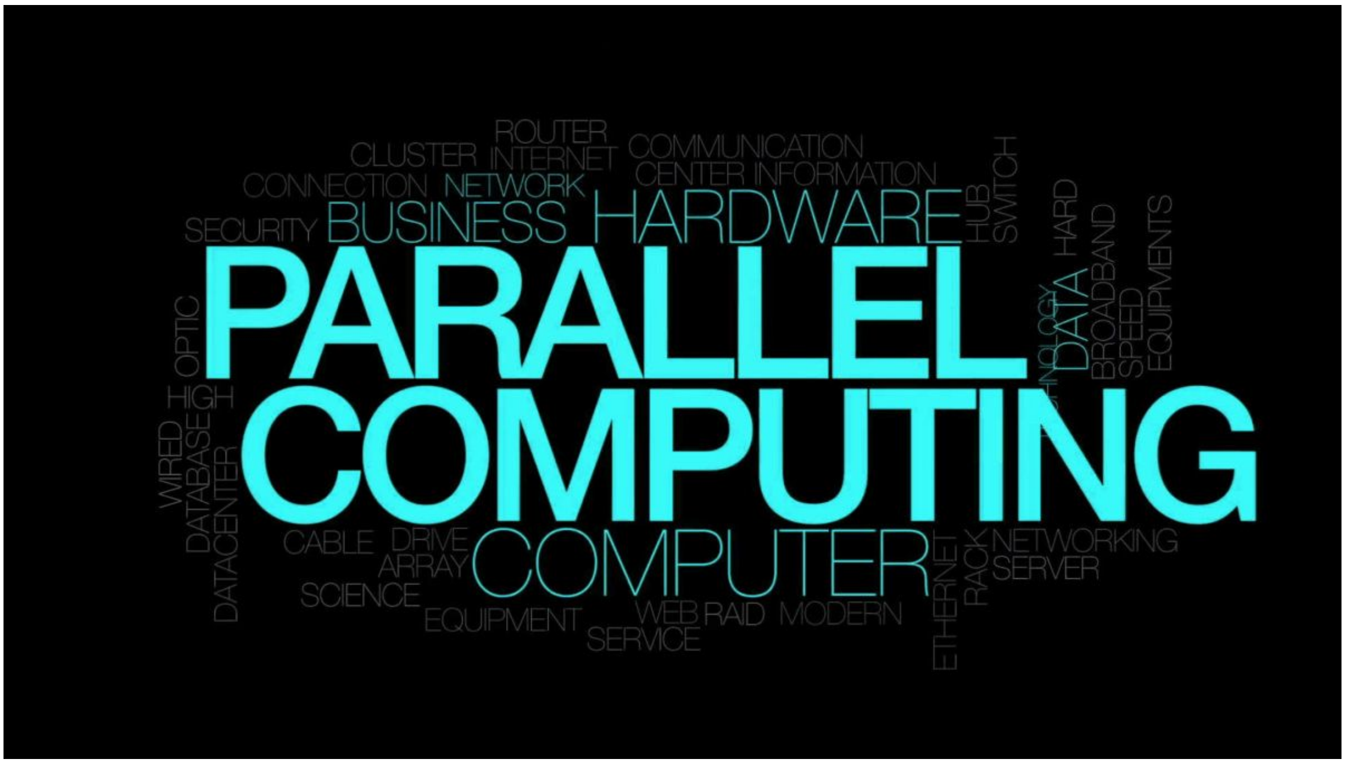
Parallel Computing

Version 7.0, Parallel Computing.
Date 16-06-2019



THE FIFTH GENERATION COMPUTER THINKING IT OVER
Now that they're gone, I'll get to work on the sixth generation
S. harris

Mats Grobben, 500714964, Mats.Grobben@hva.nl
Kostis Thanos, 500739944, Kostis.Thanos@hva.nl

Version control

Tabel with version changes.

Version	Date	Changes to the document
1.0	24-04-2019	Create the first assignment and pick a algorithm which we will use in the coming assignments.
2.0	26-04-2019	Improve the first assignment based on receiving feedback.
3.0	28-04-2019	Create the second assignment.
4.0	13-05-2019	Work on the second assignment.
5.0	25-05-2019	Create the second approach of the second assignment.
6.0	05-06-2019	Worked on assignment three and started on the CSP and RMI implementation.
7.0	08-06-2019	Creating the documentation for assignment four.
8.0	17-06-2019	Worked on the last assignment, finishing touches to the document before delivery

Table of contents

Introduction

This document contains our elaboration of the Parallel Computing assignment.
This assignment is done for the subject Parallel Computing as practical assignment, commissioned by the HvA. In this assignment we will choose three different algorithms which we will explain in terms of input, output, serial solutions for it and parallelization of it.

1. Case studies

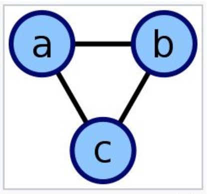
1.1 Dijkstra's Algorithm

The input is a set of nodes. The output is the shortest path combination.

It can also be used for finding the shortest paths from a single node to a single destination node by stopping the algorithm once the shortest path to the destination node has been determined. The steps of finding the shortest path are as following:

1. Mark all nodes unvisited. Create a set of all the unvisited nodes called the *unvisited set*.
2. Assign to every node a tentative distance value: set it to zero for our initial node and to infinity for all other nodes. Set the initial node as current.
3. For the current node, consider all of its unvisited neighbors and calculate their *tentative* distances through the current node. Compare the newly calculated *tentative* distance to the current assigned value and assign the smaller one. For example, if the current node *A* is marked with a distance of 6, and the edge connecting it with a neighbor *B* has length 2, then the distance to *B* through *A* will be 6 + 2 = 8. If B was previously marked with a distance greater than 8 then change it to 8. Otherwise, keep the current value.
4. When we are done considering all of the neighbors of the current node, mark the current node as visited and remove it from the *unvisited set*. A visited node will never be checked again.
5. Move to the next unvisited node with the smallest tentative distances and repeat the above steps which check neighbors and mark visited.
6. If the destination node has been marked visited (when planning a route between two specific nodes) or if the smallest tentative distance among the nodes in the *unvisited set* is infinity (when planning a complete traversal; occurs when there is no connection between the initial node and remaining unvisited nodes), then stop. The algorithm has finished.
7. Otherwise, select the unvisited node that is marked with the smallest tentative distance, set it as the new "current node", and go back to step 3.

Our implementation of Dijkstra's Algorithm will make use of Adjacency list.
An adjacency list is a collection of unordered lists used to represent a finite graph. Each list describes the set of neighbors of a vertex in the graph. An example can be seen on the right.



This undirected cyclic graph can be described by the three unordered lists {b, c}, {a, c}, {a, b}.

The implementation that we use will also be using a Priority Queue. The program will basically keep running until the Priority Queue is empty. Elements will be added from beginning to end (e.g. from Element A to the last element Z) and this is also the order in which the priority queue will be called ("Graph and its representations - GeeksforGeeks", 2018).

Testing

Using perhaps a different algorithm to test the solution for instance the algorithm of Bellman-Ford (which is a bit slower but can also calculate the shortest path for negative distances) or the Fredman & Tarjan algorithm that was designed in 1984 which uses a Fibonacci heap. Or by inserting a graph which has been precreated (and which already gives the correct answer beforehand), a test graph from a reliable source. Such an example could be SNAP, which is a library that “easily scales to massive networks with hundreds of millions of nodes, and billions of edges.” (snap.stanford.eduw)

Test generation technique or class.

Inside our main class we import 10 datasets from the website of Stanford University, these input sets will be used inside our code and during testing(“Index of /~yyye/yyye/Gset,” 2003). The datasets are then imported, after this splitted based on the space between the input and then used to run our tests. The class function will keep on running until we reach the end of the input sets, as we use input sets.length to measure this.

Input set

At first we suggested to create the input sets on our own, by generating a desired amount of edges (objects of the Edge class) which will be randomly connected to each other (by using the Math.Random class). Afterwards weight will randomly be added to each connection (again by using the Math.Random class) given a desired minimum and maximum weight range.

However after some research, we found the option to use existing sources of the Stanford University. We decided to start out with these because it seemed like a credible source to us and it would help us to quickly advance to the crux of the course. Below we will explain how we will read and use these input sets in our situation.

The user of the program has to put all the desired files as filenames in string format into our inputsets array, as seen below.

```
String[] inputsets = new String[]{ "G1.txt", "G23.txt", "G25.txt", "G36.txt", "G45.txt",
"G54.txt", "G58.txt", "G60.xt", "G50.txt", "G70.txt" };
```

The inputsets from this array will then consecutively iterated with Dijkstra’s Algorithm. After which the output is in the following string form for each of the edges:

Source Vertex: 0 to vertex 6241 distance: 3

This will result in a huge output, since the files we use often have 10000 edges and each get their own printed line. Which is why we think it might be better to store this information in a separate text file.

Expected concurrency gains from parallelisation

We have chosen to parallelise the nodes. We can basically say that the concurrency gains will be seen where a node has multiple connections to other nodes. Since the length of the previous path always has to be known (which means that there is no way to start in the middle of the process), we can only work from there on forward. Let’s say we have just calculated the length of node A to node B, and node B has 10 new connections. We can now use 10 threads to calculate the distances to all those 10 nodes at the same time. The maximum concurrency gains will be depend on “the maximum depth of the graph”, or the furthest path to possibly take.

address all the points from grading matrix Concurrency gains and expectations are discussed: the expected speedup and efficiency of the parallel solution, task granularity, problem decomposition and what the communication overhead is**(feedback expected concurrency gains)**

Dijkstra’s algorithm complexity

The complexity of Dijkstra’s algortihm is O(VlogV) + O(ElogV). E being an irrational and transcendental number approximately equal to 2.718281828459

Expected Speedup

Basically the speed up depends on the amount of interconnected vertices, if the average amount of connections between the vertices is high, it means that there is a high opportunity for speed up, whilst when it would be low there is not much to be gained by parallelising. And of course the speedup will b

Efficiency of the parallel solution

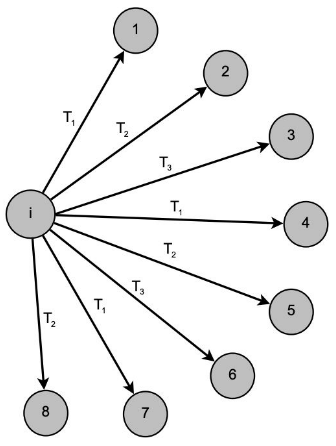
in general we would say that the efficiency is very high with our proposed input sets since they all have very interconnected graphs. In general the solution will be quite efficient since in most graphs that we have found in our research on web pages and books, nodes will have multiple connections, thus we can assume that in general a parallel Dijkstra will be efficient.

Task Granularity

As we have said before and as can be seen in the figure on the right The task to calculate the new eight to each new node from node i can be granularized to 8 separate tasks to be performed by different threads at the same time. So this is a positive sign from our point of view.

(If Tcomp is the computation time and Tcomm denotes the communication time, then the Granularity G of a task can be calculated as:

$$G = \frac{Tcomp}{Tcomm} \quad)$$



Problem Decomposition

As the existing Dijkstra algorithm is serial and thus executes the codes in a serial matter. We will be parallelizing this process during the continuation of the course.
In the first assignment we will be parallelizing the full program without making any changes on the existing algorithm. As we proceed to the next assignments we will have to put more time and thinking into the decompositioning and come up with possible solutions to parallelize in a more complicated and deeper way (for example parallelising the nodes and even the edges themselves).

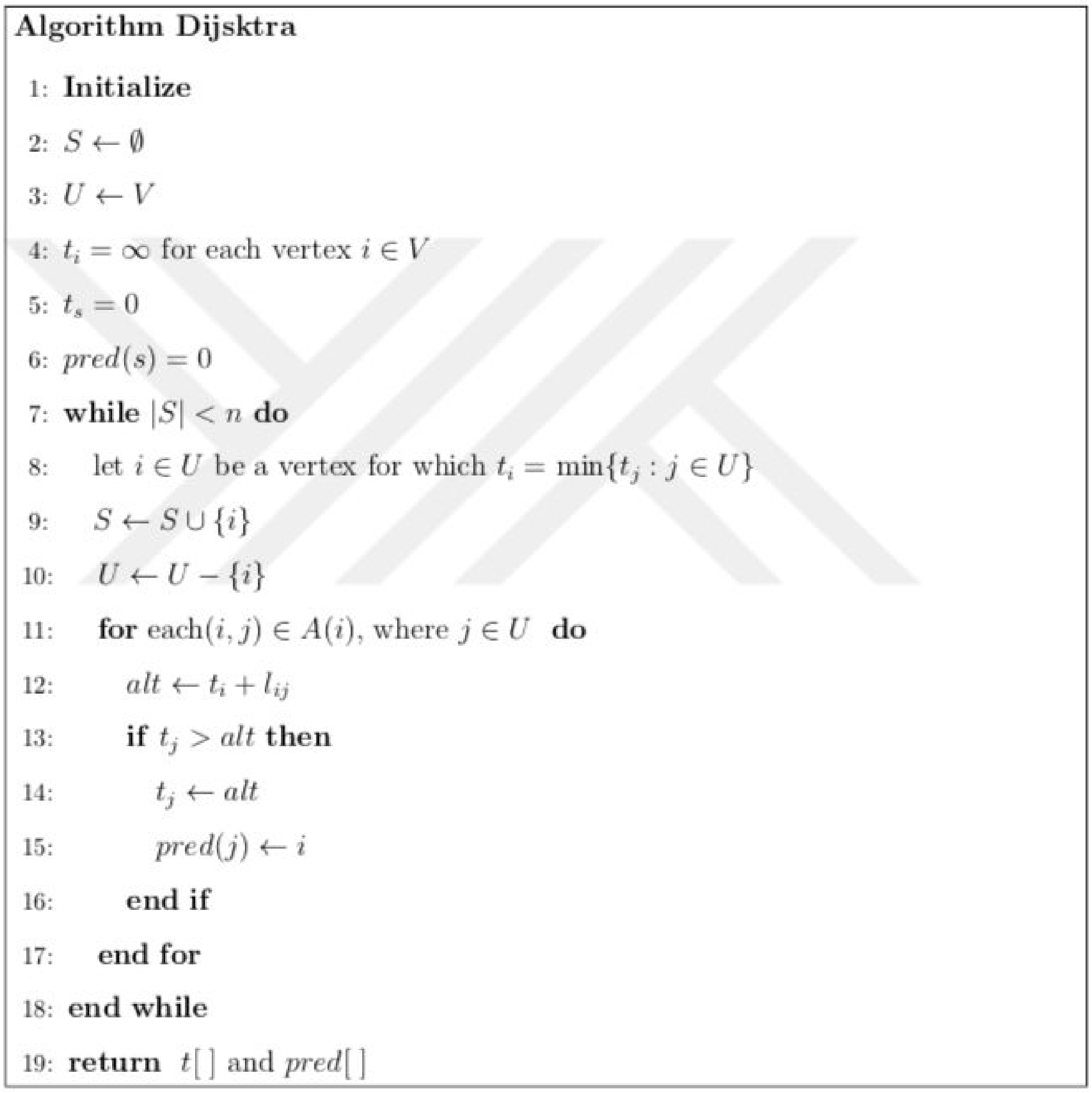
The first problem that we will be facing is the fact that a thread need its own list of Graphs depending on the amount of cores. So this is why we have made the method called prepareGraphListsForEachThread(). There are also a couple more problems for which we have made methods which will be discussed in the following pages

Communication overhead

As of how the software is written right now, there is no problem resulting from communication overhead. When we proceed to a more complex implementation (on node or edge level as mentioned before), the threads may suffer from the communication overhead, namely whenever they need to wait for the time needed to request the previous weight of the node to add the new number up to.

PseudoCode

Below we describe the Dijkstra algorithm in pseudo code (Ersoy, 2012).



```
Algorithm Dijsktra
Initialize
$S \leftarrow \emptyset$
$U \leftarrow V$
$t_i = \infty$ for each vertex $i \in V$
$t_s = 0$
$pred(s) = 0$
while $|S| < n$ do
    let $i \in U$ be a vertex for which $t_i = \min\{t_j : j \in U\}$
    $S \leftarrow S \cup \{i\}$
    $U \leftarrow U - \{i\}$
    for each$(i, j) \in A(i)$, where $j \in U$ do
        $alt \leftarrow t_i + l_{ij}$
        if $t_j > alt$ then
            $t_j \leftarrow alt$
            $pred(j) \leftarrow i$
        end if
    end for
end while
return $t[\,]$ and $pred[\,]$
```

In the pseudocode of the algorithm given in the figure above, the following notation is used:

- S is the set of visited vertices, so far, in graph.
- U is the set of unvisited vertices, so far, in graph.
- s is the source node.
- V is the set of vertices in graph.
- .n is the number of vertices in graph.
- ti is the current minimum distance of node i to the source node s.
- pred(i) is the predecessor vertex of node i in the shortest path route from s to i.
- A(i) is the list of adjacencies of node i.
- alt is the alternative route time that is tested for the fastest route from vertex s to j.

Dijkstra’s algorithm

Below is our proposed implementation of Dijkstra’s Algorithm using PriorityQueue and adjacency list (J, 2018).

```
import javafx.util.Pair;
       import java.util.Comparator;
       import java.util.LinkedList;
       import java.util.PriorityQueue;

public class DijkstraPQ {
 static class Edge {
   int source;
   int destination;
   int weight;

   public Edge(int source, int destination, int weight) {
     this.source = source;
     this.destination = destination;
     this.weight = weight;
   }
 }

 static class Graph {
   int vertices;
   LinkedList<Edge>[] adjacencylist;

   Graph(int vertices) {
     adjacencylist = new Lin     this.vertices = vertices;
kedList[vertices];
     //initialize adjacency lists for all the vertices
     for (int i = 0; i <vertices ; i++) {
       adjacencylist[i] = new LinkedList<>();
     }
   }

   public void addEdge(int source, int destination, int weight) {
     Edge edge = new Edge(source, destination, weight);
     adjacencylist[source].addFirst(edge);

     edge = new Edge(destination, source, weight);
     adjacencylist[destination].addFirst(edge); //for undirected graph
   }

   public void dijkstra_GetMinDistances(int sourceVertex){

     boolean[] SPT = new boolean[vertices];
     //distance used to store the distance of vertex from a source
     int [] distance = new int[vertices];

     //Initialize all the distance to infinity
     for (int i = 0; i <vertices ; i++) {
       distance[i] = Integer.MAX_VALUE;
     }
     //Initialize priority queue
```

```
    //override the comparator to do the sorting based keys
    PriorityQueue<Pair<Integer, Integer>> pq = new PriorityQueue<>(vertices, new Comparator<Pair<Integer,
Integer>>() {
      @Override
      public int compare(Pair<Integer, Integer> p1, Pair<Integer, Integer> p2) {
        //sort using distance values
        int key1 = p1.getKey();
        int key2 = p2.getKey();
        return key1-key2;
      }
    });

 //create the pair for for the first index, 0 distance 0 index
    distance[0] = 0;
    Pair<Integer, Integer> p0 = new Pair<>(distance[0],0);
    //add it to pq
    pq.offer(p0);

    //while priority queue is not empty
    while(!pq.isEmpty()){
      //extract the min
      Pair<Integer, Integer> extractedPair = pq.poll();

      //extracted vertex
      int extractedVertex = extractedPair.getValue();
      if(SPT[extractedVertex]==false) {
        SPT[extractedVertex] = true;

        //iterate through all the adjacent vertices and update the keys
        LinkedList<Edge> list = adjacencylist[extractedVertex];
        for (int i = 0; i < list.size(); i++) {
          Edge edge = list.get(i);
          int destination = edge.destination;
          //only if edge destination is not present in mst
          if (SPT[destination] == false) {
            ///check if distance needs an update or not
            //means check total weight from source to vertex_V is less than
            //the current distance value, if yes then update the distance
            int newKey =  distance[extractedVertex] + edge.weight ;
            int currentKey = distance[destination];
            if(currentKey>newKey){
              Pair<Integer, Integer> p = new Pair<>(newKey, destination);
              pq.offer(p);
              distance[destination] = newKey;
            }
          }
        }
      }
    }
    //print Shortest Path Tree
    printDijkstra(distance, sourceVertex);
  }

  public void printDijkstra(int[] distance, int sourceVertex){
    System.out.println("Dijkstra Algorithm: (Adjacency List + Priority Queue)");
    for (int i = 0; i <vertices ; i++) {
      System.out.println("Source Vertex: " + sourceVertex + " to vertex " +   + i +
              " distance: " + distance[i]);
    }
  }

  public static void main(String[] args) {
    int vertices = 6;
    Graph graph = new Graph(vertices);
    graph.addEdge(0, 1, 4);
    graph.addEdge(0, 2, 3);
    graph.addEdge(1, 2, 1);
    graph.addEdge(1, 3, 2);
    graph.addEdge(2, 3, 4);
    graph.addEdge(3, 4, 2);
    graph.addEdge(4, 5, 6);
    graph.dijkstra_GetMinDistances(0);
  }
}
```

1.2 Bucketsort

Bucket sort is mainly useful when input is uniformly distributed over a range. The numbers will be distributed into a number of buckets. Each bucket has a specific range value. Each number in the array will be put into the bucket that fits the range. Eventually all numbers will be in a bucket. The buckets themselves are already in the good order, so now we just need to sort the numbers in the buckets. This could be done with selection sort.

Correctness of the solution

The correctness of the solution can be tested by measuring the speed of the parallelized in comparison with the sequential solution. By benchmarking the results we can give an accurate result of the performance increase, or possible decrease.

Generating sizeable input sets

A sizable large input can be generated by generating a big array with random numbers from a desired range.

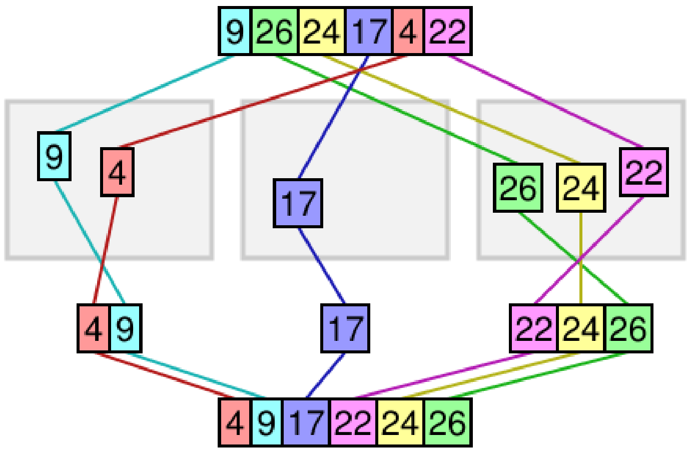
Expected concurrency gains from parallelisation

Bucket Sort it’s main logic is to divide different parts of array into different buckets, then sort them at the same time with another algorithm and put it back together in one big result array (Smaizys, 2013).

For each number we are checking in which bucket it fits. If we checked multiple numbers at once, our time would improve by *n* times. Each thread could check 1 number and put that in the corresponding bucket.

But checking for buckets could also be done simultaneously. Instead of having to check each consecutive bucket. If multiple threads or cores are used they can all be checked by one immediately. And thus there is even more improvement

After putting the numbers into the specific buckets, it’s possible to sort them out in a parallel way which will be faster than the original sequential way.
Below we can see an example of how the Bucket Sort normally would function, as we can see all numbers are sorted and splitted up into buckets until the outcome is fully sorted.



Testing

By using different algorithms to test the solution and outcome of the Bucket Sort, for instance the Merge Sort algorithm. Will help us check our solution on correctly and accurately.
Besides the Merge Sort algorithm it would be possible to use similar algorithms like for example the Quick Sort algorithm or the Heap Sort algorithm.

Test generation technique or class.

Inside the main class we will be able to generate input sets ourself of sizes as big as wished for. These input sets can be generated by using the Math.Random class().
The input sets are after importing, used to run our test, these test will split these inputs into several buckets and repeats this process until all data is processed and sorted out in the end.

Input set

As mentioned above, we will generate our own input sets for the Bucket Sort algorithm. These input sets can variate in size, the size of the input sets are defined by us and created in consultation with Mr. Srinivasan.

Bucket Sort algorithm complexity

The average time complexity for Bucket Sort is $\theta(n^2)$, the best time complexity for Bucket Sort is $\Omega(n)$ and as last the worst time complexity would be $O(n^2)$.

Expected Speedup

Basically the speed up depends on the amount of parts which are parallelized during the assignment. The more tasks are parallelized, the higher the changes that the speed up will be as efficient and fast as possible.

Efficiency of the parallel solution

We expect the efficiency of the parallel solution to be very high, as the parallelization of the sorting of numbers and buckets will speed up the entire process drastically.
Because of this we expect the algorithm complexity to speed up compared to the serial version, also the concurrency gains form the new parallelization Bucket Sort algorithm should be huge. Thus we have to keep in mind, that these gains do depend on the size of the used input sets, the smaller the input sets the less speed efficiency would come out of parallel solution.

Task Granularity

What basically could be done is when the array gets divided into buckets in the first place. This is a task that can already be done parallel (Thread 0 assigns array 0 to a bucket, Thread 1 assigns array 1, etc…). The next task is the sorting within the buckets (for which insertion sort is used) itself that can be parallelized. And the final task of filling up the array again can be parallelized. So we basically have 3 parts that are eligible for parallelisation.

Problem Decomposition

As the Bucket Sort algorithm is in default serial and thus shall executes its code in a serial matter. When choosing this algorithm we will be parallelizing the different processes or one of them during the continuation of the course.

With bucket sort we see clearly there are three steps that can describe the whole process. The first is dividing the numbers of the unsorted array into their correct number range (or the bucket). The second part is the sorting within each bucket. And the last part being the remerging of the array.

There are multiple ways in which the sorting within the buckets is done. We shall talk about one of these which is the insertion sort. The insertion sort itself can also be parallelized in the way that we addressed in the lesson.

Communication overhead

As the current implementation of the Bucket Sort states right now, there is no problem regarding the communication overhead.
When we proceed in our assignments to a more complex implementation, for example at multithreading of the buckets dividing, the threads may possibly suffer from communication overhead. This may possibly occur when the buckets are done sorting and compare each others value's to create an fully sorted outcome.

Bucket Sort algorithm

Below is our proposed implementation of the simple serial version of Bucket Sort Algorithm (Paul, 2019).

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.List;

/* * Java Program sort an integer array using radix sort algorithm.
* input: [80, 50, 30, 10, 90, 60, 0, 70, 40, 20, 50] *
output: [0, 10, 20, 30, 40, 50, 50, 60, 70, 80, 90]
public class BuckeSort {
        public static void main(String[] args) {
                System.out.println("Bucket sort in Java");
                int[] input = { 80, 50, 30, 10, 90, 60, 0, 70, 40, 20, 50 };
                System.out.println("integer array before sorting");
                System.out.println(Arrays.toString(input));

                // sorting array using radix Sort Algorithm
                bucketSort(input);
                System.out.println("integer array after sorting using bucket sort algorithm");
                System.out.println(Arrays.toString(input));
        }

        /** * * @param input */
        public static void bucketSort(int[] input) {
                // get hash codes
                final int[] code = hash(input);
                // create and initialize buckets to ArrayList: O(n)
                List[] buckets = new List[code[1]];
                for (int i = 0; i < code[1]; i++) {
                        buckets[i] = new ArrayList();
                }

                // distribute data into buckets: O(n)
                for (int i : input) {
                        buckets[hash(i, code)].add(i);
                }

                // sort each bucket O(n)
                for (List bucket : buckets) {
                        Collections.sort(bucket);
                }

                int ndx = 0;
                // merge the buckets: O(n)
                for (int b = 0; b < buckets.length; b++) {
                        for (int v : buckets[b]) { input[ndx++] = v;
                        }
                 }
        }

        /** * * @param input * @return an array containing hash of input */
        private static int[] hash(int[] input) {
                int m = input[0];
                for (int i = 1; i < input.length; i++) {
                        if (m < input[i]) {
                                m = input[i];
                        }
                 }

                return new int[] {
                        m, (int) Math.sqrt(input.length)
                };
        }

        /** * * @param i * @param code * @return */
        private static int hash(int i, int[] code) {
                 return (int) ((double) i / code[0] * (code[1] - 1));
        }
}
```

1.3 Merge Sort

Similar to for example QuickSort, Merge Sort is a divide and conquer algorithm. A divide-and-conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly.

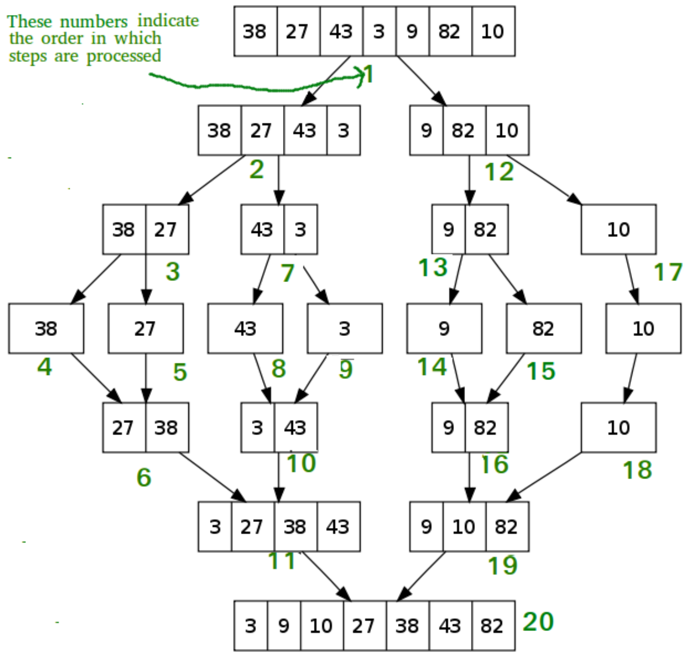
Merge Sort is an algorithm that splits up the given array of values into two halves until there are two elements per array. It then compares the numbers within each of the arrays of 2 elements and orders those. After that it merges all the arrays of 2 elements together and orders the resulting arrays of 4 elements. This continues until there is one array left which has all elements completely ordered in the ascending order.
The outcome of the Merge Sort is thus an ordered array of values, thus the input is logically an unordered array of values.

Correctness of the solution

- The solution can be tested by using any other algorithm, for instance the BucketSort which we have mentioned above.
- Or another possible solution would be to have a sheet with an ordered list (which is the answer), which afterwards is put in the algorithm in a random order (this can be done by using a function which will put the numbers in a random position), by using for example Math.Random, so we can make sure we have the right answer.

Generating sizeable input sets

A sizable large input can be generated by generating a big array with random numbers from a desired range.



Expected concurrency gains from parallelisation

The algorithm is dividing the array each time into two separate halves. If the array is for example composed of 1024 elements, it means the first execution handles 1024 elements, then the second one 512, then 256 etc. Until the last executions which have to perform a task with only 2 elements. Which will mean that there will be a bunch of tasks consisting of simply sorting two integers.

Those tasks are going to be available for work-stealing which results in causing a huge performance decreasing. Work-stealing is when an idle thread can query other threads of the pool (not directly, in a work-stealing world each thread has also its own local queue) and steal from them a task.
Because of this it is way faster to have one thread performing two times a simple job rather than having two threads synchronizing each other to split and share this job.

So the idea is to fix a limit regarding the number of elements to handle. If we are above this limit, we trigger the parallel execution. This will make the new parallelisation implementation faster then the sequential implementation (Harsanyi, 2018).

Testing

By using different algorithms to test the solution and outcome of the Merge Sort, for instance the Bucket Sort algorithm. Will help us check our solution on correctly and accurately. Besides the Bucket Sort algorithm it would be possible to use similar algorithms like for example the Quick Sort algorithm or the Heap Sort algorithm.

Test generation technique or class.

Inside the main class we will be able to generate input sets ourself of sizes as big as wished for. These input sets can be generated by using the Math.Random class().
The input sets are after importing, used to run our test, these test will split these inputs into several arrays and repeats this process until all data is processed and sorted out in the end.

Input set

As mentioned above, we will generate our own input sets for the Merge Sort algorithm. These input sets can variate in size, the size of the input sets are defined by us and created in consultation with Mr. Srinivasan.

Merge Sort algorithm complexity

The average time complexity for Merge Sort algorithm is $\theta(n \log(n))$, the best time complexity for Merge Sort is $\Omega(n \log(n))$ and as last the worst time complexity would be $O(n \log(n))$.

Expected Speedup

Basically the speed up depends on the amount of threads are created, reused and passed true into the subsections of the algorithm. The choices we make during the assignment based on the way of parallelisation will have impact on the final outcome and speedup, the more advanced choices we make the higher our final speed up will be and the maximum efficiency of the parallel solution will be accomplished.

Efficiency of the parallel solution

We expect the efficiency of the parallel solution to be very high, as it’s possible to parallelize the array sorting and possibly sorting of the array subsections. This will affect the speed up positively and basically speed up the entire process drastically.
Because of this we expect the algorithm complexity to speed up compared to the serial version, also the concurrency gains form the new parallelization Merge Sort algorithm should be huge. Thus we have to keep in mind, that these gains do depend on the size of the used input sets, the smaller the input sets the less speed efficiency would come out of parallel solution.

Task Granularity

The dividing part of the program is the first task which is being called recursively. The second part being the comparison and the sorting of individual numbers amongst each other. And the last part being the remerging (by adding all the branches together to form a whole again),

Problem Decomposition

The first part of the algorithm that divides can be parallelized in our opinion this can be done after the moment of first division. After that the number of tasks available for parallel computation is basically multiplied by a number of two. The second part the individual sorting can also be parallelized (multiple threads can do multiple sorting computations until the numbers are sorted individually). Then the remerging part can be done by a thread number moving down back to one only (for the last remerge).

Communication overhead

As we implement the current version of Merge Sort which is documented below, there is no problem regarding the communication overheid.
When we proceed in our assignments to a more complex implementation, for example at multithreading of the splitting, sorting and dividing the arrays, the threads may possibly suffer from communication overhead. This may possibly occur each time when the arrays are done sorting and compared to each others value’s to create an fully sorted outcome.

Merge Sort algorithm

Below is our proposed implementation of the simple serial version of Merge Sort Algorithm (Paul, 2019).

```
// Merges two subarrays of arr[].
  // First subarray is arr[l..m]
  // Second subarray is arr[m+1..r]
  void merge(int arr[], int l, int m, int r)
  {
     // Find sizes of two subarrays to be merged
     int n1 = m - l + 1;
     int n2 = r - m;

     /* Create temp arrays */
     int L[] = new int [n1];
     int R[] = new int [n2];

     /*Copy data to temp arrays*/
     for (int i=0; i<n1; ++i)
        L[i] = arr[l + i];
     for (int j=0; j<n2; ++j)
        R[j] = arr[m + 1+ j];
        /* Merge the temp arrays */

        // Initial indexes of first and second subarrays
        int i = 0, j = 0;

        // Initial index of merged subarry array
        int k = l;
        while (i < n1 && j < n2)
        {
            if (L[i] <= R[j])
            {
                arr[k] = L[i];
                i++;
            }
            else
            {
                arr[k] = R[j];
                j++;
            }
            k++;
        }

        /* Copy remaining elements of L[] if any */
        while (i < n1)
        {
            arr[k] = L[i];
            i++;
            k++;
        }

        /* Copy remaining elements of R[] if any */
        while (j < n2)
        {
            arr[k] = R[j];
            j++;
            k++;
        }
    }

    // Main function that sorts arr[l..r] using
    // merge()
    void sort(int arr[], int l, int r)
    {
        if (l < r)
        {
            // Find the middle point
            int m = (l+r)/2;

            // Sort first and second halves
            sort(arr, l, m);
            sort(arr , m+1, r);

            // Merge the sorted halves
            merge(arr, l, m, r);
        }
    }

    /* A utility function to print array of size n */
    static void printArray(int arr[])
    {
        int n = arr.length;
        for (int i=0; i<n; ++i)
            System.out.print(arr[i] + " ");
        System.out.println();
    }
    // Driver method
    public static void main(String args[])
    {
        int arr[] = {12, 11, 13, 5, 6, 7};

        System.out.println("Given Array");
        printArray(arr);

        MergeSort ob = new MergeSort();
        ob.sort(arr, 0, arr.length-1);

        System.out.println("\nSorted array");
        printArray(arr);
    }
}
```

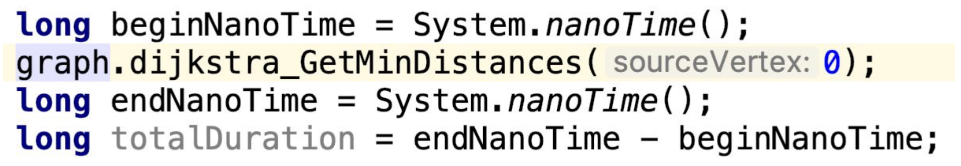
2. Basic thread and locks

2.1 Serial implementation

We have already documented the serial version above inside our document , at the section of “**Dijkstra’s algorithm”** (pages 10,11,12). We have also explained above in our case study of the Dijkstra’s Algorithm, that we have found an option to use existing sources of the Stanford University. The user of the program has the option to put all the wished for files into our inputsets array, as seen below.

*String[] inputsets = **new** String[]{ **"G1.txt"**, **"G23.txt"**, **"G25.txt"**, **"G36.txt"**, **"G45.txt"**, **"G54.txt"**, **"G58.txt"**, **"G60.xt"**, **"G50.txt"**, **"G70.txt"** };*

Serial benchmarks



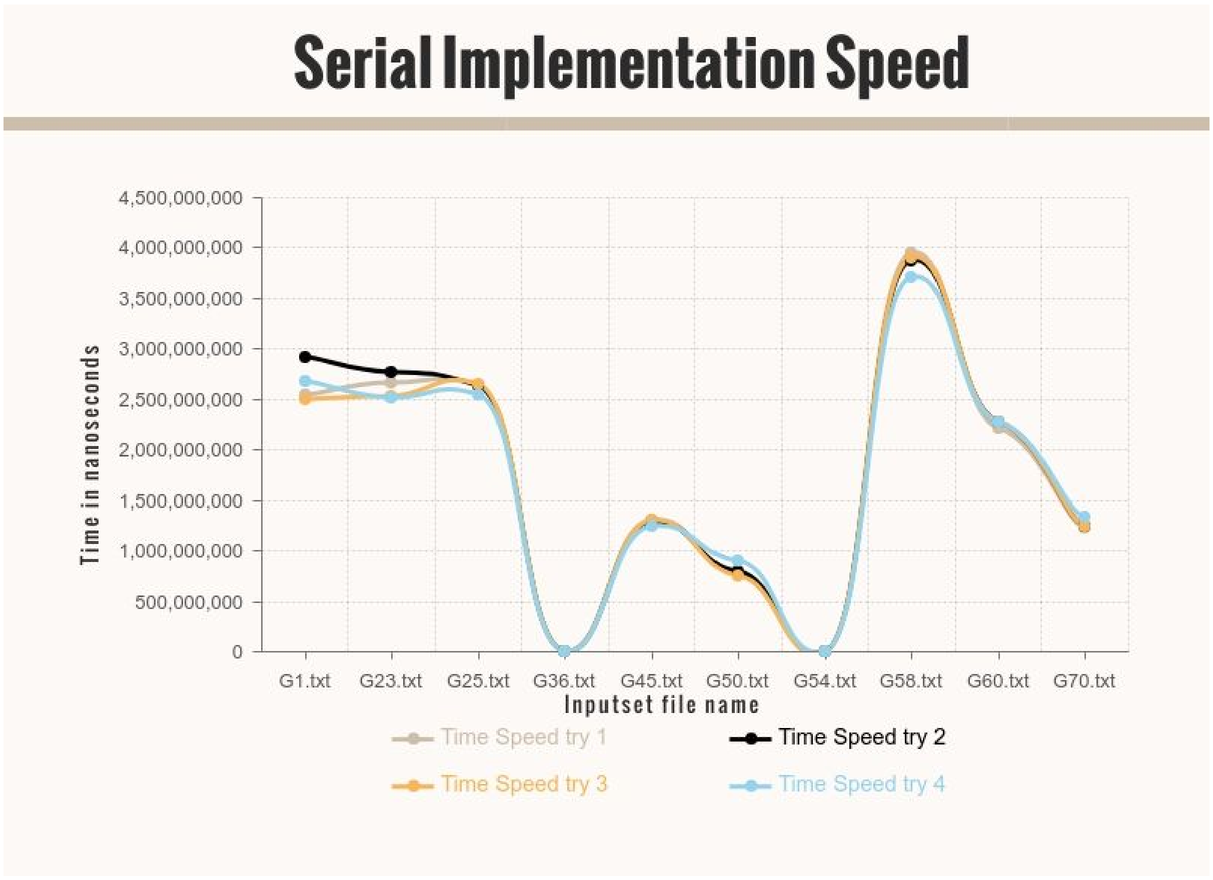
```
long beginNanoTime = System.nanoTime();
graph.dijkstra_GetMinDistances( sourceVertex: 0);
long endNanoTime = System.nanoTime();
long totalDuration = endNanoTime - beginNanoTime;
```

The program will then run for all the selected files serially and calculate the shortest path distance from the begin node to each of the nodes in the graph. The code for that is seen below:

In order to collect the wished for statistics we chose to use the Nanotime() measurement instead of the CurrentTimeMillis() method because it is more accurate, the only drawback being that it takes a bit more time because of this.

Below you can find the output and the needed time to process all the input sets(10) a couple of times so we can get an idea of the average process time for each individual set:

Inputset file	Time Speed try 1	Time Speed try 2	Time Speed try 3	Time Speed try 4
G1.txt	2537425200 ns	2913099500 ns	2501976900 ns	2674222900 ns
G23.txt	2661408500 ns	2760552200 ns	2529261700 ns	2518631900 ns
G25.txt	2629985500 ns	2633302301 ns	2647941800 ns	2546761900 ns
G36.txt	2800 ns	25999 ns	23200 ns	21200 ns
G45.txt	1302768500 ns	1281840701 ns	1296907000 ns	1247125900 ns
G50.txt	775069200 ns	786263999 ns	753254800 ns	896529700 ns
G54.txt	2400 ns	2900 ns	1700 ns	2000 ns
G58.txt	3941400700 ns	3873665600 ns	3900910500 ns	3708562200 ns
G60.txt	2217551500 ns	2268035600 ns	2265861800 ns	2268944500 ns
G70.txt	1223144600 ns	1244166199 ns	1245026500 ns	1326795900 ns

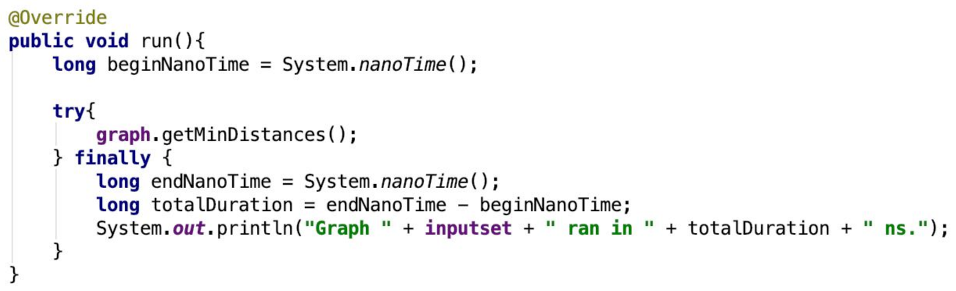

Serial Implementation Speed
Time in nanoseconds
4,500,000,000
4,000,000,000
3,500,000,000
3,000,000,000
2,500,000,000
2,000,000,000
1,500,000,000
1,000,000,000
500,000,000
0
G1.txt
G23.txt
G25.txt
G36.txt
G45.txt
G50.txt
G54.txt
G58.txt
G60.txt
G70.txt
Inputset file name
Time Speed try 1
Time Speed try 2
Time Speed try 3
Time Speed try 4

2.2 Threads and Locks implementation

In the serial Dijkstra’s Algorithm implementation we already make use of several (10) different inputs sets. In order to compare the results of the serial implementation to the basic threads and locks implementation we will be using the same input sets as we have done inside our serial implementation of the Dijkstra’s Algorithm.

Threads and Locks benchmarks, approach 1; Single Graphs per Thread

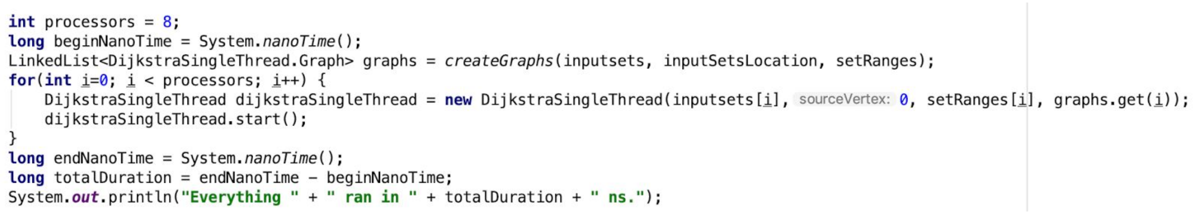
Below we can see our elaboration of the 1 approach (Parallelize the whole Algorithm, in a for loop). We will be calling the getMinDistances() method in the run() method of a Thread.
At the start and at the end of the run the duration is calculated in nanoseconds.
We are able to select an X amount of processors to work with.



```
@Override
public void run(){
    long beginNanoTime = System.nanoTime();

    try{
        graph.getMinDistances();
    } finally {
        long endNanoTime = System.nanoTime();
        long totalDuration = endNanoTime - beginNanoTime;
        System.out.println("Graph " + inputset + " ran in " + totalDuration + " ns.");
    }
}
```

there is also a nice option in Java that checks how much processors are available at the moment of execution. Based on this information, the amount of processors will be chosen.

```
int processors = Runtime.getRuntime().availableProcessors();
```

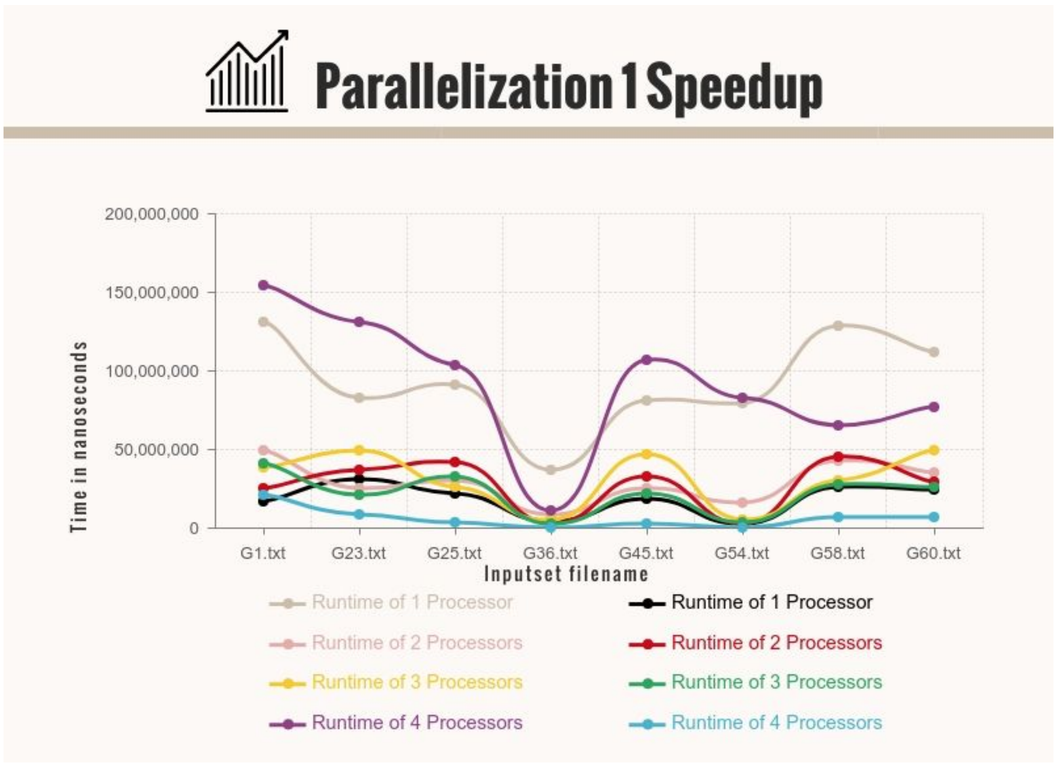


```
int processors = 8;
long beginNanoTime = System.nanoTime();
LinkedList<DijkstraSingleThread.Graph> graphs = createGraphs(inputsets, inputSetsLocation, setRanges);
for(int i=0; i < processors; i++) {
    DijkstraSingleThread dijkstraSingleThread = new DijkstraSingleThread(inputsets[i], sourceVertex: 0, setRanges[i], graphs.get(i));
    dijkstraSingleThread.start();
}
long endNanoTime = System.nanoTime();
long totalDuration = endNanoTime - beginNanoTime;
System.out.println("Everything " + " ran in " + totalDuration + " ns.");
```

With this approach one core is used to execute exactly one graph as we can see above (graphs.get(i) only passes one graph in the creation of the DijkstraSingleThread class). The disadvantage of this approach is that whenever there are more input sets compared to the cores, not all input sets will be executed as one core can exactly execute one graph.
We have created another approach in order to develop a more efficient way of parallelizing the algorithm in the next section.

Below we can see the outcome of approach 1.1(Parallelize the whole Algorithm, in a for loop). We have benchmarked the results for 1, 2, 3 and 4 processors, each outcome has been calculated and displayed twice to make sure the results are realistically.

Inputset file	Runtime of 1 Processor	Runtime of 1 Processor	Runtime of 2 Processors	Runtime of 2 Processors	Runtime of 3 Processors	Runtime of 3 Processors	Runtime of 4 Processors	Runtime of 4 Processors
G1.txt	130717999 ns	16776499 ns	48922700 ns	25221200 ns	37927300 ns	41130000 ns	153766100 ns	20867400 ns
G23.txt	82442699 ns	30654499 ns	25162700 ns	36395100 ns	48754600 ns	20573800 ns	130424400 ns	8214600 ns
G25.txt	90904400 ns	21408900 ns	30056600 ns	41874500 ns	25788600 ns	32142300 ns	103318100 ns	3077400 ns
G36.txt	36779100 ns	3780200 ns	8408100 ns	2701100 ns	4730200 ns	2352200 ns	10755900 ns	4900 ns
G45.txt	81147501 ns	18593101 ns	25291400 ns	32398400 ns	46754800 ns	21724300 ns	106597600 ns	2559400 ns
G54.txt	79438700 ns	2322001 ns	15546600 ns	2857600 ns	5001300 ns	3456400 ns	82912200 ns	11800 ns
G58.txt	128109400 ns	25958400 ns	42912000 ns	45199800 ns	30015600 ns	27213500 ns	65357000 ns	6697600 ns
G60.txt	111415001 ns	24264000 ns	35059500 ns	29192500 ns	49202300 ns	25527600 ns	76825000 ns	6418000 ns
total runtime	254822701 ns	112653900 ns	112534000 ns	126416100 ns	111891500 ns	138677300 ns	218153400 ns	126273600 ns



Result analysis

First of all we can see that the input sets usually run from about 110 million ns to 130 million ns, aside from 2 anomalies which run the double time, which is quite peculiar (more info about this later).
If we have a look at the clock running time of the threads and locks implementation, we can see that even with the anomalies included the general computation time is already much smaller.

<u>Note</u>

Now this does make sense since one processor will only compute one graph (while the serial implementation computes ten).
However when we look at a larger amount of processors, we do see that the speed up is more than proportional to the serial implementation. So it gives us a hint that Threads might indeed improves the speed up!

Bottlenecks

As we compare the results we can conclude that with the previous approach we are not really able to measure the differences in speed, it rather limits the amount of input sets that are being processed. So another approach would be to have the following scenario.
Theoretically let us suppose we have 12 input sets, then:
when using 1 processor:
It would have to do all 12 input sets alone
when using 2 processors:
They would both do 6 input sets (2 x 6)
when using 3 processors:
They would all do 4 input sets (3 x 4)
and lastly, when using 4 processors:
The four of them would all process 3 input sets (4 x 3)
As we can expect it will have an impact on the speed because it will not compromise on the amount of graphs processed.

Communication overhead

The communication here is basically non existent since the processors all calculate their own graphs independently of each other, no data is being send to other cores and thus the communication overhead is zero.

Task Granularity

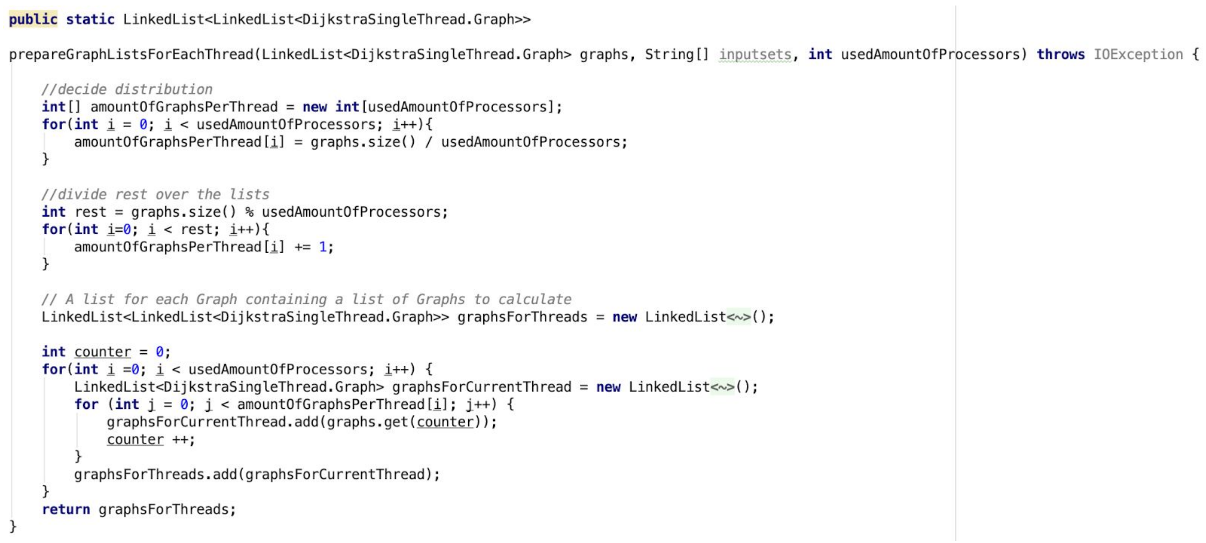
The tasks are the graphs themselves, these get granularized. Each Graph is a different task to be executed by a Thread.

Is the expected speedup and efficiency true?

This was basically the serial implementation with Threads and Locks we did not expect a speed up. We just tinkered around with the code to make it work. Even though the times do range widely in execution time. If we would take the average of all of time we would still get about the same results as the serial implementation.

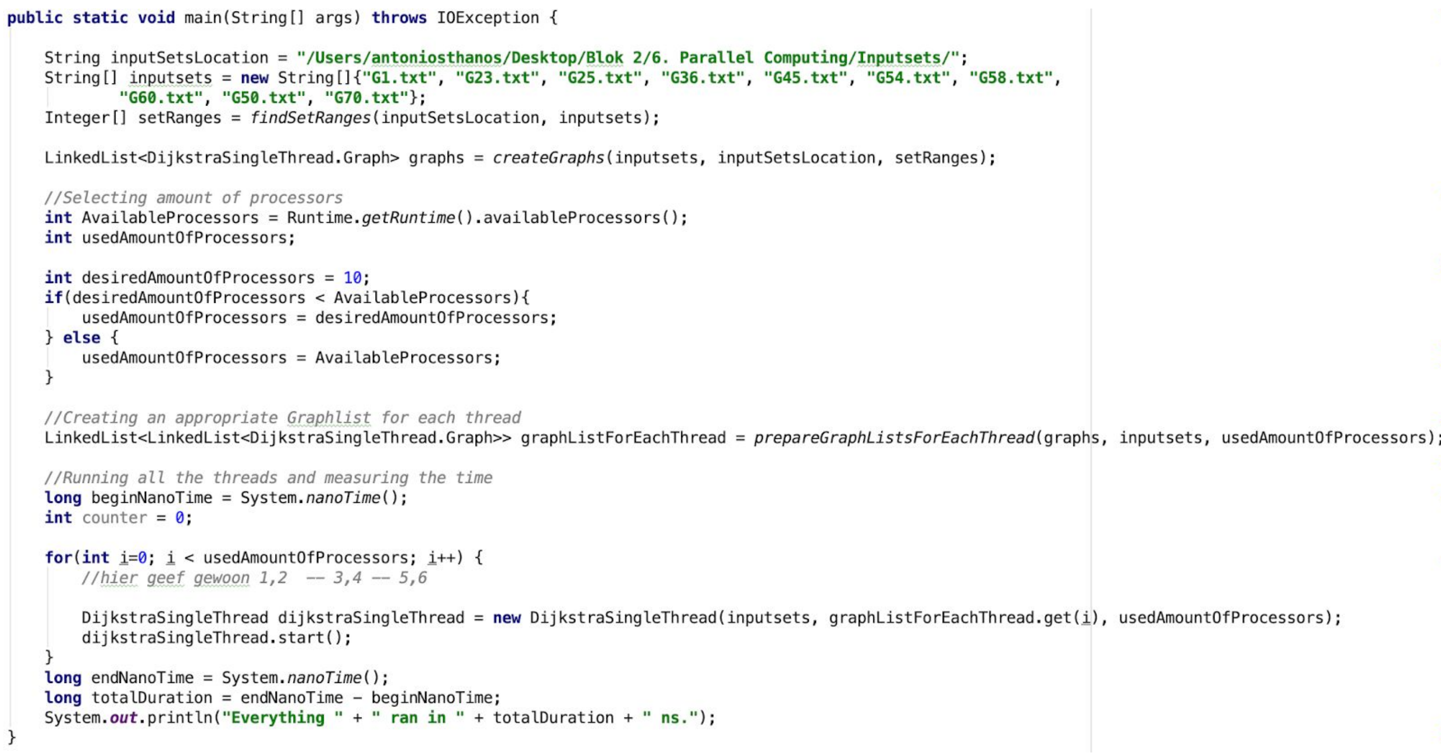
2.3 Threads and Locks benchmarks, approach 2 Multiple Graphs per Thread

Now we have created a piece of code (a method) which will return to us a LinkedList of a LinkedList of Graphs, of which we shall send each part to its corresponding thread:



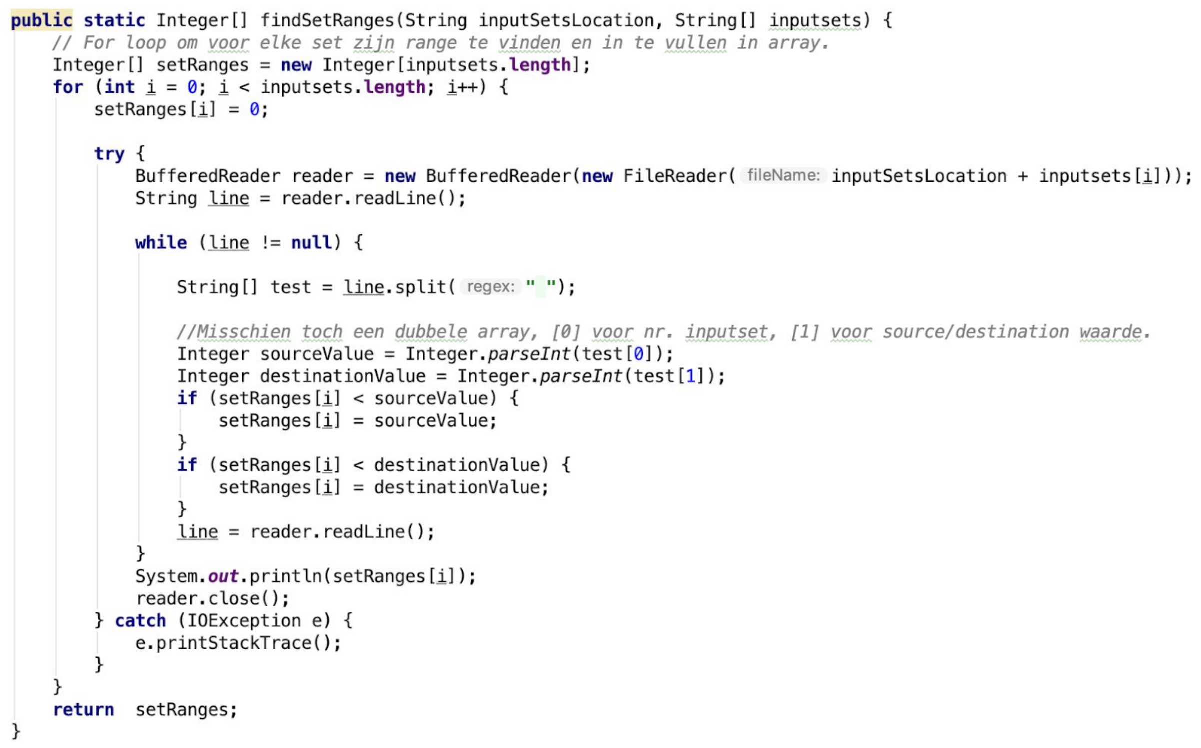
```java
public static LinkedList<LinkedList<DijkstraSingleThread.Graph>>

prepareGraphListsForEachThread(LinkedList<DijkstraSingleThread.Graph> graphs, String[] inputsets, int usedAmountOfProcessors) throws IOException {

    //decide distribution
    int[] amountOfGraphsPerThread = new int[usedAmountOfProcessors];
    for(int i = 0; i < usedAmountOfProcessors; i++){
        amountOfGraphsPerThread[i] = graphs.size() / usedAmountOfProcessors;
    }

    //divide rest over the lists
    int rest = graphs.size() % usedAmountOfProcessors;
    for(int i=0; i < rest; i++){
        amountOfGraphsPerThread[i] += 1;
    }

    // A list for each Graph containing a list of Graphs to calculate
    LinkedList<LinkedList<DijkstraSingleThread.Graph>> graphsForThreads = new LinkedList<~>();

    int counter = 0;
    for(int i =0; i < usedAmountOfProcessors; i++) {
        LinkedList<DijkstraSingleThread.Graph> graphsForCurrentThread = new LinkedList<~>();
        for (int j = 0; j < amountOfGraphsPerThread[i]; j++) {
            graphsForCurrentThread.add(graphs.get(counter));
            counter ++;
        }
        graphsForThreads.add(graphsForCurrentThread);
    }
    return graphsForThreads;
}
```

Main (approach 2)

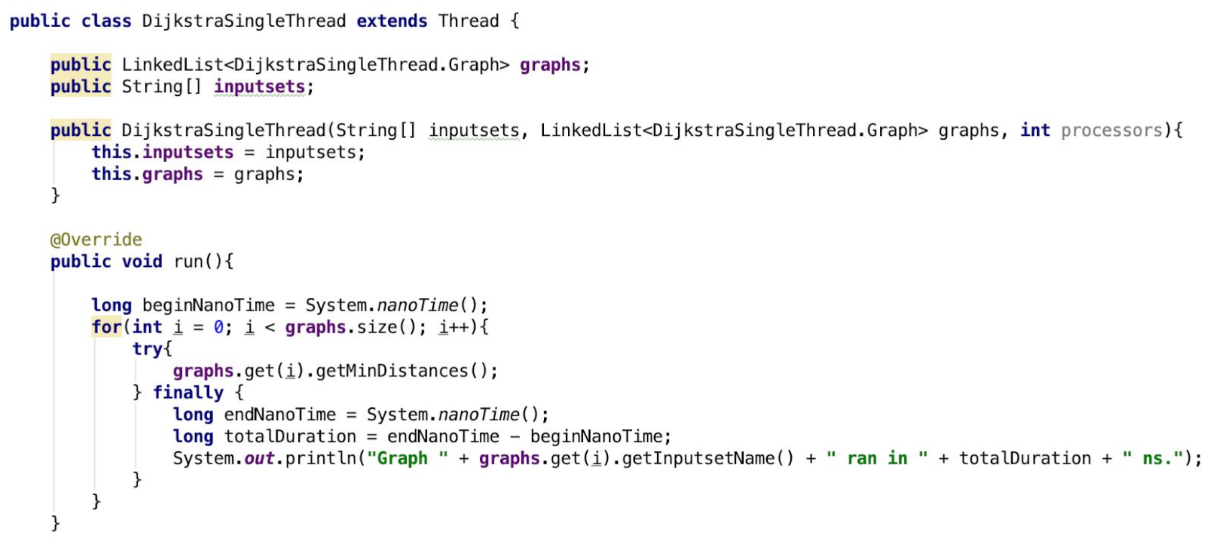


```java
public static void main(String[] args) throws IOException {

    String inputSetsLocation = "/Users/antoniosthanos/Desktop/Blok 2/6. Parallel Computing/Inputsets/";
    String[] inputsets = new String[]{"G1.txt", "G23.txt", "G25.txt", "G36.txt", "G45.txt", "G54.txt", "G58.txt",
            "G60.txt", "G50.txt", "G70.txt"};
    Integer[] setRanges = findSetRanges(inputSetsLocation, inputsets);

    LinkedList<DijkstraSingleThread.Graph> graphs = createGraphs(inputsets, inputSetsLocation, setRanges);

    //Selecting amount of processors
    int AvailableProcessors = Runtime.getRuntime().availableProcessors();
    int usedAmountOfProcessors;

    int desiredAmountOfProcessors = 10;
    if(desiredAmountOfProcessors < AvailableProcessors){
        usedAmountOfProcessors = desiredAmountOfProcessors;
    } else {
        usedAmountOfProcessors = AvailableProcessors;
    }

    //Creating an appropriate Graphlist for each thread
    LinkedList<LinkedList<DijkstraSingleThread.Graph>> graphListForEachThread = prepareGraphListsForEachThread(graphs, inputsets, usedAmountOfProcessors);

    //Running all the threads and measuring the time
    long beginNanoTime = System.nanoTime();
    int counter = 0;

    for(int i=0; i < usedAmountOfProcessors; i++) {
        //hier geef gewoon 1,2  -- 3,4 -- 5,6

        DijkstraSingleThread dijkstraSingleThread = new DijkstraSingleThread(inputsets, graphListForEachThread.get(i), usedAmountOfProcessors);
        dijkstraSingleThread.start();
    }
    long endNanoTime = System.nanoTime();
    long totalDuration = endNanoTime - beginNanoTime;
    System.out.println("Everything " + " ran in " + totalDuration + " ns.");
}
```

Findsetranges



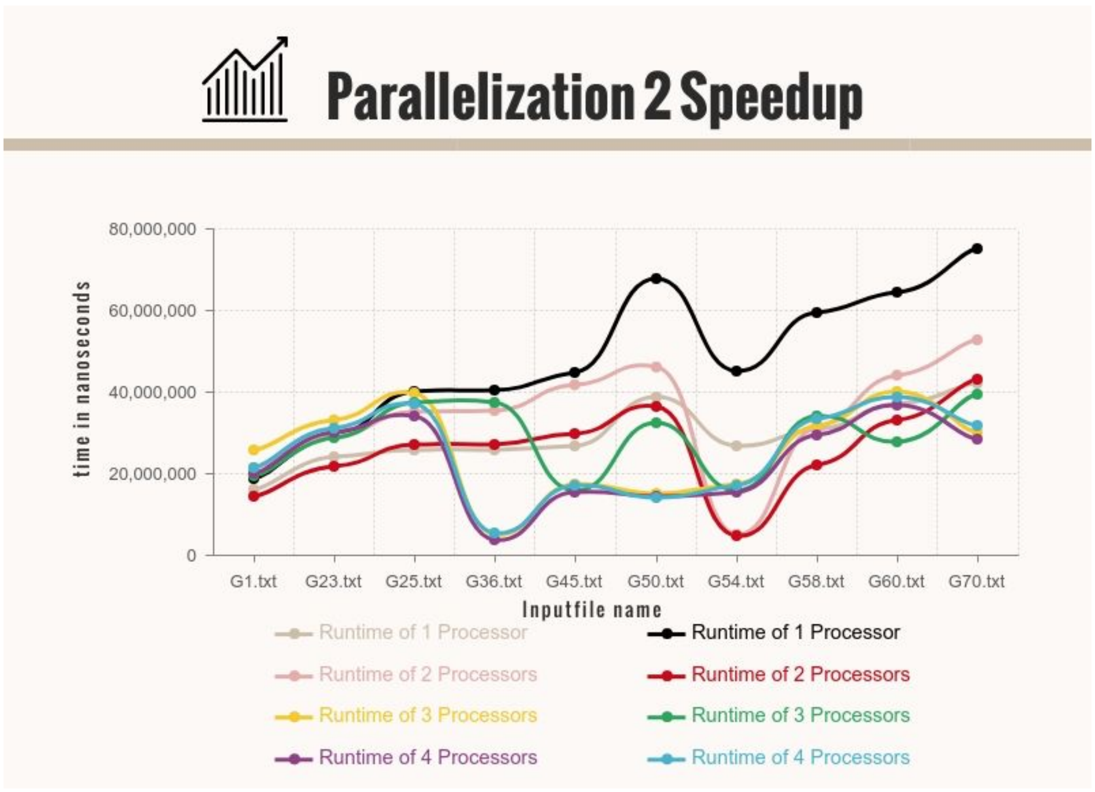
```
public static Integer[] findSetRanges(String inputSetsLocation, String[] inputsets) {
    // For loop om voor elke set zijn range te vinden en in te vullen in array.
    Integer[] setRanges = new Integer[inputsets.length];
    for (int i = 0; i < inputsets.length; i++) {
        setRanges[i] = 0;

        try {
            BufferedReader reader = new BufferedReader(new FileReader( fileName: inputSetsLocation + inputsets[i]));
            String line = reader.readLine();

            while (line != null) {

                String[] test = line.split( regex: " ");

                //Misschien toch een dubbele array, [0] voor nr. inputset, [1] voor source/destination waarde.
                Integer sourceValue = Integer.parseInt(test[0]);
                Integer destinationValue = Integer.parseInt(test[1]);
                if (setRanges[i] < sourceValue) {
                    setRanges[i] = sourceValue;
                }
                if (setRanges[i] < destinationValue) {
                    setRanges[i] = destinationValue;
                }
                line = reader.readLine();
            }
            System.out.println(setRanges[i]);
            reader.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    return  setRanges;
}
```



```
public class DijkstraSingleThread extends Thread {

    public LinkedList<DijkstraSingleThread.Graph> graphs;
    public String[] inputsets;

    public DijkstraSingleThread(String[] inputsets, LinkedList<DijkstraSingleThread.Graph> graphs, int processors){
        this.inputsets = inputsets;
        this.graphs = graphs;
    }

    @Override
    public void run(){

        long beginNanoTime = System.nanoTime();
        for(int i = 0; i < graphs.size(); i++){
            try{
                graphs.get(i).getMinDistances();
            } finally {
                long endNanoTime = System.nanoTime();
                long totalDuration = endNanoTime - beginNanoTime;
                System.out.println("Graph " + graphs.get(i).getInputsetName() + " ran in " + totalDuration + " ns.");
            }
        }
    }
```

Here the Graph basically calls the method ***get_min_distances which is now renamed to run(),*** which is thus automatically called when graph.start() is executed.
Now again we shall show the results with different processors and now truly, with all input sets at all times! Again we have included two times the results, to make sure the results are as realistically as possible.

Inputset file	1 Processor	1 Processor	2 Processors	2 Processors	3 Processors	3 Processors	4 Processors	4 Processors
G1.txt	18655024 ns	18655024 ns	21264548 ns	14276466 ns	25747945 ns	19188016 ns	19879102 ns	21382227 ns
G23.txt	29109634 ns	29109634 ns	29012973 ns	21785438 ns	33150972 ns	28582124 ns	30073243 ns	31075819 ns
G25.txt	4007299 ns	40007299 ns	35164743 ns	27019074 ns	39545719 ns	37198430 ns	33968848 ns	36871436 ns
G36.txt	40368179 ns	40368179 ns	35327668 ns	27108620 ns	5075758 ns	37311423 ns	3664019 ns	5340059 ns
G45.txt	4468530 ns	44684530 ns	41571409 ns	29654480 ns	17310765 ns	15659101 ns	15385214 ns	16894257 ns
G50.txt	67607672 ns	67607672 ns	46040645 ns	36357891 ns	15154402 ns	32466488 ns	14442867 ns	13884861 ns
G54.txt	45088982 ns	45088982 ns	4938946 ns	4532319 ns	17404951 ns	15761814 ns	15468649 ns	16988057 ns
G58.txt	59496835 ns	59496835 ns	31310330 ns	21833804 ns	31587542 ns	34104346 ns	29276527 ns	33199065 ns
G60.txt	64258158 ns	64258158 ns	43846394 ns	32975086 ns	40080339 ns	27607382 ns	36800844 ns	38731522 ns
G70.txt	74880296 ns	74880206 ns	52620291 ns	42903795 ns	29877745 ns	39305873 ns	28205774 ns	31812054 ns
total runtime	704658 ns	704658 ns	1127603 ns	882631 ns	1182577 ns	1488096 ns	2100962 ns	2911867 ns



Result analysis

As we can see here the results are quite disappointing but at least they are consistent. It seems that the program takes increasingly more time when choosing an increasing amount of processors. Which would indicate that it takes a longer time to start up a thread.

We think that this is because we are still using relatively small input sets (1000 to 10000 edges), which also between each other vary greatly in size which could also cause other problems). We chose these datasets because we found them on the Stanford website and though they would be good to start with.

Now that we have come to the impression that we will need more input sets in order to test more correctly we would suggest that we create our own method which will generate input sets in the same format as the input sets used from Stanford University.

Bottlenecks

Because we are using the existing Stanford University input sets which are relatively small, we can conclude from the output that the efficiency of the parallel solution is now not being used at his full capacity. In our future iterations the decision can be made to generate our own input sets by a custom method. These new input sets will have a larger amount of edges compared to the input sets we are currently using.
We expect the solution to be more efficient when processing these larger input sets and show a significant speedup compared to the serial version of the Dijkstra algorithm. Obviously we expect to see the same changes when using more than one processor, two processors should have an higher speed up, three even higher and so on.

Note

Inside our result output we just found out that the completed duration time is not being calculated correctly, instead of calculating this after processing all input sets the program does now calculate it as soon as all threads are started. Because of this the time of the complete duration is not realistic. However it does explain why the total run time seems to take longer when using more processors, since more processors equals the start up of more Threads (with a smaller amount of Graphs). We will address this problem in the next iteration, we could use some hints perhaps.

Communication overhead

There is still no communication overhead since the cores do not have to exchange information with each other, because we are parallelizing on Graph level and each graph is independent of each other (independent information).

Task Granularity

Again the tasks are the graphs themselves, these get granularized. Each Graph is a different task to be executed by a Thread.

Is the expected speedup and efficiency true?

We did expect a better speed up here since we are running 10 graphs with multiple processors (since our system had a maximum amount of 8 processors available, it would mean that the maximum time would be 2 time units). We can also see from the tables and graphs that there is a significant speed up in performance (which luckily indicates that some of our work bears fruit!).

2.3 Threads and Locks, approach 3 Parallelize the nodes

We have also worked on the parallelisation of nodes, which we are still currently working on. We took a break from that since we found it very difficult to implement and visualize and decided to work on Approach 1 and 2. But now that we have completed approach 1 and 2, we think we can use some of their implementation to construct Approach 3 with a clearer vision this time.

We have refactored the code and some methods in order to be able to run Threads for all the nodes (and also for multiple input sets). This code runs our simple TL implementation for all the input sets.

The main method passes on the required variables vertices (the number of vertices) and sourceVertex (which is usually 0)

The main method calls the SingleThreaded Dijkstra class which is basically the same as the original but it extends Thread and in it’s run method it creates a new Graph after which it calls the graph’s run method .

So we have also made the static Graph class extend Thread of course. Here we pass the variables vertices and sourceVertex yet again.

Result analysis

Because we have not fully implemented this approach inside our application and the algorithm is not yet parallelized on node level, at this moment unable to conclude what the result analysis for this approach will be. In future developing and implementing the node level based parallelization we will include the result analysis and compare this to the other approaches. We expect this approach to be the most efficient and provide with the highest speedup of all approaches.

Bottlenecks

A bottleneck here might be that if the the nodes have a very small amount of edges (Say 1, 2 or even 0), there might be a risk that threads will not have work to do (tasks to execute), which will cause a slow down. On the other hand, at this point we do not implement anything of the sort. Which means that it will always help since we only do Graph parallelization now (we are suggesting that both could be possible!).

Communication overhead

Communication overhead is again zero, because the nodes that are being calculated are all going from a Node (and a previous path) that has already been calculated. Since the previous path is known, for each node it is just a matter of adding the old distance to the distance of each new node.

Task Granularity

The task that is granularized here is the calculation from each known node to all the new nodes it’s connected to (so all the edges to new nodes).

Is the expected speedup and efficiency true?

This implementation has not fully been reached yet, this can be an future improvement for the algorithm. We expect the the efficiency and speedup with this approach to be at the maximum capacity compared to the other approaches. This is because the parallelization will be done on node level which we think will provide the maximum amount of efficiency and speedup.

3. Optimal thread and locks

3.1 Data structures:

1. Concurrent Queue
ConcurrentLinkedQueue is an unbounded thread-safe queue based on linked nodes. This queue orders elements as a FIFO (first-in-first-out). we thought this is good when reading the Edges from the txt file since the order is important (the first one is always from the zero node. and the Dijkstra Graph is also calculated starting from the bottom (low source value nodes) to the top (high source value nodes). So in this way this data structure is very compatible with the format of our inputsets and the manner of loading.

2. Concurrent Hashmap
If you're doing lots of reads and writes on it, a ConcurrentHashMap is possibly the best choice, if it's mostly reading, a common Map wrapped inside a collection using a ReadWriteLock (since writes would not be common, you'd get faster access and locking only when writing). Collections.synchronizedMap() is possibly the worst case, since it might just give you a wrapper with all methods synchronized, avoid it at all costs.

ConcurrentHashMap is designed to be more efficient and have better performance in a multi-threaded environment when compared to hashtable. In hashtable concurrency is achieved by obtaining a lock on the entire object for performing a single operation such as put() or get(). ConcurrentHashMap allows concurrent access to the map. Part of the map called Segment (internal data structure) is only getting locked while adding or updating the map. So ConcurrentHashMap allows concurrent threads to read the value without locking at all. This data structure was introduced to improve performance.

The allowed concurrency among update operations is guided by the optional concurrency Level constructor argument (default 16), which is used as a hint for internal sizing. The table is internally partitioned to try to permit the indicated number of concurrent updates without contention. Ideally, you should choose a value to accommodate as many threads as will ever concurrently modify the table. Using a significantly higher value than you need can waste space and time, and a significantly lower value can lead to thread contention. The internal structure of ConcurrentHashMap in Java 8 is different previous java versions. Instead of an Entry there is a Node class. The major difference is that for a given segment if the Node size increases significantly this it changes the structure from a linked list to a TreeNode and optimize the performance. The TreeNode uses red black tree which is a balancing tree and making sure that operations on the tree is in the order of lg(n)

3. Executor service
ExecutorService abstracts away many of the complexities associated with the lower-level abstractions like raw Thread. It provides mechanisms for safely starting, closing down, submitting, executing, and blocking on the successful or abrupt termination of tasks (expressed as Runnable or Callable).
Rather than spending your time implementing (often incorrectly, and with great effort) the underlying infrastructure for parallelism, the j.u.concurrent framework allows you to instead focus on structuring tasks, dependencies, potential parallelism.

Overcomes:
Poor Resource Management i.e. It keep on creating new resource for every request. No limit to creating resource. Using Executor framework we can reuse the existing resources and put limit on creating resources.

Not Robust : If we keep on creating new thread we will get StackOverflowException exception consequently our JVM will crash.

Overhead Creation of time : For each request we need to create new resource. To creating new resource is time consuming. i.e. Thread Creating > task. Using Executor framework we can get built in Thread Pool.

The Java ExecutorService is a construct that allows you to pass a task to be executed by a thread asynchronously. The executor service creates and maintains a reusable pool of threads for executing submitted tasks. The service also manages a queue, which is used when there are more tasks than the number of threads in the pool and there is a need to queue up tasks until there is a free thread available to execute the task.

3.2 Concurrency Patterns:

1. Thread Pool
As of now we have just splitted up the data into to precisely splitted lists in order to be calculated in the most efficient amount of time. However with the thread pool it would just be possible to let the Threads always work when there are Tasks available. This is done in the way that a watcher keeps watching queue (usually BlockingQueue) for any new tasks. As soon as tasks come, threads again start picking up tasks and execute them. This eliminates the preparationary work that we force the system to do every time. and allows us to implement the parallelisation on edge level perhaps.

Real life thread pool, example
Imagine having a facility where 12 people are working. There are 3 sections of this facility. Kitchen, restrooms and security. If you would not use the thread pool technique, this is how it will end up working:
All 12 people will be standing in a meeting room, if new customers come by facility and ask for tasks, then you will separate people in groups and send them to do their work, and come back to meeting room. But, before they go to their duty, there is a preparation phase. They need to wear correct uniform, equip certain devices and walk to that section, finish work and come back. So, once every time they finish their job (thread ends), they need to walk back to meeting room, undress uniform, take out equipment and wait for next job. These refer to creating thread context, it's memory allocation and tracking information by OS. It is too much time consuming for OS to re-organize new thread needs.

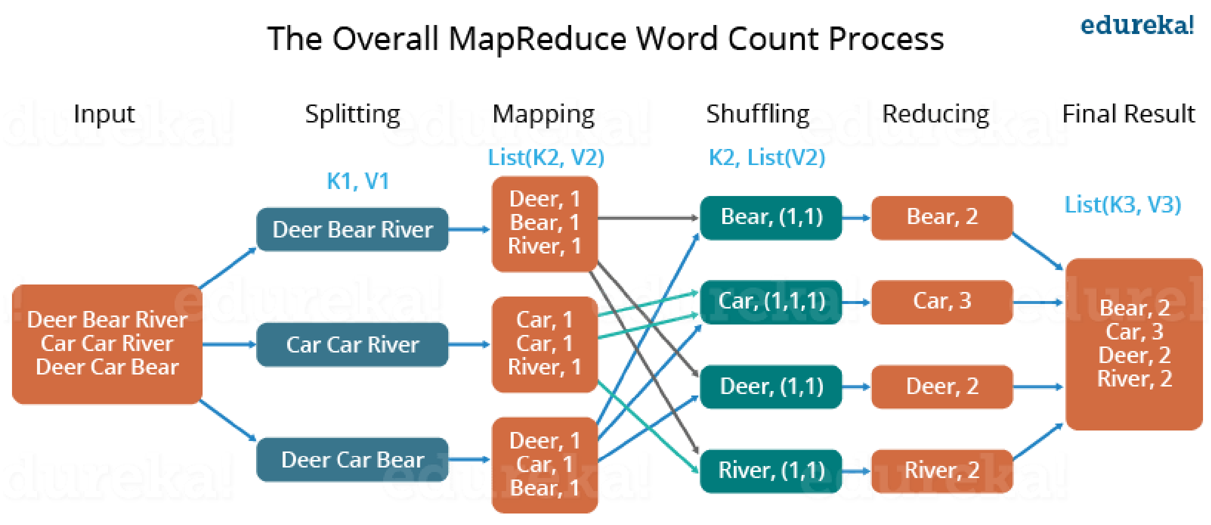
When you would use thread pooling, then the situation would look as following:
In the early morning, you will assign 6 people to kitchen, 2 people to restroom and 4 people to security. So, they will only do their preparation once in a day. Even if there is no customers at the kitchen, those 4 people will be there, idling, for any upcoming tasks. They do not need to go back to meeting room until kitchen closes (application ends). These 4 people are in the Kitchen app pool, and ready to serve quickly. But, you cannot promise they are working all day along, since kitchen may become idle time to time. Same logic applies for restrooms and security as well.

In the first scenario, you do not waste any thread for any task, but keep in mind, it will take good amount of time to prepare every single thread for each task. In the second scenario which is regarding the thread pool concurrency pattern; you prepare the threads in advance, so it’s not guarantee you will be using all threads for all tasks but, OS mostly makes great optimization on it, so you can safely rely on it.

2. Map reduce

MapReduce is a programming framework that allows us to perform distributed and parallel processing on large data sets in a distributed environment.
-So, the first is the map job, where a block of data is read and processed to produce key-value pairs as intermediate outputs.
-The output of a Mapper or map job (key-value pairs) is input to the Reducer.
-The reducer receives the key-value pair from multiple map jobs.
-Then, the reducer aggregates those intermediate data tuples (intermediate key-value pair) into a smaller set of tuples or key-value pairs which is the final output.

We would think here it would be possible to pass as Key-Value pairs, node sources and destination, basically edges (without weight however, since its value is always one in our case). If our datasets would be much larger this would be a favorable data structure to choose.



3. Nuclear Reaction

is a type of computation which allows threads to either spawn new threads or converge many threads to one. We thought of Nuclear Reaction in the context of Dijkstra in the way of having a thread for a Node, creating new threads for each Edge of the Node and afterwards merging to one again. Which is good since often the number of edges is unpredictable. And if we can let this unpredictability be solved by the Nuclear Reaction pattern that would be optimal.

4. Producer / Consumer

In computing, the producer–consumer problem (also known as the bounded-buffer problem) is a classic example of a multi-process synchronization problem. The problem describes two processes, the producer and the consumer, which share a common, fixed-size buffer used as a queue.

- The producer's job is to generate data, put it into the buffer, and start again.
- At the same time, the consumer is consuming the data (i.e. removing it from the buffer), one piece at a time.

The problem, to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

The solution for this will be for the producer to either go to sleep or discard data if the buffer is full. The next time the consumer removes an item from the buffer, it notifies the producer, who starts to fill the buffer again. In the same way, the consumer can go to sleep if it finds the buffer to be empty. The next time the producer puts data into the buffer, it wakes up the sleeping consumer.

An inadequate solution could result in a deadlock where both processes are waiting to be awakened (Mahrsee, 2018).

Consideration and motivation

In the last chapter we have considered three different data structures and three different concurrency patterns. In terms of data structures we have considered concurrent que, map reduce and concurrent hashmap. Below we have documented the pro's and con's of each concurrency pattern and done the same for each data structure.

Pro's data structures	Concurrent Queue	Map Reduce	Concurrent Hashmap	Producer / Consumer
Best suitable for lots of reads and writes.			X	
Allows concurrent threads to read the value without locking at all.			X	
Applies FIFO (First In First Out)	X			
Distributed and parallel processing on large data sets in a distributed environment.		X		
Data Locality		X		
Can be used independently and Concurrently				X
Converts to CSP easily				X
Separating producer and Consumer functionality result in more clean, readable and manageable code.				X
Con's data structures	**Concurrent Queue**	**Map Reduce**	**Concurrent Hashmap**	**Producer / Consumer**
Difficult to implement		X		
Doesn't provide any guarantee over the way the elements are arranged in the Map.			X	
Data may not be changed after initialization.			X	
An empty queue will block the calling thread until the item is ready to be retrieved	X			
Producer and Consumer in some circumstances have to wait until one of the two is finished				X

Pro’s Concurrency Patterns	ThreadPool	Executor Service	Nuclear Reaction
Continuous iteration over available Tasks.	X	X	
Helps you avoid creating or destroying more threads, than would really be necessary.	X		
Large Compatibility with data structures	X		
Merging and splitting threads			X
Efficient Resource Management		X	X
Allows separation between task submission and task execution		X	
Con’s Concurrency Patterns	**ThreadPool**	**Executor Service**	**Nuclear Reaction**
Difficult to implement			X
If the pool size is too large (compared to hardware capacity or to processing needs), there might be an unnecessary switching overhead for unused threads.	X		
The ExecutorService is not automatically destroyed when there are no tasks waiting to be executed, so you have to manually shut it down, you can use the shutdown() or shutdownNow().		X	

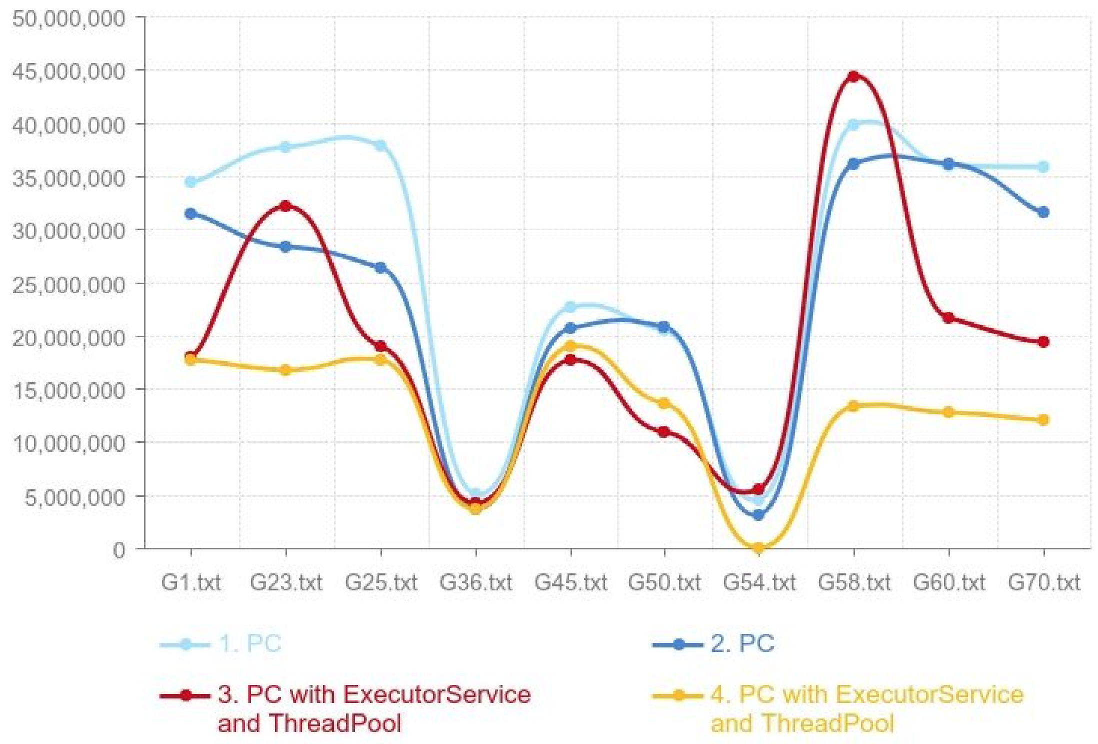
Final choice
After gathering all the pro’s and con’s of both the data structures and the concurrency patterns, and considering all of these features. We found out that the **Executor Service** combined with **Threadpool** and **Producer Consumer** are the most suitable for our solution, we think that ,these will provide us with the most gains in concurrency and improved efficiency. Because of this we have officially chosen to implement the **Executor Service**, **Threadpool** and **Producer Consumer.**

3.3 Chosen Approaches

Benchmark
Below we have documented the results of first the implementation with just our data structure Producer/Consumer, we have included the results two times, to make sure the results are as realistically as possible.
Right next to this, the implementation of the data structure Producer/Consumer combined with the concurrency patterns both Executor Service and Threadpool, again we have included the results two times, to make sure the results are as realistically as possible.

Input set file	Producer/ Consumer	Input set created	Producer/ Consumer	Input set created	Producer/ Consumer with ExecutorService and Thread Pool	Input set created	Producer/ Consumer with ExecutorService and Thread Pool	Input set created
G1.txt	34440416 ns	3697 ns	31423809 ns	4055 ns	17986173 ns	759 ns	17775235 ns	681 ns
G23.txt	37737264 ns	736 ns	28327416 ns	827 ns	32132745 ns	1070 ns	16782310 ns	593 ns
G25.txt	37861170 ns	925 ns	26415528 ns	1915 ns	19028764 ns	654 ns	17657420 ns	558 ns
G36.txt	5068823 ns	206 ns	3744515 ns	198 ns	4203761 ns	658 ns	3730080 ns	628 ns
G45.txt	22691028 ns	175 ns	20665375 ns	207 ns	17638298 ns	3842 ns	18968842 ns	2526 ns
G50.txt	20477291 ns	117 ns	20864529 ns	183 ns	10927554 ns	1150 ns	13659900 ns	189 ns
G54.txt	4501765 ns	208 ns	3086340 ns	207 ns	5492057 ns	687 ns	35930 ns	617 ns
G58.txt	39732464 ns	144 ns	36049441 ns	196 ns	44400883 ns	157 ns	13248647 ns	152 ns
G60.txt	36019312 ns	183 ns	36159044 ns	183 ns	21620718 ns	139 ns	12725694 ns	189 ns
G70.txt	35853653 ns	232 ns	31626864 ns	363 ns	19353473 ns	147 ns	12069266 ns	691 ns
Total runtime	274.383.186 ns	6623 ns	238.362.861 ns	7944 ns	192.784.426 ns	9263 ns	126.653.324 ns	6824 ns

Producer Consumer Speed Graphs
Default vs Pattern and Datastructures



Result analysis

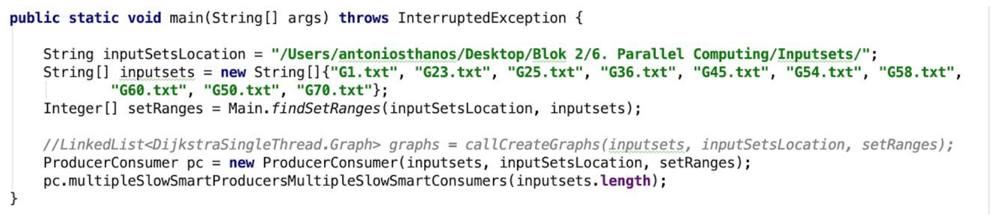
Above we have the outcome shown in a line graph, found on the Y-axis the time in nanoseconds and on the X-axis the input sets which we use for calculation.
As we look at the results of the outcome of the two new implementations, first the Producer / Consumer (PC) concept and second the Producer / Consumer concept combined with Executorservice and Thread Pool and compare these to the implementations of the earlier assignments we are able to conclude that there is a significant speed up in nanoseconds. Compared to the serial solution the speed up is huge, if the trend continues in the direction that the speedup is going, our final solution is a truly worthy and developed result that meets our expectations and even exceeds them at some levels.

Also a couple of oddities would be: The second run of PC with Executorservice for input set G54.txt shows an extremely low number. We find it strange that the first run then has a higher execution time than the regular PC. After checking this again by rerunning the algorithm we can see that it sometimes opts for a much shorter execution of G54 and sometimes one that is just twice as fast. We would try explain this with the phenomenon that the distribution amount of memory to each Thread will differ each time.

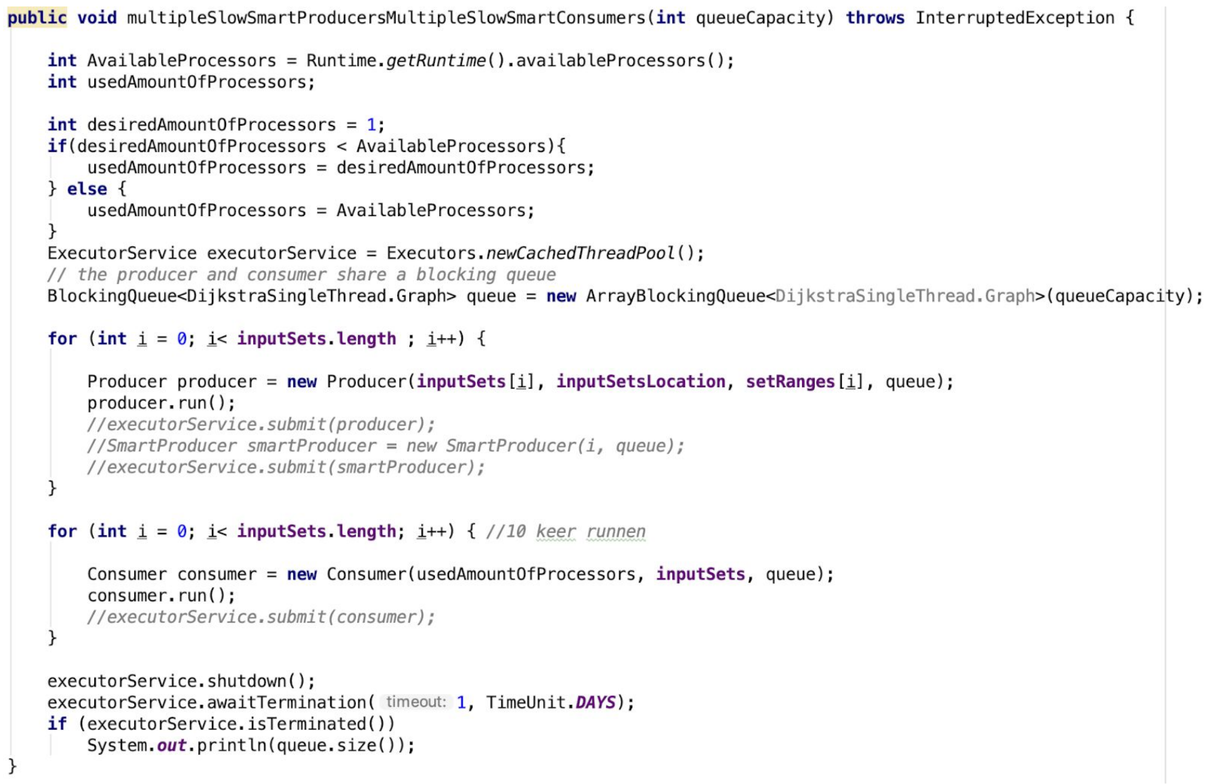
Bottlenecks

It is clear to see from the performance graph that for input sets it does not improve the input set creation time much, so instead the approach by just creating them serially, could be chosen. Perhaps the other processors could be spent in a more efficient manner when Dijkstra would be run alongside other executable code.

Also, as we can see in the output our producer consumer solution it does not execute the preparation task of the producer and the calculation task of the consumer in an alternating fashion.
This could be seen as a bottleneck inside the solution, however this is because the preparation and creation of the input sets for the producer takes less time than the first consumer takes while calculating the outcome of the first input set. Because of this we made the choice to keep the program functioning the way it functions right now.



```java
public static void main(String[] args) throws InterruptedException {

    String inputSetsLocation = "/Users/antoniosthanos/Desktop/Blok 2/6. Parallel Computing/Inputsets/";
    String[] inputsets = new String[]{"G1.txt", "G23.txt", "G25.txt", "G36.txt", "G45.txt", "G54.txt", "G58.txt",
            "G60.txt", "G50.txt", "G70.txt"};
    Integer[] setRanges = Main.findSetRanges(inputSetsLocation, inputsets);

    //LinkedList<DijkstraSingleThread.Graph> graphs = callCreateGraphs(inputsets, inputSetsLocation, setRanges);
    ProducerConsumer pc = new ProducerConsumer(inputsets, inputSetsLocation, setRanges);
    pc.multipleSlowSmartProducersMultipleSlowSmartConsumers(inputsets.length);
}
```



```java
public void multipleSlowSmartProducersMultipleSlowSmartConsumers(int queueCapacity) throws InterruptedException {

    int AvailableProcessors = Runtime.getRuntime().availableProcessors();
    int usedAmountOfProcessors;

    int desiredAmountOfProcessors = 1;
    if(desiredAmountOfProcessors < AvailableProcessors){
        usedAmountOfProcessors = desiredAmountOfProcessors;
    } else {
        usedAmountOfProcessors = AvailableProcessors;
    }
    ExecutorService executorService = Executors.newCachedThreadPool();
    // the producer and consumer share a blocking queue
    BlockingQueue<DijkstraSingleThread.Graph> queue = new ArrayBlockingQueue<DijkstraSingleThread.Graph>(queueCapacity);

    for (int i = 0; i< inputSets.length ; i++) {

        Producer producer = new Producer(inputSets[i], inputSetsLocation, setRanges[i], queue);
        producer.run();
        //executorService.submit(producer);
        //SmartProducer smartProducer = new SmartProducer(i, queue);
        //executorService.submit(smartProducer);
    }

    for (int i = 0; i< inputSets.length; i++) { //10 keer runnen

        Consumer consumer = new Consumer(usedAmountOfProcessors, inputSets, queue);
        consumer.run();
        //executorService.submit(consumer);
    }

    executorService.shutdown();
    executorService.awaitTermination( timeout: 1, TimeUnit.DAYS);
    if (executorService.isTerminated())
        System.out.println(queue.size());
}
```

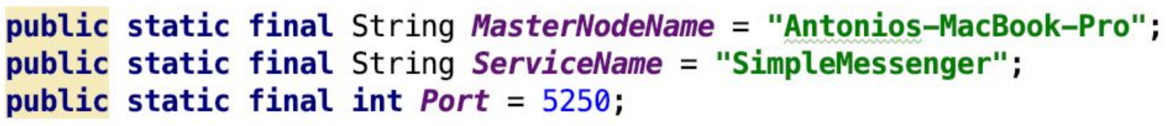
Communication overhead
Since producer creates all the graphs which consumer then has to take to produce the actual result of the graph, here there seems to be a significant communication overhead. For instance Thread 2 can only work after the producer has created it’s second Graph (and for that producer also has to have created it’s first Graph).

Task Granularity
In this case the tasks are slightly different from before since the Graph generation is now also a separate task which is done by the producer. The consumer still as before is the calculating of the shortest distances, which is also a Task for every Graph.

Is the expected speedup and efficiency true?
Yes it is indeed true, since it gives the fastest speed ups as of yet. The results seem much more stable than before as well.

4. RMI

In our example we have started the RMI-concept from the Parallel Computing example with the following variables:



```
public static final String MasterNodeName = "Antonios-MacBook-Pro";
public static final String ServiceName = "SimpleMessenger";
public static final int Port = 5250;
```

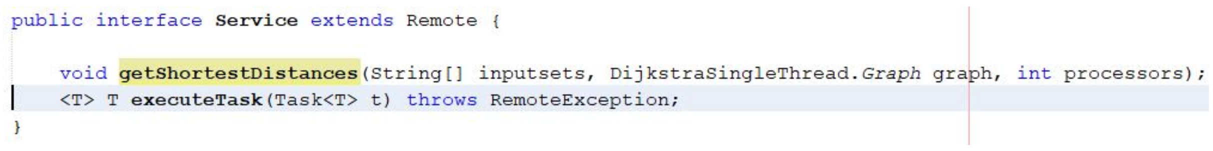
After having tried multiple data types we found out that most of these do not function because we get a hash/Remote Exception. So we decided to keep using the given data type in the example namely: String. We thought we could send the input sets that should be used in the iteration to come. And then splitting the string on the space character to get all the inputsetnames to work with in the current client. After running the program with one of our systems we would get the following output:

This is host:Antonioss-MBP
Connecting to localhost
Client side:G1.txt G23.txt G25.txt G36.txt G45.txt G54.txt G58.txt G60.txt G50.txt G70.txt
SendMessage took 2 ms.

This shows that the Server gets the string and sends it back. Which is necessary for the CSP implementation. After that we thought that the client could create an instance of the ProducerConsumer class which it could then initialize and start by itself. We did this with the following code:

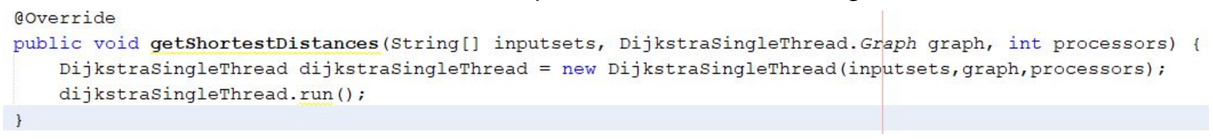
```
String greeting = service.sendMessage("G1.txt G23.txt G25.txt G36.txt G45.txt G54.txt
G58.txt G60.txt G50.txt G70.txt");
String[] inputsetsFromMessage = greeting.split(" ");
Integer[] setRanges = {800,1000,10000,1000,2000,7000,800,};
ProducerConsumer pc = new ProducerConsumer(inputsetsFromMessage,
""/Users/antoniosthanos/Desktop/Blok 2/6. Parallel Computing/Inputsets/",
setRanges);
pc.multipleSlowSmartProducersMultipleSlowSmartConsumers(10,4,4);
```

Now after having done this test we go to our real RMI implementation which is the following:



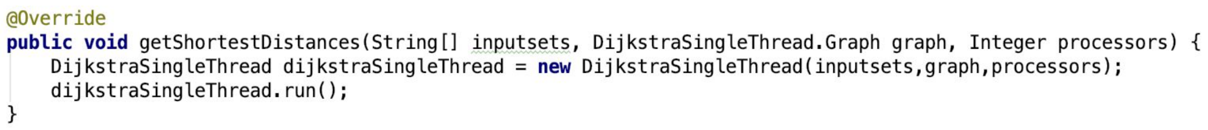
```
public interface Service extends Remote {

    void getShortestDistances(String[] inputsets, DijkstraSingleThread.Graph graph, int processors);
    <T> T executeTask(Task<T> t) throws RemoteException;
}
```

The server is the same as the RMI concept code with the following method added:



```
@Override
public void getShortestDistances(String[] inputsets, DijkstraSingleThread.Graph graph, int processors) {
    DijkstraSingleThread dijkstraSingleThread = new DijkstraSingleThread(inputsets,graph,processors);
    dijkstraSingleThread.run();
}
```

Our client class has the following code which calls for the service method to be executed:



```
@Override
public void getShortestDistances(String[] inputsets, DijkstraSingleThread.Graph graph, Integer processors) {
    DijkstraSingleThread dijkstraSingleThread = new DijkstraSingleThread(inputsets,graph,processors);
    dijkstraSingleThread.run();
}
```

By first running the server and then running our client class it returns us the integer array with the shortest distances from the beginning node (0) to each Node in the graph (the solution of the dijkstra Graph).

We have composed a table of our benchmark results down below:

Benchmark

Input set file	RMI	Input set created	RMI	Input set created
G1.txt	11870874 ns	3437 ns	11731655 ns	2591 ns
G23.txt	21207611 ns	606 ns	11076324 ns	592 ns
G25.txt	12558984 ns	834 ns	11653897 ns	834 ns
G36.txt	2774482 ns	303 ns	2461852 ns	303 ns
G45.txt	11641276 ns	182 ns	12519435 ns	182 ns
G50.txt	7212185 ns	101 ns	9015534 ns	101 ns
G54.txt	3624757 ns	200 ns	23713 ns	200 ns
G58.txt	29304582 ns	139 ns	8744107 ns	139 ns
G60.txt	14269673 ns	293 ns	8398958 ns	293 ns
G70.txt	12773292 ns	187 ns	7965715 ns	187 ns
Total runtime	127237721 ns	6108 ns	83591193 ns	6108 ns

Result analysis
We can see that for RMI we get the fastest results for most of the graphs (except with the PC-executorService sometimes, which is faster now and then). Also the input set creation seems a bit faster.

Bottlenecks
In the current implementation of the RMI approach we don't really see any bottlenecks. In case we combine RMI with CSP and by using the raspberry PI, we expect the Dijkstra's algorithm to possibly suffer from data loss. Mainly because the threads are working with separate memory at this point.

Communication overhead
After looking at the RMI code in depth again, we realized once more that the Client of course depends on the Service class, which needs to run first before being able to execute any sort of task with the client. This is the only way we could see in which there is a significant communication overhead in this implementation.

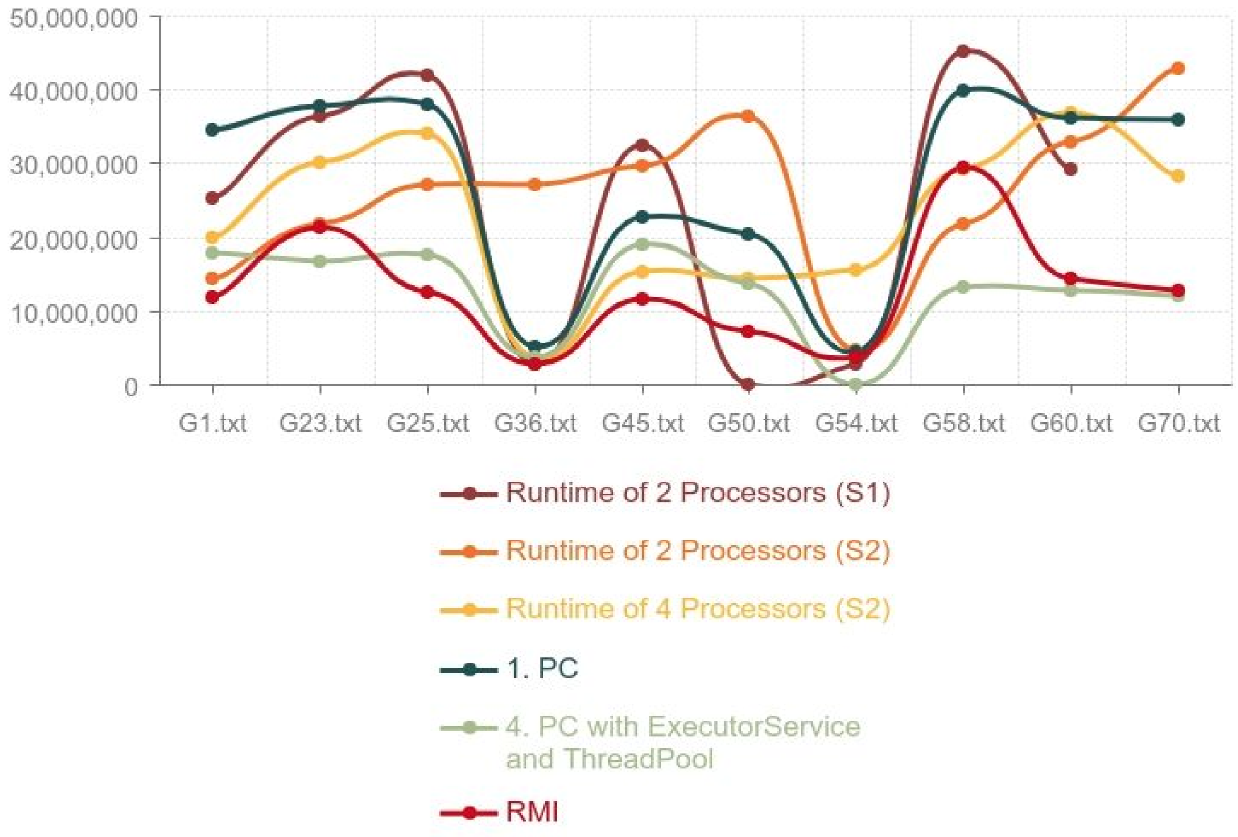
Task granularity
So as we can see the RMI has a server and a client. The server has two tasks, one of creating the thread. And the second one is to run the thread, which calculates our desired distances array. For the client (if we take the creation of the input sets into account) we have the tasks of firstly finding the setranges and creating the graphs. So all in all a total of four tasks. If Node parallelisation was added this could be another task which would account to a grand total of 5!

Is the expected speedup and efficiency true?
If we compare the current achieved speedup from the implementations we have developed before RMI and after this look at the speed up that RMI provides us. The line of positive improvement in the line charts is still rising. Our expectations for this implementation were an postive growing line on terms of speedup and a higher efficiency. This means that our expectation based on speedup and efficiency are indeed reached..

5. Case study conclusion

5.1 Best performing implementation

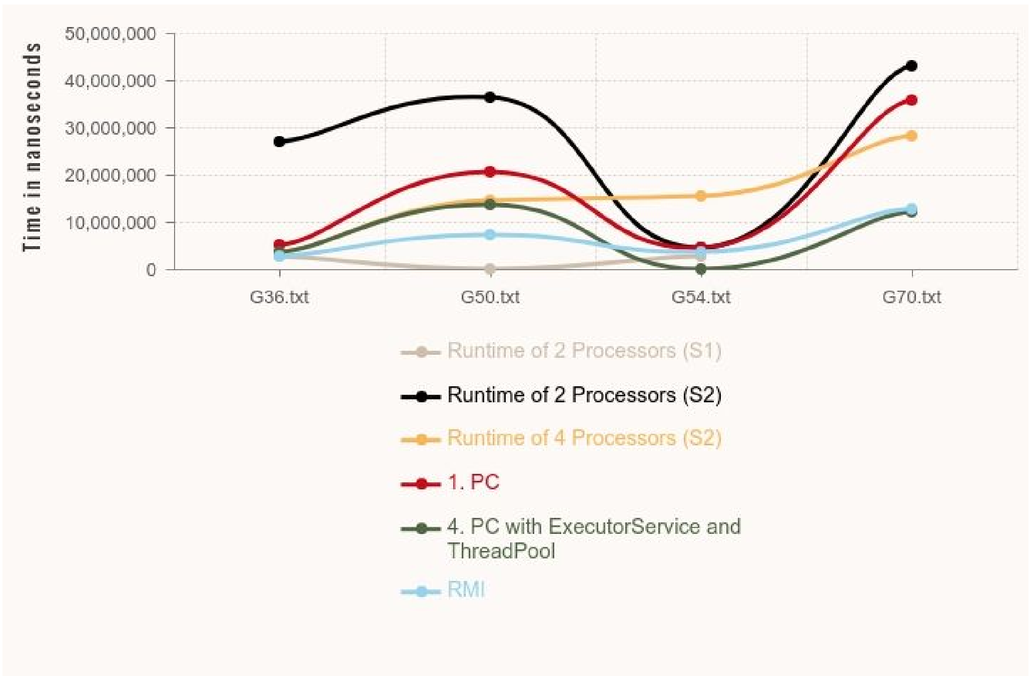
After completing the final approach and collecting all the statistic of the out come, we are able to to compare these results.
If we compare the results of all the different implemented approaches, we can conclude that for as the application states now, the best performing implementation is the RMI approach and as second best the Producer / Consumer combined with the ThreadPool and Executor Service.
Below we have documented the differences of speed in nanoseconds between all completed approaches:



Unfortunately we are unable to conclude with one hundred percent certainty which of the approaches, is the best performing solution. This is because we did not completely succeed the CSP implementation and because of this we do not have the statistics of this approach. This is the reason for the uncertainty in the final conclusion, the conclusion is now based on the current statistics and implementation of the Dijkstra’s algorithm.

5.2 What implementation performs best on what size of problem input

When looking at the performance of all approaches there are some similarities between the speed of the calculation with in the use of certain input sets.
For example if we look at the processing of the input set G36.txt we see similar speed outcomes for pretty much each approach. On the other hand if we look at the G70.txt input set, the speed up difference between the approaches are clearly distinguished.
This is because the input set G36.txt has a total of 11.768 lines of text and a total of 2000 nodes. In comparison to the G70.txt input set which has a total of 10.000 lines and a total of 10.000 nodes. We can conclude out of this situation that the lines of text have no influence on the performance of the approaches.
However the nodes of the input sets have a clear amount of influence on the performance of the approaches, as the G36.txt input set with only 2000 nodes is being almost equally processed in terms of speed in all approaches.



The same situation is being sketched in the outcome of speed of the input set G50.txt and G54.txt. As shown below there is a clear distinguished between the speedup of the approaches for G50.txt, just like the situation above G54.txt speed is quite similar by all the used approaches.
Again if we look at the statistics of the input sets in this situation we see that the amount of lines and the amount of nodes both have different influences on the outcome in terms of speed of the Dijkstra’s algorithm.
As the G50.txt input set has 6000 lines of text and 3000 nodes and the G54.txt has also 6000 lines of text and 1000 nodes.

With this knowledge we are able to draw a final conclusion on the performance part of the implementations. In this case study the best performing solution is RMI, the next implementation would be Producer / Consumer with ExecutorService and ThreadPool, and so on. The performance of dependent on the amount of nodes given in a input set. The larger the amount of nodes, the faster the threaded solution and improved threaded solutions like RMI compared to the serial version.
However when the input set exists of a low number of nodes the serial version and the versions with just basic single thread or multiple thread, do perform better in general compared to for example to RMI solution.
Therefore the amount of nodes of each input set drastically influences the speed of calculation of all distances.

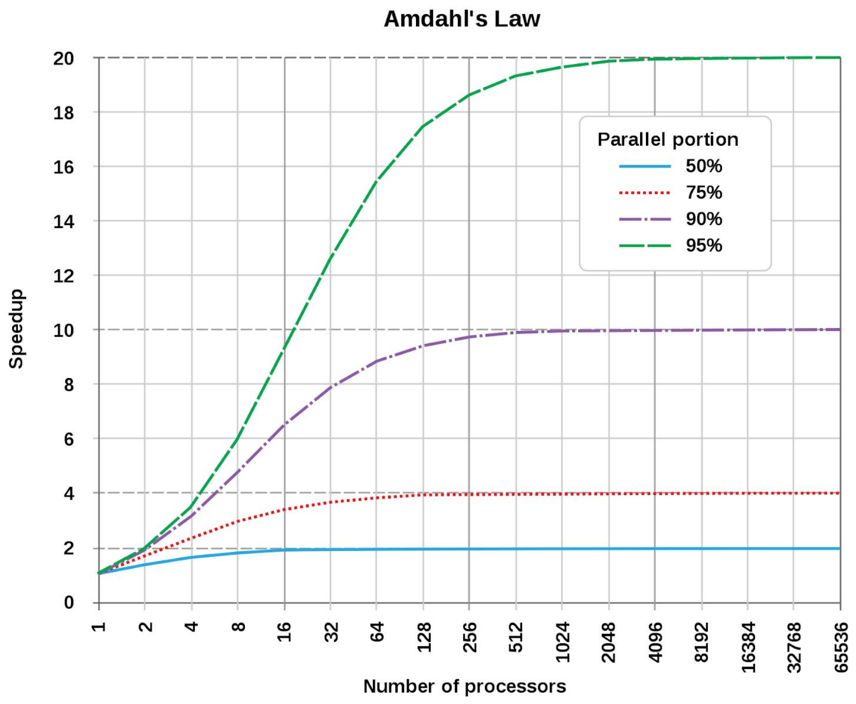
5.3 Scalability

If we look at all the approaches we have used and developed, we can conclude some approaches will have better scalability than others. In order to scale the scalability of the Dijkstra algorithm, it will receive more graphs or more nodes, this will cause a bigger workload for the algorithm.

As the serial approach only uses one processor, just like the single thread approach. Because of this the scalability potential of these approaches is lower.
The scalability potential will grow in the next approaches if we look at 2, 3, 4 or more processors it will be able to handle more graphs, nodes each time, besides this the efficiency of the algorithm grows.

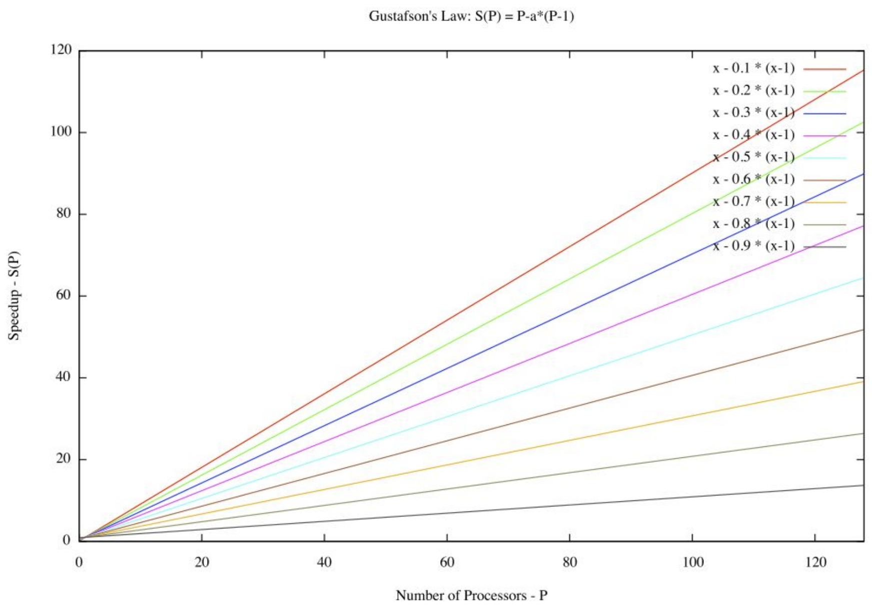
The scalability will grow significant in the following approaches as the Producer / Consumer has an higher efficiency compared to the approaches done before. This efficiency will only grow as the scale increases. Though a higher efficiency can possibly be established by using data structures and concurrency patterns, by knowing this it’s possible to conclude that the scalability potential of the Producer / Consumer combined with Executorservice and ThreadPool approach is even higher than before. This is caused by the Producer / Consumer concept combined with Executorservice and TheadPool as these patterns stimulate a more efficient usage of the algorithm which will result in an speedup pattern, used more efficient and a higher scalability.

In the current state of the application the best approach is the RMI approach, as this maximizes our scalability and efficiency of the Dijkstra’s algorithm. This approach also offers the highest speed up compared to the approached that have been executed before. If we increase the graph or nodes input the RMI approach will grow in efficiency and so increase the potential scalability.



Evolution according to Amdahl's law of the theoretical speedup is latency of the execution of a program in function of the number of processors executing it. The speedup is limited by the serial part of the program.

If we look at Amdahl’s law and use this towards our implementations, it is mostly correct except for the anomalies in our graphs (which are probably due to performance issues, such as having many browser tabs open at the same time).



Gustafson's law addresses the shortcomings of Amdahl's law, which is based on the assumption of a fixed problem size, that is of an execution workload that does not change with respect to the improvement of the resources. Gustafson also seems to apply in most of our results from calculating graphs. Since it usually indicates that using more processors (see previous graphs) increases the speed up by a constant amount. Again we have some exceptions (because of performance variability).

5.4 Task Granularity

As we have seen with most of our approaches the task granularity is pretty similar (they all have calculating of one whole graph as a task). In some production is also a similar task because it fits the specific pattern. Which we would advise because in the real world the time of the whole program is important (which means also the production of the graphs). Lastly node implementation would be a good bonus for the Dijkstra algorithm aside from all the other tasks.

5.6 What implementation seems best

Since we think that using Dijkstra with the largest possible Nodes and edge sizes is the most useful and interesting and thus the important, we think that the node implementation with RMI would seem to us the best implementation (since it performs the best in most cases and shows the best improvement with larg(er) input sets).

5.7 (Generic) Lessons from the case study

The main lesson we have learned and experienced from the executing the case study, is that each layer in the case study offers a new positive speedup and efficiency in the Dijkstra’s algorithm. At the start of the case study we were kind of sceptical regarding the more efficient or speedup the Dijkstra’s algorithm would experience. The results inside the provided graphs actually proves the complete opposite of our skepticism.
As in comparison to the serial implementation, the parallel solution of the Dijkstra’s algorithm already speeds up by an ‘shocking’ average of X ns.

Below the speedups are displayed while being compared based on each approach, for example the regular serial implementation versus the basic parallel approach.
On the Y-axis is displayed the time in nanoseconds and on the X-axis the input sets which we use for calculation.

Also we would definitely advice to think more thoroughly about deciding what kind of input set size to use. And also ALWAYS to run the computer on similar circumstances (don’t leave 15 google chrome tabs open while you are running the tests!).

5.8 Possible future improvements

After everything we see that the input sets are a more integral part of the project than we would have thought at first. So in hindsight we should have probably used a more extensive amount of time in deciding what kind of input sets to use. Our reasoning was that in our point of view in which we have had experience with the Dijkstra algorithm before a graph with a thousand nodes is certainly much larger than we had seen before. However we can see that in order to effectively test the effectiveness of parallelization, large datasets are rather preferable. And our tip would be to any future team working on this assignment to choose LARGE datasets (as long as their computers don’t crash).

We have started implementing the CSP solution and worked with the ActiveMQ approach, unfortunately we did not find the time to complete this solution.

References list

Harsanyi, T. (2018, October 29). Parallel Merge Sort in Java.
Retrieved April 24, 2019, from
https://hackernoon.com/parallel-merge-sort-in-java-e3213ae9fa2c?gi=af4aa7700814

Smaizys, R. (2013, January 2). Bucket sort parallel algorithm using C++ OpenMPI.
Retrieved April 24, 2019, from
https://www.smaizys.com/programing/bucket-sort-parallel-algorithm-using-c-openmpi/

J, S. (2018, August 17). Dijkstra's – Shortest Path Algorithm (SPT) – Adjacency List and Priority Queue – Java Implementation | Algorithms.
Retrieved April 24, 2019, from
https://algorithms.tutorialhorizon.com/dijkstras-shortest-path-algorithm-spt-adjacency-list-and-priority-queue-java-implementation/

Graph and its representations - GeeksforGeeks. (2018, October 4).
Retrieved June 16, 2019, from
https://www.geeksforgeeks.org/graph-and-its-representations/

Index of /~yyye/yyye/Gset. (2003, September 11).
Retrieved May 13, 2019, from https://web.stanford.edu/%7Eyyye/yyye/Gset/

Babu, R. (2009, December 30). Java: How to scale threads according to cpu cores?
Retrieved May 28, 2019, from
https://stackoverflow.com/questions/1980832/java-how-to-scale-threads-according-to-cpu-cores

Paul, J. (2019, March 9). Bucket Sort in Java with Example - How Algorithm Works.
Retrieved June 7, 2019, from
https://javarevisited.blogspot.com/2017/01/bucket-sort-in-java-with-example.html

Mahrsee, R. (2018, September 11). Producer-Consumer solution using threads in Java - GeeksforGeeks.
Retrieved June 9, 2019, from
https://www.geeksforgeeks.org/producer-consumer-solution-using-threads-java/

Ersoy, M. (2012, January 1). Parallel algorithms for shortest path problem on time dependent graphsarallel algorithms for shortest path problem on time dependent graphs.
Retrieved May 21, 2019, from
https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=3&ved=2ahUKEwi4__Hx1KziAhUJ36QKHbm9D4QQFjACegQIBxAC&url=https%3A%2F%2Ftez.yok.gov.tr%2FUlusalTezMerkezi%2FTezGoster%3Fkey%3DcbOXH84ZayrLjc0tI-QXKgbqtDJSNZhpiPXKmdu7KMEqOF5NftCkbVu3i-iMkOSr&usg=AOvVaw1BhA7FNn5wkTR1bw4Bdk_1

Baeldung. (2019, May 7). A Guide to the Java ExecutorService.
Retrieved June 16, 2019, from
https://www.baeldung.com/java-executor-service-tutorial

Paraschiv, E. (2019, May 23). Finally Getting the Most out of the Java Thread Pool.
Retrieved June 16, 2019, from https://stackify.com/java-thread-pools/

Bakshi, A. (2019, May 22). MapReduce Tutorial – Fundamentals of MapReduce with MapReduce Example.
Retrieved June 16, 2019, from https://www.edureka.co/blog/mapreduce-tutorial/

Sehgal, K. (2018, June 3). An Introduction to Bucket Sort.
Retrieved June 16, 2019, from
https://medium.com/karuna-sehgal/an-introduction-to-bucket-sort-62aa5325d124

Gupta, L. (2018, August 6). Java Thread Pool – ThreadPoolExecutor Example.
Retrieved June 16, 2019, from
https://howtodoinjava.com/java/multi-threading/java-thread-pool-executor-example/