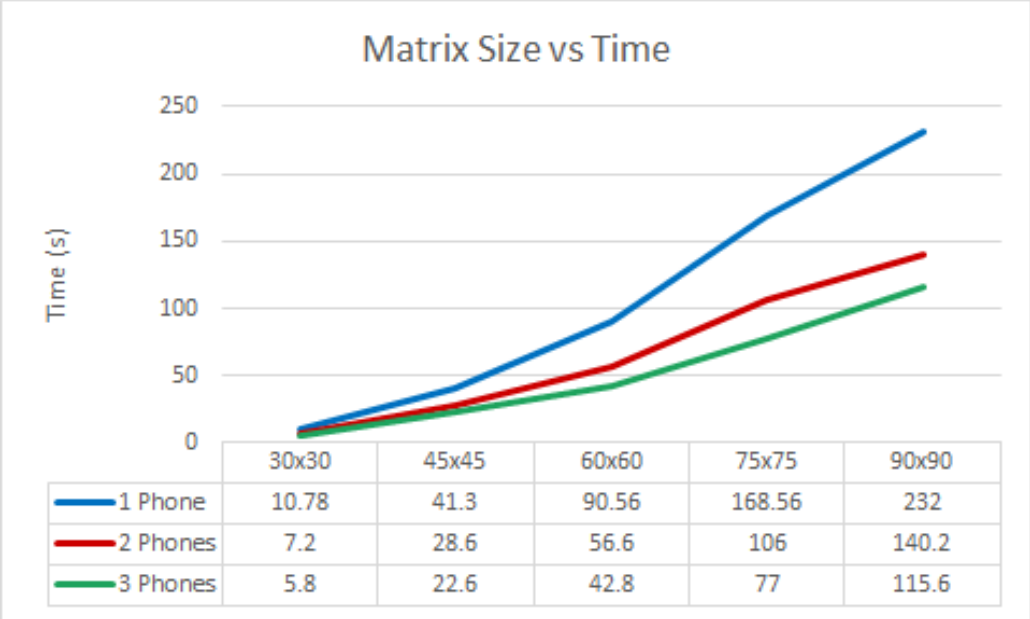


Figure 4 Matrix size compared to the time of execution

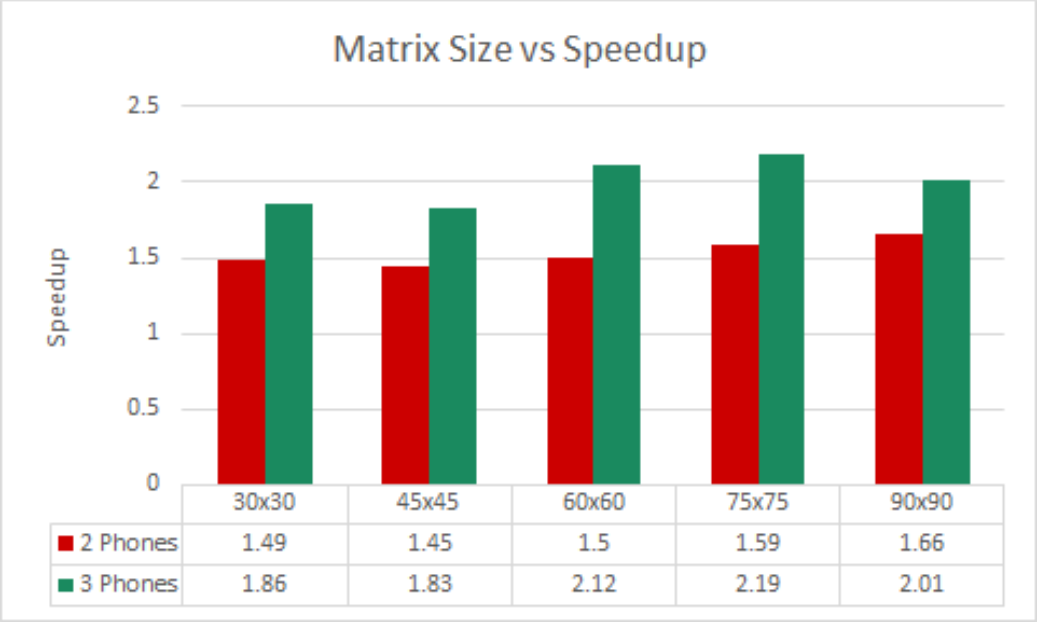


	30x30	45x45	60x60	75x75	90x90
1 Phone	10.78	41.3	90.56	168.56	232
2 Phones	7.2	28.6	56.6	106	140.2
3 Phones	5.8	22.6	42.8	77	115.6

Figure 5 Line graph comparing 3 phones' execution time

As expected, the time to perform the LU decomposition decreases as the size of the matrix increases and the number of phones increases. The parallelized version of the computation will continue to outperform the serial

version because the amount of parallelized work (Jobs) only increases with size of the matrix. However, when comparing the two parallelized versions, the difference in computation time increases from the 30x30 matrix until the 75x75 matrix where it begins to decrease. The decrease is much more apparent on the graph below that compares the matrix size to the speedup obtained.



	30x30	45x45	60x60	75x75	90x90
2 Phones	1.49	1.45	1.5	1.59	1.66
3 Phones	1.86	1.83	2.12	2.19	2.01

Figure 6 Speedup for each matrix

A possible reason for the decreased speedup in later tests, is that the window size remained constant. A larger window size would send more Jobs to the server's resources, which would in turn decrease the amount of communication overhead and allow the clients to spend more time performing computations. For smaller matrices, the window size must remain smaller in order to ensure that the computations are performed in parallel instead of one client performing most of the computation serially.

The results confirm the initial hypothesis that the speedup of the computation will be greater than 1 and will be directly related to the number of Android devices used to perform the computation.

7. FUTURE AND ON-GOING WORK

Although the Distributed Framework was successful in obtaining computational speedup, the framework is far from optimized. The group has suggested several extensions that could potentially increase the speed up. If time had permit, these would have been implemented.

7.1 CREATE CUSTOMIZED LINKEDLISTS

Within the framework, the Java LinkedList data structure was used (java.util.LinkedList<E>). Although these linked lists provide efficient implementations of most operations used, some operations were as slow as O(n). An example is the `remove(Object)` function. The Java implementation likely traverses the LinkedList until finding an occurrence of the object, this will be an O(n) operation. However, this operation is often used within critical sections (E.g. removing a Job from the main Job queue). Thus, it would be beneficial to reduce the time of the operation. One possibility of reducing this run time, would be to write a custom queue class, containing a linked list data structure to impose a FIFO ordering, while also having an array (or possibly hashmap) to track the exact location of each object in the linked list. Although this would require O(n) contiguous space, it could offer significant speed-ups.

7.2 DYNAMIC TIME-OUT PERIODS

Currently the timeout period for a heartbeat response is constant. There may be merit in making this timeout period dynamically set. It is suggested, that setting the timeout period should be done similarly to TCP's protocol. TCP uses an exponential moving average to predict the round trip time, and sets the timeout period to the predicted round trip time, plus a constant multiplied by the deviation (also calculated as an exponential moving average) [1]. A technique like this would allow the server to repossess resources more hastily, and would reduce the number of false disconnects.

7.3 INCREASING COHERENCY

With the current framework, client Job queues are only purged if out of date iterations are detected, it may be beneficial to also purge Jobs in the queue, which have sequence numbers that have already been computed. This would reduce the amount of redundant computation. One method of implementing this, would be to associate a queue of ServerThreads with each row of the matrix. Upon updating that row of the matrix (by getting possession of the Job lock associated with that row), all ServerThreads could be notified the computation has been complete. The ServerThreads could then purge their queues as necessary, and send a message to their ClientThread to also purge the results. This extra information could possibly be included in the heartbeat, or a new message type could be created to hold the information. Tests should however be run. It is possible that the extra communication overhead and the extra contention time may slow the computation.

7.4 VARYING WINDOW SIZE

Currently the window size for each Boss is constant. It would be beneficial if this also varied with the Android client. The window size is used to pipeline Jobs to the client. However, it is also desirable to keep the window size as small as possible, that still allows the pipelining to occur (the more Jobs sent out, the larger the chance of redundant computation). Thus, at the start of each iteration, the following could occur:

I. The Heart thread sends a heartbeat, and records the time taken to receive a response. Say this time is t_{RTT}.
II. The Boss thread sends a single Job, and records the time taken to receive a response. Say this time is $t_{RTT + COMP}$.
III. Calculate the time taken for a single computation: $t_{COMP} = t_{RTT + COMP} - t_{RTT}$.
IV. Using these values, set the window size. One possibility would be: window_size = $t_{RTT} / t_{COMP} + 1$, where the result is rounded up (See Appendix II for explanation).

7.5 TESTING SCALABILITY

Since physical devices were used to test the parallelism, it was impossible for the team to test the scalability of the framework. With only four group members, we only had access to three Android devices. Emulators could not be used, as real network communication times should be used. In extending this project, the framework's scalability should also be more rigorously tested.

7.6 ON SITE CACHING

As a final improvement, Android clients could be set to save the results they computed over the last iteration. This would reduce the number of Jobs to be sent. All the device would need, is the new pivot row. With that, it could compute the new rows based off of the cached values of the old rows. This could be highly efficient. A new message could be created to contain only the pivot row. Starting each iteration, every Boss would send this special message to the client. With the new pivot row the client would compute all cached rows (as long as the sequence number does not equal the new iteration number) and return the results. Upon re-computing all cached values, the client could then send a response to the special message, asking for new work. This would launch the ServerThread to function as it does without the cached results. This would complicate the process of Job stealing and coherency. As an example, what if a device does not finish re-computing the cached rows, before another iteration begins. It would have out of date rows that it may think are up to date. Despite the increase in complexity, and the possible coherency issues, using cached results should be further explored. As communication times encumber distributed computing, caching values could provide a significant bump in speedup. Another item of consideration for caching results, is the amount of memory each device possesses. Android devices may not have the storage space to cache all rows complete.

8. REFERENCES

1. Jim Kurose and Keith Ross, “Computer Networking, A Top-Down Approach”, 6th ed, Pearson, New Jersey, USA, 2013
2. David Culler, J.P. Singh, Anoop Gupta, “Parallel Computer Architecture: A Hardware/Software Approach”, Morgan Kaufmann Publishers, San Francisco, USA, 1998
3. Autar Kaw, “LU Decomposition”, [Online], http://mathforcollege.com/nm/mws/gen/04sle/mws_gen_sle_ppt_ludecomp.pdf
4. Weisstein, Eric W. "LU Decomposition." From *MathWorld*--A Wolfram Web Resource. http://mathworld.wolfram.com/LUDecomposition.html

9. APPENDIX

APPENDIX I: LU DECOMPOSITION

Lower Upper (LU) decomposition is a form of Gaussian elimination that factors a square matrix as the product of an upper triangular matrix and a lower triangular matrix. Thus:

$$A = LU$$

$$\begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$ [4]

Forward elimination will be used to find the Lower and Upper values of the matrix.

For the Lower:

$$l_{11} = \frac{a_{11}}{a_{11}} = 1$$

$$l_{21} = a_{21}/a_{11}$$

$$l_{31} = a_{31}/a_{11}$$

…

For the Upper, for the first iterations (to reduce first columns to 0):

$$U = A$$

$$u_2 = u_2 - u_1 * l_{21}$$

$$u_3 = u_3 - u_1 * l_{31}$$

For the next iterations:

$$u_3 = u_3 - u_2 * l_{32}$$

….

This will eventually lead to:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}.$$ [3]

APPENDIX II: SELECTING WINDOW SIZES

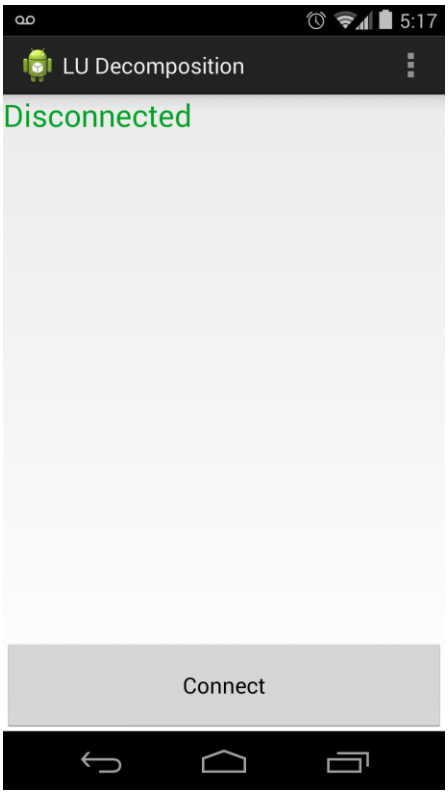
In the case where $t_{RTT} <= t_{COMP}$ the window size should be set to 2. This makes sense. Originally, two Jobs are sent. Upon receiving the two Jobs, t_{COMP} passes, and the first Job is complete and sent back to the server. It will take $t_{RTT} / 2$ time for the server to receive the Job, and a minimum addition of $t_{RTT} / 2$ for the client to receive a new Job (there may be an additional processing delay). Thus, after completing a single Job, the client will wait a minimum of $(t_{RTT} / 2) + (t_{RTT} / 2) = t_{RTT}$ **before** receiving a new Job. Since $t_{RTT} <= t_{COMP}$, the client will not have completed the second Job before a new Job is received. Thus, the client becomes saturated when the window size = 2.

In the case where $t_{RTT} > t_{COMP}$ things become more complicated. As mentioned above, after completing a Job, a client must wait a minimum of t_{RTT} before receiving a new job. Since $t_{RTT} > t_{COMP}$, we know that in the time the client will wait to receive new Jobs, it will have completed t_{RTT} / t_{COMP} Jobs. One solution would be to set $t_{RTT} = t_{COMP}$, however, this may be difficult to calculate as each Android device would have a different t_{RTT} and a different t_{COMP} so the Job size would be different for every device. This would pose serious problems with the current decomposition strategy.

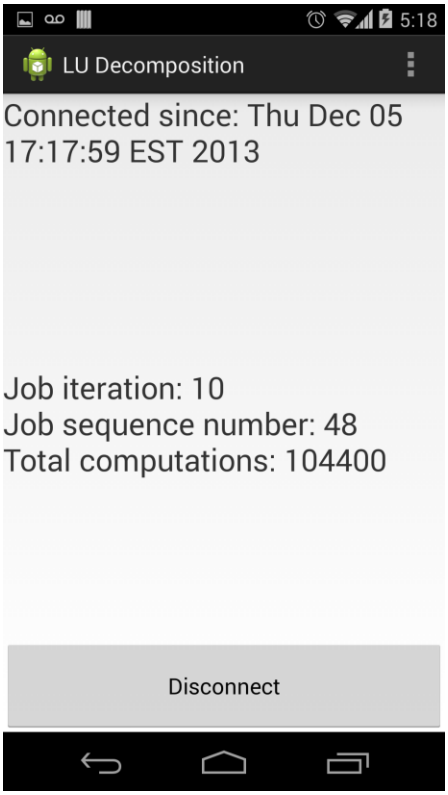
A better solution would be to simply set the window_size = $t_{RTT} / t_{COMP} + 1$. Say the client queue has $t_{RTT} / t_{COMP} + 1$ Jobs (as it would after receiving the first batch). It would complete the first Job after t_{COMP} and send the response, mark this as time t = 0. Thus, if no Jobs were to be received, the queue would be depleted at $t = t_{COMP} * (t_{RTT} / t_{COMP}) = t_{RTT}$. At $t = t_{COMP}$ the second Job would be complete, and sent, the first Job would still be propagating back to the server. At $t = t_{RTT} / 2$, the first Job would have reached the server, and a new Job would be sent to the client. At $t = t_{COMP} + t_{RTT} / 2$, the second Job would have reached the server, and a new Job would have been sent, the response to the first Job would still be propagating towards the client. At $t = t_{RTT}$ the client Job queue would have depleted (as mentioned above), and a new Job would be received (in response to the very first Job being completed). At $t = t_{RTT} + t_{COMP}$ the Job queue would again be depleted, and again, a new Job would be received (in response to the second Job being complete). Thus, the system would enter a steady state, and never be idle. This again minimizes the window size, while ensuring saturation.

In fact, this window size works in the case where $t_{RTT} <= t_{COMP}$. In this case, $0 <= t_{RTT} / t_{COMP} <= 1$. If $t_{RTT} > 0$ taking the ceiling of t_{RTT} / t_{COMP} would yield a window size of 2, as expected. Similarly, if $t_{RTT} = 0$, the window size would be 1, as expected.

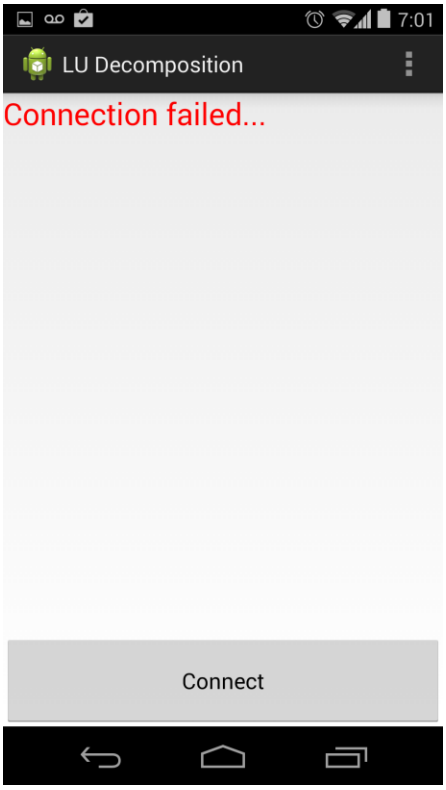
APPENDIX III: GRAPHICAL USER INTERFACE



The LU Decomposition App before it has attempted to connect to the server.



The LU Decomposition App while it's decomposing a matrix.



The LU Decomposition app it failed to connect to the server.