DEGREE PROJECT IN THE FIELD OF TECHNOLOGY
INFORMATION AND COMMUNICATION TECHNOLOGY
AND THE MAIN FIELD OF STUDY
COMPUTER SCIENCE AND ENGINEERING,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2017

Distributed Dominant Resource Fairness using Gradient Overlay

ALEXANDER ÖSTMAN

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF COMPUTER SCIENCE AND COMMUNICATION

Distributed Dominant Resource Fairness using Gradient Overlay

ALEXANDER ÖSTMAN

Master in Computer Science
Date: October 4, 2017
Supervisor: Danupon Nanongkai, Amir H. Payberah
Examiner: Mads Dam
School of Computer Science and Communication

Abstract

Resource management is an important component in many distributed clusters. A resource manager handles which server a task should run on and which user's task that should be allocated. If a system has multiple users with similar demands, all users should have an equal share of the cluster, making the system fair. This is typically done today using a centralized server which has full knowledge of all servers in the cluster and the different users. Having a centralized server brings problems such as single point of failure, and vertical scaling on the resource manager.

This thesis focuses on fairness for users during task allocation with a decentralized resource manager. A solution called, *Parallel Distributed Gradient-based Dominant Resource Fairness*, is propoesd. It allows servers to handle a subset of users and to allocate tasks in parallel, while maintaining fairness results close to a centralized server. The solution utilizes a gradient network topology overlay to sort the servers based on their users' current usage and allows a server to know if it has the user with the currently lowest resource usage.

The solution is compared to pre-existing solutions[33, 35, 18], based on fairness and allocation time. The results show that the solution is more fair than the pre-existing solutions based on the gini-coefficient. The results also show that the allocation time scales based on the number of users in the cluster because it allows more parallel allocations by the servers. It does not scale as well though as existing distributed solutions. With 40 users and over 100 servers the solution has an equal time to a centralized solution and outperforms a centralized solution with more users.

Sammanfattning

Resurshantering är en viktig komponent i många distribuerade kluster. En resurshanterare bestämmer vilken server som skall exekvera en uppgift, och vilken användares uppgift som skall allokeras. Om ett system har flera användare med liknande krav, bör resurserna tilldelas jämnlikt mellan användarna. Idag implementeras resurshanterare oftast som en centraliserad server som har information om alla servrar i klustret och de olika användarna. En centraliserad server skapar dock problem som driftstopp vid avbrott på ett enda ställe, även enbart vertikal skalning för resurshanteraren.

Denna uppsats fokuserar på jämnlikhet för användare med en decentraliserad resurshanterare. En lösning föreslås, *Parallel Distributed Gradient-based Dominant Resource Fairness*, som tillåter servrar att hantera en delmängd av användare i systemet, detta med en liknande jämnlikhet jämförande med en centraliserad server. Lösningen använder en så kallad *gradient network topology overlay* för att sortera servrarna baserat på deras användares resursanvändning och tillåter en server att veta om den har användaren med lägst resursanvändning i klustret.

Lösningen jämförs med existerande lösningar baserat på jämnlikhet och allokeringstid. Resultaten visar att lösningen ger en mer jämnlik allokering än existerande lösningar utifrån gini-koefficienten. Resultaten visar även att systemets skallbarhet angående allokeringstid är beroende på antalet användare i klustret eftersom det tillåter fler parallella allokeringar. Lösningen skalar inte lika bra dock som existerande distribuerade lösningar. Med 40 användare och över 100 servrar har lösningen liknande tid som en centraliserad server, och är snabbare med fler användare.

Contents

Chapter 1

Introduction

Resource management is an important component in many computer systems. It can, for instance, be found in operating systems where it manages the amount of memory each program has access to[25]. The core concept of a resource manager is to manage resources with limited availability. Today resource management is not only handling resources in a single machine though. With the increase in popularity of data collection and data analytics, large clusters have been built to store and process the data. These clusters can contain thousand of machines and require resource managers to orchestrate where and when a data processing task should be run and should try to keep the resource distribution fair between its users. Fairness is an important part of resource managers since every user should get the same amount of resources. This thesis will focus on fairness for distributed resource managers for large scale clusters, also called *distributed resource managers*, handling many different types of resources.

1.1 Problem

There are many different problems a distributed resource management have to solve. One existing problem today is the use of a centralized resource manager. This hinders the scalability of the cluster and its availability. One example is YARN[28], which has a max capacity of around 10 000 nodes[11], this hinders scalability of the cluster. A trend in growing cluster sizes has been observed by different sources[2, 34, 32]. One way to solve scalability is using a decentralized algorithm[20]. Systems today though use a centralized solution since

it allows a global view of all the available resources and the different users which enable a *best-fit* algorithm to allocate a task for a user that has the lowest resource usage to the best suitable node. In section 2.1 it can be seen that different fairness algorithms require some global view of system resources which is then also a major advantage of using a centralized solution.

Allocation time is a requirement identified in the development of the resource manager Sparrow[18]. The authors of Sparrow mention that there is a trend for shorter tasks in data analytics frameworks, where a job may finish under 100 ms. Resource managers today may take over one second to schedule a task, which gives a significant overhead compared to the run-time of a task[18]. An issue with sparrow though is that it does not account for heterogeneous resource demand on tasks. It works instead based on slots where a task take up a certain slot with a fixed resource cost.

Another requirement is that allocations in the system should be fair for all users[7]. This means that all users should feel that they received a fair amount of resources compared to the other users of the system. Fairness is a property important in Yarn[28], Borg[30] and Mesos[9].

The problem this thesis aims to solve is how one can create a decentralized resource manager with a focus fairness policy. While also trying to utilize the cluster's resources with heterogeneous resource demands for tasks, with a shorter allocation time in comparison to a centralized solution. The main challenges in this problem are:

- How can one implement a fairness policy without access to a global view of currently allocated tasks and all the users in the system? In section 2.1 it can be seen that most fairness policies require some global knowledge, which may require an approximation to be made of global data.
- How can one minimize the allocation time in regards to cluster utilization. It may require nodes to allocate in parallel.
- How can one ensure scalability of the system? No node, for example, should have to contain a global view of all users which would limit the system based on memory requirements.

1.2 Aim

The aim of this thesis is to evaluate if it is possible to apply a network topology to the servers, such as a *gradient overlay* to create a fair resource manager.

1.3 Limitations

This thesis builds upon the ideas from the papers *Dominant Resource Fairness Heterogeneous*[33] and *Distributed Dominant Resource Fairness (DDRF)*[35]. Both have a limitation that a specific user will not change their resource demand of a task, to create comparable results to these papers, this limitations was applied as well. DDRF also have further limitations to create simulation results: a user will submit an endless amount of tasks and a task does not end. These limitations are applied in this thesis as well, again to create comparable results.

The limitations does not simulate real world scenarios though, and this can be seen as a basic scenario. The limitations help to create easy to compare results between the solutions, and can show if there is potential to further investigate and build upon specficic solutions.

1.4 Contribution

This thesis investigates a solution to the challenges above by building by building upon *Distruted Dominant Resource Fairness*(DDRF)[35]. DDRF creates a directed graph with the servers in the cluster and assigns users to different servers. A user gets a task allocated if the server it is on, has no connections (neighbors) to other servers with a user with a lesser resource share. The proposed solution uses a gradient network topology overlay to create a dynamic directed sorted graph based on their users' resource share. If possible a server only has links to servers with a lower value than itself. This is done to reduce the dependency of the initial graph in DDRF.

To achieve a faster allocation time, it is proposed to allow servers to allocate in parallel. To do this, a server calculates an approximation of the Gini-coefficient (a fairness evaluation method from economics)

based on its neighbors' users. If the Gini-coefficient can be reduced by allocating the task (meaning a more fair result), the task is allowed to be allocated.

The proposed solutions are evaluated based on fairness and allocation time against a centralized solution implementing *Dominant Resource Fairness*[7], a *Sparrow*[18] inspired solution, and *DDRF*.

1.4.1 Achievements

This thesis evaluates four different distributed solutions, two existing: *Distributed Dominant Resource Fairness*, and *Sparrow*, and two proposed solutions: *Distributed Gradient-based Dominant Resource Fairness*, and, *Parallel Distributed Gradient-based Dominant Resource Fairness*. The solutions are evaluated based on fairness and their allocation time (time between a task has been submitted until it is allocated).

It is shown that the proposed solutions both give a better fairness result than the pre-existing ones with the tested datasets. Only one proposed solution, *Parallel Distributed Gradient-based Domiannt Resource Fairness* (PDGDRF) showed potential of being able to have a faster allocation time than a centralized server, and both proposals were beaten in allocation time by the pre-existing solutions. PDGDRF's allocation time is shown to scale based on the number of users in the cluster and passes a centralized solution with 40 users in a cluster with 100 machines. *PDGDRF* is a good candidate to do future work on if one want to have a distributed solution instead of a centralized one, to enable parallel allocations on multiple machines. It can also be a candidate against other distributed solutions if fairness is of more importance than allocation speed.

1.5 Definitions

In this section the definition of different terms are explained, what they mean in the context of this thesis.

1.5.1 Fairness

Fairness in the sense of resource allocation means that all users feel that they have received a fair amount of resources compared to other users. In the simplest case consider of two users, A and B. Both want to allocate tasks with a demand vector of 2 CPUs and 2 GB of ram, $< 2CPU, 2GB >$, for an unlimited amount of tasks, in a system with 12 CPUs and 12 GB of RAM, $< 12CPU, 12GB >$. Both users should then have three tasks running at the same time, giving them $< 6CPU, 6GB >$ at all time, which equalizes their resource usage. Fairness in a system can also be defined by different fairness properties that are useful to look at[7]:

- **Envy-freeness**: a user should not prefer the allocation of another user.
- **Truthfulness**: a user should not benefit by lying about their resource demand.
- **Pareto-efficiency**: it is not possible to increase the allocation of one user, without decreasing it from another user.
- **Sharing incentive**: each user is better of by sharing the cluster with others. Encouraging sharing their cluster.
- **Single resource fairness**: if there exist only one resource in the system, the solution should be reduced to a max-min fairness.
- **Bottleneck fairness**: if there is one resource that is demanded most by every user the solution should reduce to a max-min fairness for that resource.
- **Population monotonicity**: when a user leaves the system, none of the resources of the remaining users should reduce.
- **Resource monotonicity**: if more resources are added to the system, no users allocation should be reduced.

Envy-freeness can be compared to the simple example given above, where one user should not prefer another users allocation. *Truthfulness* and *Pareto-efficiency* is necessary to maximize the amount of tasks that can run in the system. If a user lies about their demand, it will take

resources from another user, while *Pareto-efficiency* means that all resources should be utilized. Lastly *sharing-incentive* is a desirable property in data-center environments, which means that one user should not be better of by keeping a part of the cluster to themselves. The last four properties are considered to be nice-to-have properties for a fairness algorithm[7, 33].

1.5.2 Cluster utilization

Cluster utilization is similar to *pareto-efficiency* mentioned in section 2.1. In this thesis *cluster utlization* means that the resources in the system should be used to its fullest capacity. This will be looked at by checking the ratio between the usage of the system against the total capacity.

$$\overline{utility} = \frac{\overline{usage}}{\overline{capacity}} \tag{1.1}$$

1.5.3 Scalability

Scalability refers to *horizontal scalability* in the cluster. *Horizontal scalability* means that the performance of resource allocation should be dependent on the amount of machines in the cluster. In this thesis, scalability will be looked upon in the context of allocation time, how long it takes to allocate a task, and how many tasks that can be allocated in parallel by multiple machines.

1.5.4 Latency

Latency refers to the time for a task to get an allocation in the system. Latency is measured from the moment a user submits a task to the resource manager until the resource manager has proposed a suitable node and the task have started executing on that node.

1.5.5 Run-time

Run-time refers to when all user tasks have been allocated. In this thesis, all tasks are running endlessly, and users submit an endless amount of tasks. The run-time is, therefore, the time between starting a simulation and when no new task from any user can be allocated to the cluster.

1.5.6 Fault-tolerance

Fault-tolerance is a property a distributed system has if it tolerates node failures while still being operational[8]. In this thesis, fault-tolerance will not be looked at in the perspective of restarting tasks to guarantee task completion. Instead, fault-tolerance will be considered so that new tasks can be submitted to the system.

1.5.7 Heterogeneous resource demand

Heterogeneous resource demand in this thesis means that tasks do not need to want similar resources or resource amounts. A task from user *x* may want 1 CPU, 2 GB of ram and no GPU, $< 1CPU, 2GB, 0GPU >$, while user *y* want $< 4CPU, 1GB, 1GPU >$. This is what is called a *heterogeneous resource demand*, the opposite would instead be a fixed resource demand where every task would receive the same amount of resources, such as $< 4CP, 2GB, 1GPU >$ to cover both users demand.

1.5.8 Allocation tick

In the thesis, allocation tick refers to a time-based interval that ticks each node between a set time. When a node gets an allocation tick, it tries to allocate a task from a user.

Chapter 2

Relevant theory and related work

2.1 Fairness techniques

This section explains *fairness* techniques and how fairness can be implemented in a distributed environment using *dominant resource fairness*.

2.1.1 Max-Min fairness

Max-Min Fairness is called *Max-Min* since it "maximizes the minimum share of a source whose demand is not fully satisfied"[17]. It works by guaranteeing that each user will get at least $\frac{1}{N}$ of the shared resource, where N is the amount of users in the system. If a user has a lesser demand, he/she will only get its requested share, and the users with unsatisfied demands will share the remaining resources.

2.1.2 Asset fairness

Asset fairness is an extension to Max-Min fairness which allows multiple different resource types by assigning each resource type a different weight[19]. One example for this would be a system with two resources, CPU cores and memory. One could set that one CPU core is equal to 2 GB of RAM. This is then reduced to Max-Min fairness. A problem with *Asset Fairness* is that it can violate that each user gets at least $\frac{1}{N}$ of the shared resources. Consider a system with 28 CPU cores and 56 GB of ram, with two users each wanting to allocate as many tasks as possible. User A has a demand of $< 1CPU, 2GB >$ for each

task and user B has a demand of $< 1CPU, 4GB >$ per task. If one weighted the resources at 1 CPU core = 2 GB of ram, one would get the following equation system:

$$\begin{aligned} & max(x, y) \\ & A + B < 28 \\ & 2A + 4B < 56 \\ & 4A = 6B \end{aligned} \tag{2.1}$$

This has the solution: $A = 12, B = 8$, where user A gets 12 CPUs, and 24 GB of ram. User B, on the other hand, gets 8 CPUs and 32 GB of ram. User A does then not get at least $\frac{1}{N}$ of any resource in the system. Asset fairness also break the propeties of *Sharing incentive*, *bottleneck fairness* and *resource monotonicity*[7].

2.1.3 Dominant Resource Fairness (DRF)

Dominant resource fairness (DRF)[7] is a generalization of max-min fairness to allow multiple different resource types. It works by picking the *dominant resource* for each user. For example if user A wants to allocate 1 CPU and 2 GB of ram on a machine with 3 CPUs and 8 GB of ram, the dominant resource would be calculated as following: $< \frac{1}{3}CPU, \frac{2}{8}GB >$, thus the *dominant resource* would be the CPU since $\frac{1}{3} > \frac{2}{8}$. When every user's dominant resource has been found, *max-min fairness* is applied on their *dominant resource*. The implementation of *DRF* differs from *max-min fairness* though in that it does not give partial resources to a user. Every task gets resources equal to their demand vector. *DRF* ensures that the system is envy-free, truthful, Pareto-efficient and have a sharing incentive[7].

2.1.4 Dominant Resource Fairness Heterogeneous (DRFH)

Dominant resource fairness heterogeneous (DRFH)[33] is an extension of *DRF* to handle a large number of heterogeneous servers, since *DRF* only handles a single server in theory. *DRFH* reformulates the definition of *DRF* by defining a cluster of heterogeneous server as: $S = \{1, ...k\}$, and the users as $U = \{1, ..., N\}$. The capacity of a server is defined as $c_l = (c_{l1}, ..., c_{lm})^T, l \in S$, the capacities are normalized based on the total amount of resources in the cluster:

$$\sum_{l \in S} c_{lr} = 1, r = 1, 2, ..., m \tag{2.2}$$

Every user $i \in U$ have a resource demand vector $D_i = (D_{i1}, ..., D_{im})^T$ where D_{ir} is the fraction of the resource demand over the total resource capacity in the system. The global dominant resource of user *i* is then defined as $r_i^* \in \arg\max_{r \in R} D_{ir}$. A user *i's* allocation share on a server *l* is denoted by $A_{il} = (A_{il1}, ..., A_{ilm})^T$. So the number of tasks user *i* can allocate resources for on server *l* is $\min_{r \in R}\{A_{ilr}/D_{ir}\}$. Wei Wang et. al. introduce $G_{il}(A_{il}) = \min_{r \in R}\{A_{ilr}/D_{ir}\}D_{ir_i^*}$ which is the *global dominant share* that user *i* receives from a server *l* under an allocation A_{il}. $G_i(A_i) = \sum_{l \in S} G_{il}(A_{il})$ is then the users global dominant share based on all its allocations in the cluster. From this the problem can be defined as:

$$\begin{aligned} &\max_{A} \min_{i \in U} G_i(A_i) \\ &\text{s.t.} \sum_{i \in U} A_{ilr} \leq c_{lr}, \forall l \in S, r \in R \end{aligned} \tag{2.3}$$

This aims to maximize the minimum global dominant share among all users in the cluster. They prove that the solution to this problem ensures *envy-freeness*, *pareto-optimality* and *truthfulness*. In the implementation, they compared two approximation algorithms to the problem, a first fit solution which allocates resources on the first server that can fit the task. The second solution was a best-fit algorithm that choose the best server based on the heuristic $H(i, l) = \|D_i/D_{i1} - \overline{c}_l/\overline{c}_{l1}\|_1$, where $\overline{c}_{lr}$ is the remaining resources on server *l*. Their experiments showed that the best-fit solution gave the best cluster utilization compared to both first-fit and a slot based scheduler. One major setback of this solution is that it requires a global view of the cluster resources and its users.

2.1.5 Distributed dominant resource fairness (DDRF)

Distributed dominant resource fairness (DDRF)[35] builds up on *DRFH* and tries to solve the problems of it having a centralized server containing a global view. *DDRF* defines an additional set on the *DRFH* model, $U_l \subset U$, $\bigcup_{l \in S} U_l = U$, where U_l are the users on a specific server *l*. The reformulated problem from *DRFH* is then instead dependent on U_l instead of U:

$$
\begin{aligned}
& f(l) = \min_{i \in U_l} G_i(A_i) \\
& \max_{A} \min_{l \in S} f(l) \\
& \text{s.t.} \sum_{i \in U} A_{ilr} \leq c_{lr}, \forall l \in S, r \in R
\end{aligned}
\tag{2.4}
$$

This does require some global resource knowledge, such as the global resource capacity in the system and the global resource allocations of a specific user *i*. Based on (2.4) they show that since $\bigcup_{l \in S} U_l = U$ it gives $\min_{l \in S} f(l) = \min_{l \in S} \min_{i \in U_l} G_i(A_i) = min_{i \in U} G_i(A_i)$. Which results in:

$$
\max_{A} \min_{l \in S} f(l) = \max_{A} \min_{i \in U} G_i(A_i) \tag{2.5}
$$

Equation (2.5) shows that (2.4) then gives the same problem as seen in equation (2.3). Thus it keeps the same properties as *DRFH* such as *envy-freeness*, *pareto-optimality* and *truthfulness*. The main benefit of *DDRF* over *DRFH* is that each server can compute its global dominant share based on its own users.

Qinyun Zhu and Jae C. Oh implementation of DDRF was an approximation algorithm, where each user in the system is assigned to a specific server. Each server in the cluster also has knowledge of a subset of other servers, called neighbors. To allocate a task, a server calculates the dominant share of its users in U_l. It selects the user with the lowest dominant share and checks with its neighbors if it is the lowest among them as well. If it is, the server allocates a task for that user. The result then depends on what neighbors a server have, and what users that are allocated on those servers.

2.2 Fairness evaluation

This section describes different metrics to evaluate the fairness in the system. These metrics are based on measuring income equality in economics but have been used for measuring fairness[35].

2.2.1 Gini-coefficient

The *gini-coefficient* can be used to measure inequality. It is based on the *Lorenz-Curve* which in economics and ecology is used to describe

inequality[4]. The *gini-coefficient* is defined as the area between the *lorenz-curve* and the *uniform distribution line*, also called the *45 degree line*. For unordered data the gini-coefficient can be calculated as follows[3]:

$$G = \frac{\sum_{i=1}^{n} \sum_{j=1}^{n} |x_i - x_j|}{2n \sum_{i}^{n} x_i} \tag{2.6}$$

For testing inequality in a resource manager, each x_i represents a user *i* and is equal to either its total resource allocations, or its current allocations at time *t*. Following on section 2.1.5, using the DDRF model, calculation of the gini-coefficient on their *global dominant share* given an allocation matrix *A* this would equal to:

$$G = \frac{\sum_{i \in U} \sum_{j \in U} |G_i(A_i) - G_j(A_j)|}{2|U| \sum_{i \in U} G_i(A_i)} \tag{2.7}$$

2.2.2 20:20 ratio

The *20:20 ratio* is another inequality test that was considered but later skipped, it is a measure of inequality which is the ratio between the 20% richest in the population in comparison to the 20% poorest. The 20:20 ratio is used by the United Nations Development Programme[21], and is calculated by their total share of income. If $X \subset U$ is the richest 20% and $Y \subset U$ is the poorest 20% the 20:20 ratio based on global dominant share can be calculated as follows:

$$R = \frac{\sum_{x \in X} G_x(A_x) / \sum_{i \in U} G_i(A_i)}{\sum_{y \in Y} G_y(A_y) / \sum_{i \in U} G_i(A_i)} \tag{2.8}$$

If the system is completely fair *R* would be equal to 1, while $R > 1$ indicates unfairness. One downside of the *20:20 ratio* is that it does not account for complete inequality where one user has all the resources. This would create a division by zero, and R would go to infinity.

2.3 Gossip protocols

In peer-to-peer systems gossip protocols are an information spreading mechanism which takes inspiration of rumor spreading in the real world[24]. A gossip is an unreliable and asynchronous message which contains information that may be useful to another node. A gossip

protocol is based on that each node has a set of links to other nodes, called neighbors. A node sends a gossip with information to a neighbor, which that neighbor stores and can send to one of its neighbors. This is the basis of gossip based information dissemination. A simple pseudocode for a gossip based information dissemination protocol can look like the following:

Algorithm 1 Simple push based gossip algorithm

```
loop
    p ← random neighbor
    update ← random known information
    sendUpdate(p, update)
end loop

procedure ONUPDATE(U)
    store U to known information
end procedure
```

Algorithm 1 would run on every node in the network, which would spread information in the network.

2.3.1 Aggregation protocols, Push Sum

Aggregation Protocols are a subset of gossip protocols and can be used to create a summary of data[12]. One example is calculating the average of a value across all nodes or calculating the sum. Some of the benefits of an aggregation protocol is that it is scalable to large systems based on that is has a small message size and sends a low amount of messages per node and provides local access to global data, but it does come with the cost of not providing it in real-time[14].

In this thesis aggregations will be used to compute the sum of values. The sum can be computed by initializing two variables on all nodes, $s_{t,i}$ and $w_{t,i}$ where *t* is the value at a certain timestep and *i* is a node in the network[15]. Each node initialize $s_{0,i} = x_i$ where x_i is their value to be summed, and the initiating node of the aggregating sum set $w_{0,i} = 1$, while the other nodes set $w_{0,i} = 0$. The algorithm for calculating push sum looks as follows:

Algorithm 2 Push Sum aggregation for a single round

procedure PUSHSUMROUND()
  $p \leftarrow$ random neighbor
  sendData(p, $\frac{1}{2}s_{t,i}$, $\frac{1}{2}w_{t,i}$)
  sendData(self, $\frac{1}{2}s_{t,i}$, $\frac{1}{2}w_{t,i}$)
  $s_{t+1,i} \leftarrow 0$
  $w_{t+1,i} \leftarrow 0$
end procedure

procedure ONDATARECIEVE(m)
  $s_{t+1,i} \leftarrow s_{t+1,i} + m.s$
  $w_{t+1,i} \leftarrow w_{t+1,i} + m.w$
end procedure

procedure GETSUM()
  return $\frac{s_{t,i}}{w_{t,i}}$
end procedure

This kind of push sum aggregation in algorithm 2 have been proven with probability at least $1 - \delta$ that the approximation error drops to ϵ in at most $O(\log n + \log \frac{1}{\epsilon} + \log \frac{1}{\delta})$ rounds[15].

2.4 Topologies

A decentralized network can be built using different overlay topologies, and these topologies can for example help increase the robustness of the system to hinder network partitions, but also allow the network to be traversable in search for a particular node.

2.4.1 Random topology

A *random overlay topology* is an unstructured topology which resembles a random graph. A random topology can be created by using a peer sampling service such as Cyclon[31] or Croupier sampling[5]. These algorithms work by creating a partial view of the network for each node, which is called its neighbors. Through gossiping rounds, the nodes exchange a random subset of their partial view with each other to receive a random subset of the graph. A random graph comes with

the property that it is a good expander[6], and thus random walks mix fast[29].

2.4.2 Gradient topology

Gradient Topology can be defined as: For any two nodes p and q that have local utility values *U(p)* and *U(q)*, if $U(p) \geq U(q)$ then $dist(p, r) \leq dist(q, r)$, where *r* is a node with the highest utility in the system and $dist(x, y)$ is the shortest path length between node x and y[26]. This creates a topology overlay where nodes are ordered in descending order from the gradient core, which contains the nodes with the highest utility. The gradient core will be referenced to as the gradient center in the thesis. A gradient topology is built by having some peer sampling service providing random peers to each node. Each node prefers other nodes with as similar utility as possible but higher than itself. This creates the topology. There are ways to speed up the creation of a gradient topology such using the T-Man protocol[13]. The gradient topology enables efficient search for nodes with a high utility value, while still being robust because of its *p2p* nature[23].

2.5 Resource management systems

This section describes some resource management systems in use today. An implementation based on the concepts of *Sparrow* will be used in the thesis to see how it compares to the other solutions. *Yarn* and *Mesos* gives an overview on how resource managers are used today in large clusters.

2.5.1 Sparrow

Sparrow is a distributed resource manager which aims to reduce the latency of scheduling tasks. The motivation behind this is because there is a trend for shorter jobs in data analytics frameworks, where a job may finish in under 100 ms. Their results show that *Sparrow* can provide median response times within 12 % of an ideal scheduler (a scheduler which has full network knowledge and schedules on the first available machine). This is solved by treating each node in the network as a resource manager. A scheduler sends probes to different nodes in the cluster and requests task allocation. If a node has a free

slot, it proposes that slot to the scheduler, in which the scheduler can accept or reject the offer. If a node does not have a free slot, it reserves a place for the task on the node and proposes that slot to the scheduler when available. *Sparrow* does not handle different resource requirements though. Instead, it has a fixed number of slots on a machine, in which each task can be run. This means that different tasks can not have different resource requirements and may result in tasks having more resources than they need. This is one of the challenges to find a solution to in this thesis, allowing heterogeneous resource demand can increase cluster utilization. *Sparrow* also implements fairness policies such as *min-max fairness*. This is done by running *min-max fairness* individually on each node, independently from other nodes. This can be seen as a naive solution though[33], since it violates Pareto optimality. Similar to task demand this is also a challenge for the thesis derived from Sparrow. A non-naive implementation of fairness requires access to a global view of a certain extent.

2.5.2 Yarn

YARN[28] is a resource manager/negotiator which is designed for use in Apache Hadoop. It was created since the usage of MapReduce in Hadoop had shifted from indexation of web crawlers to more complex usage areas which required workarounds. Yarn operates with three main components: *Application master* (AM), *Resource Manager* (RM) and *Node Manager* (NM). The RM runs on a single dedicated machine and is the heart of *YARN*. The RM's job is to mediate resources to the different applications that run in the cluster. This includes allocating containers on worker nodes for different applications that they can use. All resource requests from an application go through the RM. The RM works together with different NM's which are nodes that handle a specific part of the cluster. Each NM keeps track on their workers' available resources and heartbeats them to check for liveness, and this data is then transferred to the RM. Comparing Yarn to the proposed solution in this thesis, the RM and NM's would be replaced with the decentralized system which may both reduce stress and allow increased scalability of the system. Finally, the AM handles the resources for a single application, it handles which tasks should be placed on which machines and send the resource requirements to the RM. For fairness there are implementations of DRF with YARN, one example is Horton-

works implementation[10].

2.5.3 Mesos

Mesos is a platform for sharing a cluster between several different frameworks, such as Hadoop, Elasticsearch, etc. It works by allowing different type of schedulers from frameworks, such as YARN for Hadoop to request resources. These schedulers are connected to a *mesos master*. Mesos is built on the idea that there is no best-generalized scheduler, and the different frameworks handle their own task scheduling. Instead, the Mesos-master can receive resource requests from the schedulers, and propose an allocation on a certain worker to the scheduler. This is similar to the implementation in this thesis where the resource manager trusts the ordering of tasks from the users and does not include a scheduler of its own. A scheduler does not have to send a request though: *Mesos master* works proactively and sends proposals to schedulers using a fairness policy when resources are available. Fairness is implemented in Mesos by having fairness between different schedulers (or frameworks). Mesos fairness policy for multiple resources is based on *dominant resource fairness*, which is relevant for this thesis since Mesos allows the original *DRF* algorithm to be run without any major alternations. Making the thesis implementation proactive is a possible enhancement to make the implementation work together with the Mesos framework.

Chapter 3

Method

3.1 System Model

The system model assumed in the following parts, is similar to those seen in *DRFH*[33] and *DDRF*[35]. In the system, there are different types of resources that can be allocated, this set of resources will be denoted $R = \{1, ..., m\}$. There exists a set of *K* servers called $S = \{1, ..., K\}$, each server with a capacity vector $c_l = \{c_{l1}, c_{l2}, ..., c_{lm}\}$. The total capacity of the cluster is called $C = (C_1, ...C_m)^T$ and can then be computed by the following:

$$C_r = \sum_{l \in S} c_{lr} \tag{3.1}$$

Each server $l \in S$ will allocate a set number of tasks that will run endlessly until the servers capacity is met. There exists a set of users $U = \{1, ..., N\}$. Each user $i \in U$ have a non-heterogeneous demand vector $D_i = (D_{i1}, ..., D_{im})^T$, which is the demand for the tasks that each user will submit to the servers in *S*. The allocations made in the system at a time-step $t = [0, P]$ for a specific user $i \in U$ at server $l \in S$ is denoted as $A^t_{il} = (A^t_{il1}, ..., A^t_{ilm})^T$, all allocations for users at a specific server $l \in S$ will be called $A^t_l = (A^t_{1l}, A^t_{il})^T$, and the allocations for all users at time-step *t* will be denoted $A^t = (A^t_1, ..., A^t_N)^T$. A user $i \in U$ allocation on a server $l \in S$ is increased by adding its demand vector to the allocation vector of server *l* if the total allocation on that server does not exceed its capacity. A users allocation can never be reduced since it is assumed that tasks will never stop running.

$$
\begin{aligned}
&A_{il}^{t+1} = A_{il}^{t} + D_i \\
&\text{s.t.} \sum_{i \in U} A_{ilr}^{t+1} \leq c_{lr}, r \in R
\end{aligned}
\tag{3.2}
$$

3.1.1 Adding dominant resource fairness

The fairness method chosen for this thesis is dominant resource fairness, since it has been proven in a system with several heterogeneous servers to the following properties explained in section 1.5.1: *envy-freeness*, *pareto-efficiency*, *truthfulness*, *single resource fairness*, *bottleneck fairness*, *population monotonicity*, and *resource monotonicity*[7]. This with support for several resources compared to *Max-Min fairness*. *Sharing incentive* is not mentioned in the above properties since it is not well defined yet for a system with multiple heterogeneous servers[33]. This can be compared to *Asset Fairness* which has been proven to break *sharing incentive*, *bottleneck fairness* and *resource monotonicity*[7].

As *DRF* was the selected algorithm to use, each user $i \in U$ needs to have a global dominant share G_i, which is the resource that they have used the most in regards to the cluster capacity. This is calculated by using the method seen in *DRFH*[33] by taking the sum of the dominant share G_{il} that a user has on all servers. The dominant share for a user $i \in U$ on a specific server $l \in S$ can be calculated as follows[33]:

$$
\begin{aligned}
G_{il}(A_{il}^{t}) &= \max_{r \in R} \frac{A_{ilr}^{t}}{C_r} \\
G_i(A_i^{t}) &= \sum_{l \in S} G_{il}(A_{il}^{t})
\end{aligned}
\tag{3.3}
$$

The idea of dominant resource fairness is to achieve *Max-Min* fairness on each user's global dominant resource. To achieve this and allow distributing the algorithm, methods from *DDRF* is used which has a subset of users contained on each server which will be called $U_l \subset U$, $\bigcup_{l \in S} U_l = U$. The maximization problem is then identical to that in *DDRF*[35] which can be seen in equation 3.4.

$$\begin{aligned} &\max_{A^P} \min_{l \in S} \min_{i \in U_l} G_i(A_i^P) \\ &\text{s.t.} \sum_{i \in U} A_{ilr}^P \leq c_{lr}, \forall l \in S, r \in R \end{aligned} \tag{3.4}$$

Equation 3.4 can be explained as that at the last time-step P of the system, the allocation made should have maximized the server that contained the user with the minimum global dominant share.

3.1.2 Difference from the original DRFH

The model differs slightly from *DRFH* in that Wei Wang et. al. assumed that the capacity of the servers is normalized and the demand vector of a user is the fraction of the resource demand over the total resource capacity. Below it is shown that using Wei Wang et. al.'s method gives equal results of the global dominant resource as seen in section 3.1. Their calculation for the dominant resource can be seen in equation 3.5[33]:

$$G_{il}(A_{il}^t) = \min_{r \in R}\{\frac{A_{ilr}^t}{C_r} / \frac{D_{ir}}{C_r}\} \max_{r \in R} \frac{D_{ir}}{C_r} = \min_{r \in R} \frac{A_{ilr}^t}{D_{ir}} \max_{r \in R} \frac{D_{ir}}{C_r} \tag{3.5}$$

In equation 3.5, $\min_{r \in R}\{A_{ilr}^t / D_{ir}\}$ is the maximum number of tasks user *i* can have allocated on server *l*[33]. The global dominant share in equation 3.5 is thus calculated by taking the number of tasks allocated multiplied with the dominant share of the demand vector. Since the demand vector is not heterogeneous, A_{il}^t can be seen as an integer multiple of the demand vector.

$$\begin{aligned} &A_{il}^t = z * D_i, z = 1, 2, ... \\ &\text{s.t.} \sum_{i \in U} A_{ilr}^t \leq c_{lr}, \forall l \in S, r \in R \end{aligned} \tag{3.6}$$

Using the fact that $\min_{r \in R}\{A_{ilr}^t / D_{ir}\}$ gives the multiple z seen in equation 3.6, it can be seen that one gets the same formula as presented in equation 3.3.

$$G_{il}(A_{il}^t) = \max_{r \in R} \frac{z * D_{ir}}{C_r} = \max_{r \in R} \frac{A_{ilr}^t}{C_r} \tag{3.7}$$

This modification was made to make the calculations more similar to the original DRF algorithm[19]. Since it also only uses the original demand and capacity vectors, no values have to be converted by the system. This while maintaining that no allocation on a server can exceed its capacity.

3.2 Problems with Sparrow and DDRF

This section looks upon some of the problems with other suggested distributed fairness protocols before looking at the suggested implementation in this thesis. Firstly let's consider Sparrow[18]. Every server in Sparrow handles the fairness individually by themselves based on what users that have allocated on that server. It is therefore similar to the naive model mentioned by Wei Wang et. al.[33]. In the original Sparrow implementation, servers handled fairness completely by themselves, in this example, let's assume that the servers also have access to a particular user's global dominant share, extending the solution. Let's consider two servers (S1, S2) with two users (U1, U2). U1 has a demand vector of $< 1, 1 >$ and U2 $< 0.1, 0.1 >$. S_1 has a capacity $C_1 =< 1.2, 1.2 >$ and S_2 has a capacity of $C_2 =< 1, 1 >$, with a total cluster capacity of $C_1 + C_2 =< 2.2, 2.2 >$. One way to create a bad allocation can then be made by the following:

Table 3.1: Table showing the steps of a bad allocation with a naive model. Operation explains what happens at that step, G_i is each users dominant share, C_i is each servers current capacity, Q_i is the queue of tasks on a server from each user.

Time	Operation	G_1	G_2	C_1	C_2	Q_1	Q_2
1	U_1 submit on S_1	0	0	$< 1.2, 1.2 >$	$< 1, 1 >$	$\{U_1\}$	{}
2	U_2 submit on S_1	0	0	$< 1.2, 1.2 >$	$< 1, 1 >$	$\{U_1, U_2\}$	{}
3	S_1 allocate U_2 task	0	0.045	$< 1.1, 1.1 >$	$< 1, 1 >$	$\{U_1\}$	{}
4	U_2 submit on S_1	0	0.045	$< 1.1, 1.1 >$	$< 1, 1 >$	$\{U_1, U_2\}$	{}
5	S_1 allocate U_1 task	0.45	0.045	$< 0.1, 0.1 >$	$< 1, 1 >$	$\{U_2\}$	{}
6	U_1 submit on S_2	0.45	0.045	$< 0.1, 0.1 >$	$< 1, 1 >$	$\{U_2\}$	$\{U_1\}$
7	S_1 allocate U_2 task	0.45	0.09	$< 0, 0 >$	$< 1, 1 >$	{}	{}
8	S_2 allocate U_1 task	0.9	0.09	$< 0, 0 >$	$< 0, 0 >$	{}	{}

As seen in table 3.1 an allocation was achieved where one user had ten

times as high dominant resource than the other. The main problem seen in table 3.1 is that S_2 has no knowledge of the existence of U_2 on S_1. To follow the *DRF* algorithm a global knowledge of the user with the lowest dominant resource is needed as well. This enables a server to hold allocating its lowest user in its queue until that user have the global lowest dominant share.

If one looks on *DDRF* implementation instead, it is based on that each server $l \in S$ handles a subset of users $U_l \subset U$. Each server has a set of neighbors S_l, which is random links to other servers in the system. A server $l \in S$ is allowed to allocate for one of its users based on the following requirement:

$$\forall q \in S_l, \min_{i \in U_l} G_i(A_i^t) < \min_{i \in U_q} G_i(A_i^t) \tag{3.8}$$

If a server has the user with the lowest dominant resource, compared to all its neighbors it is allowed to allocate for its lowest user. It can be seen though that this solution also allows the creation of bad allocation scenarios in regards to fairness. Consider a scenario with four nodes: A, B, C, D. That start with their lowest dominant share set to: A = 0.1, B = 4, C = 3, D = 4 and their demand vector is the same as their initial dominant share. Each user only requires a single resource and the total cluster capacity is 18.3. Figure 3.1 shows how this configuration creates an unfair allocation based on this scenario.

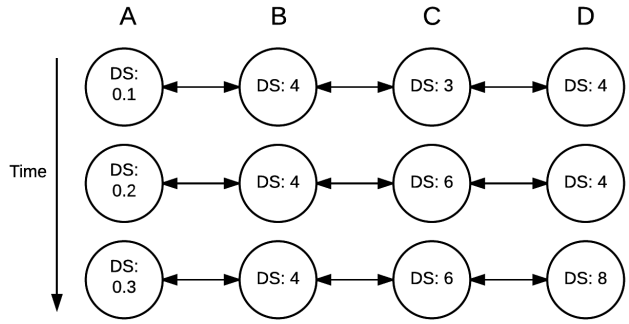

Figure 3.1: A scenario where DDRF returns an unfair allocation. DS represents each nodes lowest users dominant share.

In figure 3.1 it can be seen that the node with the least amount of resources (node A) only got around 1.6 % of the total cluster capacity

with four users. While a centralized solution would give the final result of: {4.3, 4, 6, 4}, where the lowest user receives 21.8 % of the cluster resources. *DDRF* is therefore dependent on the initial cluster configuration, how the servers are linked together. If they would instead be ordered in increasing order:

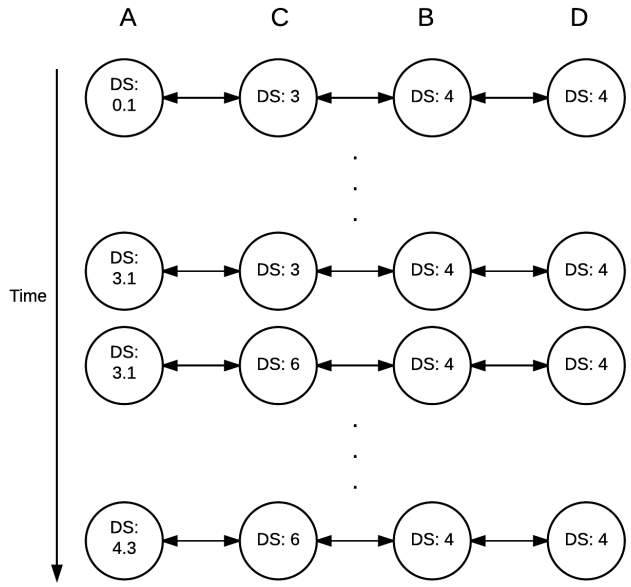

Figure 3.2: Shows when the servers have been sorted in increasing order based on users dominant resource share.

The results from the allocations shown in figure 3.2 is that the user with the lowest cluster usage has 21.8 % of the total cluster resources, same results as with the *DRF* algorithm.

3.3 Distributing DRF with a gradient topology

As seen in section 3.1.1, the idea of the distributed dominant resource fairness is to maximize the server that has the user with the lowest dominant resource. A gradient topology can help solve this problem since it orders the server in a network topology based on a utility function, which can help to locate the server that handles the user with the lowest dominant resource, as also in section 3.2 where sorting the servers resulted in the correct allocations. But firstly let's define what

is needed to be added to the system model. In a gradient topology, each server $i \in U$ requires a set of neighbors which will be denoted S_i^t, which is the neighbors a server has at time-step t. The out-degree of each server will be constant to a predefined number P. Each server $l \in S$ also have a utility function as mentioned which indicates the value of that server, which will be denoted $U(A_l^t)$, which is dependent on the current allocations made on a server l at time-step t.

The gradient topology, as mentioned in section 2.4.2, will order the servers so that for two servers *p*, *q*, with utility $U(A_p^t) \geq U(A_q^t)$, in a fully converged gradient topology, so that the distance (shortest number of hops in the topology) will be $dist(p, r) \leq dist(q, r)$, where r is a node with the highest utility in the system. This ordering is done with a preference function as seen in *Converging an overlay network to a gradient topology*[27], where a server l prefers server *a* over *b* if:

$$\begin{aligned} &(i).\ U(A_a^t) \geq U(A_l^t) \geq U(A_b^t) \text{ or if} \\ &(ii).\ |U(A_a^t) - U(A_l^t)| < |U(A_b^t) - U(A_l^t)| \text{ when} \\ &\quad U(A_a^t), U(A_b^t) > U(A_l^t) \text{ or } U(A_a^t), U(A_b^t) < U(A_l^t) \end{aligned} \tag{3.9}$$

From the neighbor selection function above, it is possible to see how a server can know that it has the highest utility in the cluster. Consider a server *r* which has the highest utility in the cluster:

$$\forall l \in S \setminus \{r\}, U(A_r^t) > U(A_l^t) \tag{3.10}$$

Then the only neighbor selection condition from equation 3.9 which applies is (ii), which will give server *r* a neighbor set S_i^t containing the servers with the highest utility value in the cluster, excluding *r*: $\forall l \in S \setminus \{r, S_r^t\}, \forall n \in S_r^t, U(A_n^t) > U(A_l^t)$. For any other server in the cluster that does not have the maximum utility, condition (i) can be applied which means that it has a neighbor that has a higher utility than itself. Based on this, a server can know that it has the highest utility value in a fully converged gradient topology by checking if it only has neighbors that have a smaller utility than itself, equation 3.11. From equation 3.9 it is is also seen that each server $l \in S_r^t$ have neighbors only to each other and create a complete graph, which will be called the gradient center and will be denoted G^t.

$$\forall l \in S_r^t, U(A_r^t) > U(A_l^t) \tag{3.11}$$

To keep the properties of the *DRF* algorithm, it requires that the user with the lowest dominant resource should allocate for the next time-step. Therefore the utility function is defined so that the server that has the user with the lowest dominant resource should be located in the center. For any server $l \in S$ the utility function looks the following:

$$U(A_l^t) = -\min_{i \in U_l} G_i(A_i^t) \tag{3.12}$$

Equation 3.11 and 3.12 now provides a way to locate which user that should allocate in the system to maintain the properties of *DRF* by having only the user with the lowest dominant resource to allocate.

$$\begin{aligned} A_{il}^{t+1} &= \begin{cases} A_{il}^t + D_i \text{ if } \forall n \in S_l^t, U(A_l^t) > U(A_n^t) \\ A_{il}^t \text{ otherwise} \end{cases} \\ \text{s.t.} \sum_{i \in U} A_{ilr}^{t+1} &\leq c_{lr}, \forall l \in S, r \in R \end{aligned} \tag{3.13}$$

This only covers when the gradient topology is fully converged though, and every server has found its optimal neighbor set. After the node with the maximum utility value has allocated for its user, assume that its utility value becomes lower than all other servers in its neighbor set, $\forall l \in S_r^t, U(A_r^t) < U(A_l^t)$. The servers in its neighbor set S_r^t will now remove *r* as a neighbor in favor of another server. One of these servers, called *q*, will now have the maximum utility function and will create a new gradient center with its neighbor set S_q^t that will contain $S_r^t \setminus \{q\}$, with the addition of a random node until the gradient topology have converged.

The node *q* will be able to check with equation 3.11 that it has the highest utility value and allocate for the correct user. But this is only true while the number of allocations made in the system is less than the out-degree for the servers. If the number of alloations is above the out-degree, and the gradient has not converged, the center might be full of nodes selected at random which cannot guarantee that the correct user is allocated to. The main problem to keep the algorithm consistent is then to always have the correct nodes at the gradient center.

3.3.1 Keeping the gradient center nodes converged

Maintaining the correct servers in gradient center can be done in two ways: wait enough number of cycles to be sure that the gradient has converged, or validate the center correctness with the use of messages between the servers located in the center. Waiting enough of cycles can be difficult because of the random selection of nodes to exchange neighbors with, and can result in either waiting too long until doing an allocation or waiting to short, resulting in allocating the wrong user eventually. To use messages to validate the correctness of the gradient center, the following method is proposed:

1. The center node *r* selects its neighbor node *p* with the lowest utility value.
2. It sends a message msg_r containing the nodes $(S_r^t \setminus \{p\}) \cup \{r\}$ to server *p*.
3. Server *p* compares the nodes in msg_r to its neighbor set S_p^t, if $msg_r = S_p^t$ is true, it is the correct center and it messages back to server *r*.

If one considers when a new server tries to enter the gradient center after the maximum utility server has allocated, it will from before the allocation has had its optimal neighbors. If it is the correct server, which will be called A, that should be included in the gradient center, and it will already have its neighbor set $S_A^t = G^t$. Meaning that the gradient center is already converged. If it is an incorrect server, called B, that is included with utility $\forall l \in G_t \cup A, U(A_B^t) < U(A_l^t)$, it is known that all its neighbors have a higher utility value than itself, $\forall l \in S_B^t, U(A_l^t) > U(A_B^t)$. It is also known that $U(A_A^t) > U(A_B^t)$ and $\forall l \in G^t, U(A_l^t) > U(A_A^t)$. Looking on equation 3.9 it is possible to see, $\forall l \in G^t \setminus A, |U(A_A^t) - U(A_B^t)| < |U(A_l^t) - U(A_B^t)|$, since server A is not included in G^t server B does not have the correct neighbor links to all servers in G^t, and an allocation can not happen until server A is included in the gradient center. This method will be tested with simulations to see how it performs in a more realistic scenario.

Chapter 4

Implementation

The implementation of the system for simulation is done using the Kompics framework[1] which is developed at SICS, which allows the simulation of distributed systems containing thousands of different nodes. Kompics toolbox[16] is also used, which contains different distributed tools such as a gradient topology implementation, and a bootstrap server, both used in the implementation. Kompics toolbox is also developed at SICS. When simulating with Kompics, the simulation time is based on ticks of the framework instead of the actual machine time. This allows the simulation of large networks, without affecting the resulting simulation time based on program complexity. This comes with side-problems though, a complex algorithm running during a single system tick will not be considered in the simulation time. This will therefore be approximated in the implementations with a set time-interval for certain algorithms, this time interval will be called *allocation tick*.

A base simulation setup was used when implementing the solutions. This was setup by having four different type of server nodes: resource nodes, users, bootstrap server, and a gateway node. The resource nodes are the servers which will have tasks allocated on them. The user nodes each represents a user which want to allocate tasks on the cluster. The bootstrap server allows the different resource nodes to locate each other during startup and create a network topology. Lastly the gateway node functions as a redirect node which depending on implementation can direct users to different resource nodes. The gateway can for instance redirect to a single server (centralized), or picking

a server uniformly at random.

The aim of the simulation is to try and get the lowest gini-coefficient based on users dominant share in the system, and maximum lowest global dominant share, when each user will allocate endless amount of forever running tasks with a non-changing demand vector. Each user will therefore send a new task to the system each time its previous task have been allocated. This to not overflow the system with user tasks. A basic pseudo code on this can be seen in algorithm 3. It first requests a server from the gateway node, when the gateway node responds, the user sends its allocation request to the returned server. When that server has found an allocation for the user, it will send a proposal. If the user accepts, the task will be allocated and the server responds with a message informing the user that the task has been allocated. When the task is allocated the user repeat the process with a new task id.

In the following subsections each implementation done will be explained. First a centralized implementation based on the DRF/DRFH paper[7, 33]. A probe implementation is also made, which is inspired by *Sparrow*[18] but with a few changes to make it comparable to the other results. DDRF is implemented without any major changes, but instead focuses on examining how different network topologies can change its results. Lastly two suggested algorithms are explained, the *Distributed Gradient-based Dominant Resource Fairness* (DGDRF) which focuses on mimicking the original *DRF* algorithm in a distributed manner, and the *Parallel Distributed Gradient-based Dominant Resource Fairness* (PDGDRF) which tries to allow parallel allocations by multiple nodes.

Algorithm 3 Pseudo code for task allocation requests for a user $i \in U$.

```
taskId ← 0
acceptedTaskId ← 0

procedure ONSTART
    RequestServerFromGateway()
end procedure

procedure ONGATEWAYRESPONSE(s)
    task ← (user: i, id: taskId, demand: D_i, dominantShare: G_i(A_i^t))
    Send task to server s
end procedure

procedure ONPROPOSAL(s, task)
    if acceptedTaskId < askId then
        Send accept to server s
        acceptedTaskId ← acceptedTaskId + 1
    end if
end procedure

procedure ONTASKALLOCATED(t)
    A_i^{t+1} ← A_i^t + D_i
    taskId ← taskId +1
    RequestServerFromGateway()
end procedure
```

4.1 Centralized implementation

The centralized server implementation, implements two algorithms. Firstly *DRFH* using a first-fit selection algorithm on which server that should handle what task. *First-Fit* was chosen to create comparable results to the distributed solutions, since they also select a server based on a first-fit scenario. First-Fit was also one of the test-cases of the original *DRFH* algorithm. The other implementation is a FIFO based algorithm which allocates the tasks in the order they come into the system. The centralized DRFH solution, will be a base case in the simulations, which shows the resulting fairness, based on full system knowledge. The FIFO based algorithm will instead give a worst-case scenario with

no fairness at all. The network setup of the centralized server can be seen in figure 4.1.

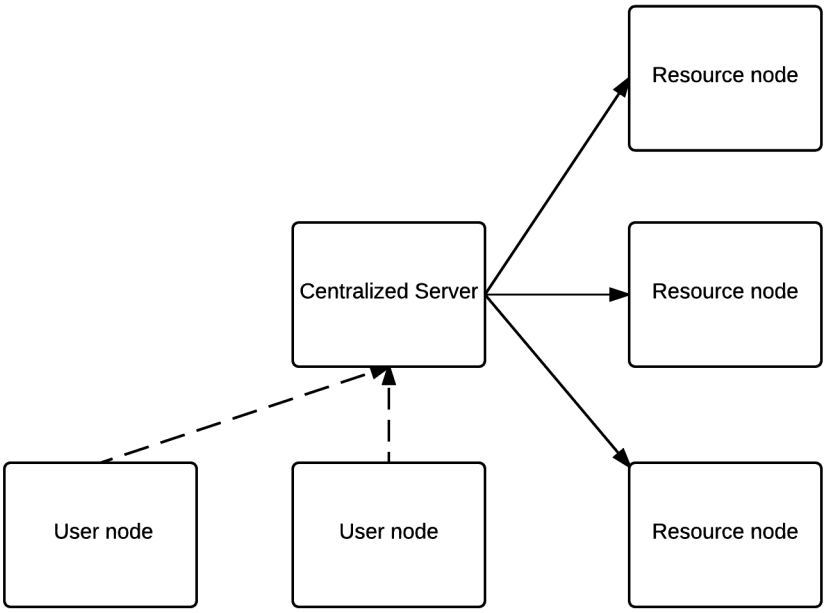

Figure 4.1: Shows the network setup of the centralized server implementation. The users contact the gateway node, to get info about the centralized server, which handles all the allocations.

As seen in figure 4.1 the users have contact to the centralized server. In the implementation each user gets redirected by the gateway node to the centralized server when they want to allocate a task. The centralized server will store the users requested tasks, and select the next task from the user with the lowest dominant share to allocate on a time-based interval (allocation tick interval). This time interval is added to simulate the complexity of finding the user with the lowest dominant share and a suitable server. This time interval also allows all users to send their new task to the centralized server before the next allocation is made.

4.2 Probe-based implementation

This implementation works similar to *Sparrow*[18] with a few additions. Each user gets a resource node uniformly at random each time

it wants to allocate a task from the gateway node and sends its request to that node. In Sparrow, each server handles fairness based only on what allocations that has been made on that node. In this implementation an addition is that users keep track of their own global dominant share, which allows a server to know how much a user has allocated globally. This addition was added since this is assumed to exist for other implementations later on, and may help improve the fairness results. It does not have information about other users that did not send an allocation request though, therefore a server can only handle fairness based on which users sent a request to that specific node.

Every resource node $l \in S$ has a set of fixed neighbors S_l with a static out-degree which is generated at random to create a random graph. A resource node can get information about its neighbors, which contains how much of its resources that are currently in use.

When a user sends a request to a resource node, it stores that request in a list, and that server is the primary handler of that request. Sparrow utilized the power of two choices technique, where it sends a a request to two of its neighbors with the lowest load. Since sparrow was a slot based scheduler, and did not consider heterogeneous resource demand, it could approximate load based on number of slots left, or the size of its request queue. Approximating load with multiple resources is difficult since even though one resource on the server might be completely used, a task may not require that resource. In this simulation, the load will be estimated based on CPU usage though, since the test data in the experiments which is from *Google cluster data*, are CPU heavy[35]. The load is therefore calculated as:

$$load = CPU_{used}/CPU_{capacity} \tag{4.1}$$

Sparrow uses the power of two choices technique to reduce the latency on task allocations, but since the task get propagated to more resource nodes, it can have an effect on fairness as well. A resource node, therefore sends a request to its two neighbors that have the lowest CPU load. The first server that sends its proposal to the user, allocates the task. To ensure that a server gets to see tasks from other servers, a time-interval is added as in the centralized server implementation where only one task is allocated between each time interval. A pseudo code for task allocation can be seen in algorithm 4.

Algorithm 4 Pseudo code that shows the allocation algorithm for any server $l \in S$.

```
list ← []

procedure ONREQUEST(task)
    add task to list
    send task to two neighbors with the lowest load
end procedure

procedure ONTASKFROMSERVER(task)
    add task to list
end procedure

procedure ONTICK
    sort list in ascending order, based on t.dominantShare, t ∈ list
    available = Σ_{i∈U} A^t_{ilr}, ∀r ∈ R
    for each t ∈ list do
        if t.Demand_r < c_lr − available_r, ∀r ∈ R then
            send proposal to t.user
            list ← list \ t
            break
        end if
    end for
end procedure

procedure ONACCEPT(task)
    run task
    send submit message to task.user
end procedure
```

In regards to cluster utilization, the hypothesis is that it will be lower than in a centralized solution, since a resource node can only allocate on itself. If a user is selected to allocate on an already full server, and all its neighbors are full as well, that task and user cannot allocate a task anymore. *Sparrow* was not designed though to have endless running tasks, and have to be taken into consideration when comparing the results.

4.3 DDRF implementation

The *DDRF* implementation was made using Algorithm 1 (algorithm 5 in this thesis), from its paper, DDRF without task forwarding[35]. Which in its paper, received good results in fairness, based on the resulting gini-coefficient.

As seen in section 3.2, dependent on initial neighbor selection for each node, and its allocation condition can be seen in equation 3.8. Since in *DDRF* there is no change in neighbors during runtime, two different setups of selecting neighbors will be made in this implementation. First a uniformly at random selection will be made of all resource nodes in the system. The second neighbor selection implementation will utilize a gradient topology for the initial setup of the neighbors. The utility function seen in equation 3.12 is used, but with slight modifications.

$$U(A_l^t) = \begin{cases} -\min_{i \in U_l} G_i(A_i^t) \text{ if } |U_l| > 0 \\ -\infty \text{ otherwise} \end{cases} \tag{4.2}$$

By using the utility function in equation 4.2, nodes without users will have the lowest possible utility, while nodes with users on them will start with the utility of 0. This will create a network topology where nodes with users will prioritize each other as neighbors. An example can be seen in figure 4.2.

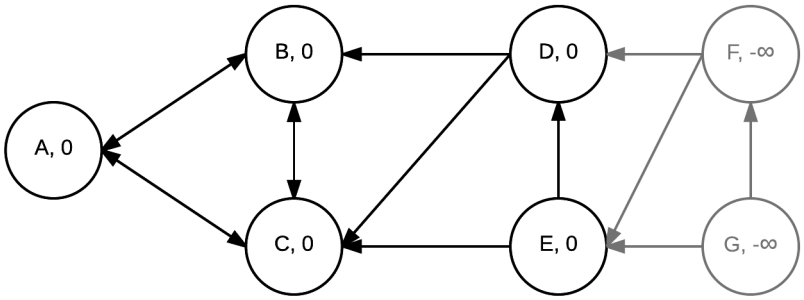

Figure 4.2: Shows a resulting network teopology from a gradient overlay, where each node has two neighbors.

When the gradient has converged, resulting in a similar result as in figure 4.2, the neighbors at that time-step is saved, and is non-changing for the remainder of the simulation. This scenario was created to get

Algorithm 5 Pseudo code that shows the allocation algorithm for any server $l \in S$.

```
users ← [[]]
neighbors ← [∞, ∞, ..., M]

procedure ONREQUEST(task)
    if users does not contain task.user then
        add {user: task.user,tasks: [], dominantShare: 0} to users
    end if
    add task to users[task.user]
    users[task.user].dominantShare ← task.dominantShare
end procedure

procedure ONTICK
    for each i ∈ users do
        t ← get task from i.tasks
        if t.dominantShare ≤ min_{n∈neighbors} n and t.Demand_r < c_lr −
available_r, ∀r ∈ R then
            send proposal to t.user
            i.tasks ← users[i] \ t
            break
        end if
    end for
    for each n ∈ S_l do
        send minimum dominant share update request to n
    end for
end procedure

procedure ONRESOURCEUPDATEREQ(sender)
    neighbors[server] ← dominantShare
end procedure

procedure ONRESOURCEUPDATERESP(server, dominantShare)
    neighbors[server] ← dominantShare
end procedure

procedure ONACCEPT(task)
    run task
    send submit message to task.user
end procedure
```

a good initial neighbor selection where the nodes with users have knowledge about each other, and does not contain neighbor links to servers without users. When the neighbors have been set, the nodes will begin to allocate tasks from users. Each resource node will ask its selected neighbors about their lowest dominant share, at a set time interval, to get updates of the current state of its neighbors.

Similar to the centralized and naive implementation, there is a set time interval between task allocations on a resource node. This to create comparable results to the centralized server. If a resource node cannot allocate a task on its own machine, it will utilize random walk, on an underlying random graph of the network. The node sends the task it cannot allocate to a random neighbor, and if the receiving node cannot allocate it as well, it sends it further. Random walk of tasks does not change the receiving nodes minimum dominant share, random walk is simply a method to allow further cluster utilization. It is not optimal in a real system, but for simulation purposes it allows the cluster to be fully utilized.

By looking at algorithm 5 it can be seen that each node sends a update request to all its neighbors each tick to get their latest information. This creates an overhead in terms of message cost, when comparing to the centralized server. Every node in the cluster does not need to send these requests though: it is only necessary for the nodes that have users on them. The worst case scenario though is that each server in the cluster has a user located on them. This gives the following message cost per tick:

$$O(2|S|P) \tag{4.3}$$

Equation 4.3 is then the number of servers in the cluster times the constant out-degree of each server. It is multiplied by two since each request has to be answered with a response as well. The information necessary in this message is only the minimum dominant share from a servers local users. In this case a double, giving the message 8 byte cost in addition to the transmission overhead. The total cost in bytes then becomes:

$$O(2|S|P * (8 + overhead) \tag{4.4}$$

4.4 Distributed Gradient-based Dominant Resource Fairness (DGDRF)

This section describes the implementation of the first solution proposed in the thesis, *Distributed Gradient-based Dominant Resource Fairness (DGDRF)*. It tries to mimic the *DRF* algorithm by always allocating the user with the lowest dominant share. The implementation follows section 3.3 and section 3.3.1. A user belongs to a single server $l \in S$, and sends all its tasks to that server, as in the DDRF implementation. The server can therefore keep track of the users allocations and its global dominant share. The same utility function is used as in DDRF, equation 4.2, which allow the resource nodes to create a gradient topology where the nodes without any users will not be considered to be near or in the center. The nodes will also be ordered based on their lowest dominant share. The difference to DDRF with gradient neighbor selection, is that the gradient will be used continuously to get a dynamically changing graph.

Each node will at a set time-interval check its neighbors if it has the user with the lowest global dominant share, based on equation 3.11. If the node consider itself to have the user with the lowest global dominant share in the network, it sends a message to its neighbor with the lowest utility value, with descriptors to all its neighbors excluding that neighbor. The receiving node then compares the descriptors to its own neighbors, if all neighbors match it returns an acknowledgement to the sending node, that it can allocate. First the node tries to allocate on itself, if that does not work, random walk is used as explained in the DDRF implementation, section 4.3.

The most important parameters for the *DGDRF solution*, which will be looked upon in the experiments are *view size* and *shuffle period*. The *view size* sets out-degree of a node, meaning how many neighbors it will have in the gradient topology. The *shuffle period* is how often a node will exchange information with one of its neighbors and may change neighbor(s) if a more suitable node is found. The implementation of the gradient topology is as mentioned from SICS, in the Kompics toolbox framework[16].

In the gradient topology implementation, a node will randomly select one of its neighbors to exchange information with. In these messages a node will send information about its current neighbors and their utility value together with its own utility value. The message size is therefore dependent on the number of neighbors. The response message will send an equal size of information, but with information about its own neighbors. If one assumes that the utility is expressed as the minimum dominant share, and an address size to a neighbor is expressed by adr one gets the following cost per message:

$$(P + 1)(8 + adr) \tag{4.5}$$

If each node sends this message x times during one tick, one gets the following cost in bytes including overhead:

$$2 * x|S|((P + 1)(8 + adr) + overhead) \tag{4.6}$$

This is the cost for getting the nodes to receive information about each other, but also to be able to update the gradient topology correctly and locate better suited neighbors.

4.5 Parallel Distributed Gradient-based Dominant Resource Fairness (PDGDRF)

Parallel Distributed Gradient-based Dominant Resource Fairness (PDGDRF) is the second proposed solution and builds upon the previous section. This solution will look upon if the performance can be increased by allowing resource nodes to allocate in parallel. This needs to be done with an approximation since if one follows the DRF algorithm, only one node can allocate at the same time. This means that the only change in performance is dependent on that the resource nodes can have a hypothetical lower load than the centralized server.

The addition to section 4.4 is that nodes will have the possibility to allocate for its users if an approximated gini-coefficient based on neighbors lowest dominant share will be reduced. The idea behind this is, that it is the benchmark that is used for the results. This gini-coefficient is calculated in the following way for any server $l \in S$:

$$Gini = \frac{\sum_{i \in S_l^t} \sum_{j \in S_l^t} |(-U(i)) - (-U(j))|}{2|U| \sum_{i \in S_l^t} (-U(i))} \tag{4.7}$$

This information is already available from the gradient in the form of the utility function, which does not require any new information to be shared. A pseudo code on the allocation conditions can be seen in algorithm 6.

In algorithm 6 it can be seen that there is no constant speed up. Initially each node checks if they consider themselves the center node, the requirement of comparing neighbors to a neighboring node is removed, this to see the potential difference in allowing a non-perfect gradient center. If a resource node does not consider itself the one with the highest utility, it calculates the gini-coefficient at that time *t*, and how it would become if its lowest user would allocate.

Algorithm 6 Algorithm that shows the conditions when a server $l \in S$ can allocate for its lowest user.

```
procedure SHOULDALLOCATE(user)
    if isCenterNode() then
        return true
    end if
    utility ← −G_user(A^t_user)
    giniOld ← calculateGini(utility)
    newUtility ← −G_user(A^t_user + D_user)
    giniNew ← calculateGini(newUtility)
    if giniNew < giniOld then
        return true
    end if
    return false
end procedure

procedure ISCENTERNODE
    for each q ∈ S^t_l do
        if U(q) > U(l) then
            return false
        end if
    end for
    return true
end procedure

procedure CALCULATEGINI(utility)
    N ← {-utility} //N is a list containing utility
    for each q ∈ S^t_l do
        if U(q) ≠ ∞ then
            add -U(q) to N
        end if
    end for
    gini ← calculate gini-coefficient based on N
    return gini
end procedure
```

Chapter 5

Evaluation

The simulation results will be evaluated on four different points: latency for task allocation, fairness based on gini-coefficient, fairness based on the minimum dominant share a user has, and the number of incorrect allocations a solution does in comparison to the original DRF algorithm.

Latency will be evaluated by looking at the final run-time for each allocation. All solutions will have the same time between each allocation tick (100 ms), which is how often a node can allocate a task for a user. The latency is then defined by how fast a solution can find which user that should allocate next and how many tasks that can allocate in parallel. The *DGDRF solution* based on this if it always allocates the correct task (the one belonging to the user with minimum global dominant share), can at most achieve the same speed as the centralized solution. This is because it mimics the *DRF* algorithm and since the allocation tick is the same in all solutions, only one user should allocate each allocation tick when mimicing DRF. So its latency is dependent on how fast it can locate the correct user. The *PDGDRF solution* can instead lower the latency further since it allows parallel allocations by nodes.

Fairness based on the gini-coefficient shows the overall fairness in the system and provides comparable results to the *DDRF* paper[35]. It has some problems though, for instance consider two results with five users having the following global dominant share: (A) 1, 2, 3, 4, 5 and (B) 1, 5, 5, 5, 5. The results from (A) would give a gini-coefficient of

0.267, while (B) would give 0.152. Solution (B) is more fair for the majority of users but it does not accurately reflect that the system is more unfair for a single user.

Fairness based on minimum dominant share is used to evaluate the problem mentioned above. It is also the actual representation on the maximization problem that *DRF* uses. It may therefore be a more accurate way to evaluate the different solutions. Looking at the *gini-coefficient* can still be interesting though since it highlights as mentioned if the majority of users considers it fair. Fairness based on minimum dominant share is also more dependent on cluster utilization. If a solution utilizes more resources, it have the change of getting a higher minimum dominant share. The gini-coefficient is not dependent on cluster utilization and may help bring more comparable results to solutions with different cluster utilization.

Lastly, looking at the number of incorrect allocations provides a way to see a possible correlation between the fairness results and incorrect allocations. Having zero incorrect allocations is important if one wants to keep the original properties in *DRF*.

5.1 Test-cases

The test cases used in the simulations are created from *Google cluster data*[22], in a similar way done in *DRFH* and *DDRF*[33, 35], the dataset with 100 machines and 20 users is identical to *DDRF* supplied by the authors, but the other datasets have only been generated in a similar way. *Google cluster data* contains information about 10 000 machines and their resource capacity, and also 900 users and the different tasks that was submitted to the cluster. X amount of machines are picked at random from the google cluster trace, and Y amount of users. The X amount of machines used in the datasets was: 100 and 200. The different Y values used was: 10, 20, 40, 60, 80, 100 and 120. If a user contains several different tasks, one will be picked at random from that users tasks. This task is set to be allocated constantly by the user, and run endlessly. The workload in the google cluster traces are heterogeneous in terms of resources available on a node, and also the resource demands from a task[22]. The allocation tick time for all solutions and

experiments will be set to 100 ms, meaning that a node can at maximum allocate 10 tasks per second.

The experiments for each solution is run only once, this is because some solutions required an extensive run-time to simulate all the different nodes. All the comparisions in the graph are using the same generated data-set and also the same random seed.

5.2 Fairness

In this section, the fairness is looked at for the different implementations. Both the minimum global dominant share, and the gini-coefficient is looked at to see which solutions that are closest to the centralized DRF solution.

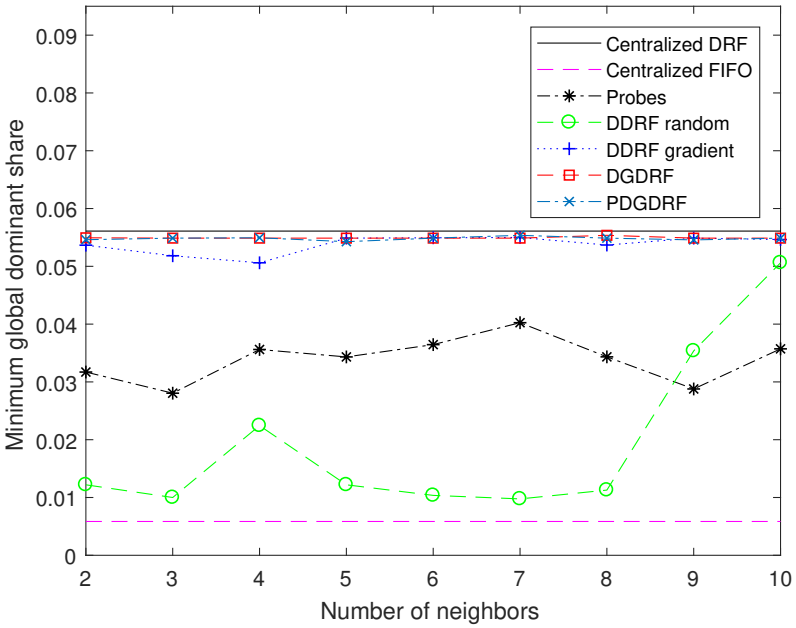

Figure 5.1: The minimum global dominant share, with 100 machines and 20 users. Higher is better.

Table 5.1: The minimum global dominant share, based on the different solutions and the number of neighbors in the network. Displays the same numbers used in figure 5.1

Solution	2	3	4	5	6	7	8	9	10
Centralized DRF	0.0561	0.0561	0.0561	0.0561	0.0561	0.0561	0.0561	0.0561	0.0561
Centralized FIFO	0.0059	0.0059	0.0059	0.0059	0.0059	0.0059	0.0059	0.0059	0.0059
Probes	0.0317	0.0280	0.0356	0.0343	0.0365	0.0402	0.0343	0.0288	0.0357
DDRF random	0.0122	0.0100	0.0224	0.0122	0.0104	0.0098	0.0113	0.0354	0.0506
DDRF gradient	0.0537	0.0518	0.0506	0.0549	0.0549	0.0550	0.0537	0.0549	0.0546
DGDRF	0.0549	0.0549	0.0549	0.0549	0.0549	0.0549	0.0554	0.0549	0.0549
PDGDRF	0.0546	0.0549	0.0549	0.0543	0.0549	0.0554	0.0549	0.0546	0.0549

The results from figure 5.1 and table 5.1 show that all algorithms give a better result than an algorithm based on a FIFO-queue implementation. Using *DDRF* with a random neighbor selection creating a random graph though presents problems when its neighbor count is lower than 9 for this specific dataset. It can almost be compared to a fifo-queue implementation with less neighbors. Using *DDRF* with a neighbor selection using a gradient, to link nodes that have users with each other provides a result almost comparable to the *centralized DRF*. It has nearly the same minimum global dominant share as the centralized DRF server, with some exceptions when reducing the neighbor count to less than 5. The probes implementation results show that it does not seem to be as dependent based on the number of neighbors, and no obvious drop in minimum global dominant share could be observed when the number of neighbors are reduced. This may be because it randomly selects a new node for every task, compared to the DDRF random solution where a user allocates to the same node every time.

The *DGDRF solution* has the second best minimum global dominant share, when looking at all neighbors. The only solution with a better minimum share is the centralized DRF solution. It also has a stable result not depending on the number of neighbors each node has in the gradient topology, if compared to DDRF with a random neighbor selection. The *PDGDRF solution* provides similar results when compared to the *DGDRF solution*. The addition to the *DGDRF solution* to allow a task to be allocated if the gini-coefficient will be improved does not seem to affect the final result much with this data-set when comparing

to the *DGDRF solution*.

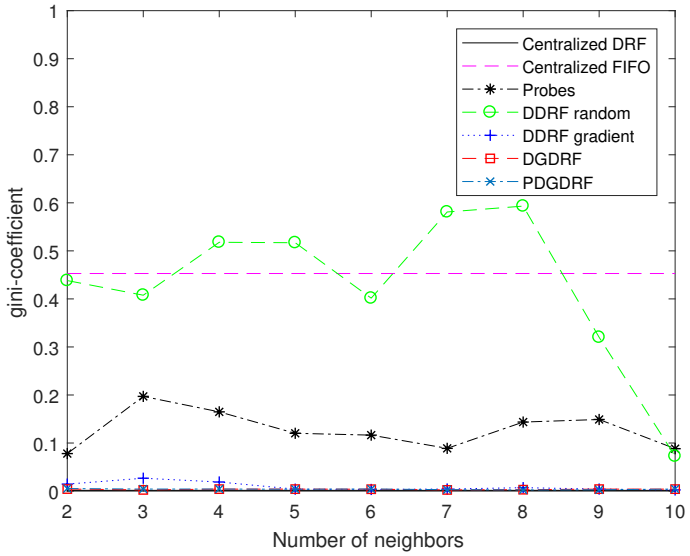

Figure 5.2: The gini-coefficient, with 100 machines and 20 users. Lower is better.

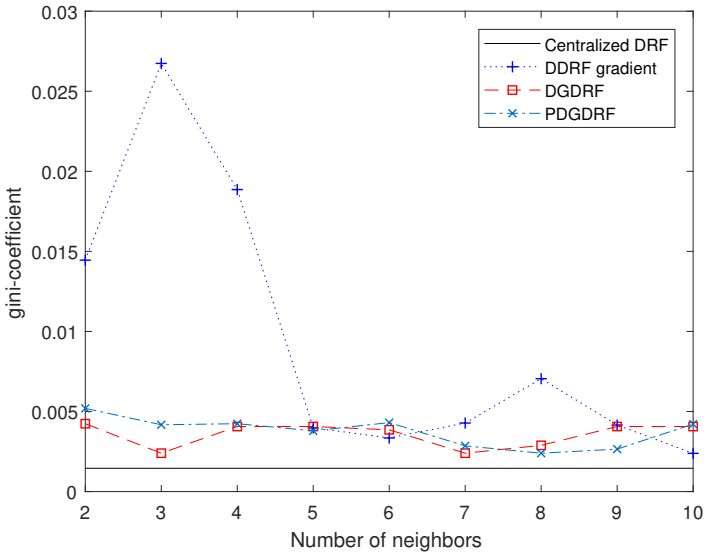

Figure 5.3: Zoomed in version of figure 5.2 on the lower values, with 100 machines and 20 users. Lower is better.

Figure 5.2 instead shows the gini-coefficient for the different solutions

when running 100 machines and 20 users. Figure 5.3 shows a zoomed in version on the four lowest value solutions. An interesting note when comparing the gini-coefficient to the minimum global dominant share is that *DDRF random* is considered more unfair in some occasions than the *FIFO solution*. This is logically the result of a larger standard deviation, since the gini-coefficient result is dependent on the distance between every users dominant share. To investigate this a study on all different solutions standard deviation when they have 6 and 8 neighbors was conducted, see table 5.2. This to check both a scenario when DDRF random has a larger gini-coefficient and one when it is slightly lower. It can be seen that DDRF random have a larger standard deviation than all other solutions with 8 neighbors, and a slightly lower standard deviation than the centralized FIFO with 6 neighbors. Next we will consider how the results change with 200 machines and 40 users.

Table 5.2: The sample standard deviation on the users final dominant share with 100 machines and 20 users

Operation	Sample standard deviation	
	8 neighbors	6 neighbors
Centralized DRF	0.000173	0.000173
Centralized FIFO	0.054938	**0.054938**
Probes	0.015692	0.011700
DDRF Random	**0.095630**	0.054062
DDRF Gradient	0.000748	0.000369
DGDRF	0.000333	0.000488
PDGDRF	0.000374	0.000483

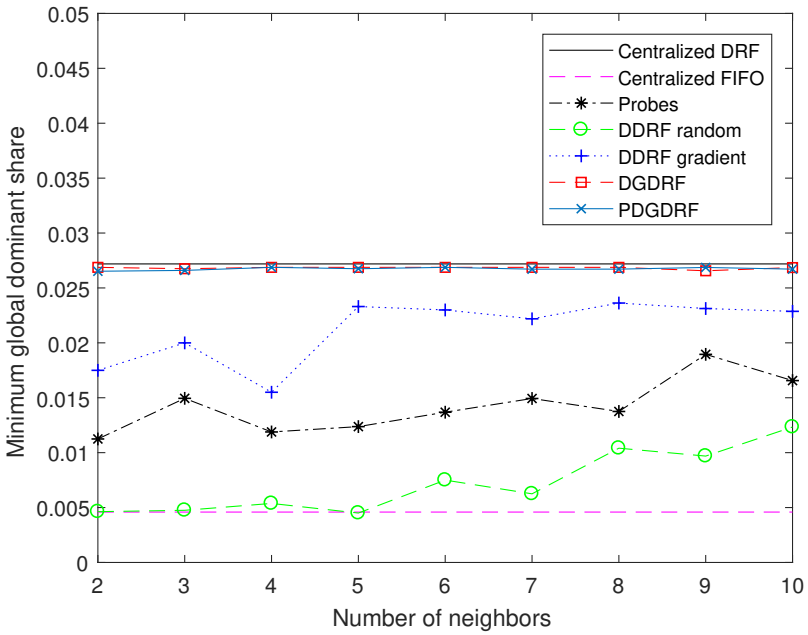

Figure 5.4: Results of looking at the minimum global dominant share, with 200 machines and 40 users. Higher is better.

Figure 5.4 shows the minimum global dominant share for the different solutions when having 200 machines with 40 users. Compared to figure 5.1 with 100 machines and 20 users, the DDRF gradient solution does not give similar results to the *DGDRF solution* or *PDGDRF solution*. The probe and DDRF random solution seem to largely unchanged, except that *DDRF random* have a lower value with 10 neighbors then before. The DDRF random solution gives a better result with more neighbors, and the probe solution does not seem as affected by the number of neighbors, having only a slightly lower minimum global dominant share with fewer neighbors. The *DGDRF solution* and *PDGDRF solution* still seem unaffected by the number of neighbors.

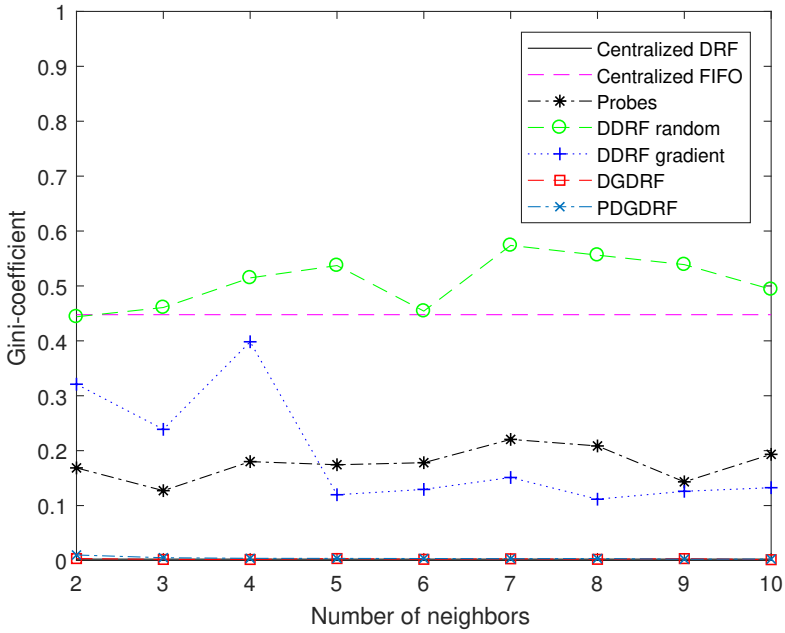

Figure 5.5: Results of looking at the gini-coefficient, with 200 machines and 40 users. Lower is better.

Looking at figure 5.5 which shows the gini-coefficient for 200 machines and 40 users, a larger difference can be seen compared to 100 machines and 20 users, where DDRF random is now considered to be more unfair in all cases than the centralized FIFO solution. DDRF gradient is also considered more unfair with neighbor count 4,3 and 2, compared to the probes solution. The *DGDRF solution* and *PDGDRF solution* remain similar to previous data-set, both with a low gini-coefficient.

From looking at these two data-sets it can be seen that two solutions, *DGDRF solution* and the *PDGDRF solution* both provide more fair allocations for all users when comparing to other distributed solutions.

5.3 Error

This section looks at the errors made by the solutions. An error is when a users task get allocated when that user did not have the lowest global dominant share. This results are looked at to see how well *DGDRF* can mimic the DRF algorithm, and also if having errors have large effects

on the fairness results seen in section 5.2.

Table 5.3: Percentage of wrong allocations made by each solution, with 100 machines and 20 users.

Neighbors	Probes	DDRF random	DDRF gradient	DGDRF	PDGDRF
2	83.53%	86.71%	79.00%	0.00%	38.45%
3	83.05%	86.32%	71.74%	0.00%	22.28%
4	79.64%	85.91%	61.35%	0.00%	15.73%
5	79.22%	89.23%	48.75%	0.00%	16.05%
6	75.89%	88.42%	23.92%	0.00%	17.56%
7	76.26%	86.41%	26.03%	0.00%	20.45%
8	80.00%	88.89%	15.17%	0.00%	20.45%
9	75.86%	82.96%	9.57%	0.00%	22.41%
10	75.28%	74.51%	10.19%	0.00%	23.30%

Table 5.3 shows the percentage of errors for the different distributed solutions with 100 machines and 20 users. For the probes solution, it does not seem like the number of neighbors affects the error percentage much. The DDRF random solution only shows slightly less errors with more neighbors, the number of errors does not seem to strongly correlate with the minimum global dominant share for this solution. The DDRF gradient solution also has a high error percentage at lower amount of neighbors, even though it had a high minimum global dominant share, the error percentage seems to have a correlation with the number of neighbors though where it drops rapidly with more neighbors. The *DGDRF solution* had a zero percent error, non dependent on the number of neighbors in this data-set. The *PDGDRF solution*, has an error around 20% for most number of neighbors, except at 2 neighbors where it almost reaches 40%.

To see how the DDRF gradient solution and the *PDGDRF solution* can still have a high minimum global dominant share, with a relatively high error percentage, the error ratio will be plotted over the number of allocations. This to see where a majority of the errors come from during the run-time execution. It will also look on if the number of errors are reduced, if a users' dominant share is only 1 % or less from the global minimum. The results can be seen in figure 5.6.

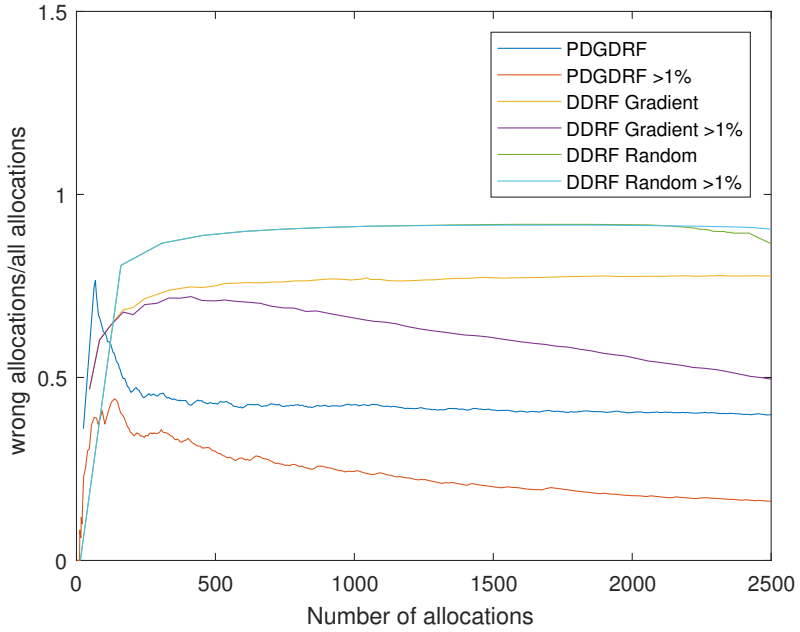

Figure 5.6: The error ratio over the number of allocations made, with 100 machines and 20 users. Each server has two neighbors. $> 1\%$ means that users with a minimum share only 1% from the global minimum is not considered as an error.

Figure 5.6 shows that the error ratio for the DDRF gradient goes up to around 79% quite quickly and does not reduce the error ratio when the cluster becomes saturated. If one looks at tasks from users that are only 1% of the global minimum though, the DDRF gradient solution has a decline in error ratio with more allocations. The *PDGDRF solution* has a similar curve, where it begins with allot of errors and the number of errors are reduced further on. This can be compared to DDRF random which had a low minimum global dominant share. It does not have any reduction in error, even if allowing tasks that are 1% of the global minimum to be considered correct.

Table 5.4: Percentage of wrong allocations made by each solution. 200 machines and 40 users

Neighbors	Probes	DDRF random	DDRF gradient	DGDRF	PDGDRF
2	86.32%	95.57%	90.30%	0.00%	64.01%
3	82.66%	95.57%	89.46%	0.00%	48.35%
4	83.78%	94.86%	76.80%	0.00%	37.80%
5	84.35%	94.93%	69.85%	0.00%	30.14%
6	81.95%	94.83%	67.38%	0.00%	23.34%
7	79.21%	94.28%	64.96%	0.00%	17.65%
8	83.13%	88.23%	62.43%	0.00%	11.67%
9	81.86%	85.47%	57.21%	0.00%	5.69%
10	82.52%	81.94%	55.68%	0.00%	5.04%

Table 5.4 shows the wrong allocations made in ratio to total number of allocations with 200 machines and 40 users. The probes solution as before does not seem to depend on the number of neighbors, with no significant drop in error ratio with more neighbors. Both DDRF random and DDRF gradient solutions now show a clear drop in error ratio with more neighbors. For the DDRF gradient solution, the error rate is increased by 62% when going from 10 neighbors to 2. The *DGDRF solution* still results in 0% errors, while the *PDGDRF solution* has had an increase in errors for fewer neighbors when compared to before, but also a decrease for more neighbors. This is interesting since it still gives similar minimum global dominant share, compared to the *DGDRF solution*. The DDRF gradient solution with 10 neighbors also has a lower error percentage compared to the *PDGDRF solution* with 2 neighbors, even though the *PDGDRF solution* resulted with a better minimum global dominant share, and gini-coefficient. These two results will be compared by looking at how their error rate changes when allowing users that have a global dominant share close to the minimum global dominant share to not count as an error. The solutions will be looked at for 1%, 5%, 10% and 15% over the minimim global dominant share, to not count as an error.

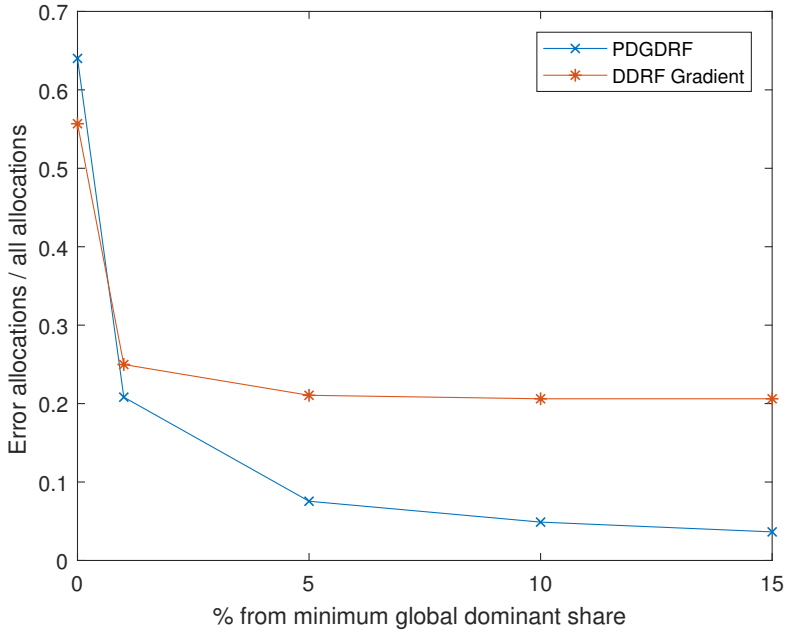

Figure 5.7: The error ratio over the number of allocations made, with 200 machines and 40 users. *PDGDRF solution* has two neighbors and DDRF gradient 10 neighbors. The x axis is how much above the minimum global dominant share a user can be to not have its task counted as an error.

As seen in figure 5.7 the DDRF gradient solution has a larger error ratio than the *PDGDRF solution* when looking at tasks from users being allocated close to the minimum global dominant share. This may then be why the *PDGDRF solution* can have a better fairness result, even though it has more incorrect allocations if one follows the original DRF algorithm.

From all results it could be seen that only looking at errors in the definition of the DRF algorithm could not explain certain behaviors of the solutions. Even though the *PDGDRF solution* had a larger amount of errors than other solutions it could still give a good final fairness result, both considering minimum global dominant share and gini-coefficient. The fairness is also a result of the type of errors.If a user with a dominant share twice as large as the minimum is allocated, gives a much larger impact to the result, than if it is only 1% larger than the minimum global dominant share.

5.4 Latency / Run-time

This section looks on the latency and run-time for the different solutions and how fast they can allocate the cluster until it becomes saturated. The solutions that will be looked upon will be the centralized DRF, Probes, DDRF gradient, *DGDRF solution* and *PDGDRF solution*. The centralized FIFO will not be looked upon since it does not implement any type of fairness algorithm.

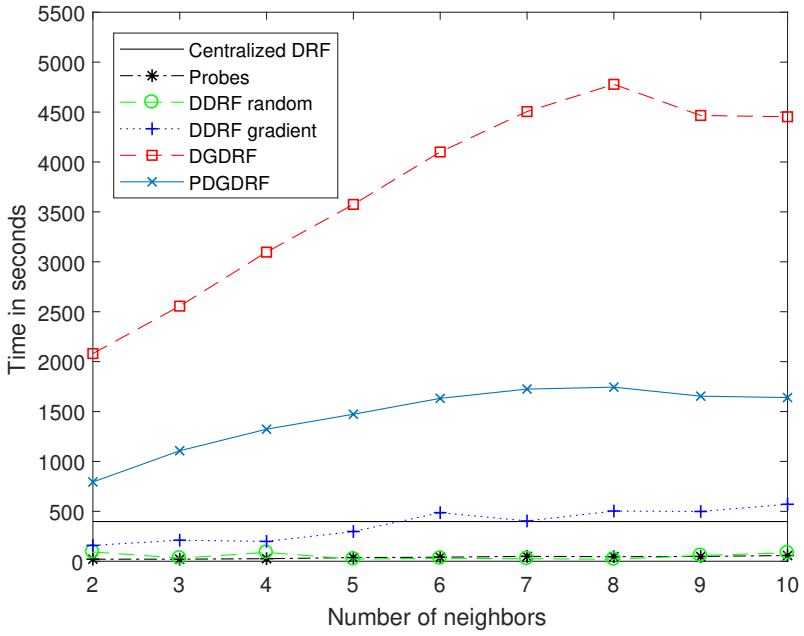

Figure 5.8: Run-time for the different solutions with 100 machines and 20 users. Lower is better.

Figure 5.8 shows the execution time for the different solutions with 100 machines and 20 users. It can be seen that the *DGDRF solution* has the worst run-time followed by the *PDGDRF solution*. The *probes solution* and *DDRF random* both have a run-time lower than the *centralized DRF solution*. This is to be expected because they both result in a worse fairness than the centralized solution, and allow parallel nodes to allocate. The *DDRF gradient solution* has a slightly higher run-time than the centralized solution when the neighbor count is 6 and above. The interesting point from this graph is that both the *DGDRF solution* and *PDGDRF solution* both have a smaller run-time with fewer neigh-

bors. The *PDGDRF solution* with two neighbors, has a run-time of 794 seconds, compared to *centralized DRFs'* 398 seconds. The increase of run-time with more neighbors in these solutions is most probably because a node only sends its updates to one of its neighbors. This can be compared to the *DDRF* solutions which sends updates to all its neighbors.

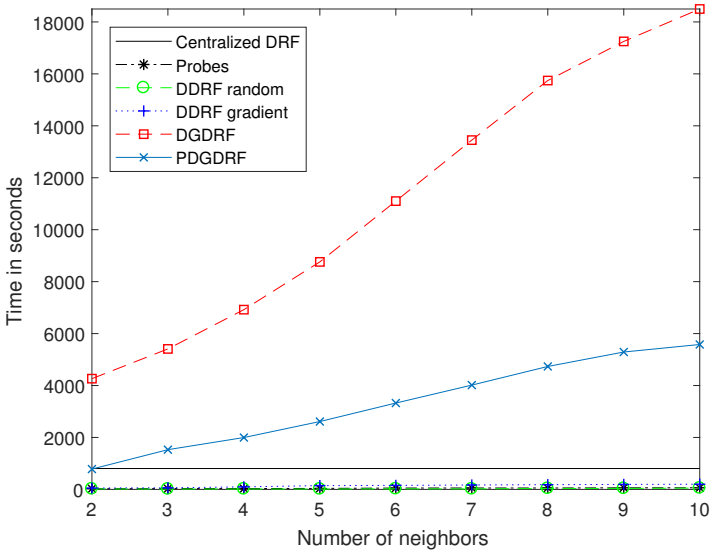

Figure 5.9: Run-time for the different solutions with 200 machines and 40 users. Lower is better.

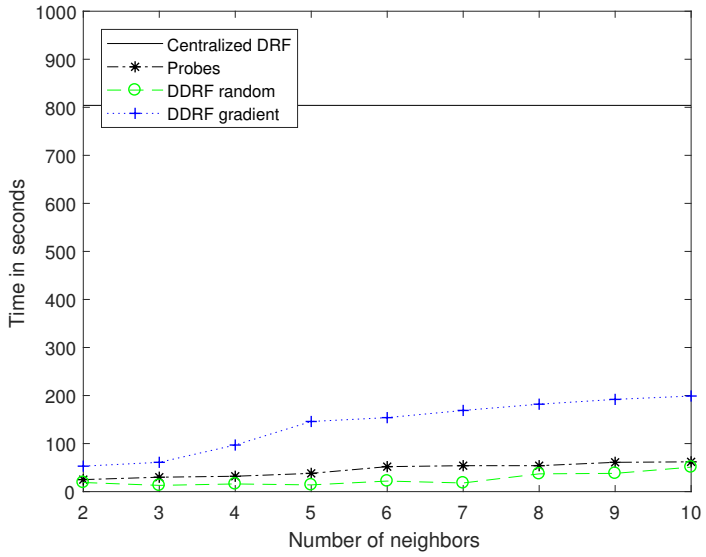

Figure 5.10: Zoomed in figure 5.9. Run-time with 200 machines and 40 users. Lower is better.

Figure 5.9 and figure 5.10 shows the execution time with 200 machines and 40 users. Figure 5.10 is a zoomed in version on the solutions with lower values. From the previous result with 100 machines and 20 users, it can be seen that DDRF gradient now is faster than the centralized DRF for all neighbor counts. The time to fully allocate the cluster for each solution seem also dependent on their fairness results. The DDRF gradient had a better fairness result for all neighbor counts compared to probes and DDRF random, and Probes had a better fairness result than DDRF random.

In figure 5.9 it can be seen though that the *PDGDRF solution* with two neighbors match the centralized DRF solution. It has a run-time of 783 seconds compared to centralized DRFs' 804 seconds. This can be compared to with 100 machines and 20 users, when it had twice as long run-time as the centralized DRF solution. The *PDGDRF solution* seems to be affected by the number of machines or users, while the *DGDRF solution* does not get closer to the centralized solution with a larger dataset, it instead seems only dependent on the number of neighbors each node has.

Further on the *PDGDRF solution* will be examined if it is possible to

speed up its allocation time by increasing the number of updates that are sent between each allocation for two neighbors, also if the allocation time is further reduced with more users or servers.

5.4.1 Investigating the PDGDRF solution

The *PDGDRF solution* was selected to be improved over the *DGDRF solution*, since it allows multiple servers to allocate tasks at the same time in parallel while still having a fair allocation. Firstly it will be investigated if the solution scales depending on the number of servers in the cluster, or the number of users. This is done by simulating the solution with both 100 machines and 200 machines with different amount of users: 10, 20, 40, 60, 80, 100, 120. The centralized DRF solution is also simulated on these data-sets, and the final result is a ratio on how the *PDGDRF solution* differs from the centralized DRF solution.

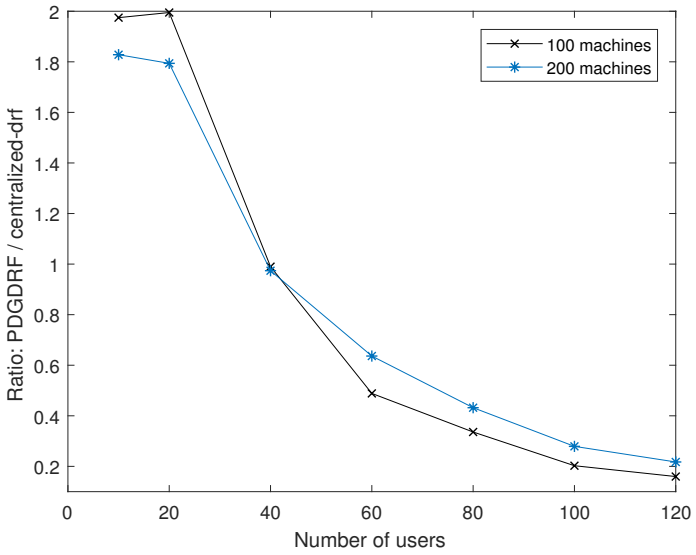

Figure 5.11: The *PDGDRF solution*s performance in ratio with the centralized DRF performance. It shows how it scales depending on the number of users. The lower, the better.

Figure 5.11 shows that for both 100 and 200 machines, the *PDGDRF solution* seem to scale based on the number of users. For both machine counts, an equal result to the centralized DRF solution was made with 40 users. For a user amount lower than 40, the *PDGDRF solution* pro-

vides a worse run-time than the centralized DRF. It can also be noted that for 100 machines, the improvement stops at 100 users, with no noticeable improvement with 120 users, meaning that it most probably scales depending on how many servers that have active users on them, and not directly the amount of users. The fairness results from these executions can be seen in figure A.1. Next it will be looked upon how frequency of gradient topology update messages affect the run-time. It will be looked upon for 20 users with 100 machines, to try and get an equal performance to the centralized DRF solution. It will also be investigated how few messages that need to be sent for 100 users and 100 machines to get an equal performance to the centralized DRF solution.

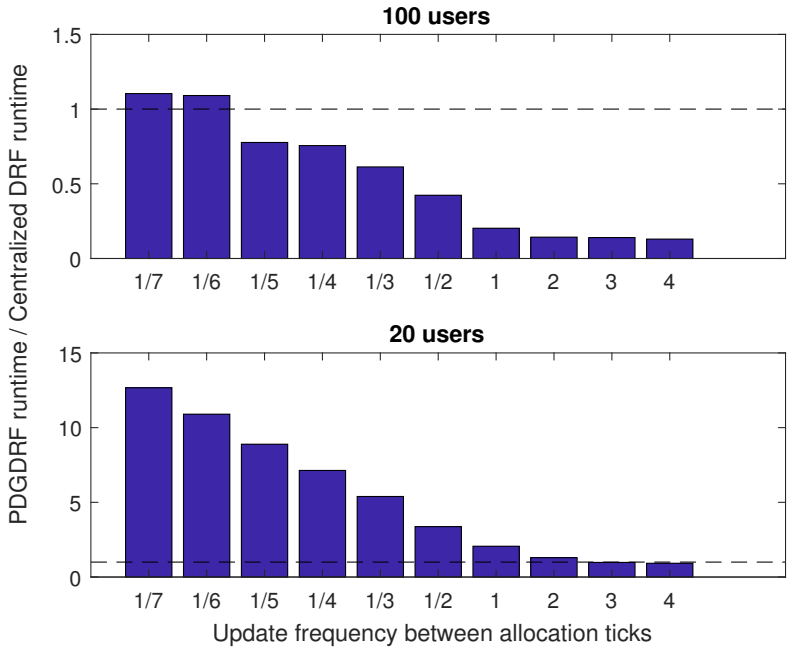

Figure 5.12: Shows how the ratio between *PDGDRF solution* and centralized DRF solution depends on the frequency of the update messages. The dashed line represents equal run-time to the centralized DRF solution. Both graphs is with 100 machines.

Figure 5.12 displays how the update frequency for gradient topology update messages affects the run-time performance. With 20 users the run-time is increasing almost linear when its not sending an update between every allocation tick. It requires 3 messages between each al-

location tick to be able to get an equal run-time as the centralized DRF solution. For 100 users instead, getting an equal performance, requires waiting at least 5 allocation ticks between updates. The fairness results for this graph can be seen in figure A.2 and figure A.3.

The performance of the *PDGDRF solution* has been observed to depend on many different type of factors. It depends on the number of neighbors, number of users and update frequency. The number of machines does not seem to affect the performance as much in regards to scaling, more machines in these data-sets have also increased the total capacity of the cluster and thus increasing the maximum number of tasks that has to be allocated.

Chapter 6

Discussion and future work

This thesis has explored different types of approaches to allocating resources to users fairly. The most accurate solution from the results was our *DGDRF solution*, which tried to mimic the DRF algorithm[7] in a distributed manner. The major problem with this solution though as seen in the results is that it has a bad scaling in terms of run-time and latency. Since it actively tries to only have one server to allocate at the same time, its performance can and should then never be faster than a centralized solution if they have the same allocation tick speed. This does not work well to solve the problem identified in Sparrow[18] that the latency in resource allocators should be reduced.

The *PDGDRF solution* on the other hand have shown potential of having a scalable speed-up based on gradient topology update frequency and number of users in the cluster. But these results only show the performance where each users dominant share is consistently growing, and where each user want to allocate endlessly. Since the *DGDRF solution* and *PDGDRF solution* only want to allocate the user with the lowest global dominant share, it does not have the *Pareto-Efficiency* property. If the lowest user does not want to allocate another task, no one else will be able to as well. This could possibly be solved by virtually growing their dominant share if no new allocation request is sent from that user, this would then allow another user to allocate. Allowing tasks to end would also give the problem of users getting a lower dominant share than before. This could lead to large fluctuations in the gradient topology.

It was observed from the results that even though the *DGDRF solution* had 0 % error, it did not achieve as good minimum global dominant share, or gini-coefficient as the centralized DRF. We believe that the difference in results is assumed to be because of the usage of random walks to locate a suitable server. The centralized server, instead could pack tasks into one server at the time, while the *DGDRF solution* picked the first one at random.

Both the solutions based on a gradient topology can have problems with shorter running tasks which was not evaluated in the thesis. When a task ends, it will decrease a users dominant share directly and the server that the user belongs to may have a higher utility than all its neighbors. This could, specifically in *PDGDRF* create an unfair allocation. In *DGDRF* this may be partially solved by checking if the server is in the gradient center. Another solution which can be tested in the future is to have two gradient topologies for the servers. One which orders the servers as seen in this thesis, where a lower dominant share means a higher utility. The other would instead be the reverse, with a higher dominant share meaning a higher utility. In the first gradient, the server with the lowest dominant share would only see servers with a lower utility than itself. In the second gradient, the neighbors for this server should be identical, since they will all have a higher utility, but with the minimum distance to the servers utility. Thus a server should be able to know if it can allocate or not by comparing the neighbors between the gradient topologies.

The probe implementation based on *Sparrows* concept of probes was used outside of its original context and problem. The idea of *Sparrow* was to allow quick allocations for short running tasks. It was also not build for heterogeneous resource demand. Setting it in the context of never ending tasks can still though show how the solution would work in those scenarios. A resource allocator/manager should be able to work for many different scenarios, both with short and long running tasks.

Both DDRF and DRFH also looked upon allocating the task on the correct server baesd on *best-fit* algorithms. This was something that was left out from this thesis since it only focused on locating the correct user to allocate for. But for a real system, this is something that should

be included to allow maximum cluster utilization. This though may give a further overhead on latency.

From the results, it is possible to see that getting a fair allocation has a large impact on the latency. So if one knows that only short tasks will be run, for example, less than the allocation time with a strongly fair solution, it may be better to use a slightly more unfair solution such as Sparrow.

This thesis did also not investigate the effect of churn on the cluster. Both DDRF, *DGDRF solution* and *PDGDRF solution* may have some problems with churn because one server is responsible for one user. Any node in Sparrow can handle any users. This is something that need to be looked at in the future, how churn can be handled effectively in the *DGDRF solution* and *PDGDRF solution*, and how that effect the run-time performance and fairness.

6.1 Ethics and sustainability

The usage of a gradient topology in a network can have sustainability concerns, because all servers need to regularly send updates to each other. With a centralized server, a working node/server can be shutdown if it does not have any tasks on it, saving on power consumption. With a gradient instead, all working nodes need to be active for the communication. This is also true for both *Sparrow* and *DDRF* as well, since they need all servers active for communication.

The concept of fairness is also a subject to ethics, as mentioned no user should envy another user, and no user should benefit from lying. This is a problem with approximate solutions such as *PDGDRF*, *Sparrow*, *DDRF*. All three solutions have a chance of different probability to create an unfair allocation. This may give a person an advantage over another in completing their work in a shorter amount of time. *DGDRF* may then be a better option in regards to ethics that the chance that someone is treated unfair is reduced.

Chapter 7

Conclusion

Four different distributed solutions have been implemented and evaluated in regards to fairness and performance (latency). Two solutions have been proposed in the thesis, the *DGDRF solution* and the *PDGDRF solution*. Both are utilizing a gradient topology overlay to create a sorted graph, based on the users dominant shares that are located on each server. The *DGDRF solution*'s purpose was to try and mimic the original DRF algorithm in a distributed manner, while the *PDGDRF solution* tries to reduce the latency for task allocation, by allowing parallel allocations, while still having a fair result.

In regards to fairness, the two proposed solutions, the *DGDRF solution* and *PDGDRF solution* both showed good results, nearly achieving similar results to a centralized solution. In terms of latency, only the *PDGDRF solution* showed potential of being able to allocate faster than a centralized solution while maintaining similar fairness results. If fairness is not as an important property, both Sparrow and DDRF can provide a lower latency while still being fairer than a FIFO-queue solution.

It has been shown that the *PDGDRF solution* scales depending on the number of users. Having more users, allows more servers to allocate tasks in parallel as long as the gini-coefficient is reduced. In regards to the problem description, the *PDGDRF solution* implements a fairness policy without access to a global view, building upon *DDRF* with the addition of a gradient topology. It also minimizes the allocation time by allowing parallel allocations. As mentioned the system scales

based on the number of users and does not require any global knowledge about all users or servers in the cluster.

The scenarios tested in this thesis does not represent the real-world though, and more extensive testing is needed, but it shows the potential of using a gradient overlay to distribute Dominant Resource Fairness.

Bibliography

[1] Cosmin Arad and Seif Haridi. "Kompics: A Message-Passing Component Model for Building Distributed Systems". In: *SICS Techinical Report* T2010:04 (2010).

[2] Eric Boutin et al. "Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing". In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, 2014, pp. 285–300. ISBN: 978-1-931971-16-4. URL: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/boutin.

[3] Christian Damgaard. "Gini Coefficient". In: MathWorld–A Wolfram Web Resource created by Eric W. Weisstein. http://mathworld.wolfram.com/GiniCoefficient.html.

[4] Christian Damgaard. "Lorenz Curve". In: MathWorld–A Wolfram Web Resource created by Eric W. Weisstein. http://mathworld.wolfram.com/LorenzCurve.html.

[5] J. Dowling and A. H. Payberah. "Shuffling with a Croupier: Nat-Aware Peer-Sampling". In: *2012 IEEE 32nd International Conference on Distributed Computing Systems*. June 2012, pp. 102–111. DOI: 10.1109/ICDCS.2012.19.

[6] Jacob Fox. *Lecture 22: Eigenvalues and expanders*. http://math.mit.edu/~fox/MAT307-lecture22.pdf.

[7] Ali Ghodsi et al. "Dominant Resource Fairness: Fair Allocation of Multiple Resource Types". In: *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*. NSDI'11. Boston, MA: USENIX Association, 2011, pp. 323–336. URL: http://dl.acm.org/citation.cfm?id=1972457.1972490.

[8] Rachid Guerraoui and André Schiper. "Fault-tolerance by replication in distributed systems". In: *Reliable Software Technologies — Ada-Europe '96: 1996 Ada-Europe International Conference on Reliable Software Technologies Montreux, Switzerland, June 10–14, 1996 Proceedings*. Ed. by Alfred Strohmeier. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 38–57. ISBN: 978-3-540-68457-2. DOI: 10.1007/BFb0013477. URL: http://dx.doi.org/10.1007/BFb0013477.

[9] Benjamin Hindman et al. "Mesos: A Platform for Fine-grained Resource Sharing in the Data Center". In: *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*. NSDI'11. Boston, MA: USENIX Association, 2011, pp. 295–308. URL: http://dl.acm.org/citation.cfm?id=1972457.1972488.

[10] Hortonworks. *Hortonworks data platform, Yarn resource management*. http://docs.hortonworks.com/HDPDocuments/HDP2/HDP-2.4.3/bk_yarn_resource_mgt/bk_yarn_resource_mgt-20160902.pdf. Sept. 2016.

[11] J. Huang, D. M. Nicol, and R. H. Campbell. "Denial-of-Service Threat to Hadoop/YARN Clusters with Multi-tenancy". In: *2014 IEEE International Congress on Big Data*. June 2014, pp. 48–55. DOI: 10.1109/BigData.Congress.2014.17.

[12] Márk Jelasity. "Gossip-based Protocols for Large-scale Distributed Systems". In: Apr. 2013.

[13] Márk Jelasity and Ozalp Babaoglu. "T-Man: Gossip-Based Overlay Topology Management". In: *Engineering Self-Organising Systems: Third International Workshop, ESOA 2005, Utrecht, The Netherlands, July 25, 2005, Revised Selected Papers*. Ed. by Sven A. Brueckner et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 1–15. ISBN: 978-3-540-33352-4. DOI: 10.1007/11734697_1. URL: http://dx.doi.org/10.1007/11734697_1.

[14] Márk Jelasity, Alberto Montresor, and Ozalp Babaoglu. "Gossip-based Aggregation in Large Dynamic Networks". In: *ACM Trans. Comput. Syst.* 23.3 (Aug. 2005), pp. 219–252. ISSN: 0734-2071. DOI: 10.1145/1082469.1082470. URL: http://doi.acm.org/10.1145/1082469.1082470.

[15] D. Kempe, A. Dobra, and J. Gehrke. "Gossip-based computation of aggregate information". In: *44th Annual IEEE Symposium on Foundations of Computer Science, 2003. Proceedings.* Oct. 2003, pp. 482–491. DOI: 10.1109/SFCS.2003.1238221.

[16] *Kompics Toolbox*. https://github.com/Decentrify/KompicsToolbox.

[17] Ivan Marsic. *Max-min Fair Share Algorithm*. http://www.ece.rutgers.edu/~marsic/Teaching/CCN/minmax-fairsh.html. Nov. 1998.

[18] Kay Ousterhout et al. "Sparrow: Distributed, Low Latency Scheduling". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. Farminton, Pennsylvania: ACM, 2013, pp. 69–84. ISBN: 978-1-4503-2388-8. DOI: 10.1145/2517349.2522716. URL: http://doi.acm.org/10.1145/2517349.2522716.

[19] Amir Payberah. *Dominant Resource Fairness*. https://www.sics.se/~amir/id2221/slides/mesos_yarn.pdf. Oct. 2016.

[20] Amir H. Payberah. "Live Streaming in P2P and Hybrid P2P-Cloud Environments for the Open Internet". QC 20130524. PhD thesis. KTH, Software and Computer systems, SCS, 2013, pp. x, 107.

[21] United Nations Development Programme. *Human development report 2003*. http://www.unic.un.org.pl/hdr/hdr2003/hdr03_complete.pdf. 2003.

[22] Charles Reiss et al. "Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis". In: *Proceedings of the Third ACM Symposium on Cloud Computing*. SoCC '12. San Jose, California: ACM, 2012, 7:1–7:13. ISBN: 978-1-4503-1761-0. DOI: 10.1145/2391229.2391236. URL: http://doi.acm.org/10.1145/2391229.2391236.

[23] Jan Sacha et al. "Decentralising a service-oriented architecture". In: *Peer-to-Peer Networking and Applications* 3.4 (2010), pp. 323–350. ISSN: 1936-6450. DOI: 10.1007/s12083-009-0062-6. URL: http://dx.doi.org/10.1007/s12083-009-0062-6.

[24] Devavrat Shah. "Gossip Algorithms." In: *Foundations and Trends in Networking* 3.1 (July 26, 2009), pp. 1–125. URL: http://dblp.uni-trier.de/db/journals/ftnet/ftnet3.html#Shah09.

[25] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. 8th. Wiley Publishing, 2008. ISBN: 0470128720.

[26] Hakan Terelius et al. "Converging an Overlay Network to a Gradient Topology". In: 2011.

[27] Hakan Terelius et al. "Converging an Overlay Network to a Gradient Topology". In: 2011.

[28] Vinod Kumar Vavilapalli et al. "Apache Hadoop YARN: Yet Another Resource Negotiator". In: *Proceedings of the 4th Annual Symposium on Cloud Computing*. SOCC '13. Santa Clara, California: ACM, 2013, 5:1–5:16. ISBN: 978-1-4503-2428-1. DOI: 10.1145/2523616.2523633. URL: http://doi.acm.org/10.1145/2523616.2523633.

[29] Ryan O'Donnell Venkatesan Guruswami. *Lecture 24: Expander Graphs*. https://www.cs.cmu.edu/~odonnell/complexity/lecture24.pdf.

[30] Abhishek Verma et al. "Large-scale cluster management at Google with Borg". In: *Proceedings of the European Conference on Computer Systems (EuroSys)*. Bordeaux, France, 2015.

[31] Spyros Voulgaris, Daniela Gavidia, and Maarten van Steen. "CYCLON: Inexpensive Membership Management for Unstructured P2P Overlays". In: *Journal of Network and Systems Management* 13.2 (2005), pp. 197–217. ISSN: 1573-7705. DOI: 10.1007/s10922-005-4441-x. URL: http://dx.doi.org/10.1007/s10922-005-4441-x.

[32] K. Wang et al. "Optimizing load balancing and data-locality with data-aware scheduling". In: *2014 IEEE International Conference on Big Data (Big Data)*. Oct. 2014, pp. 119–128. DOI: 10.1109/BigData.2014.7004220.

[33] Wei Wang, Baochun Li, and Ben Liang. "Dominant Resource Fairness in Cloud Computing Systems with Heterogeneous Servers". In: *CoRR* abs/1308.0083 (2013). URL: http://arxiv.org/abs/1308.0083.

[34] F. Wuhib, R. Stadler, and M. Spreitzer. "Gossip-based resource management for cloud environments". In: *2010 International Conference on Network and Service Management*. Oct. 2010, pp. 1–8. DOI: 10.1109/CNSM.2010.5691347.

[35] Qinyun Zhu and Jae C. Oh. "An Approach to Dominant Resource Fairness in Distributed Environment". In: *Current Approaches in Applied Artificial Intelligence: 28th International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems, IEA/AIE 2015, Seoul, South Korea, June 10-12, 2015, Proceedings*. Ed. by Moonis Ali et al. Cham: Springer International Publishing, 2015, pp. 141–150. ISBN: 978-3-319-19066-2. DOI: 10.1007/978-3-319-19066-2_14. URL: http://dx.doi.org/10.1007/978-3-319-19066-2_14.

Appendix A

Fairness results

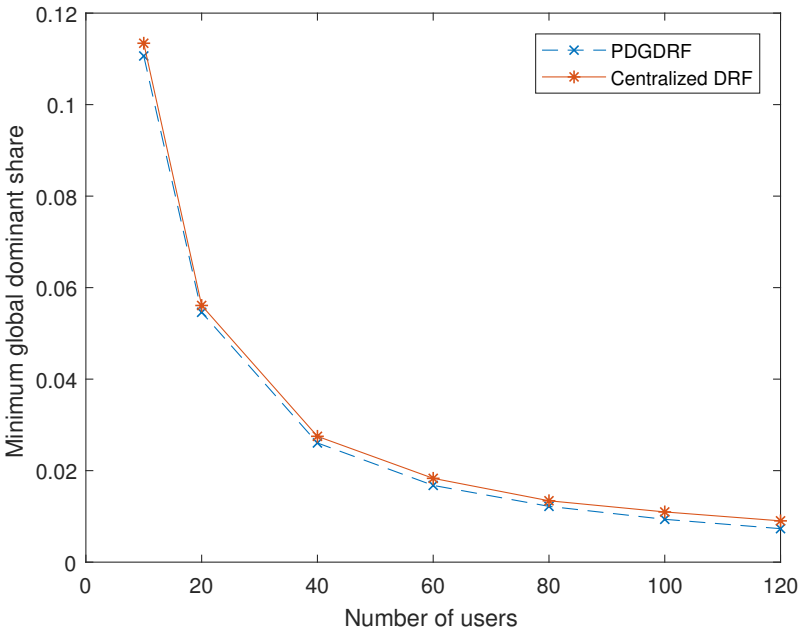

Figure A.1: The minimum global dominant share results from the executions shown in figure 5.11. Compares the centralized solution against PDGDRF.

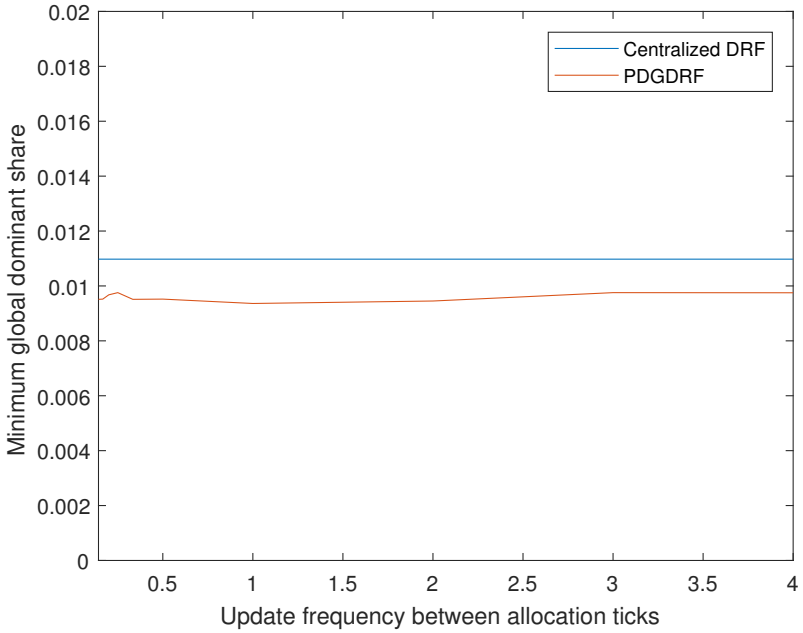

Figure A.2: The minimum global dominant share results from the executions shown in figure 5.12, with 100 users. Compares the centralized solution against PDGDRF.

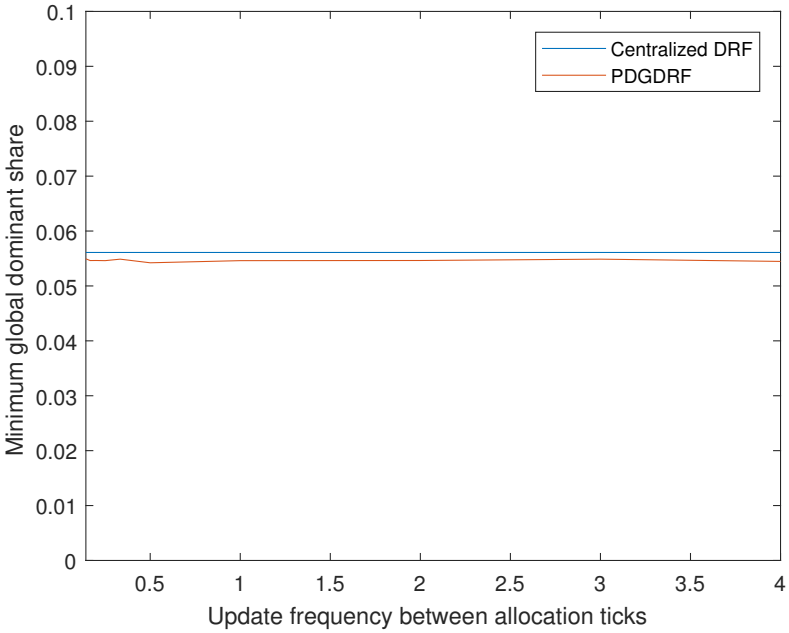

Figure A.3: The minimum global dominant share results from the executions shown in figure 5.12, with 20 users. Compares the centralized solution against PDGDRF.