DSS Coursework 2 - Time and Communication

Alhamza Alnaimi, Ben Steer

Initial Plan

Upon starting this project, it was decided the best way to accurately gather data, would be to build a full simulation of the problem using threads to represent different processes. These would each have their own ‘timeline’ where all events and communications would be stored. These timelines could then be analysed to gather all crossing paths and build a list of the < pairs that existed.

It was decided to split the project into five Objectives. The first objective was to build all of the processes, events and communications in a manner where they could be easily analysed. The second objective was to split the events and communications into resource pools of random sizes, allocating one of each to the processes. The third was to run the processes, having them concurrently add events/communications to their timelines, and ‘communication received’ events to the receiver processes timeline. The fourth objective was to build a map through the produced timelines, allowing generation of the < list. The final objective was to calculate the total number of pairs, via $C(n_e+n_c,2)$, number of || pairs, via subtracting the < pairs from total, and output these to a file. The simulation would then be run multiple times, with a varying number of events/communications/processes. This would then be read into R where the iterations would be averaged, the different test sets compared, and graphs of the results created. With this plan in mind, the implementation was begun.

Implementation

Initial Component build

As the best way to analyse the timelines once they had been filled in was not clear, shells of all the components believed to be needed were created, and added to as the project progressed.

There were four main classes that were originally created in order to run the simulation. The first of these was the Event class; this initially contained two integers, denoting the event ID (unique amongst all events and communications), and the number of the process it had been assigned to.

The second was the communication class; this extended the Event class and contained two extra variables. An event object (the ‘communication received’ event described above) and an integer pointing to the process which the communication was intended for.

The third was the process Class. Processes would be the threads used to concurrently change the timelines. These contained an integer (the processID), an ArrayList of Events (the allocated pool of events), an ArrayList of Communications (the allocated pool of communications), and an ArrayList of ArrayLists of Events (used to represent the timelines for each process). The run method was overwritten, and a while true loop was added. Within this loop, events and communications were randomly chosen from the pools until both were empty and the loop would break. When an event was picked, the processes ‘timeline’ would be retrieved from the ArrayList of ArrayLists (passing the process ID to the get method), the chosen event would then be added to the ‘timeline’ and removed from event pool (to make sure it wasn’t selected twice). If a communication was picked, all of the above would happen, but additionally, the internal event would be added to the ‘timeline’ of the receiving process ID.

The fourth component was the pair class; this was to be used to represent the < pairs, and make sure that no doubles were being counted in the analysis. It contained two integers, first and second, which stored the two numbers in the pair. The equals method was then overwritten to check if first and second of the passed pair was the same as the stored values. This was done so that when a list of pairs was created, any duplicates could be discovered and removed.

Once these Classes had been created, the framework of the main method was started, to both get started with the other objectives, and to discover what additional features the above classes would require to complete the simulation.

Simulation FrameWork

The first thing that had to be done within the main method was set the number of events/communications/processes. This was to be done by the user via args, but was initially set at the start of the method.

Next was to create all of the lists, and fill them with their initial state. Two ArrayLists of ArrayLists of Events were created (one for the event pools, on for the timelines) , One Arraylist of ArrayLists of Communication (for the communication pools) and an ArrayList of Processes. New instances were then added to all of these, given by the number of processes.

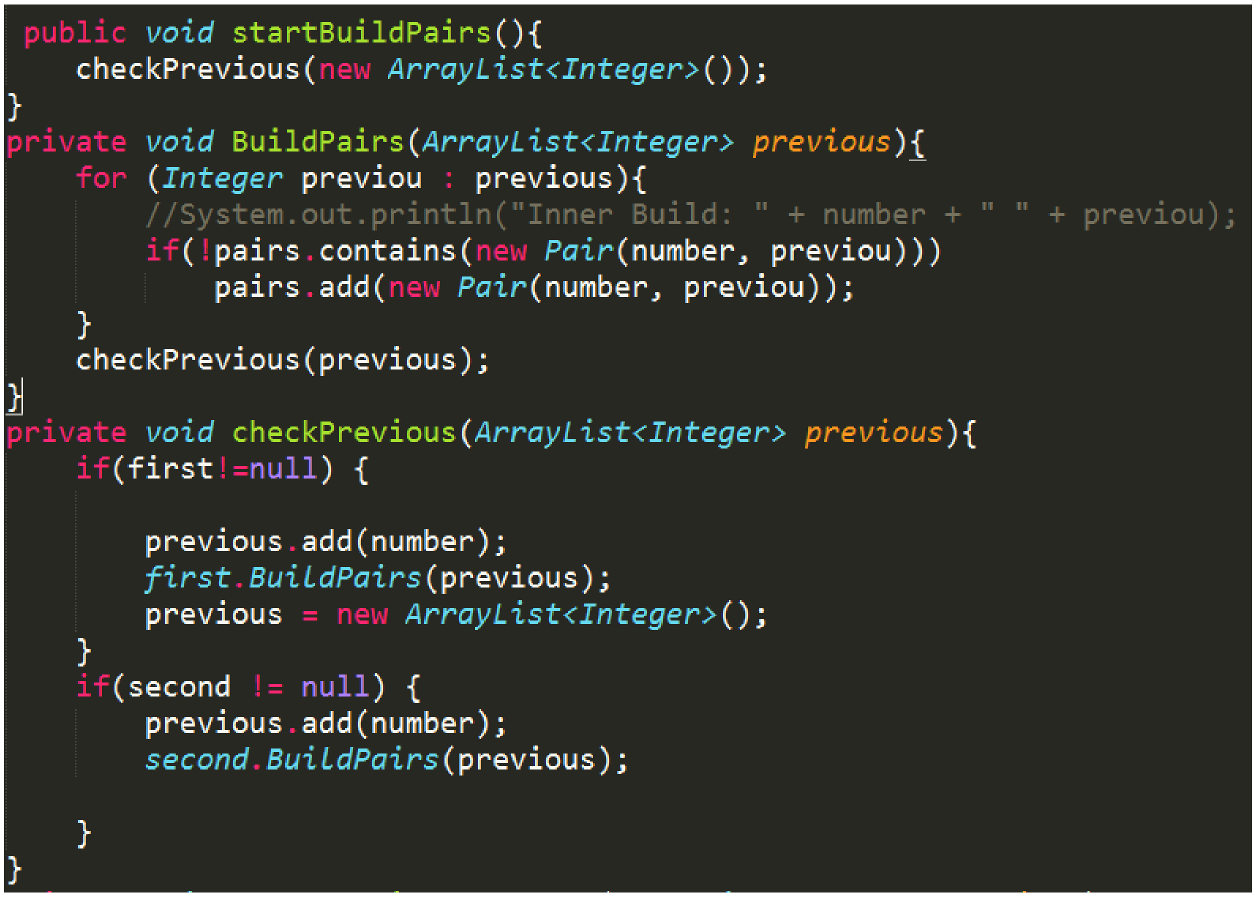
Next, for the number of events specified, a process was chosen at random and had a new event added to its pool (passing a global event counter variable as the event id). The event counter was then incremented. A similar process was carried out for communications, the only difference being that the internal event was also created causing the event counter to be incremented twice per communication.

Now that all the Lists, pools and processes had been created, the processes were started. It was realised that some sort of notification would be required to allow the main to know when all the threads were finished running. A finished boolean was added to the process class, which was set to true at the end of the run method. In the main, all of the processes would be check in a loop to see if they were finished, if they had, the loop would break and programme would start the analysis of the timelines.

Timeline Analysis

At first, it was not clear how we were going to check all of the paths that could be taken through the timelines, due to the sheer complexity once there are more than 10 events/communications. It was realised that the best way to do it would be a recursive algorithm that started at the end node of each timeline and worked its way backwards. To do this however, some changes would have to be made to the Event class.

Two event pointers were added to the class, called first and second, and initialised to null. When an event is pulled from the pool and placed into a timeline, the previous event in that timeline is set to that events first pointer. If the event is added to a timeline via a communication, that communication is set to the the events second pointer. Once this was completed, the following methods were created:



```
 public void startBuildPairs(){
    checkPrevious(new ArrayList<Integer>());
}
private void BuildPairs(ArrayList<Integer> previous){
    for (Integer previou : previous){
        //System.out.println("Inner Build: " + number + " " + previou);
        if(!pairs.contains(new Pair(number, previou)))
            pairs.add(new Pair(number, previou));
    }
    checkPrevious(previous);
}
private void checkPrevious(ArrayList<Integer> previous){
    if(first!=null) {

        previous.add(number);
        first.BuildPairs(previous);
        previous = new ArrayList<Integer>();
    }
    if(second != null) {
        previous.add(number);
        second.BuildPairs(previous);

    }
}
```

When the analysis was to be started, startBuildPairs would be called on the last event in every timeline. This would create a new arraylist of integers (to store all of the event id's the recursive calls passed through) and call the checkPrevious method. This method would check if the first and second pointers were null. If they were, it would mean that that path was dead, if not, the events ID would be added to the arraylist, and buildPairs would be called on the previous event (either first or second). Finally in this method the pairs were created, using the id of the current event for the first number and all the numbers in the arraylist for the second numbers. The contains method was used to make sure that the pairs did not already exist within the list. The checkPrevious method would then be called again to continue the process till the front of the timelines were reached. Once all paths reached a null pointer, the process would be over, and the pairs arraylist would contain all < pairs within the model.

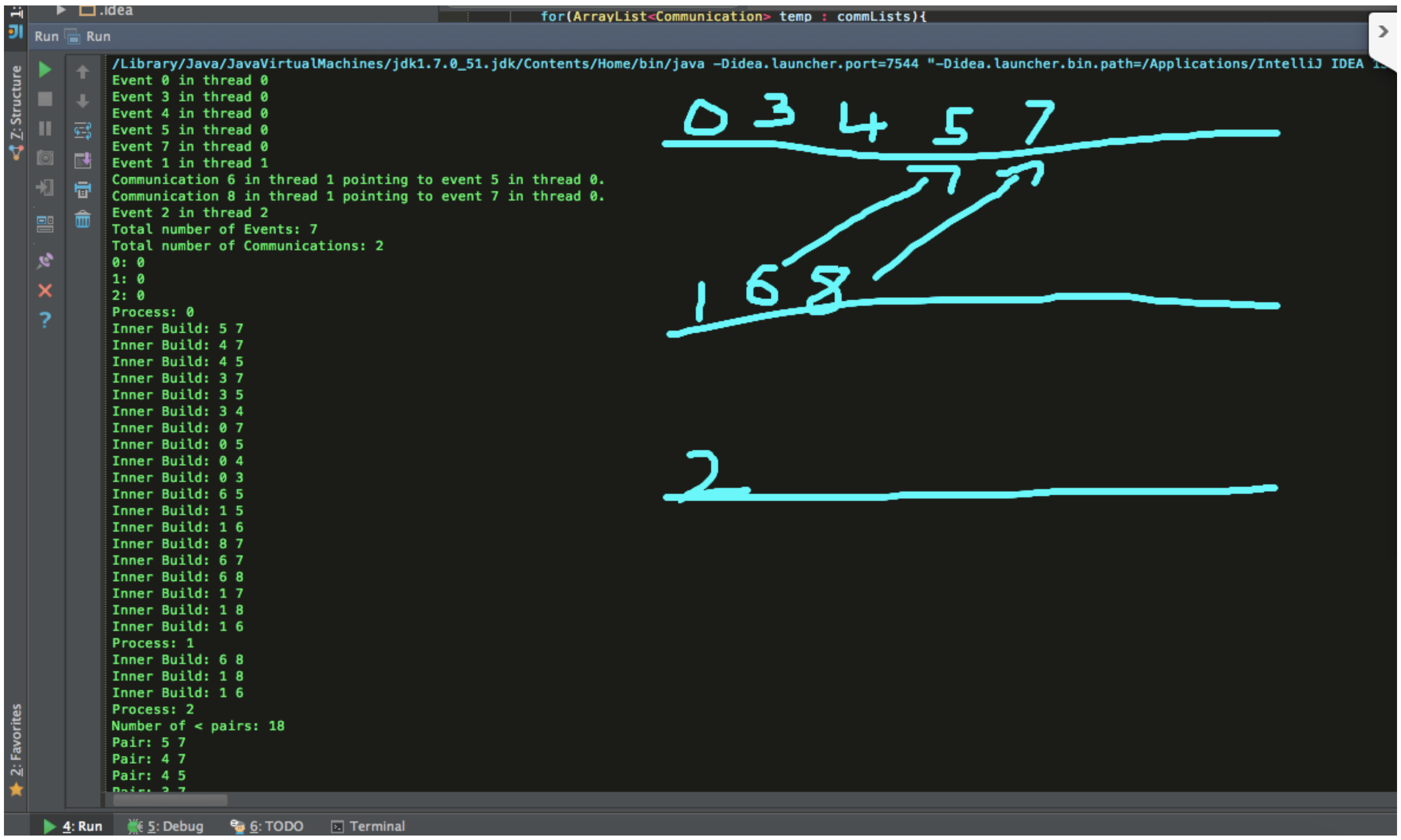
To finalise, the total possible pairs and number of || pairs was calculated:

```
int totalEvents = eventTotal+commTotal;
int nonPairs = (totalEvents * (totalEvents-1))/2;
nonPairs=nonPairs-pairs.size();
```

This could then be written out to a CSV file to be read into R.

Testing and Finalising

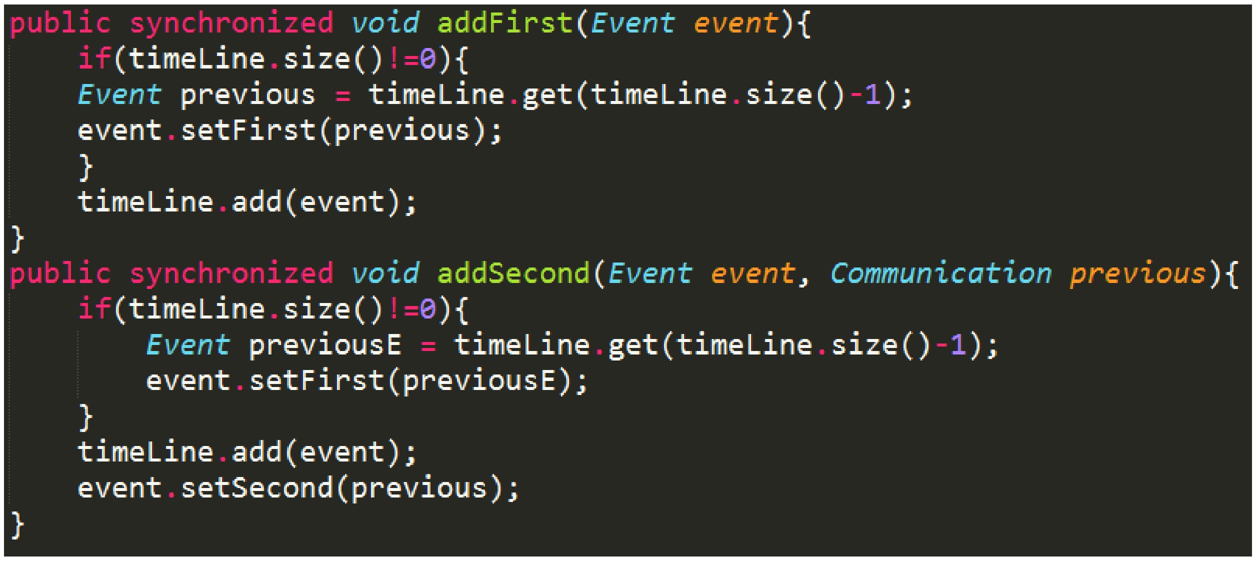
Before the simulation could be run, it had to be checked to make sure that the right output was being generated. The simulation was run with low numbers of events and communications (so that a human could work it out) and dry run, for example:



```
for(ArrayList<Communication> temp : commLists){
```

```
/Library/Java/JavaVirtualMachines/jdk1.7.0_51.jdk/Contents/Home/bin/java -Didea.launcher.port=7544 "-Didea.launcher.bin.path=/Applications/IntelliJ IDEA
Event 0 in thread 0
Event 3 in thread 0
Event 4 in thread 0
Event 5 in thread 0
Event 7 in thread 0
Event 1 in thread 1
Communication 6 in thread 1 pointing to event 5 in thread 0.
Communication 8 in thread 1 pointing to event 7 in thread 0.
Event 2 in thread 2
Total number of Events: 7
Total number of Communications: 2
0: 0
1: 0
2: 0
Process: 0
Inner Build: 5 7
Inner Build: 4 7
Inner Build: 4 5
Inner Build: 3 7
Inner Build: 3 5
Inner Build: 3 4
Inner Build: 0 7
Inner Build: 0 5
Inner Build: 0 4
Inner Build: 0 3
Inner Build: 6 5
Inner Build: 1 5
Inner Build: 1 6
Inner Build: 8 7
Inner Build: 6 7
Inner Build: 6 8
Inner Build: 1 7
Inner Build: 1 8
Inner Build: 1 6
Process: 1
Inner Build: 6 8
Inner Build: 1 8
Inner Build: 1 6
Process: 2
Number of < pairs: 18
Pair: 5 7
Pair: 4 7
Pair: 4 5
Pair: 3 7
```

The first attempts at running the simulation lead to some strange intermittent errors being thrown from the timelines, namely, index out of bounds and null pointers. As only indices between 0 and the size of the arraylists were being accessed, and none of the contents were ever set to null, this was quite perplexing. It was realised, that as the timelines were being accessed by the different processes concurrently, and as arraylists aren't thread safe by default, some of our events were being overwritten, causing memory errors, and the size of the array to be incorrect. To fix this issue a Timeline Class was created. This was basically a wrapper for the Arraylist of events, encasing all the methods used with a synchronized equivalent. This did mean, however, that the pointers within the events had to be set within the same synchronized method, to make sure that another process didn't slip an event in-between method calls. The addFirst and addSecond methods were created:



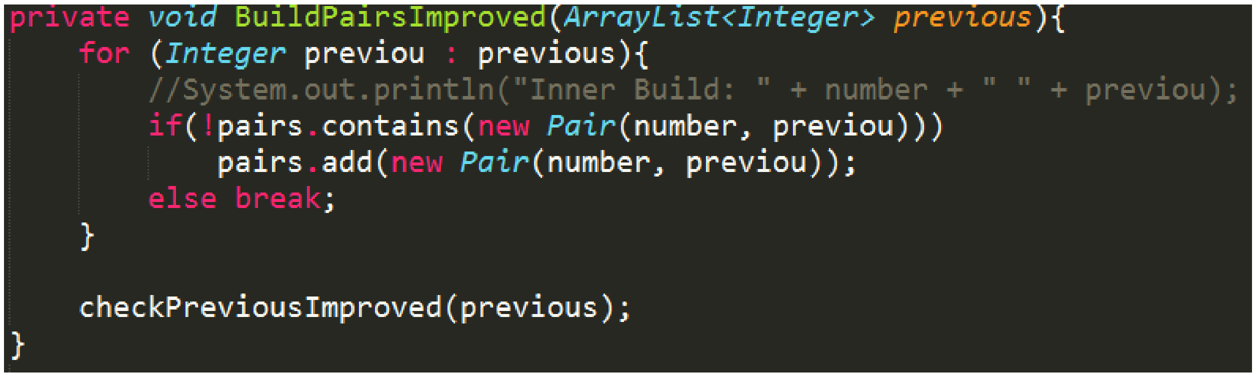
```
public synchronized void addFirst(Event event){
    if(timeLine.size()!=0){
    Event previous = timeLine.get(timeLine.size()-1);
    event.setFirst(previous);
    }
    timeLine.add(event);
}
public synchronized void addSecond(Event event, Communication previous){
    if(timeLine.size()!=0){
        Event previousE = timeLine.get(timeLine.size()-1);
        event.setFirst(previousE);
    }
    timeLine.add(event);
    event.setSecond(previous);
}
```

addFirst takes the event to be added to the timeline, sets the first pointer correctly and then adds it. addSecond does the same, but additionally takes a communication, which is then set to the events second pointer.

The exceptions stopped being thrown, and results for the simulation were being generated. However, It was discovered, through dry running, that several runs produced results that were slightly larger than they should have been. This was caused by the arraylist of integers not being reset in the checkPrevious method when returning from the recursive calls inside the first event. The list would return, still containing all of its numbers, and then be passed to the second event, meaning pairs that did not exist in the simulation were being generated and added to the list. This was easily fixed by setting the arraylist to a new instance if it returns from the first event. Once this had been implemented our simulation produced correct results every time and was felt to be ready for the second part of coursework.

Post Test Changes

After some time testing, it was agreed that the programme was not working too well at high numbers of communications. This was originally thought to be caused by the amount of times the paths would run over same pairs, spending time creating objects that would just be dropped. To attempt to remedy this a break case was added to the BuildPairs method:



```java
private void BuildPairsImproved(ArrayList<Integer> previous){
    for (Integer previou : previous){
        //System.out.println("Inner Build: " + number + " " + previou);
        if(!pairs.contains(new Pair(number, previou)))
            pairs.add(new Pair(number, previou));
        else break;
    }

    checkPreviousImproved(previous);
}
```

This meant that whenever a path began creating duplicates, it could stop and return. Benchmarking the two algorithms together showed an improvement, but only a change of about 600 milliseconds over the course of a minute. This was seen as negligible, but used in the final version to make some difference over the 100 iterations per dataset. Unfortunately, the sheer number of paths to be checked was the cause of the problem, and nothing could be done to change this.

Test Plan

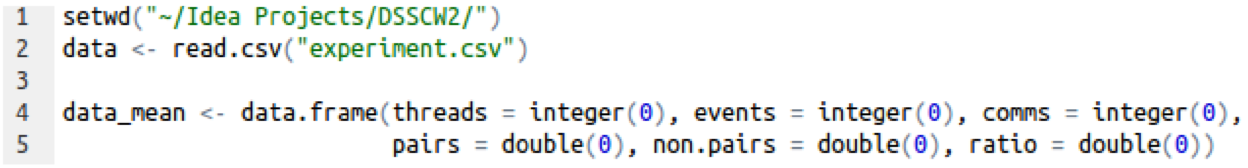
Experiment 1

In order to test the data produced. The simulation was run with different test conditions. The number of processes, events as well as communications would differ with each run. This was done in the following fashion:

```
Process:  3, Events: 5, Communications: 2
Where the number of communications would range from the values 2, 5 and 10 (anything beyond
was too slow with the recursive implementation).
The number of events would then range from 5, 10, 100, 250, 500, and 1000.
And finally the number of processes would range from 3, 5 and 10. This created 72 different
combinations.
```

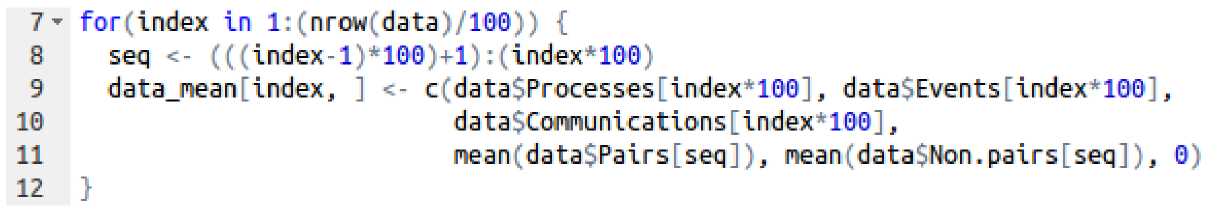
Each setup was ran 100 times, as to obtain an average value of a given sequence. The results were then saved as a csv file (Table structure) with the headers being the string literals; “Processes”, “Events”, “Communications”, “Pairs”, “Non.pairs”

This data was then read into R, in order to obtain the mean average from the 7200 lines test data, and plot the results to find the relations between them.



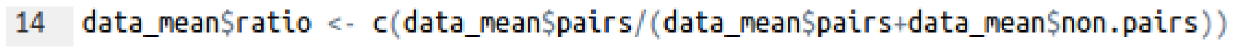
```r
setwd("~/Idea Projects/DSSCW2/")
data <- read.csv("experiment.csv")

data_mean <- data.frame(threads = integer(0), events = integer(0), comms = integer(0),
                        pairs = double(0), non.pairs = double(0), ratio = double(0))
```

The results were saved into the variable data. And an empty data frame (table type) was created, to later hold the mean averages.
In order to calculate the mean, a for loop was created, that runs 72 times (the number of rows in the original data frame (7200) divided by 100 (average iterations)). A sequence which starts at 1:100 the first time around and finishes at 7101:7200 was then initialized, as to represent the range of values for each setup. Once the mean was calculated, the data frame was then populated appropriately.



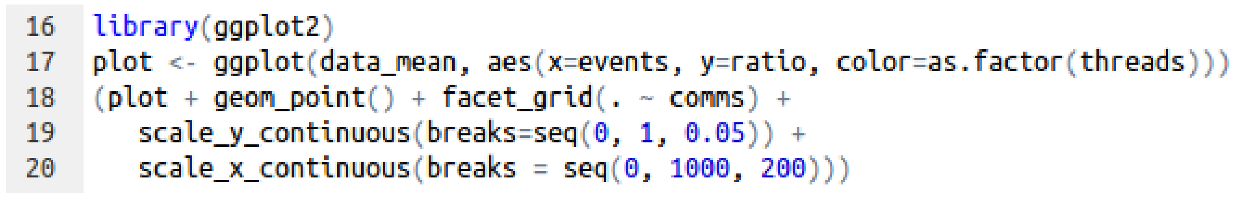
```r
for(index in 1:(nrow(data)/100)) {
  seq <- (((index-1)*100)+1):(index*100)
  data_mean[index, ] <- c(data$Processes[index*100], data$Events[index*100],
                          data$Communications[index*100],
                          mean(data$Pairs[seq]), mean(data$Non.pairs[seq]), 0)
}
```

From the mean it was then simply to deduce the ratio between the number of pairs to the number of overall combinations.



```r
data_mean$ratio <- c(data_mean$pairs/(data_mean$pairs+data_mean$non.pairs))
```

Next the data had to be plotted. Below code creates a plot with the y values representing the ratio and x values representing the number of events. The entire data set was then separated into three different graphs, with each representing the number of communications (3, 5 or 10). The data points in each graph was then assigned a color, in order to represent number of process used for the setup (3, 5, 10 or 100). This would then represent what essentially is 4D data in 2 dimensions.. Finally the x and y gap ratio had to be scaled manually. In order to be able to do all of this; the library ggplot2 was used.



```r
library(ggplot2)
plot <- ggplot(data_mean, aes(x=events, y=ratio, color=as.factor(threads)))
(plot + geom_point() + facet_grid(. ~ comms) +
   scale_y_continuous(breaks=seq(0, 1, 0.05)) +
   scale_x_continuous(breaks = seq(0, 1000, 200)))
```

The final results of this experiment can be seen under the title Results.

Experiment 2

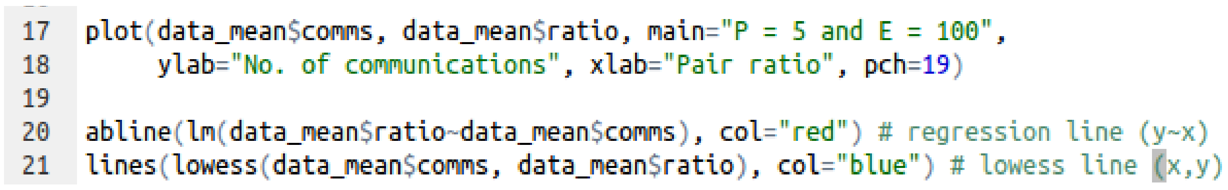
The second experiment conducted was to have a fixed number of threads and events, only ever increasing the number of communications. (Such that with 5 threads, 100 events and 20 communications; 140 events would be created, 20 of which are a communication, 20 more as a receiving event, and 100 as normal events). This setup was run with;

```
Process:  5, Events: 100, Communications: X
```

Where the number of communications would be, 5, 10, 25, 50, and 100 for each run. Similarly, 100 iterations were conducted for this setup in order to better obtain an average value.

Once the results for above experiment was collected, an average mean was calculated in a similar fashion.

Then the results were then plotted, confirming the theory behind the results of experiment 1.

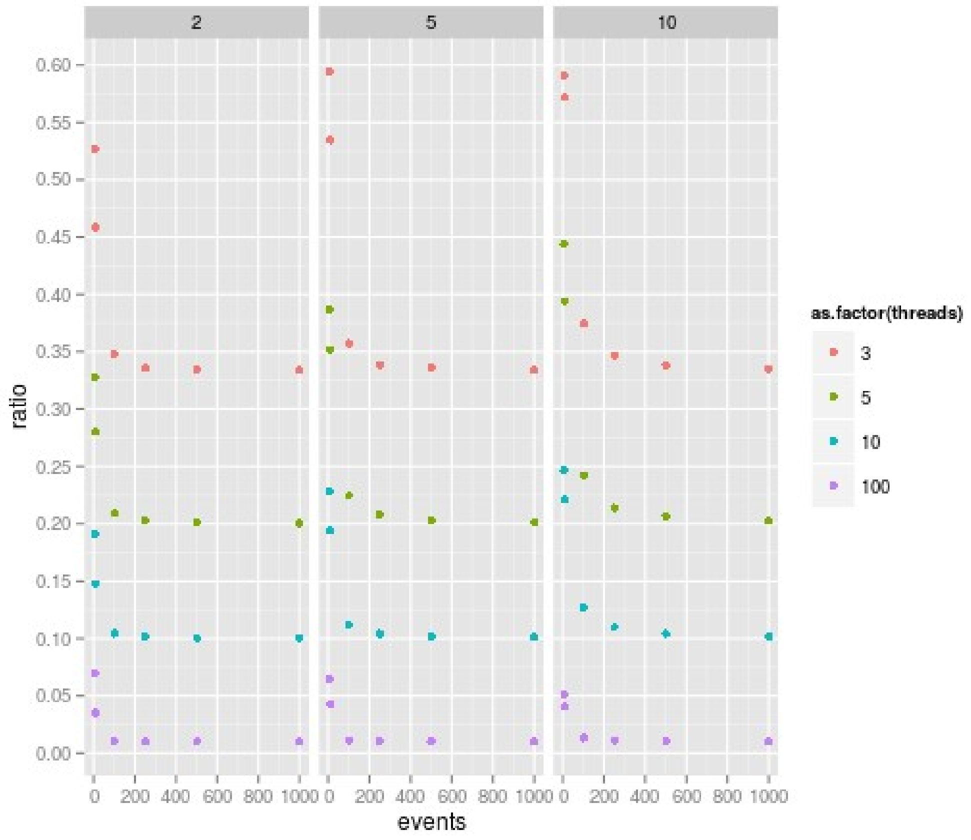


```
plot(data_mean$comms, data_mean$ratio, main="P = 5 and E = 100",
     ylab="No. of communications", xlab="Pair ratio", pch=19)

abline(lm(data_mean$ratio~data_mean$comms), col="red") # regression line (y~x)
lines(lowess(data_mean$comms, data_mean$ratio), col="blue") # lowess line (x,y)
```

The above plots the number of communications on x axis and the ratio at y axis. A linear model is then fitted to the data, with an exponential model, to showcase how the number of communications affect the ratio.

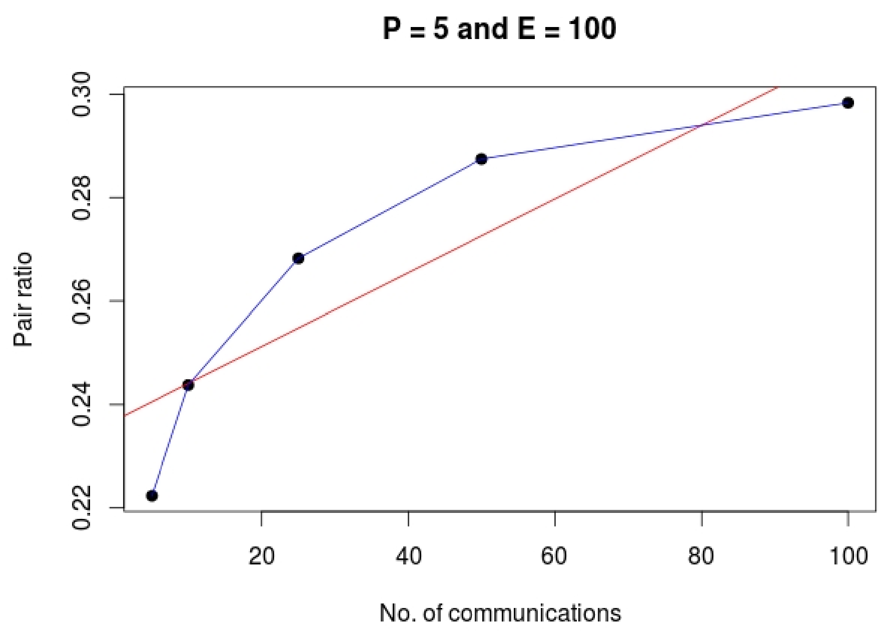
Results

Experiment 1



It can be seen from the above graph that the ratio between pairs and total possible combinations change very little between different number of events where process thread is same. The number of ratio seem to be higher the smaller amount of threads. This is obvious, as the lower amount of threads the more events there are on each (less sparsity). However, with the number of communications we can see what resembles a linear or exponential increase, affecting the ratio of pairs to total combinations. The next experiment conducted was to prove this point.

Experiment 2

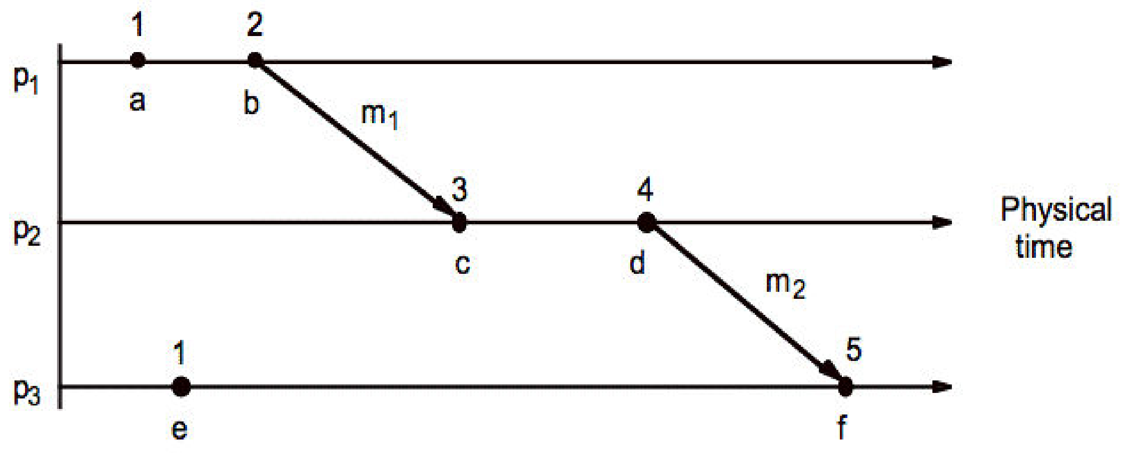


The above graph displays how the number of communications seem to affect the ratio of pairs to the total number of combinations possible. It is however not easy to tell whether if the ratio is being increased exponentially as per shown with the blue line or whether it is simply over fitting the data. An attempt was made to test with 250 communications, however the simulation, running for 8 hours, simply never finished and was later terminated prematurely.

Next was to attempt to create code that simulates logical time, quicker than what currently is being done. After one of the distributed systems lecture and a talk with the professor, it was suggested to attempt a transitive closure implementation with matrix multiplication. However once that was explained, an “easier” plan was hatched. The idea was still based on transitive relation; where we know that if pair X s a communication with event Y, and event Z is a pair with Y, then X must also be a pair with Z.

Any thread would have its *local* number of pairs calculated with the formula of n(n - 1) / 2 which is number of connections from 1 to n.

The number of pairs between a communication would then be given by the number of events previous to the current communication, and the number of events after the receiving communication. This is demonstrated as below:



P_1's pairs can be given by the formula n(n - 1)/2 which is equal to 1. Like wise P_2's and P_3's number of pairs can be given by formula, which is also equal to 1.
Contrary, the number of pairs between the communication b → c can be given by looking as the entire thing as a fourth process, say P_4 whose number of pairs is given by n (as to not include duplicates). In the above, this is given by (1 + number of events behind b) + (1 + number of events ahead of c), which is equal to 4. Following this trend, the number of pairs in the above logical graph is equal to 10. This means that the count of pairs no longer requires matrix multiplication, nor any recursive calls, rather is but simple math additions.

In order to implement this in code, during the random distribution of events on each thread, an event is created, where the belonging threads number and its position on said thread is passed along as an AbstractMap.SimpleEntry, key and value. If the chosen event is randomly chosen as a communication, then the second event of which it is paired with, is set in the method setPairWith, which also accepts an AbstractMap.SimpleEntry representing the key value of the pair's process thread and its position on said thread.

The number of pairs on each independent thread can be given by the for loop:

```
for(int i = 0; i < processCount; i++) {
   pairs.add((events.get(i).size() * (events.get(i).size() - 1)) / 2);
}
```

Where pairs is an arraylist that contains the number of pairs for each process thread given at index i.

For the number of pairs between communications, it is a bit more complex. First we have to iterate through each event at given thread, and check if its boolean value isPair equals to true. If it is, then the numbers, startLink and endLink are calculated as explained above, and finally added to pairs arraylist at index j.

```
for(int j = 0; j < processCount; j++) {
   ArrayList<Event> temp = events.get(j);
   for (Event event : temp) {
       if (event.isPair) {
           int startLink, endLink;
           // going down
           startLink = event.getBelongsTo().getValue() + 1;
           endLink = events.get(event.getPairWith().getKey()).size() -
event.getPairWith().getValue();

           int linkSize = startLink + endLink;
           int linkedPairs = linkSize-1;
           pairs.set(j, pairs.get(j) + linkedPairs);
       }
   }
}
```

And finally the ratio can then be given by:

```
int nonPairs = (eventCount * (eventCount - 1)) / 2;
nonPairs -= totalPairs;

System.out.println("There are " + totalPairs + " total pairs.");
System.out.println("Where the ratio is: " + totalPairs/(totalPairs + nonPairs) + ".");
```